umap Documentation

Release 0.5

Leland McInnes

Jan 09, 2021

User Guide / Tutorial:

Uniform Manifold Approximation and Projection (UMAP) is a dimension reduction technique that can be used for visualisation similarly to t-SNE, but also for general non-linear dimension reduction. The algorithm is founded on three assumptions about the data

1. The data is uniformly distributed on Riemannian manifold;
2. The Riemannian metric is locally constant (or can be approximated as such);
3. The manifold is locally connected.

From these assumptions it is possible to model the manifold with a fuzzy topological structure. The embedding is found by searching for a low dimensional projection of the data that has the closest possible equivalent fuzzy topological structure.

The details for the underlying mathematics can be found in our paper on ArXiv:

McInnes, L, Healy, J, *UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction*, ArXiv e-prints 1802.03426, 2018

You can find the software on github.

Installation

Conda install, via the excellent work of the conda-forge team:

```
conda install -c conda-forge umap-learn
```

The conda-forge packages are available for linux, OS X, and Windows 64 bit.

PyPI install, presuming you have numba and sklearn and all its requirements (numpy and scipy) installed:

```
pip install umap-learn
```

CHAPTER 1

How to Use UMAP

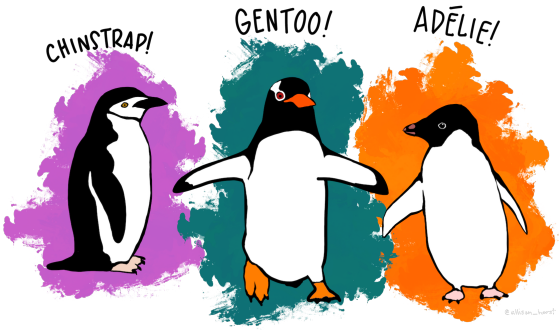
UMAP is a general purpose manifold learning and dimension reduction algorithm. It is designed to be compatible with scikit-learn, making use of the same API and able to be added to sklearn pipelines. If you are already familiar with sklearn you should be able to use UMAP as a drop in replacement for t-SNE and other dimension reduction classes. If you are not so familiar with sklearn this tutorial will step you through the basics of using UMAP to transform and visualise data.

First we'll need to import a bunch of useful tools. We will need numpy obviously, but we'll use some of the datasets available in sklearn, as well as the `train_test_split` function to divide up data. Finally we'll need some plotting tools (matplotlib and seaborn) to help us visualise the results of UMAP, and pandas to make that a little easier.

```
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
%matplotlib inline
```

```
sns.set(style='white', context='notebook', rc={'figure.figsize':(14,10)})
```

1.1 Penguin data



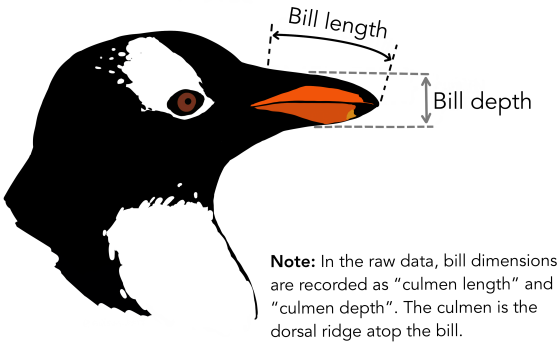
The next step is to get some data to work with. To ease us into things we'll start with the penguin dataset. It isn't very representative of what real data would look like, but it is small both in number of points and number of features, and will let us get an idea of what the dimension reduction is doing.

```
penguins = pd.read_csv("https://github.com/allisonhorst/palmerpenguins/raw/
↪5b5891f01b52ae26ad8cb9755ec93672f49328a8/data/penguins_size.csv")
penguins.head()
```

Since this is for demonstration purposes we will get rid of the NAs in the data; in a real world setting one would wish to take more care with proper handling of missing data.

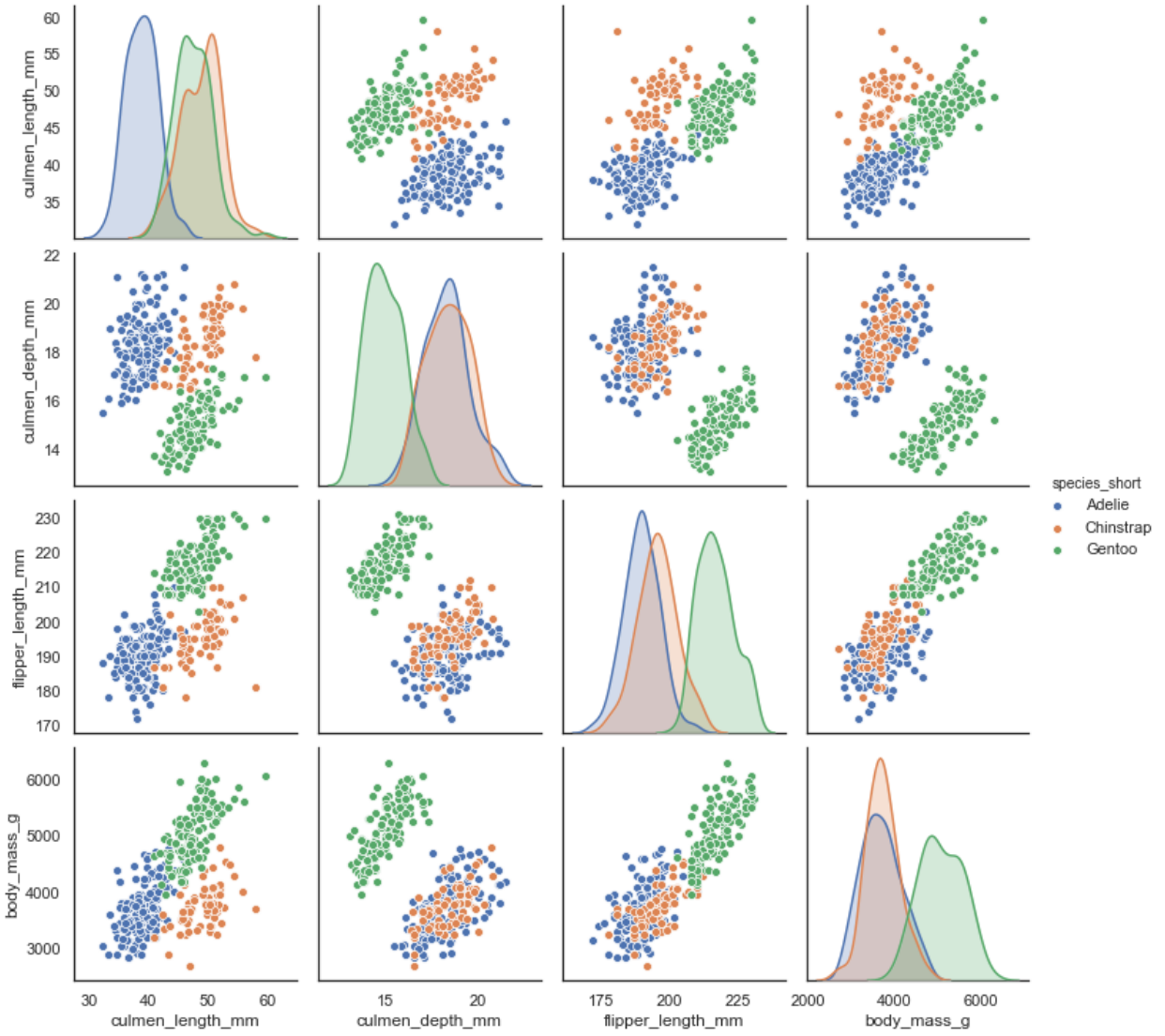
```
penguins = penguins.dropna()
penguins.species_short.value_counts()
```

```
Adelie       146
Gentoo       120
Chinstrap     68
Name: species_short, dtype: int64
```



See the github repostiory for more details about the daataset itself. It consists of measurements of bill (culmen) and flippers and weights of three species of penguins, along with some other metadata about the penguins. In total we have 334 different penguins measured. Visualizing this data is a little bit tricky since we can't plot in 4 dimensions easily. Fortunately four is not that large a number, so we can just to a pairwise feature scatterplot matrix to get an ideas of what is going on. Seaborn makes this easy.

```
sns.pairplot(penguins, hue='species_short')
```



This gives us some idea of what the data looks like by giving as all the 2D views of the data. Four dimensions is low enough that we can (sort of) reconstruct what the full dimensional data looks like in our heads. Now that we sort of know what we are looking at, the question is what can a dimension reduction technique like UMAP do for us? By reducing the dimension in a way that preserves as much of the structure of the data as possible we can get a visualisable representation of the data allowing us to "see" the data and its structure and begin to get some intuition about the data itself.

To use UMAP for this task we need to first construct a UMAP object that will do the job for us. That is as simple as instantiating the class. So let's import the umap library and do that.

```
import umap
```

```
reducer = umap.UMAP()
```

Before we can do any work with the data it will help to clean up it a little. We won't need NAs, we just want the measurement columns, and since the measurements are on entirely different scales it will be helpful to convert each feature into z-scores (number of standard deviations from the mean) for comparability.

```
penguin_data = penguins[
    [
        "culmen_length_mm",
        "culmen_depth_mm",
        "flipper_length_mm",
        "body_mass_g",
    ]
].values
scaled_penguin_data = StandardScaler().fit_transform(penguin_data)
```

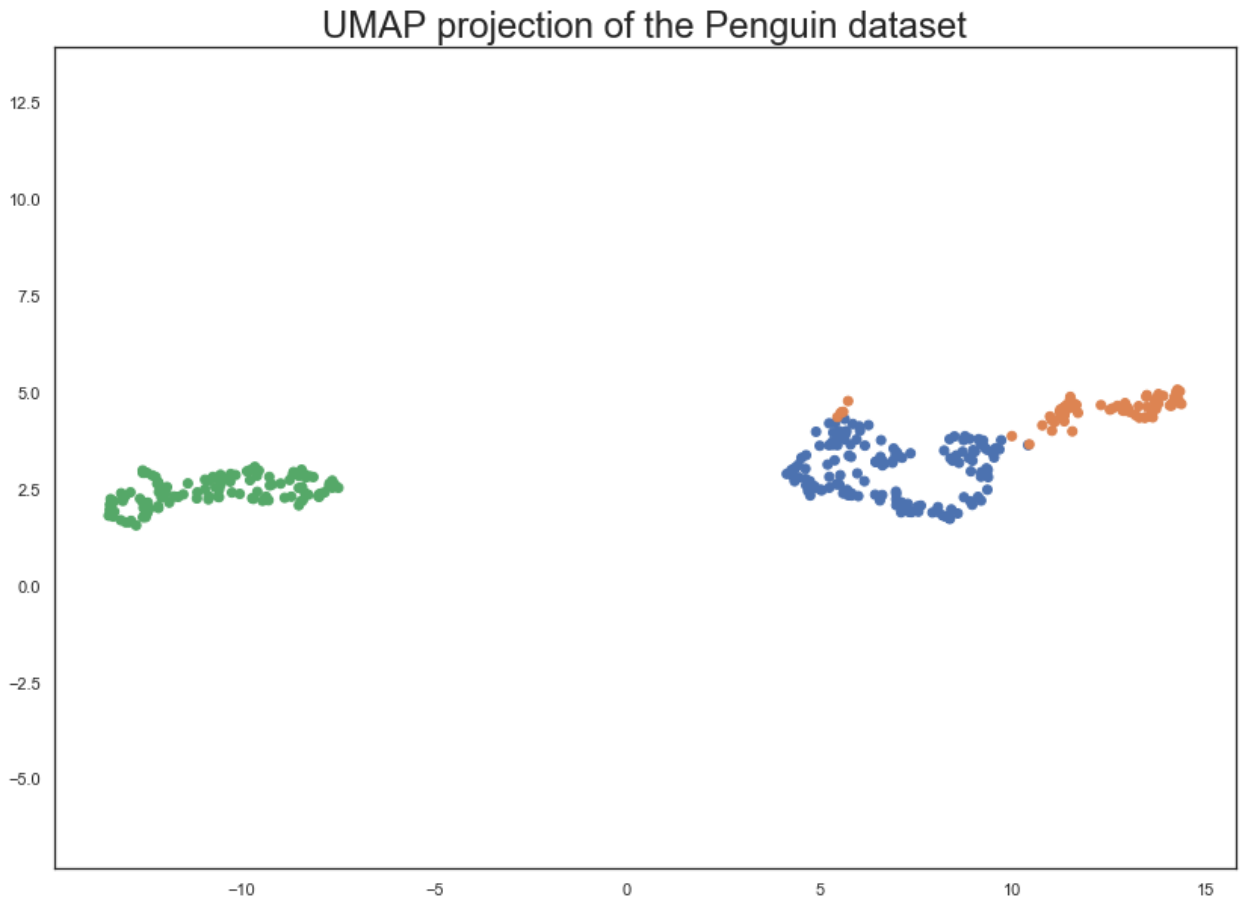
Now we need to train our reducer, letting it learn about the manifold. For this UMAP follows the sklearn API and has a method `fit` which we pass the data we want the model to learn from. Since, at the end of the day, we are going to want to reduced representation of the data we will use, instead, the `fit_transform` method which first calls `fit` and then returns the transformed data as a numpy array.

```
embedding = reducer.fit_transform(scaled_penguin_data)
embedding.shape
```

```
(334, 2)
```

The result is an array with 334 samples, but only two feature columns (instead of the four we started with). This is because, by default, UMAP reduces down to 2D. Each row of the array is a 2-dimensional representation of the corresponding penguin. Thus we can plot the `embedding` as a standard scatterplot and color by the target array (since it applies to the transformed data which is in the same order as the original).

```
plt.scatter(
    embedding[:, 0],
    embedding[:, 1],
    c=[sns.color_palette()[x] for x in penguins.species_short.map({"Adelie":0,
↪"Chinstrap":1, "Gentoo":2})])
plt.gca().set_aspect('equal', 'datalim')
plt.title('UMAP projection of the Penguin dataset', fontsize=24)
```



This does a useful job of capturing the structure of the data, and as can be seen from the matrix of scatterplots this is relatively accurate. Of course we learned at least this much just from that matrix of scatterplots – which we could do since we only had four different dimensions to analyse. If we had data with a larger number of dimensions the scatterplot matrix would quickly become unwieldy to plot, and far harder to interpret. So moving on from the Penguin dataset, let's consider the digits dataset.

1.2 Digits data

First we will load the dataset from sklearn.

```
digits = load_digits()
print(digits.DESCR)
```

```
.. _digits_dataset:

Optical recognition of handwritten digits dataset
--------------------------------------------------

Data Set Characteristics:

    :Number of Instances: 5620
    :Number of Attributes: 64
    :Attribute Information: 8x8 image of integer pixels in the range 0..16.
    :Missing Attribute Values: None
```

```
    :Creator: E. Alpaydin (alpaydin '@' boun.edu.tr)
    :Date: July; 1998

This is a copy of the test set of the UCI ML hand-written digits datasets
https://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+
Digits

The data set contains images of hand-written digits: 10 classes where
each class refers to a digit.

Preprocessing programs made available by NIST were used to extract
normalized bitmaps of handwritten digits from a preprinted form. From a
total of 43 people, 30 contributed to the training set and different 13
to the test set. 32x32 bitmaps are divided into nonoverlapping blocks of
4x4 and the number of on pixels are counted in each block. This generates
an input matrix of 8x8 where each element is an integer in the range
0..16. This reduces dimensionality and gives invariance to small
distortions.

For info on NIST preprocessing routines, see M. D. Garris, J. L. Blue, G.
T. Candela, D. L. Dimmick, J. Geist, P. J. Grother, S. A. Janet, and C.
L. Wilson, NIST Form-Based Handprint Recognition System, NISTIR 5469,
1994.

.. topic:: References

  - C. Kaynak (1995) Methods of Combining Multiple Classifiers and Their
    Applications to Handwritten Digit Recognition, MSc Thesis, Institute of
    Graduate Studies in Science and Engineering, Bogazici University.
  - E. Alpaydin, C. Kaynak (1998) Cascading Classifiers, Kybernetika.
  - Ken Tang and Ponnuthurai N. Suganthan and Xi Yao and A. Kai Qin.
    Linear dimensionalityreduction using relevance weighted LDA. School of
    Electrical and Electronic Engineering Nanyang Technological University.
    2005.
  - Claudio Gentile. A New Approximate Maximal Margin Classification
    Algorithm. NIPS. 2000.
```

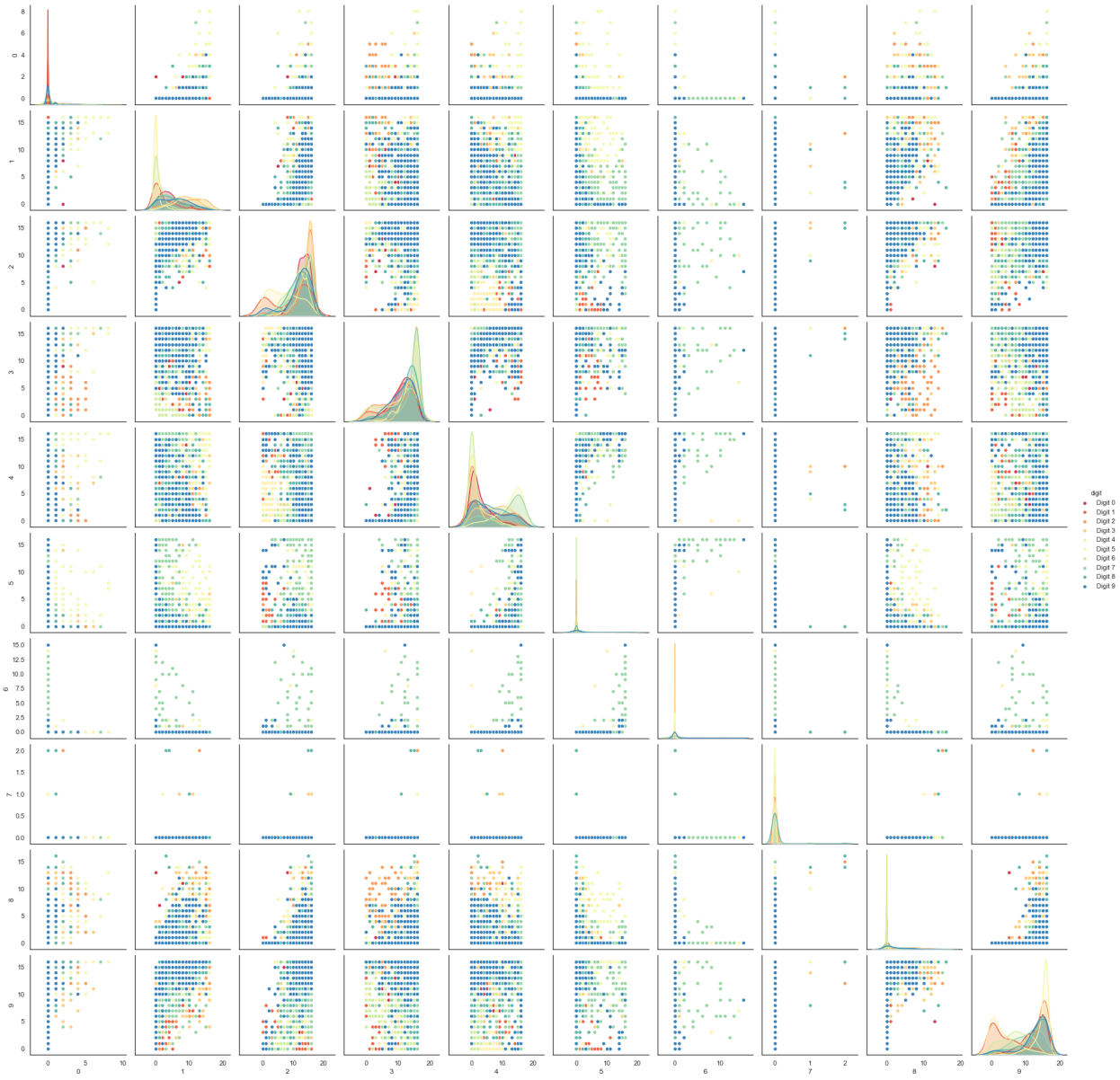
We can plot a number of the images to get an idea of what we are looking at. This just involves matplotlib building a grid of axes and then looping through them plotting an image into each one in turn.

```
fig, ax_array = plt.subplots(20, 20)
axes = ax_array.flatten()
for i, ax in enumerate(axes):
    ax.imshow(digits.images[i], cmap='gray_r')
plt.setp(axes, xticks=[], yticks=[], frame_on=False)
plt.tight_layout(h_pad=0.5, w_pad=0.01)
```

As you can see these are quite low resolution images – for the most part they are recognisable as digits, but there are a number of cases that are sufficiently blurred as to be questionable even for a human to guess at. The zeros do stand out as the easiest to pick out as notably different and clearly zeros. Beyond that things get a little harder: some of the squashed thing eights look awfully like ones, some of the threes start to look a little like crossed sevens when drawn badly, and so on.

Each image can be unfolded into a 64 element long vector of grayscale values. It is these 64 dimensional vectors that we wish to analyse: how much of the digits structure can we discern? At least in principle 64 dimensions is overkill for this task, and we would reasonably expect that there should be some smaller number of "latent" features that would be sufficient to describe the data reasonably well. We can try a scatterplot matrix – in this case just of the first 10 dimensions so that it is at least plottable, but as you can quickly see that approach is not going to be sufficient for this data.

```
digits_df = pd.DataFrame(digits.data[:,1:11])
digits_df['digit'] = pd.Series(digits.target).map(lambda x: 'Digit {}'.format(x))
sns.pairplot(digits_df, hue='digit', palette='Spectral')
```



In contrast we can try using UMAP again. It works exactly as before: construct a model, train the model, and then look at the transformed data. To demonstrate more of UMAP we'll go about it differently this time and simply use the `fit` method rather than the `fit_transform` approach we used for Penguins.

```
reducer = umap.UMAP(random_state=42)
reducer.fit(digits.data)
```

```
UMAP(a=None, angular_rp_forest=False, b=None,
     force_approximation_algorithm=False, init='spectral', learning_rate=1.0,
     local_connectivity=1.0, low_memory=False, metric='euclidean',
     metric_kwds=None, min_dist=0.1, n_components=2, n_epochs=None,
     n_neighbors=15, negative_sample_rate=5, output_metric='euclidean',
     output_metric_kwds=None, random_state=42, repulsion_strength=1.0,
     set_op_mix_ratio=1.0, spread=1.0, target_metric='categorical',
     target_metric_kwds=None, target_n_neighbors=-1, target_weight=0.5,
     transform_queue_size=4.0, transform_seed=42, unique=False, verbose=False)
```

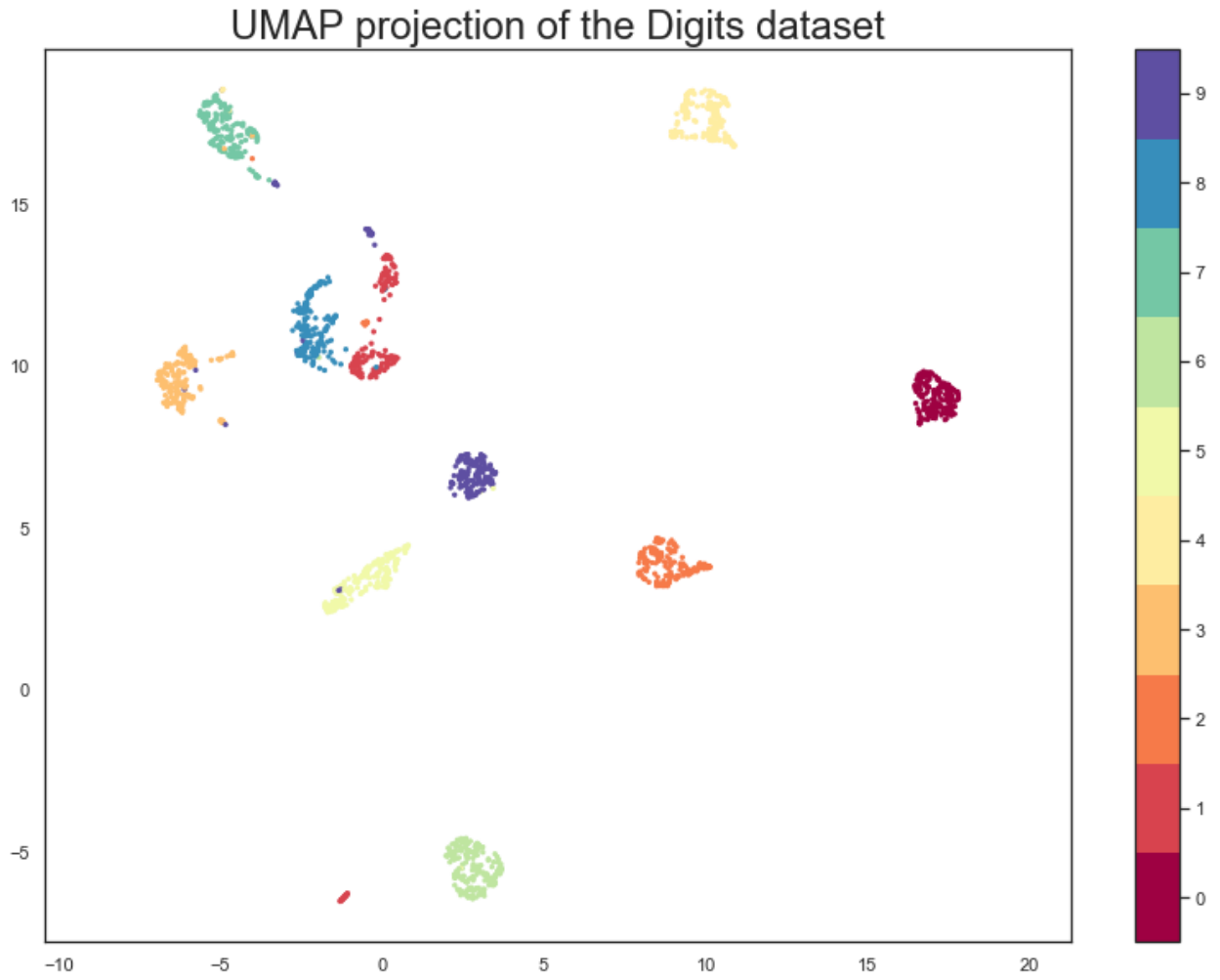
Now, instead of returning an embedding we simply get back the reducer object, now having trained on the dataset we passed it. To access the resulting transform we can either look at the `embedding_` attribute of the reducer object, or call transform on the original data.

```
embedding = reducer.transform(digits.data)
# Verify that the result of calling transform is
# idenitical to accessing the embedding_ attribute
assert(np.all(embedding == reducer.embedding_))
embedding.shape
```

```
(1797, 2)
```

We now have a dataset with 1797 rows (one for each hand-written digit sample), but only 2 columns. As with the Penguins example we can now plot the resulting embedding, coloring the data points by the class that they belong to (i.e. the digit they represent).

```
plt.scatter(embedding[:, 0], embedding[:, 1], c=digits.target, cmap='Spectral', s=5)
plt.gca().set_aspect('equal', 'datalim')
plt.colorbar(boundaries=np.arange(11)-0.5).set_ticks(np.arange(10))
plt.title('UMAP projection of the Digits dataset', fontsize=24);
```



We see that UMAP has successfully captured the digit classes. There are also some interesting effects as some digit classes blend into one another (see the eights, ones, and sevens, with some nines in between), and also cases where

digits are pushed away as clearly distinct (the zeros on the right, the fours at the top, and a small subcluster of ones at the bottom come to mind). To get a better idea of why UMAP chose to do this it is helpful to see the actual digits involve. One can do this using bokeh and mouseover tooltips of the images.

First we'll need to encode all the images for inclusion in a dataframe.

```
from io import BytesIO
from PIL import Image
import base64
```

```
def embeddable_image(data):
    img_data = 255 - 15 * data.astype(np.uint8)
    image = Image.fromarray(img_data, mode='L').resize((64, 64), Image.BICUBIC)
    buffer = BytesIO()
    image.save(buffer, format='png')
    for_encoding = buffer.getvalue()
    return '' + base64.b64encode(for_encoding).decode()
```

Next we need to load up bokeh and the various tools from it that will be needed to generate a suitable interactive plot.

```
from bokeh.plotting import figure, show, output_notebook
from bokeh.models import HoverTool, ColumnDataSource, CategoricalColorMapper
from bokeh.palettes import Spectral10

output_notebook()
```

Finally we generate the plot itself with a custom hover tooltip that embeds the image of the digit in question in it, along with the digit class that the digit is actually from (this can be useful for digits that are hard even for humans to classify correctly).

```
digits_df = pd.DataFrame(embedding, columns=('x', 'y'))
digits_df['digit'] = [str(x) for x in digits.target]
digits_df['image'] = list(map(embeddable_image, digits.images))

datasource = ColumnDataSource(digits_df)
color_mapping = CategoricalColorMapper(factors=[str(9 - x) for x in digits.target_
↪names],
                                       palette=Spectral10)

plot_figure = figure(
    title='UMAP projection of the Digits dataset',
    plot_width=600,
    plot_height=600,
    tools=('pan, wheel_zoom, reset')
)

plot_figure.add_tools(HoverTool(tooltips="""
<div>
    <div>
        <img src='@image' style='float: left; margin: 5px 5px 5px 5px'/>
    </div>
    <div>
        <span style='font-size: 16px; color: #224499'>Digit:</span>
        <span style='font-size: 18px'>@digit</span>
    </div>
</div>
"""))
```

(continues on next page)

(continued from previous page)

```
plot_figure.circle(
    'x',
    'y',
    source=datasource,
    color=dict(field='digit', transform=color_mapping),
    line_alpha=0.6,
    fill_alpha=0.6,
    size=4
)
show(plot_figure)
```

As can be seen, the nines that blend between the ones and the sevens are odd looking nines (that aren't very rounded) and do, indeed, interpolate surprisingly well between ones with hats and crossed sevens. In contrast the small disjoint cluster of ones at the bottom of the plot is made up of ones with feet (a horizontal line at the base of the one) which are, indeed, quite distinct from the general mass of ones.

This concludes our introduction to basic UMAP usage – hopefully this has given you the tools to get started for yourself. Further tutorials, covering UMAP parameters and more advanced usage are also available when you wish to dive deeper.

Penguin data information

Peguin data are from:

Gorman KB, Williams TD, Fraser WR (2014) Ecological Sexual Dimorphism and Environmental Variability within a Community of Antarctic Penguins (Genus *Pygoscelis*). PLoS ONE 9(3): e90081. doi:10.1371/journal.pone.0090081

See the full paper HERE.

Original data access and use

From Gorman et al.: "Data reported here are publicly available within the PAL-LTER data system (datasets #219, 220, and 221): http://oceaninformatics.ucsd.edu/datazoo/data/pallter/datasets. These data are additionally archived within the United States (US) LTER Network's Information System Data Portal: https://portal.lternet.edu/. Individuals interested in using these data are therefore expected to follow the US LTER Network's Data Access Policy, Requirements and Use Agreement: https://lternet.edu/data-access-policy/."

Anyone interested in publishing the data should contact Dr. Kristen Gorman about analysis and working together on any final products.

Penguin images by Alison Horst.

CHAPTER 2

Basic UMAP Parameters

UMAP is a fairly flexible non-linear dimension reduction algorithm. It seeks to learn the manifold structure of your data and find a low dimensional embedding that preserves the essential topological structure of that manifold. In this notebook we will generate some visualisable 4-dimensional data, demonstrate how to use UMAP to provide a 2-dimensional representation of it, and then look at how various UMAP parameters can impact the resulting embedding. This documentation is based on the work of Philippe Rivière for visionscarto.net.

To start we'll need some basic libraries. First `numpy` will be needed for basic array manipulation. Since we will be visualising the results we will need `matplotlib` and `seaborn`. Finally we will need `umap` for doing the dimension reduction itself.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import seaborn as sns
import umap
%matplotlib inline
```

```
sns.set(style='white', context='poster', rc={'figure.figsize':(14,10)})
```

Next we will need some data to embed into a lower dimensional representation. To make the 4-dimensional data "visualisable" we will generate data uniformly at random from a 4-dimensional cube such that we can interpret a sample as a tuple of (R,G,B,a) values specifying a color (and translucency). Thus when we plot low dimensional representations each point can be colored according to its 4-dimensional value. For this we can use `numpy`. We will fix a random seed for the sake of consistency.

```
np.random.seed(42)
data = np.random.rand(800, 4)
```

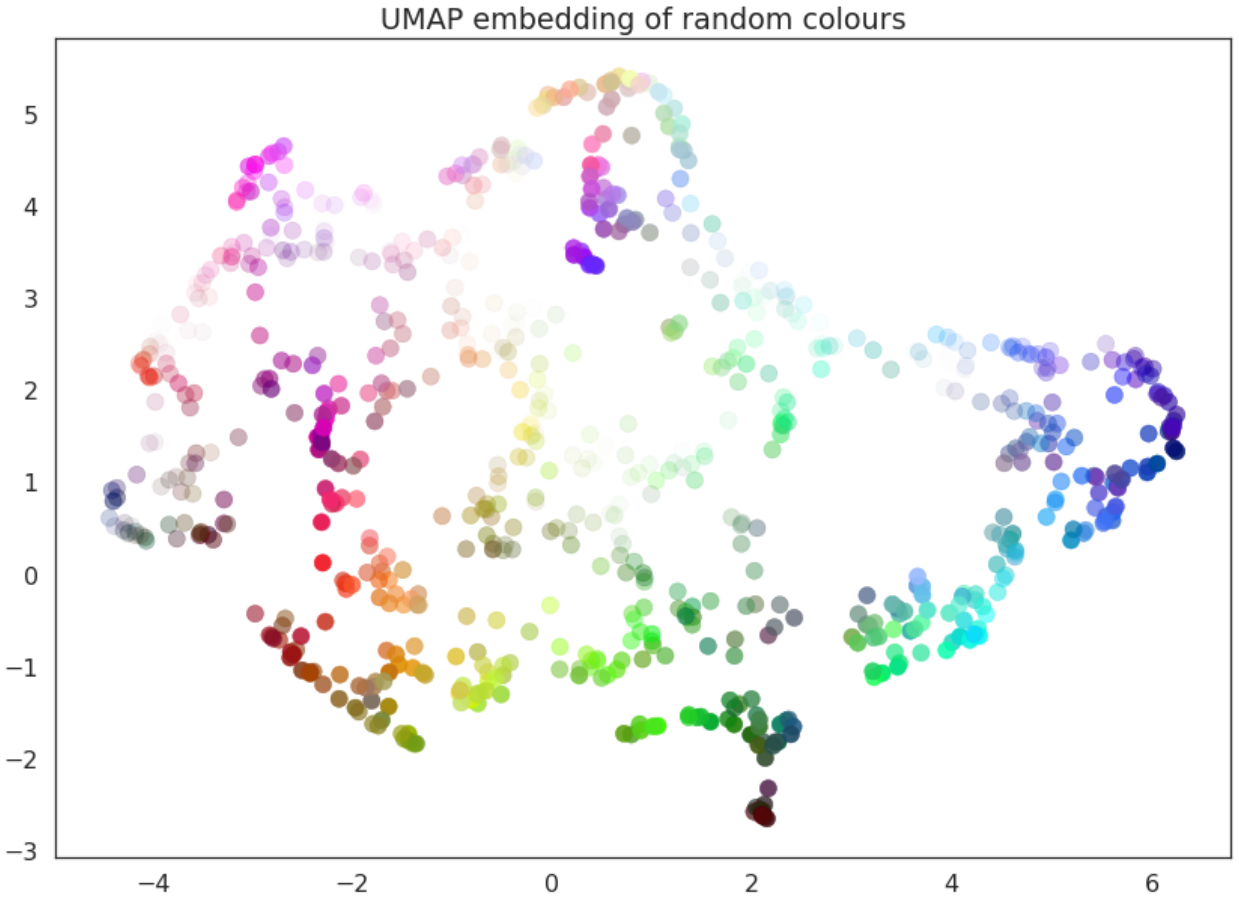
Now we need to find a low dimensional representation of the data. As in the Basic Usage documentation, we can do this by using the *`fit_transform()`* method on a *`UMAP`* object.

```
fit = umap.UMAP()
%time u = fit.fit_transform(data)
```

```
CPU times: user 7.73 s, sys: 211 ms, total: 7.94 s
Wall time: 6.8 s
```

The resulting value `u` is a 2-dimensional representation of the data. We can visualise the result by using `matplotlib` to draw a scatter plot of `u`. We can color each point of the scatter plot by the associated 4-dimensional color from the source data.

```
plt.scatter(u[:,0], u[:,1], c=data)
plt.title('UMAP embedding of random colours');
```



As you can see the result is that the data is placed in 2-dimensional space such that points that were nearby in 4-dimensional space (i.e. are similar colors) are kept close together. Since we drew a random selection of points in the color cube there is a certain amount of induced structure from where the random points happened to clump up in color space.

UMAP has several hyperparameters that can have a significant impact on the resulting embedding. In this notebook we will be covering the four major ones:

- `n_neighbors`
- `min_dist`
- `n_components`
- `metric`

Each of these parameters has a distinct effect, and we will look at each in turn. To make exploration simpler we will first write a short utility function that can fit the data with UMAP given a set of parameter choices, and plot the result.

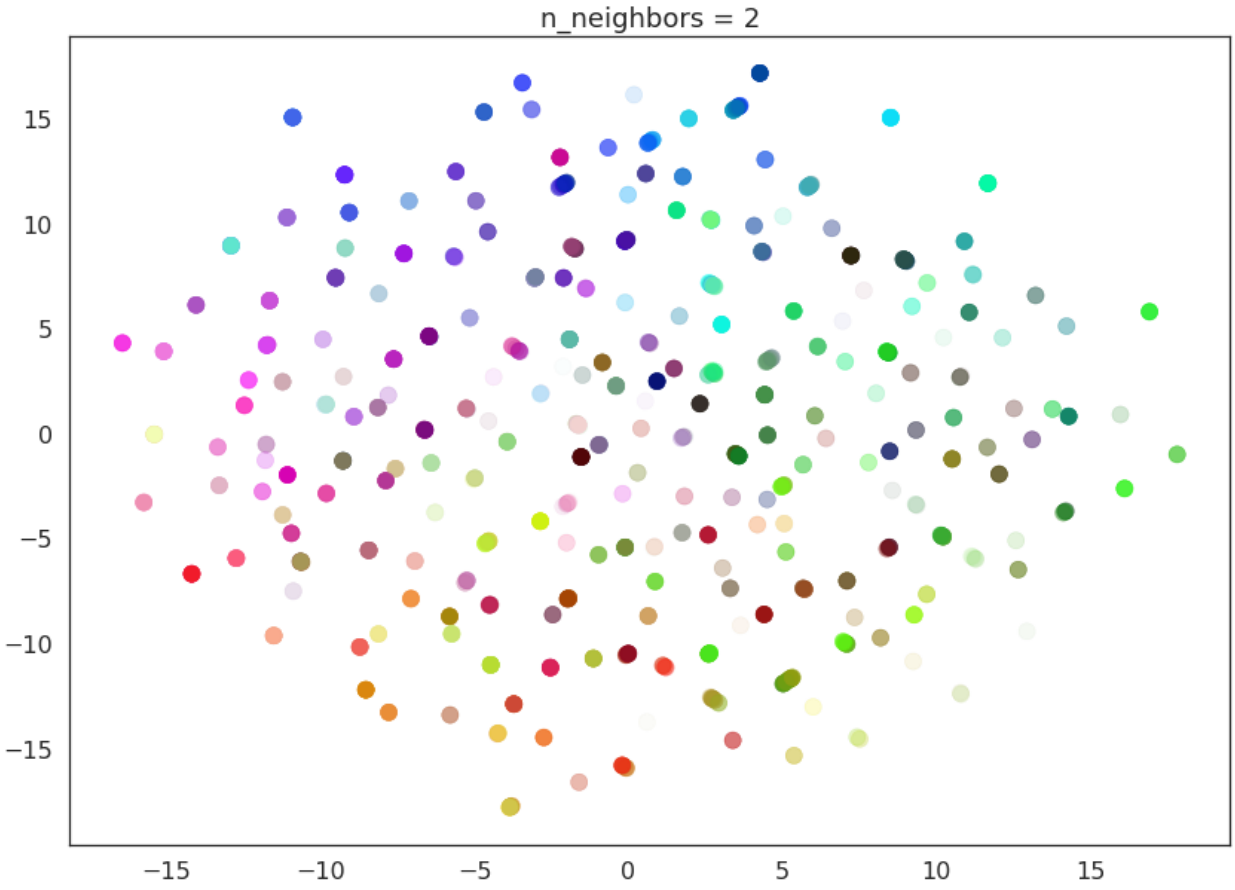
```
def draw_umap(n_neighbors=15, min_dist=0.1, n_components=2, metric='euclidean', title=
↪''):
    fit = umap.UMAP(
        n_neighbors=n_neighbors,
        min_dist=min_dist,
        n_components=n_components,
        metric=metric
    )
    u = fit.fit_transform(data);
    fig = plt.figure()
    if n_components == 1:
        ax = fig.add_subplot(111)
        ax.scatter(u[:,0], range(len(u)), c=data)
    if n_components == 2:
        ax = fig.add_subplot(111)
        ax.scatter(u[:,0], u[:,1], c=data)
    if n_components == 3:
        ax = fig.add_subplot(111, projection='3d')
        ax.scatter(u[:,0], u[:,1], u[:,2], c=data, s=100)
    plt.title(title, fontsize=18)
```

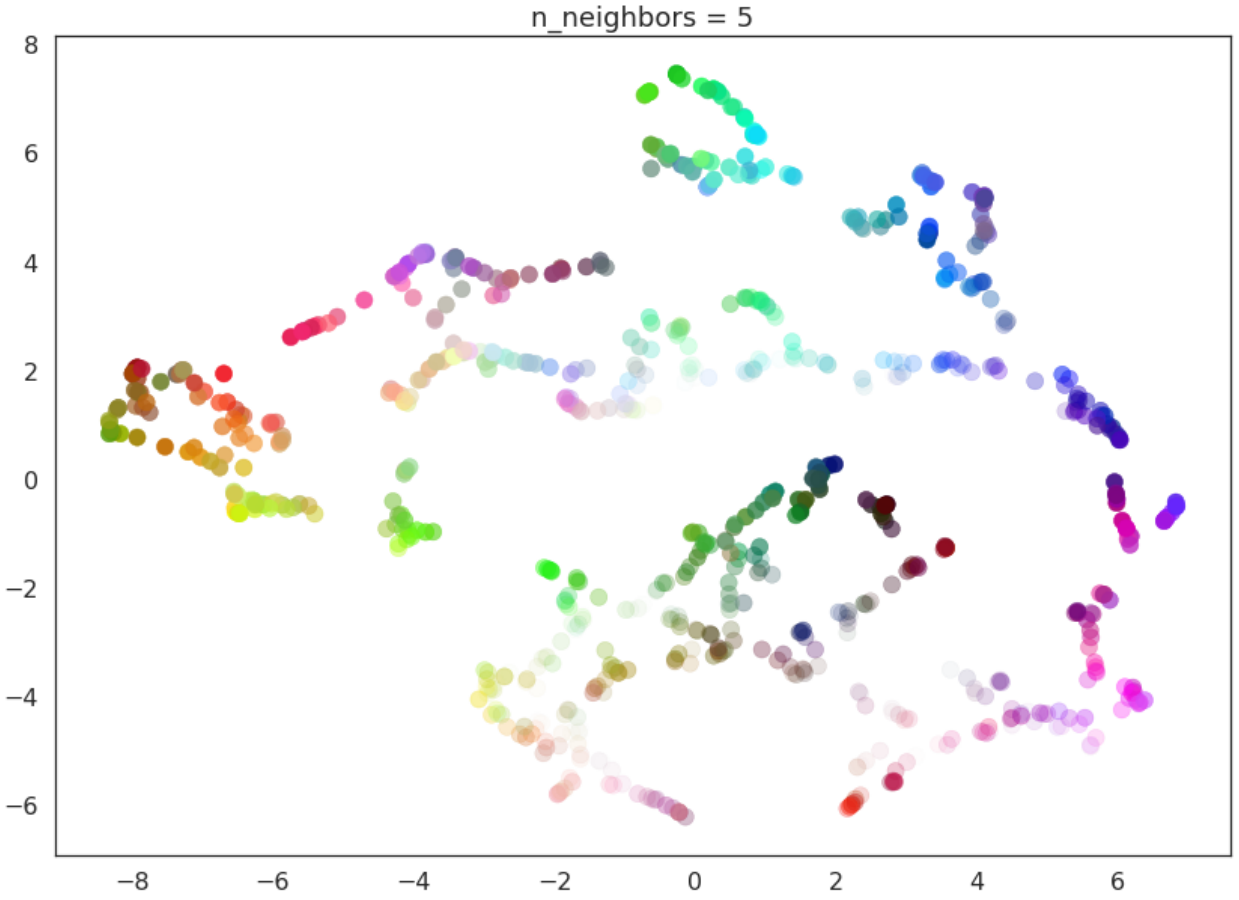
2.1 n_neighbors

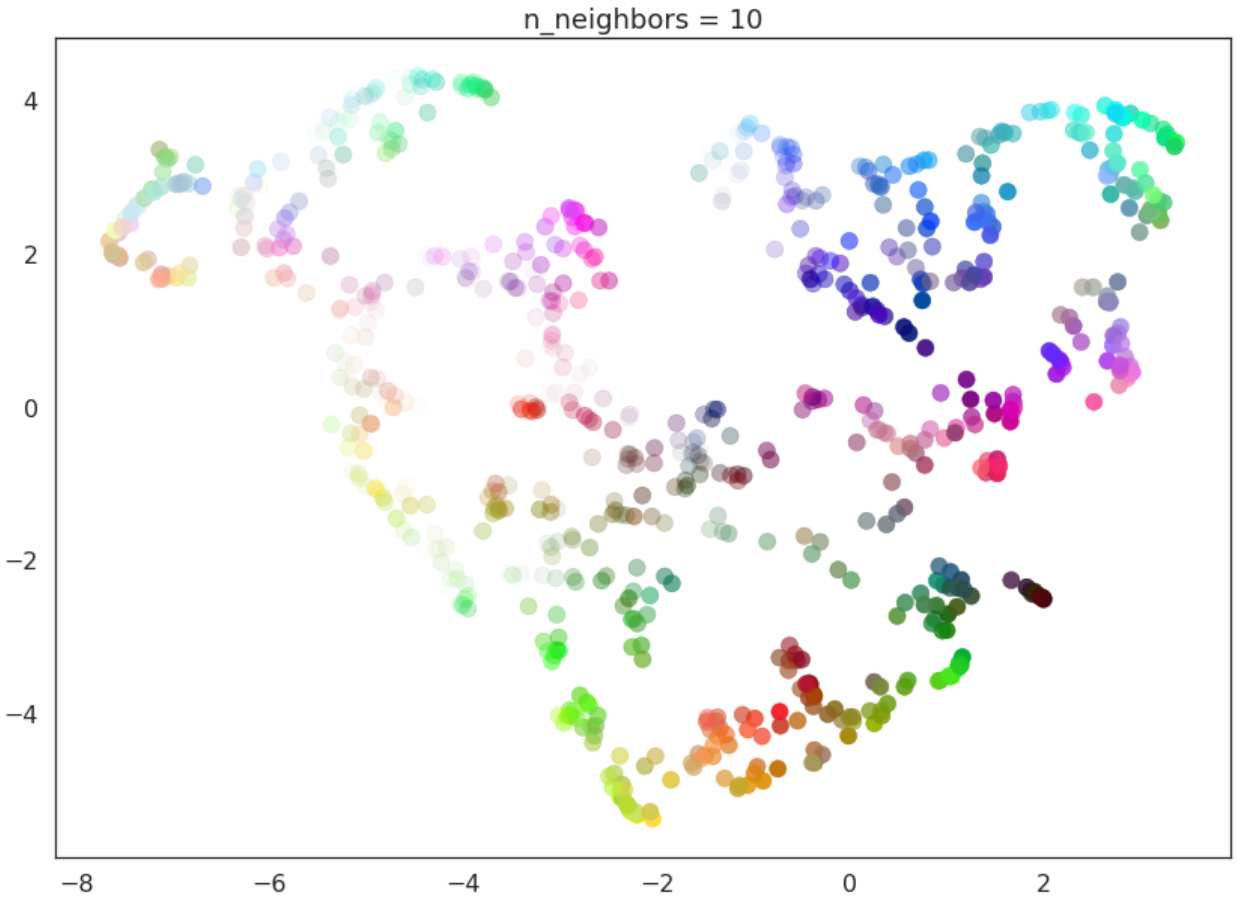
This parameter controls how UMAP balances local versus global structure in the data. It does this by constraining the size of the local neighborhood UMAP will look at when attempting to learn the manifold structure of the data. This means that low values of `n_neighbors` will force UMAP to concentrate on very local structure (potentially to the detriment of the big picture), while large values will push UMAP to look at larger neighborhoods of each point when estimating the manifold structure of the data, losing fine detail structure for the sake of getting the broader of the data.

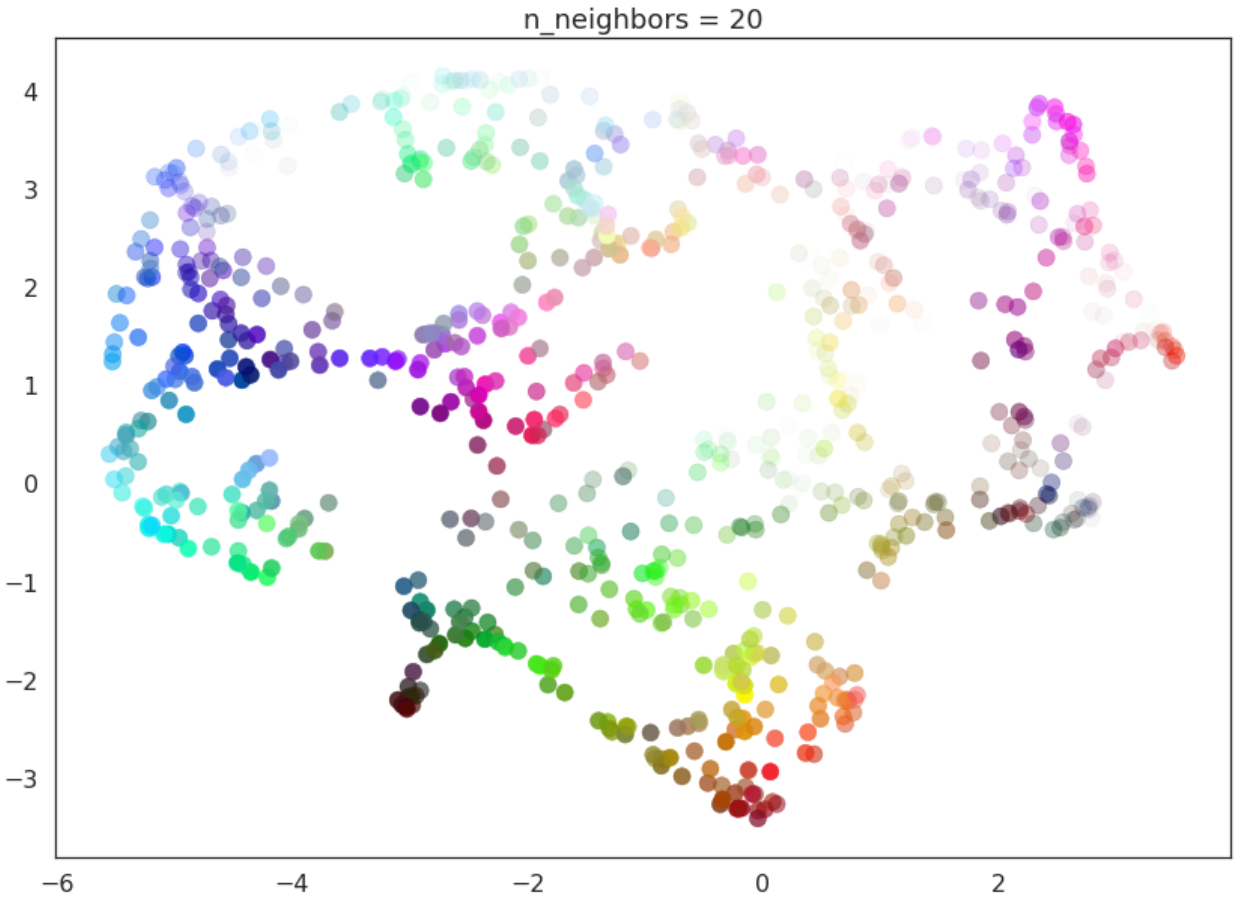
We can see that in practice by fitting our dataset with UMAP using a range of `n_neighbors` values. The default value of `n_neighbors` for UMAP (as used above) is 15, but we will look at values ranging from 2 (a very local view of the manifold) up to 200 (a quarter of the data).

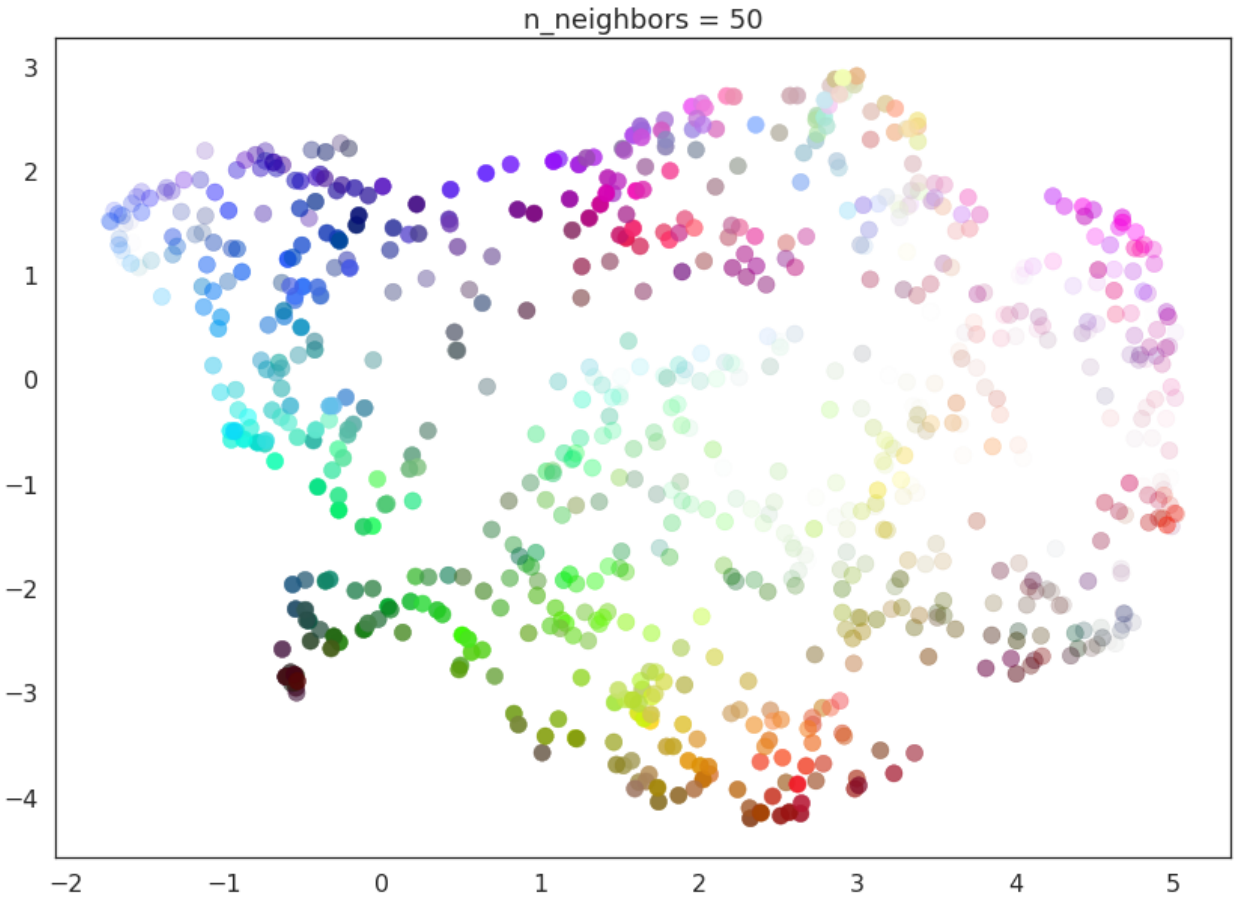
```
for n in (2, 5, 10, 20, 50, 100, 200):
    draw_umap(n_neighbors=n, title='n_neighbors = {}'.format(n))
```

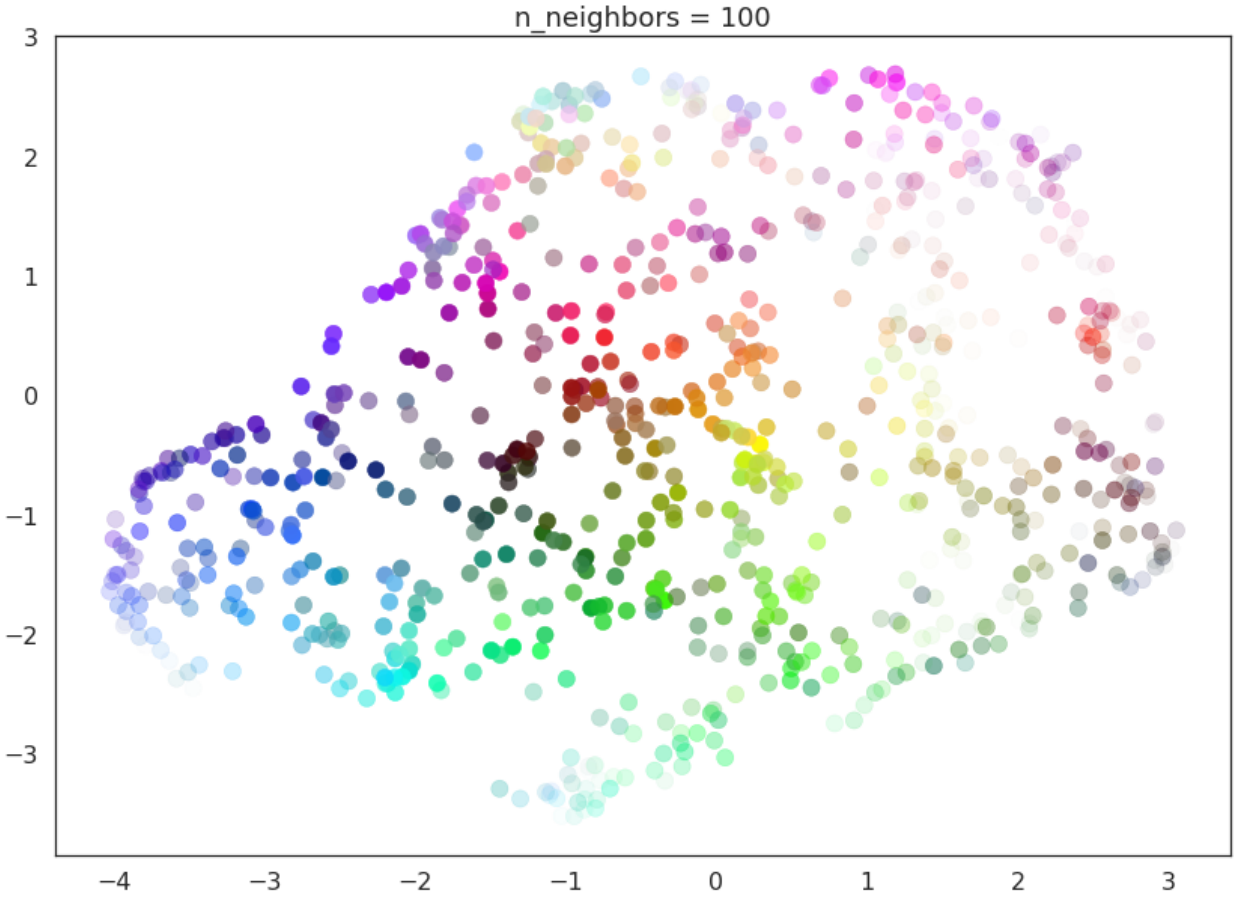

n_neighbors = 2
15
10
5
0
−5
−10
−15
−15
−10
−5
0
5
10
15

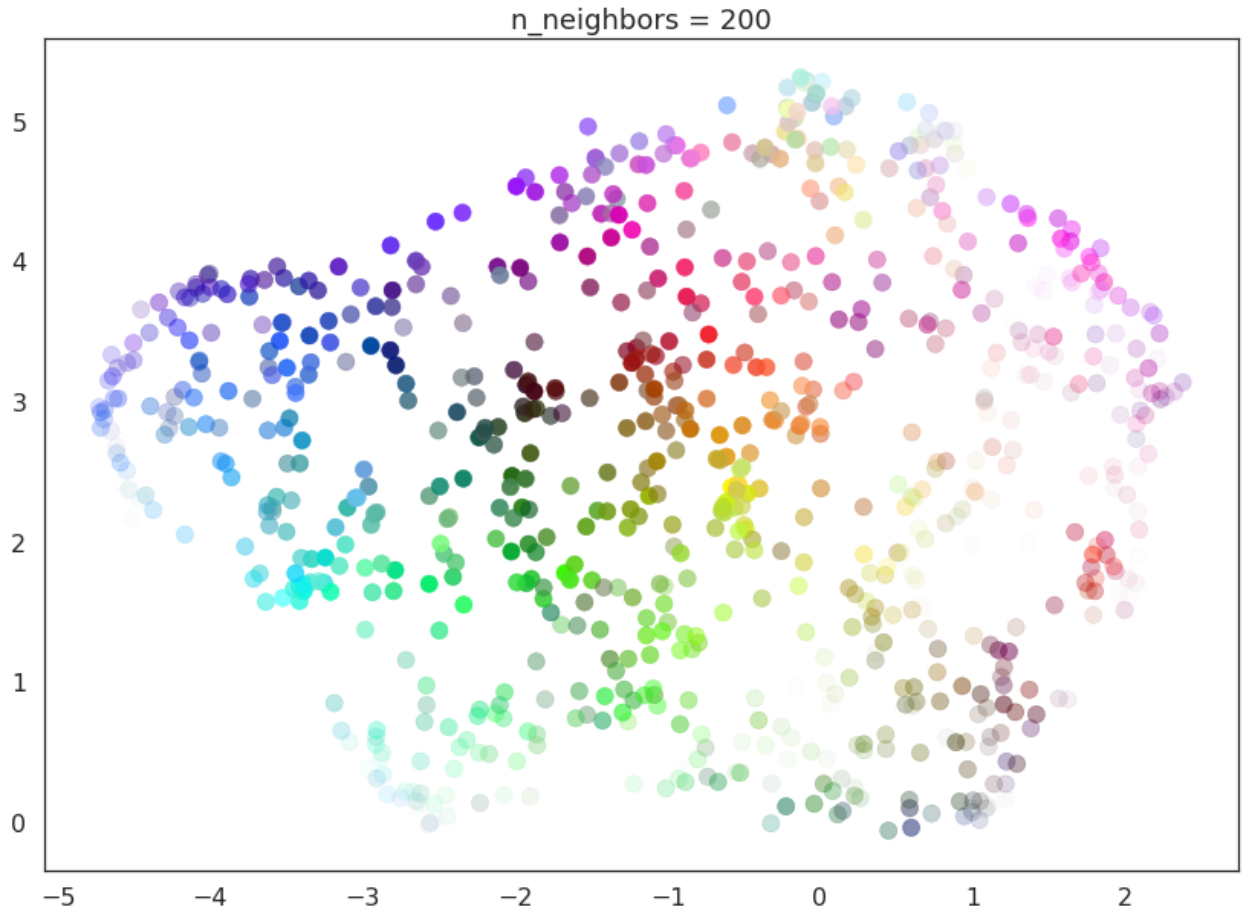

n_neighbors = 5
8
6
4
2
0
−2
−4
−6
−8
−6
−4
−2
0
2
4
6


n_neighbors = 10
4
2
0
−2
−4
−8
−6
−4
−2
0
2


n_neighbors = 20
4
3
2
1
0
−1
−2
−3
−6
−4
−2
0
2


n_neighbors = 50
3
2
1
0
−1
−2
−3
−4
−2
−1
0
1
2
3
4
5


n_neighbors = 100
3
2
1
0
−1
−2
−3
−4
−3
−2
−1
0
1
2
3



With a value of `n_neighbors=2` we see that UMAP merely glues together small chains, but due to the narrow/local view, fails to see how those connect together. It also leaves many different components (and even singleton points). This represents the fact that from a fine detail point of view the data is very disconnected and scattered throughout the space.

As `n_neighbors` is increased UMAP manages to see more of the overall structure of the data, gluing more components together, and better coverying the broader structure of the data. By the stage of `n_neighbors=20` we have a fairly good overall view of the data showing how the various colors interelate to each other over the whole dataset.

As `n_neighbors` increases further more and more focus in placed on the overall structure of the data. This results in, with `n_neighbors=200` a plot where the overall structure (blues, greens, and reds; high luminance versus low) is well captured, but at the loss of some of the finer local structure (individual colors are no longer necessarily immediately near their closest color match).

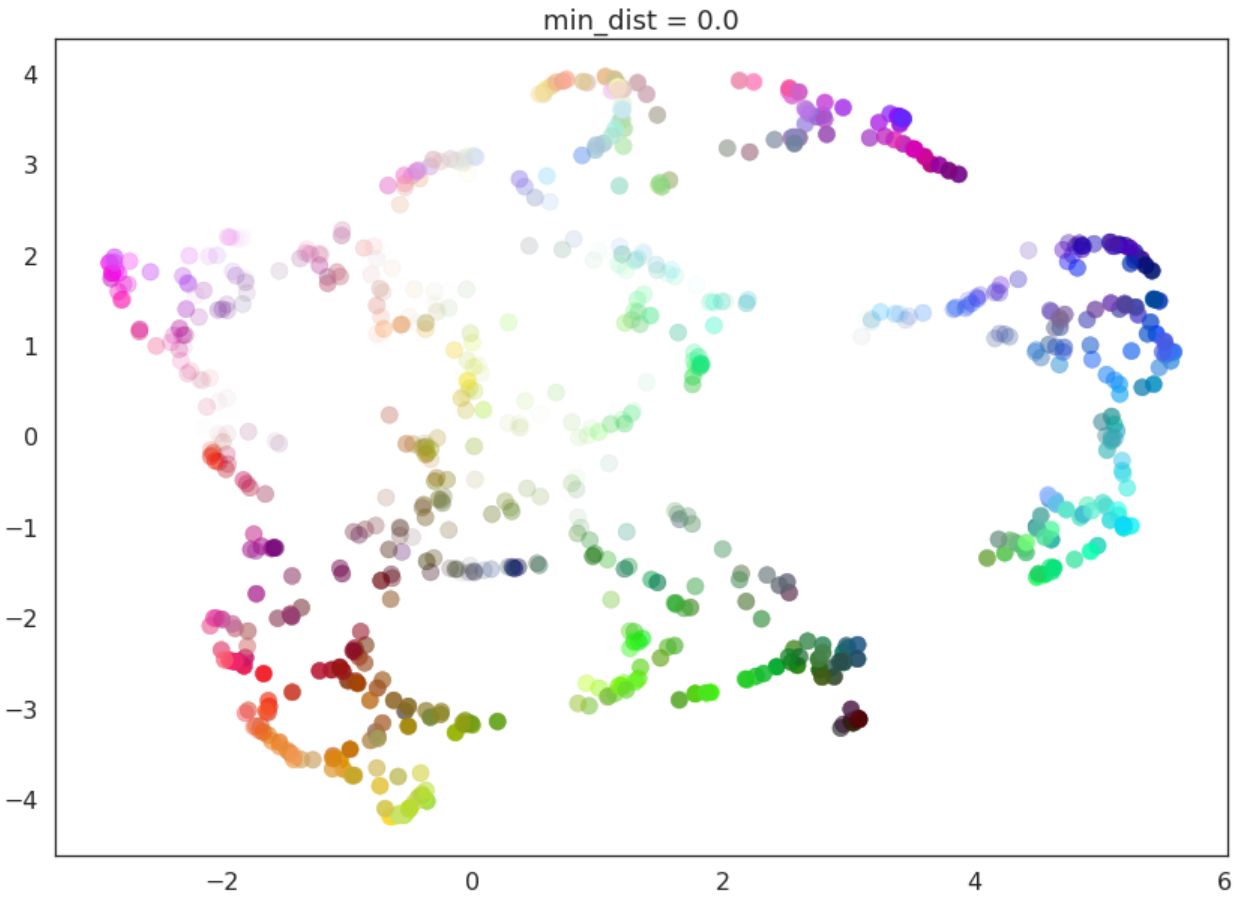
This effect well exemplifies the local/global tradeoff provided by `n_neighbors`.

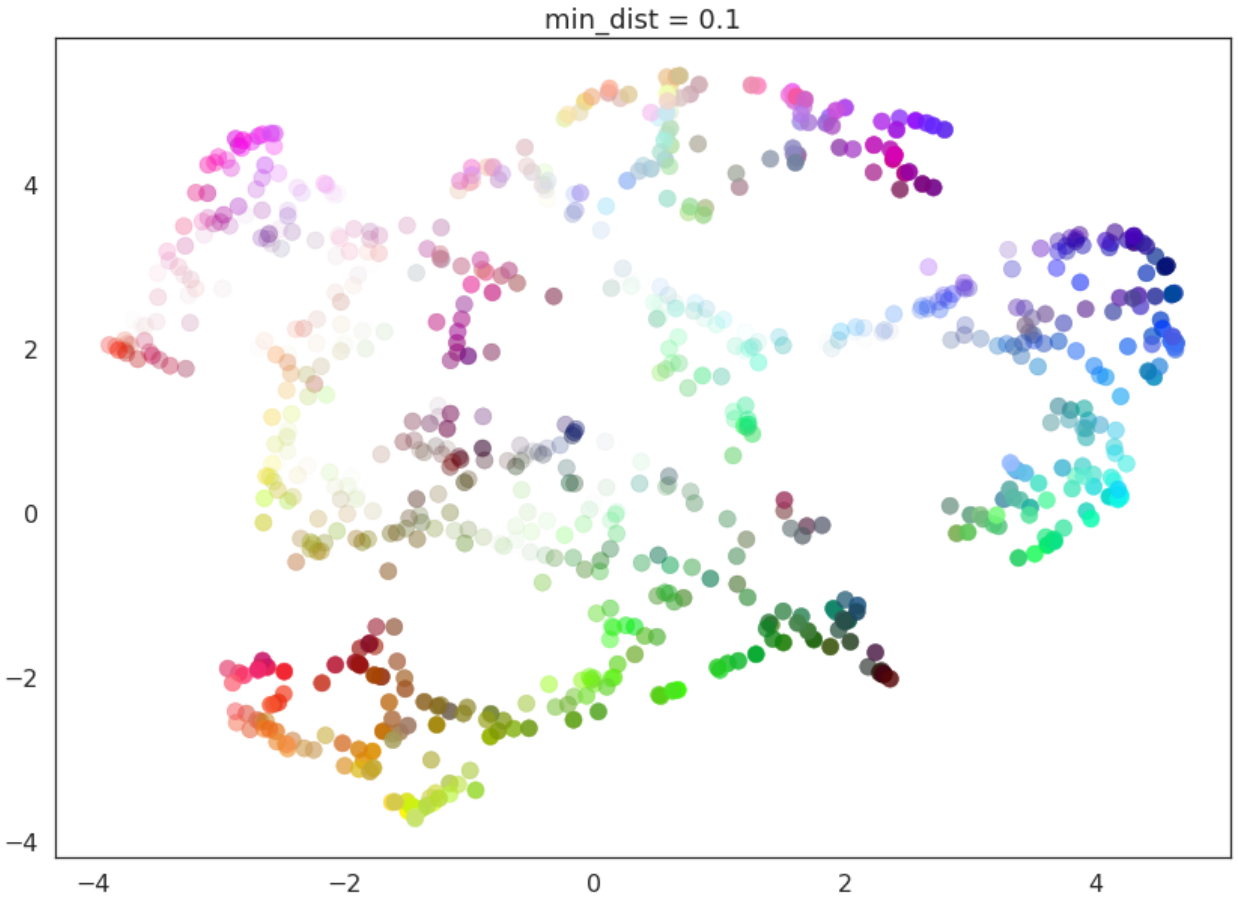
2.2 `min_dist`

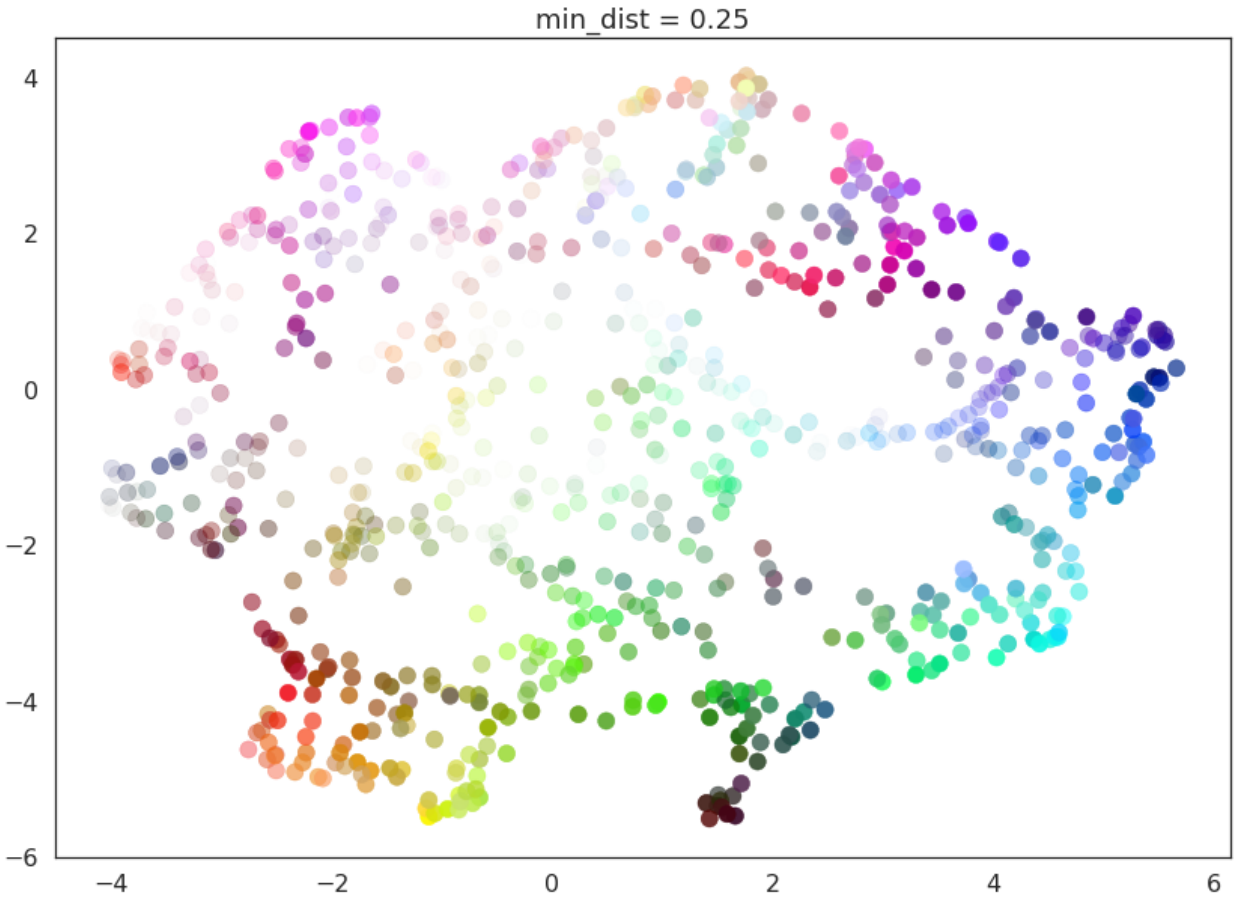
The `min_dist` parameter controls how tightly UMAP is allowed to pack points together. It, quite literally, provides the minimum distance apart that points are allowed to be in the low dimensional representation. This means that low values of `min_dist` will result in clumpier embeddings. This can be useful if you are interested in clustering, or in finer topological structure. Larger values of `min_dist` will prevent UMAP from packing points together and will focus on the preservation of the broad topological structure instead.

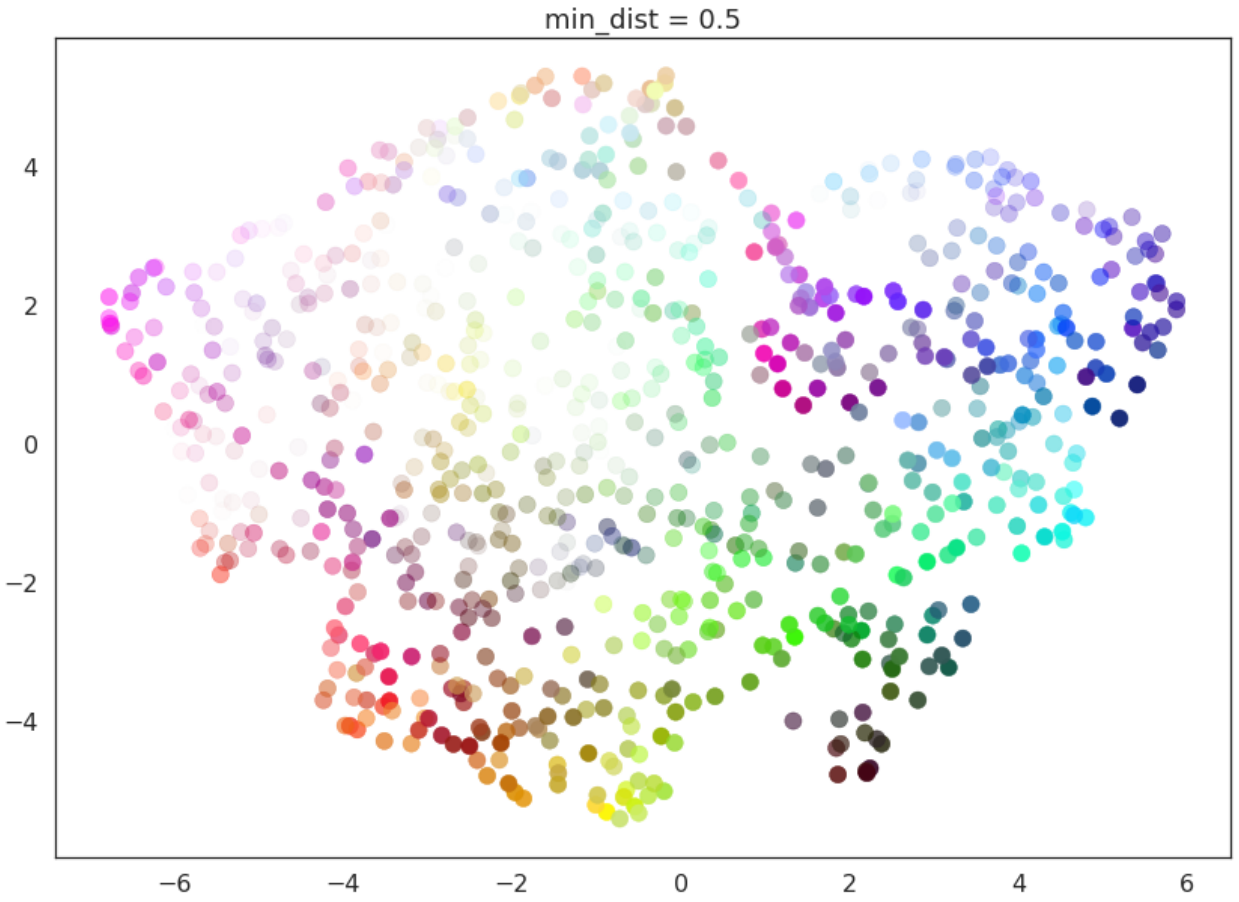
The default value for `min_dist` (as used above) is 0.1. We will look at a range of values from 0.0 through to 0.99.

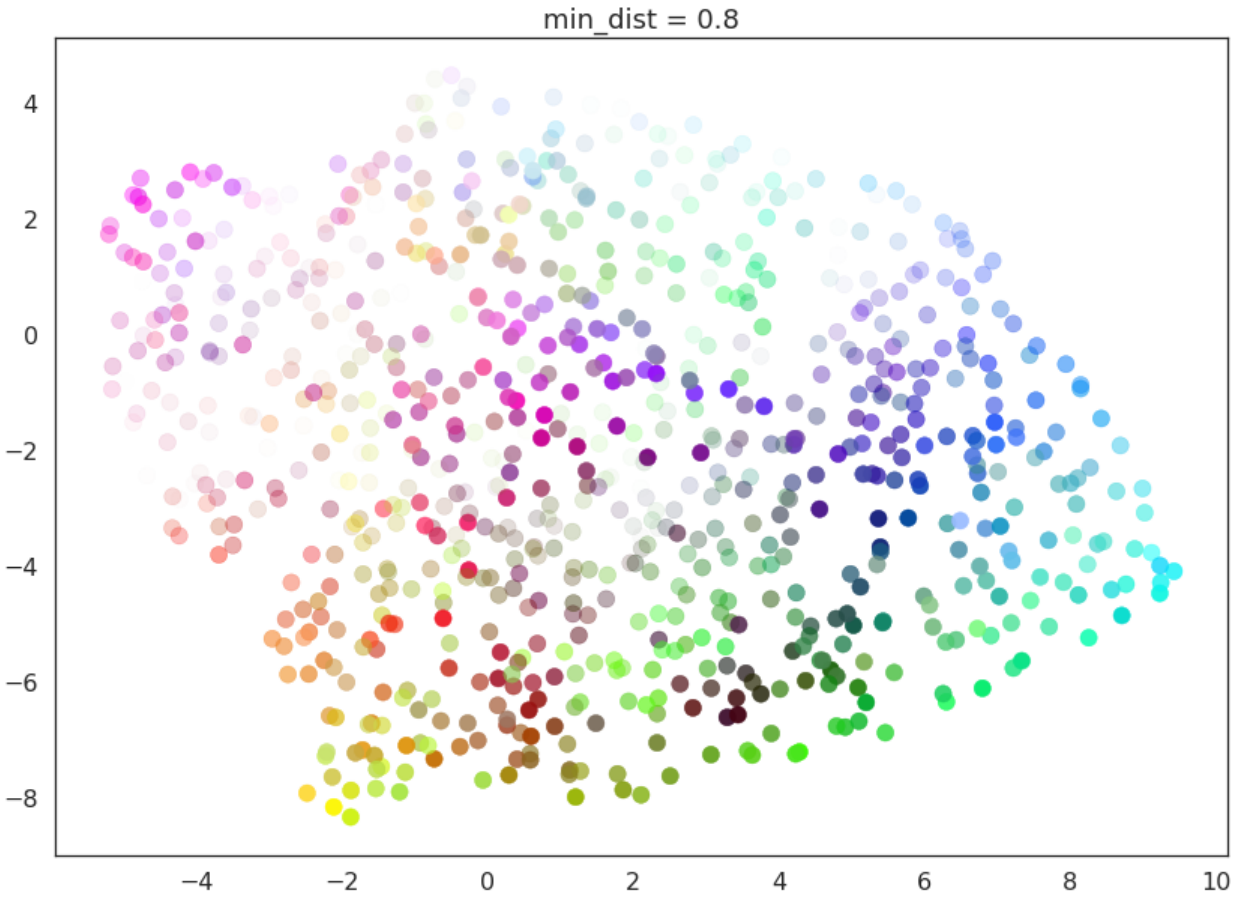
```
for d in (0.0, 0.1, 0.25, 0.5, 0.8, 0.99):
    draw_umap(min_dist=d, title='min_dist = {}'.format(d))
```

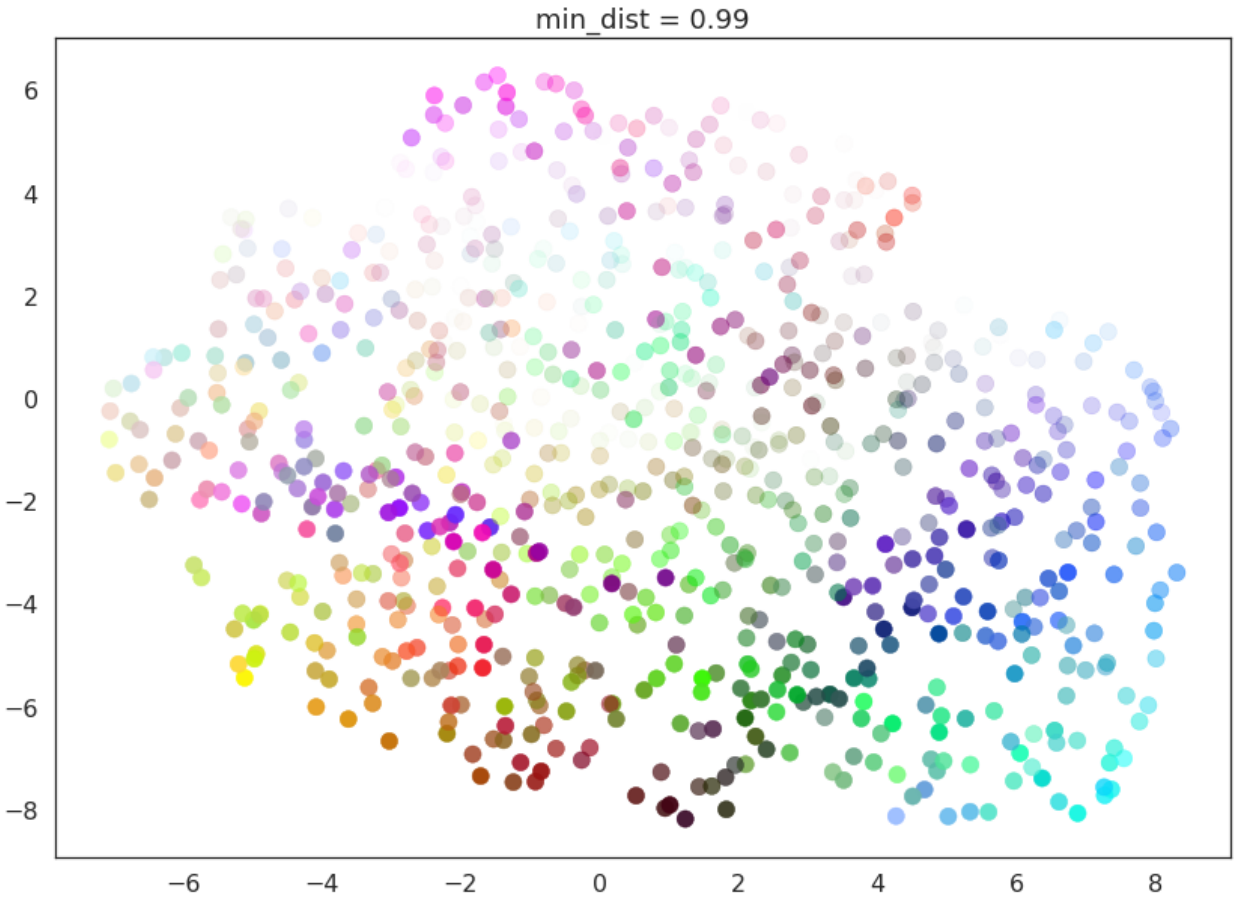



min_dist = 0.1
4
2
0
−2
−4
−4
−2
0
2
4


min_dist = 0.25
4
2
0
−2
−4
−6
−4
−2
0
2
4
6


min_dist = 0.5
4
2
0
−2
−4
−6
−4
−2
0
2
4
6


min_dist = 0.8
4
2
0
−2
−4
−6
−8
−4
−2
0
2
4
6
8
10



Here we see that with `min_dist=0.0` UMAP manages to find small connected components, clumps and strings in the data, and emphasises these features in the resulting embedding. As `min_dist` is increased these structures are pushed apart into softer more general features, providing a better overarching view of the data at the loss of the more detailed topological structure.

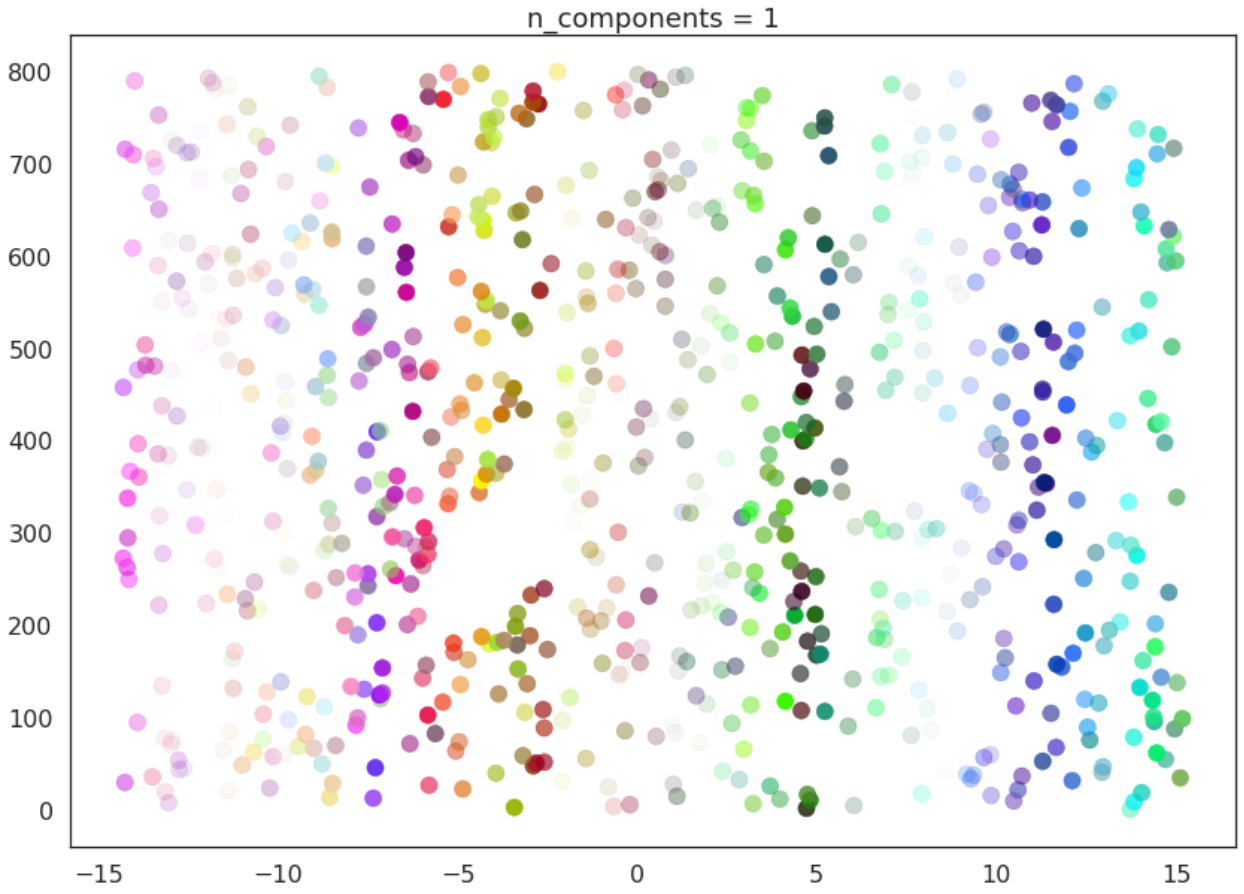
2.3 n_components

As is standard for many `scikit-learn` dimension reduction algorithms UMAP provides a `n_components` parameter option that allows the user to determine the dimensionality of the reduced dimension space we will be embedding the data into. Unlike some other visualisation algorithms such as t-SNE, UMAP scales well in the embedding dimension, so you can use it for more than just visualisation in 2- or 3-dimensions.

For the purposes of this demonstration (so that we can see the effects of the parameter) we will only be looking at 1-dimensional and 3-dimensional embeddings, which we have some hope of visualizing.

First of all we will set `n_components` to 1, forcing UMAP to embed the data in a line. For visualisation purposes we will randomly distribute the data on the y-axis to provide some separation between points.

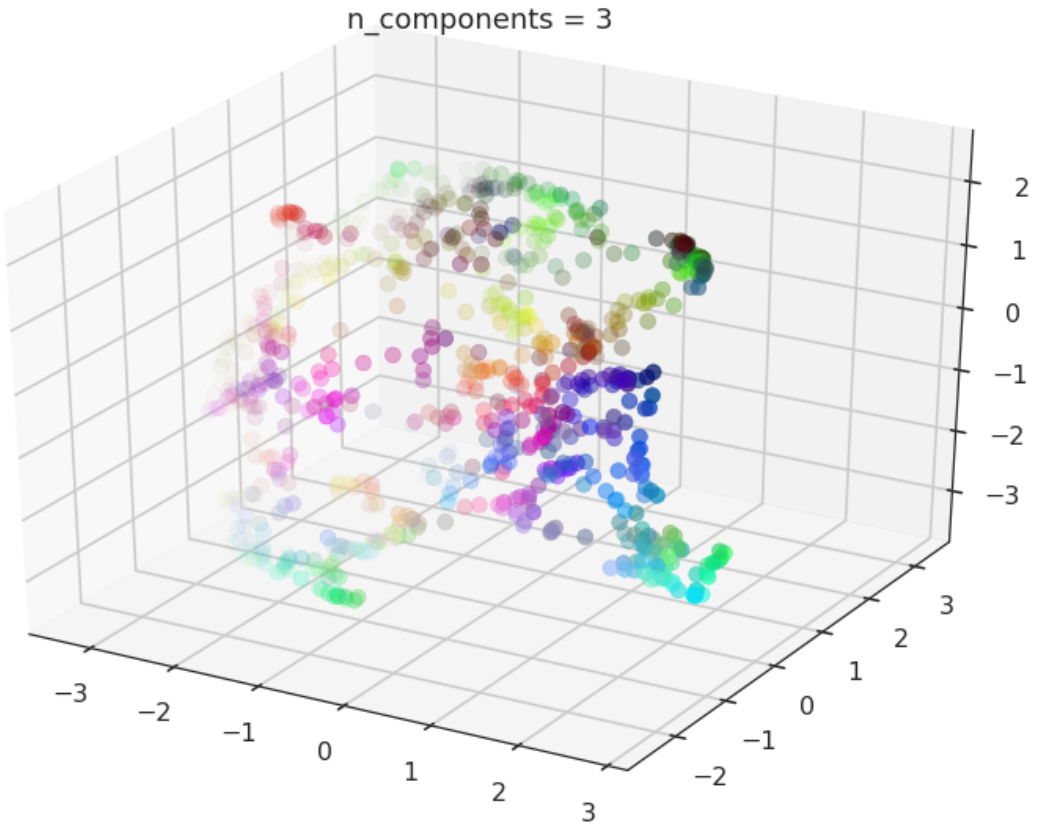
```
draw_umap(n_components=1, title='n_components = 1')
```



Now we will try `n_components=3`. For visualisation we will make use of `matplotlib`'s basic 3-dimensional plotting.

```
draw_umap(n_components=3, title='n_components = 3')
```

```
/opt/anaconda3/envs/umap_dev/lib/python3.6/site-packages/sklearn/metrics/pairwise.
↪py:257: RuntimeWarning: invalid value encountered in sqrt
  return distances if squared else np.sqrt(distances, out=distances)
```



Here we can see that with more dimensions in which to work UMAP has an easier time separating out the colors in a way that respects the topological structure of the data.

As mentioned, there is really no requirement to stop at `n_components=3`. If you are interested in (density based) clustering, or other machine learning techniques, it can be beneficial to pick a larger embedding dimension (say 10, or 50) closer to the the dimension of the underlying manifold on which your data lies.

2.4 `metric`

The final UMAP parameter we will be considering in this notebook is the `metric` parameter. This controls how distance is computed in the ambient space of the input data. By default UMAP supports a wide variety of metrics, including:

Minkowski style metrics

- euclidean
- manhattan
- chebyshev
- minkowski

Miscellaneous spatial metrics

- canberra
- braycurtis

- haversine

Normalized spatial metrics

- mahalanobis
- wminkowski
- seuclidean

Angular and correlation metrics

- cosine
- correlation

Metrics for binary data

- hamming
- jaccard
- dice
- russellrao
- kulsinski
- rogerstanimoto
- sokalmichener
- sokalsneath
- yule

Any of which can be specified by setting `metric='<metric name>'`; for example to use cosine distance as the metric you would use `metric='cosine'`.

UMAP offers more than this however – it supports custom user defined metrics as long as those metrics can be compiled in `nopython` mode by numba. For this notebook we will be looking at such custom metrics. To define such metrics we'll need numba ...

```
import numba
```

For our first custom metric we'll define the distance to be the absolute value of difference in the red channel.

```
@numba.njit()
def red_channel_dist(a,b):
    return np.abs(a[0] - b[0])
```

To get more adventurous it will be useful to have some colorspace conversion – to keep things simple we'll just use HSL formulas to extract the hue, saturation, and lightness from an (R,G,B) tuple.

```
@numba.njit()
def hue(r, g, b):
    cmax = max(r, g, b)
    cmin = min(r, g, b)
    delta = cmax - cmin
    if cmax == r:
        return ((g - b) / delta) % 6
    elif cmax == g:
        return ((b - r) / delta) + 2
    else:
```

(continues on next page)

(continued from previous page)

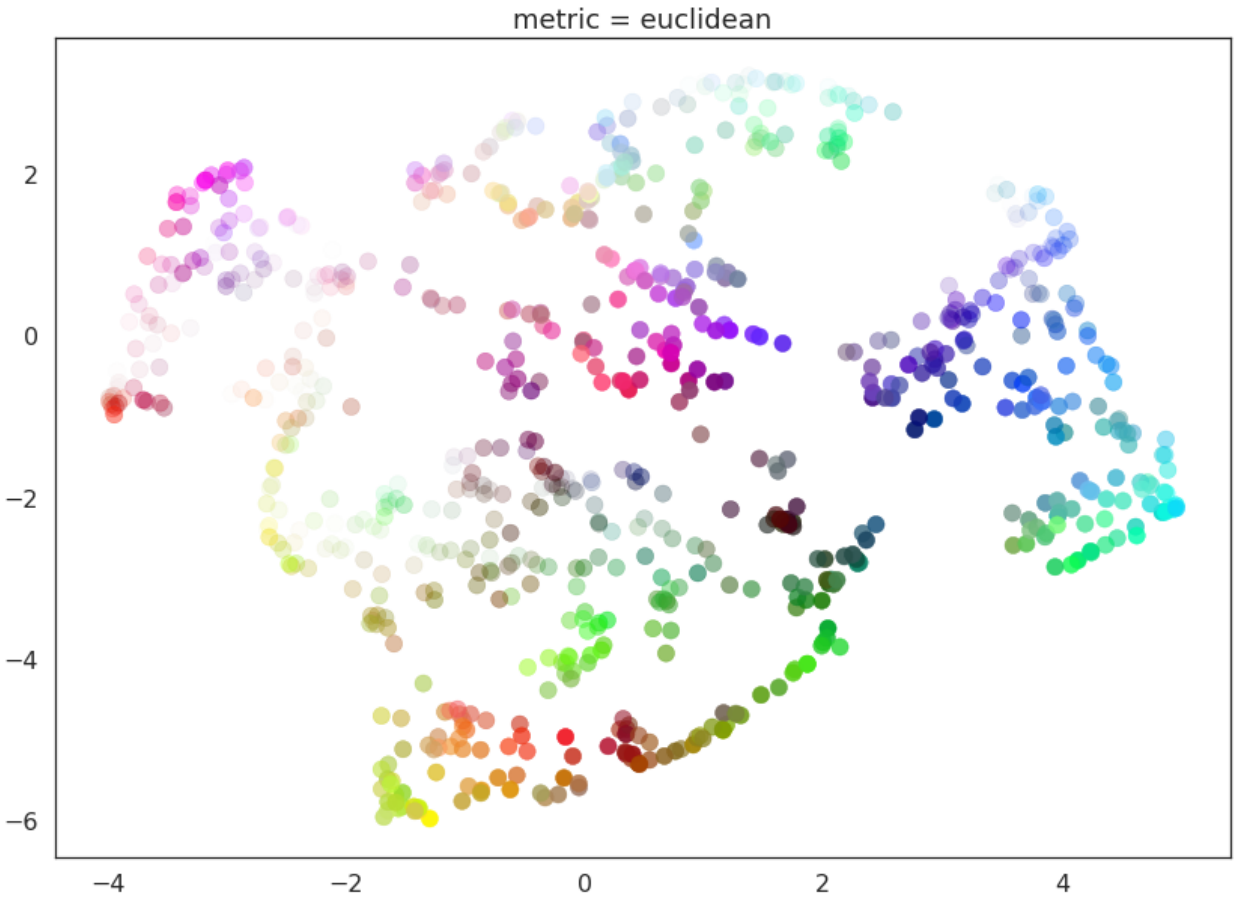
```
        return ((r - g) / delta) + 4

@numba.njit()
def lightness(r, g, b):
    cmax = max(r, g, b)
    cmin = min(r, g, b)
    return (cmax + cmin) / 2.0

@numba.njit()
def saturation(r, g, b):
    cmax = max(r, g, b)
    cmin = min(r, g, b)
    chroma = cmax - cmin
    light = lightness(r, g, b)
    if light == 1:
        return 0
    else:
        return chroma / (1 - abs(2*light - 1))
```

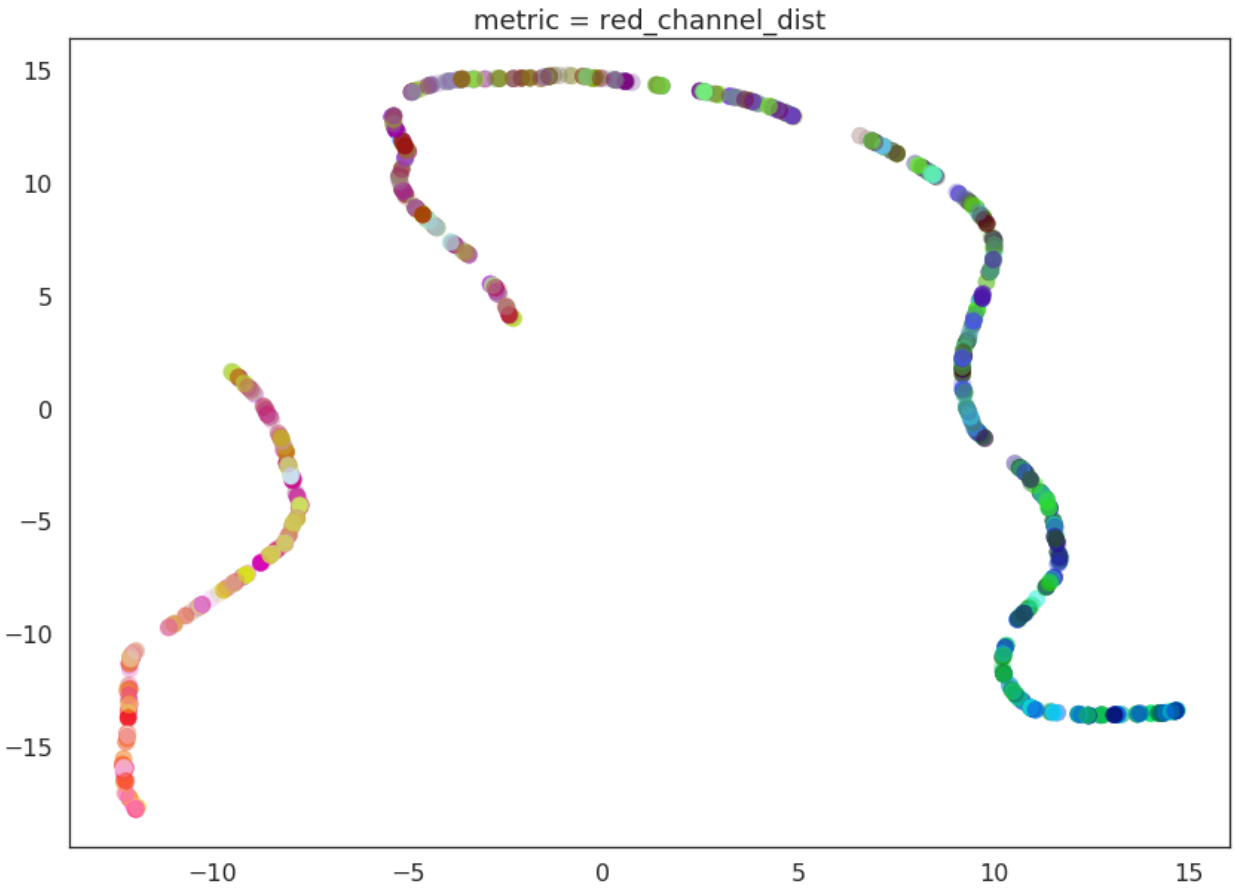
With that in hand we can define three extra distances. The first simply measures the difference in hue, the second measures the euclidean distance in a combined saturation and lightness space, while the third measures distance in the full HSL space.

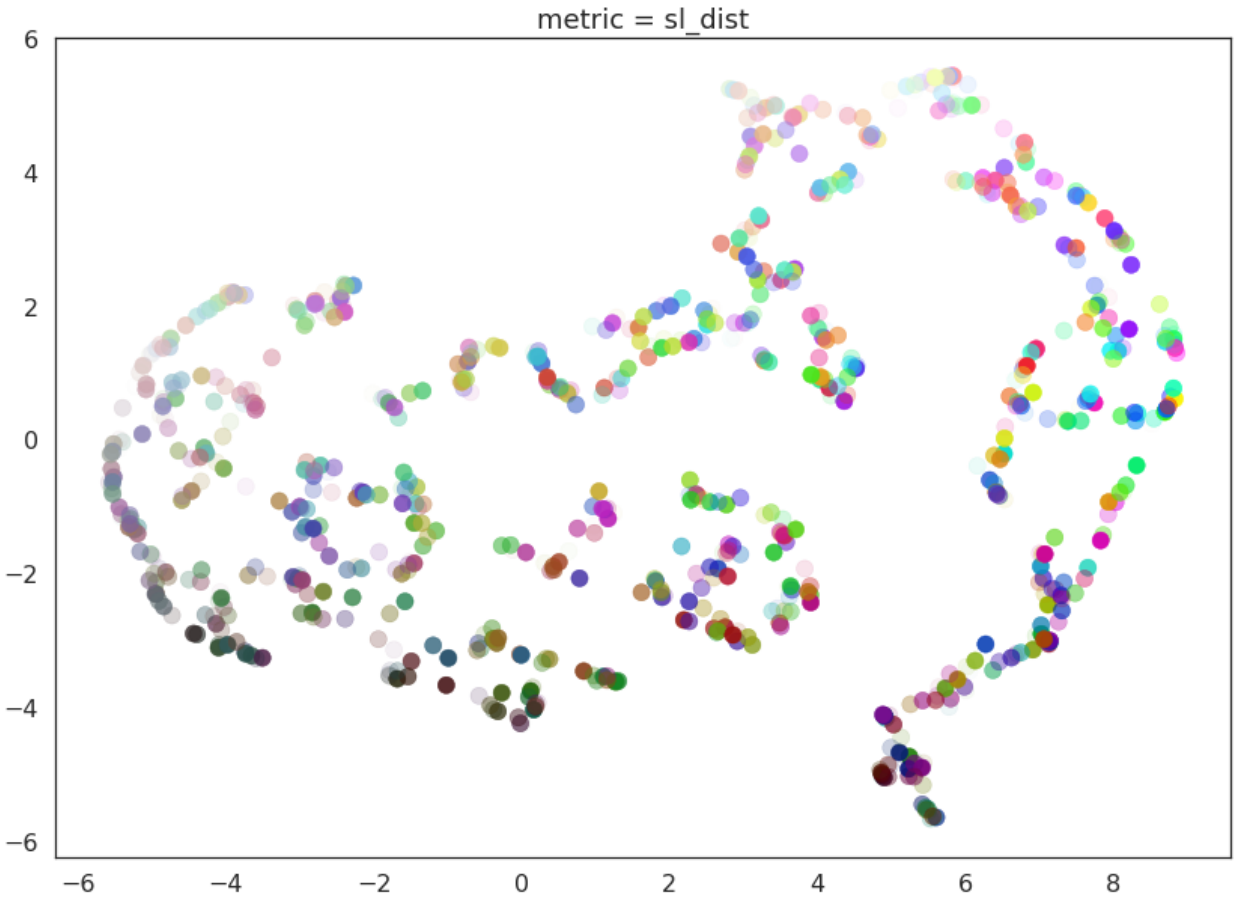
```
@numba.njit()
def hue_dist(a, b):
    diff = (hue(a[0], a[1], a[2]) - hue(b[0], b[1], b[2])) % 6
    if diff < 0:
        return diff + 6
    else:
        return diff

@numba.njit()
def sl_dist(a, b):
    a_sat = saturation(a[0], a[1], a[2])
    b_sat = saturation(b[0], b[1], b[2])
    a_light = lightness(a[0], a[1], a[2])
    b_light = lightness(b[0], b[1], b[2])
    return (a_sat - b_sat)**2 + (a_light - b_light)**2

@numba.njit()
def hsl_dist(a, b):
    a_sat = saturation(a[0], a[1], a[2])
    b_sat = saturation(b[0], b[1], b[2])
    a_light = lightness(a[0], a[1], a[2])
    b_light = lightness(b[0], b[1], b[2])
    a_hue = hue(a[0], a[1], a[2])
    b_hue = hue(b[0], b[1], b[2])
    return (a_sat - b_sat)**2 + (a_light - b_light)**2 + (((a_hue - b_hue) % 6) / 6.0)
```

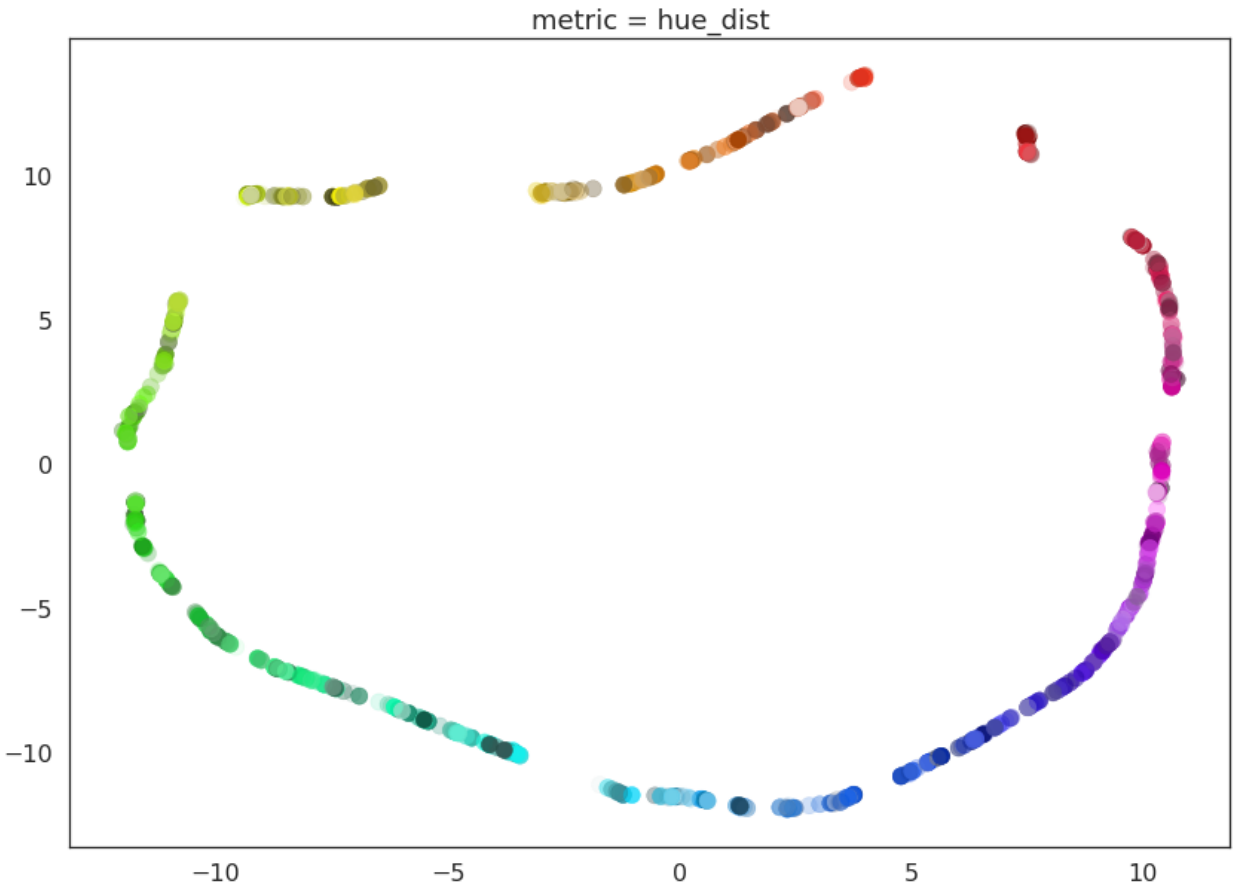
With such custom metrics in hand we can get UMAP to embed the data using those metrics to measure the distance between our input data points. Note that `numba` provides significant flexibility in what we can do in defining distance functions. Despite this we retain the high performance we expect from UMAP even using such custom functions.

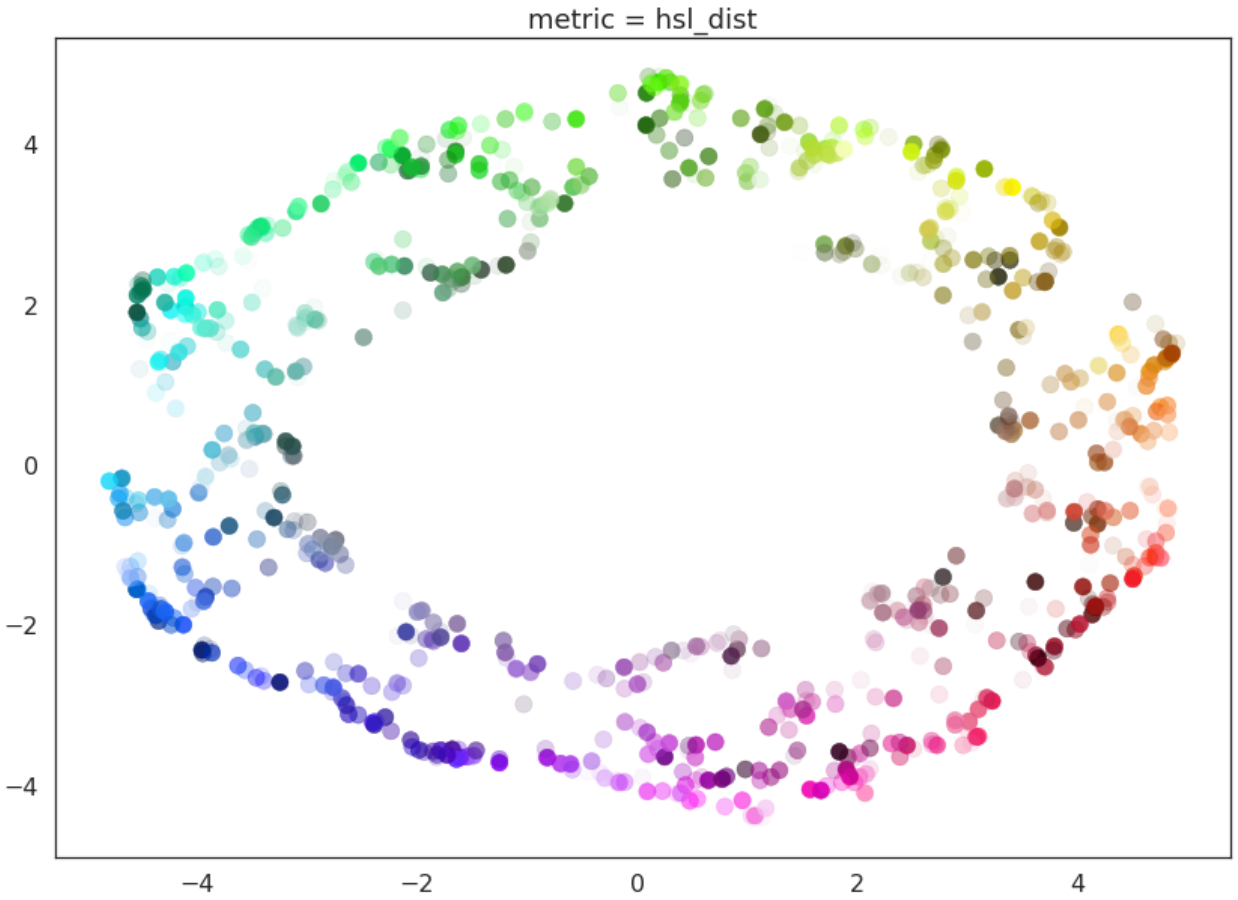
```
for m in ("euclidean", red_channel_dist, sl_dist, hue_dist, hsl_dist):
    name = m if type(m) is str else m.__name__
    draw_umap(n_components=2, metric=m, title='metric = {}'.format(name))
```


metric = euclidean
2
0
−2
−4
−6
−4
−2
0
2
4


metric = red_channel_dist
15
10
5
0
−5
−10
−15
−10
−5
0
5
10
15


metric = sl_dist
6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6
8


metric = hue_dist
10
5
0
−5
−10
−10
−5
0
5
10



And here we can see the effects of the metrics quite clearly. The pure red channel correctly sees the data as living on a one dimensional manifold, the hue metric interprets the data as living in a circle, and the HSL metric fattens out the circle according to the saturation and lightness. This provides a reasonable demonstration of the power and flexibility of UMAP in understanding the underlying topology of data, and finding a suitable low dimensional representation of that topology.

CHAPTER 3

Plotting UMAP results

UMAP is often used for visualization by reducing data to 2-dimensions. Since this is such a common use case the umap package now includes utility routines to make plotting UMAP results simple, and provide a number of ways to view and diagnose the results. Rather than seeking to provide a comprehensive solution that covers all possible plotting needs this umap extension seeks to provide a simple to use interface to make the majority of plotting needs easy, and help provide sensible plotting choices wherever possible. To get started looking at the plotting options let's load a variety of data to work with.

```
import sklearn.datasets
import pandas as pd
import numpy as np
import umap
```

```
pendigits = sklearn.datasets.load_digits()
mnist = sklearn.datasets.fetch_openml('mnist_784')
fmnist = sklearn.datasets.fetch_openml('Fashion-MNIST')
```

To start we will fit a UMAP model to the pendigits data. This is as simple as running the fit method and assigning the result to a variable.

```
mapper = umap.UMAP().fit(pendigits.data)
```

If we want to do plotting we will need the `umap.plot` package. While the umap package has a fairly small set of requirements it is worth noting that if you want to using `umap.plot` you will need a variety of extra libraries that are not in the default requirements for umap. In particular you will need:

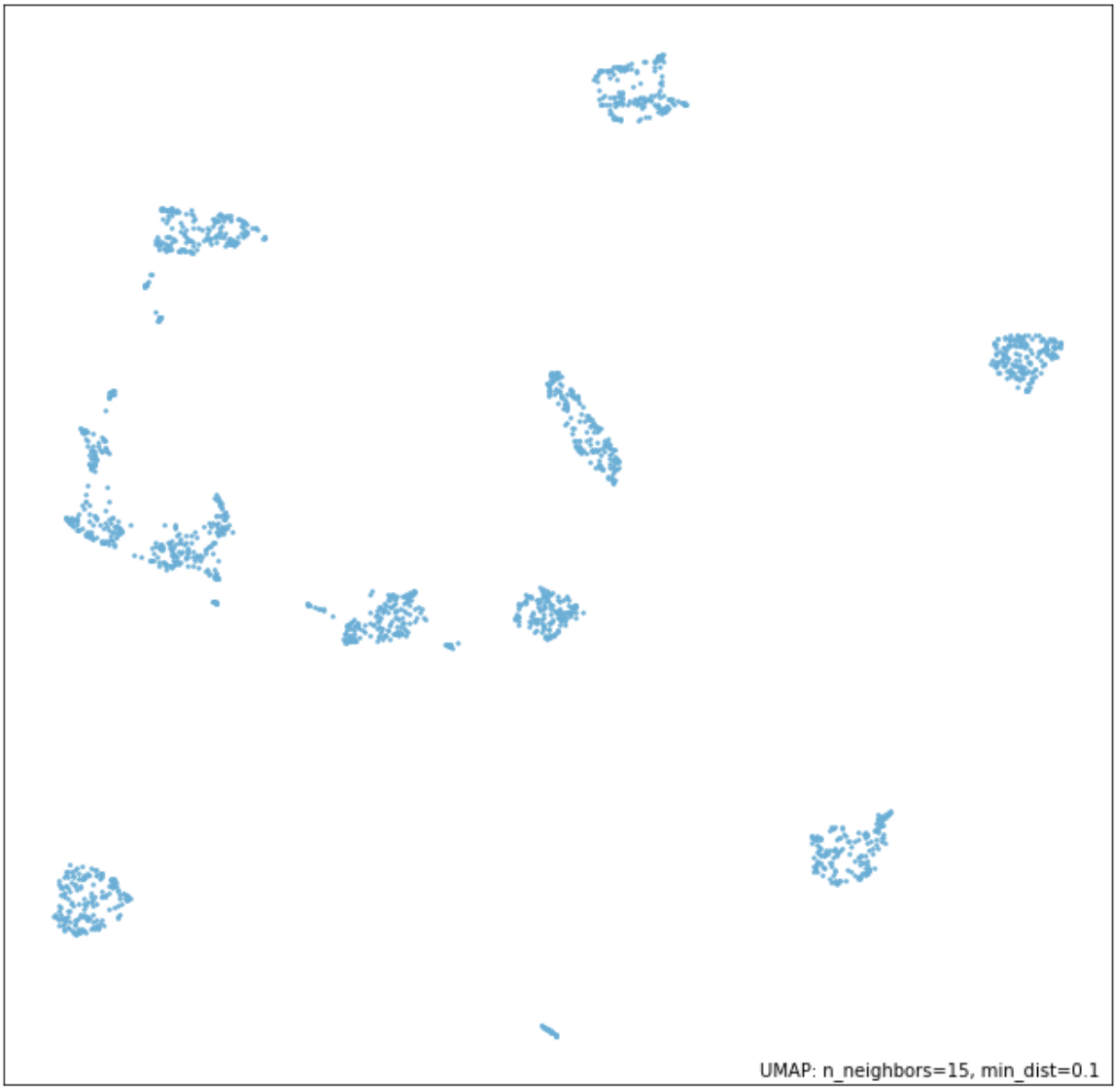
- matplotlib
- pandas
- datashader
- bokeh
- holoviews

All should be either pip or conda installable. With those in hand you can import the `umap.plot` package.

```
import umap.plot
```

Now that we have the package loaded, how do we use it? The most straightforward thing to do is plot the umap results as points. We can achieve this via the function `umap.plot.points`. In its most basic form you can simply pass the trained UMAP model to `umap.plot.points`:

```
umap.plot.points(mapper)
```

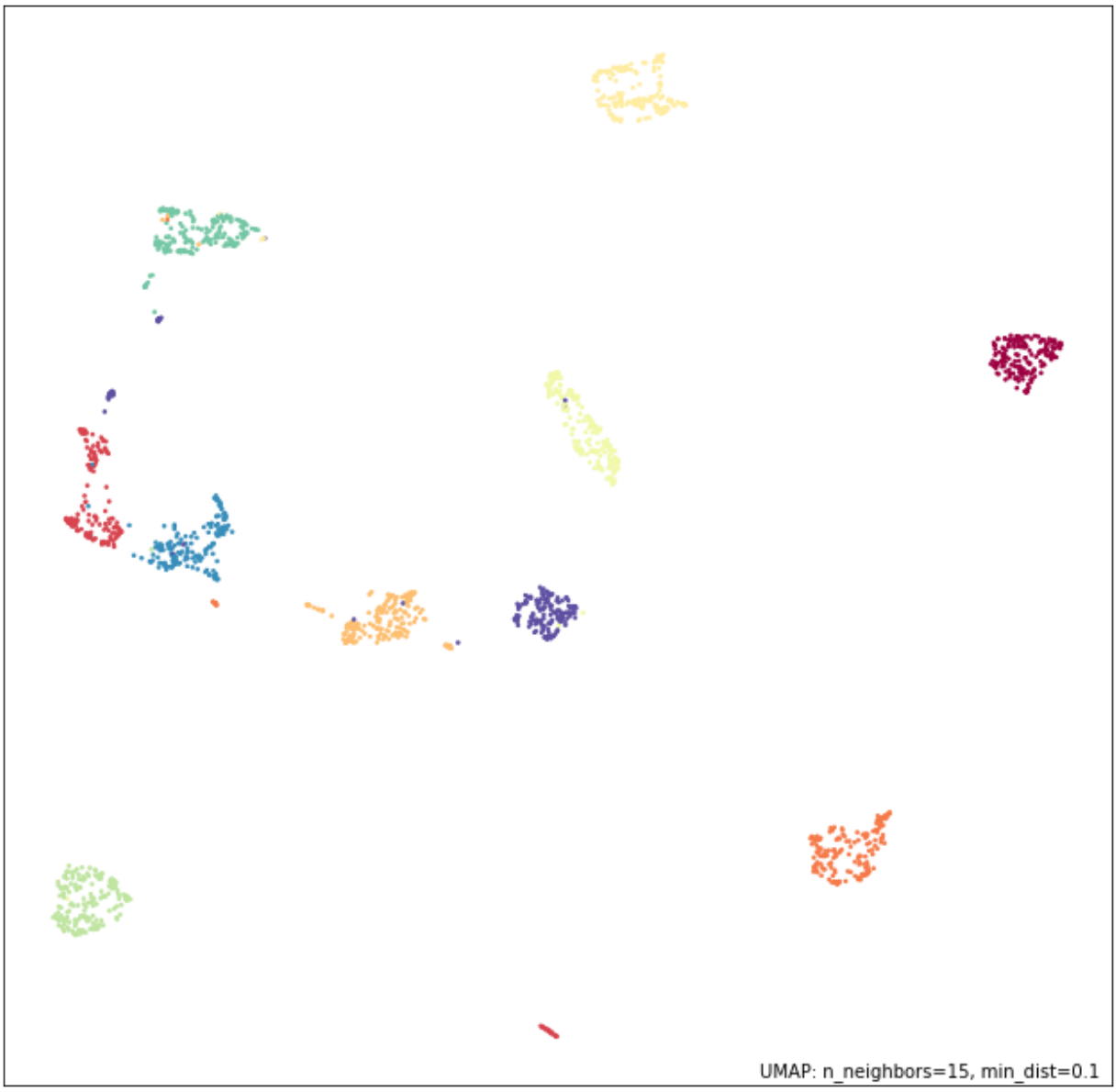


As you can see we immediately get a scatterplot of the UMAP embedding. Note that the function automatically selects a point-size based on the data density, and watermarks the image with the UMAP parameters that were used (this will include the metric if it is non-standard). The function also returns the matplotlib axes object associated to the plot, so further matplotlib functions, such as adding titles, axis labels etc. can be applied by the user if required.

It is common for data passed to UMAP to have an associated set of labels, which may have been derived from ground-truth, from clustering, or via other means. In such cases it is desirable to be able to color the scatterplot according

to the labelling. We can do this by simply passing the array of label information in with the `labels` keyword. The `umap.plot.points` function will color the data with a categorical colormap according to the labels provided.

```
umap.plot.points(mapper, labels=pendigits.target)
```



Alternatively you may have extra data that is continuous rather than categorical. In this case you will want to use a continuous colormap to shade the data. Again this is straightforward to do – pass in the continuous data with the `values` keyword and data will be colored accordingly using a continuous colormap.

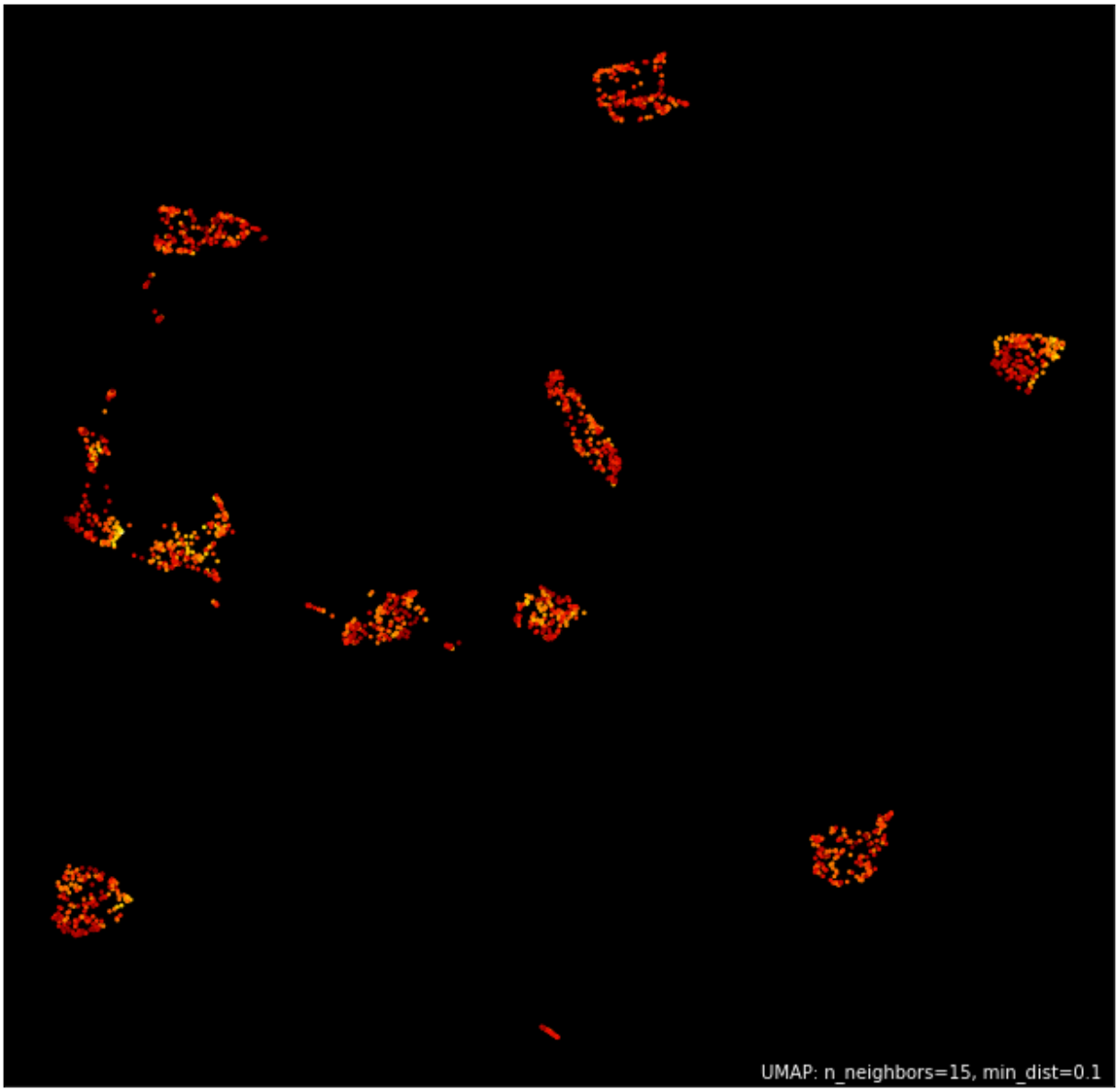
Furthermore, if you don't like the default color choices the `umap.plot.points` function offers a number of 'themes' that provide predefined color choices. Themes include:

- fire
- viridis
- inferno

- blue
- red
- green
- darkblue
- darkred
- darkgreen

Here we will make use of the 'fire' theme to demonstrate how simple it is to change the aesthetics.

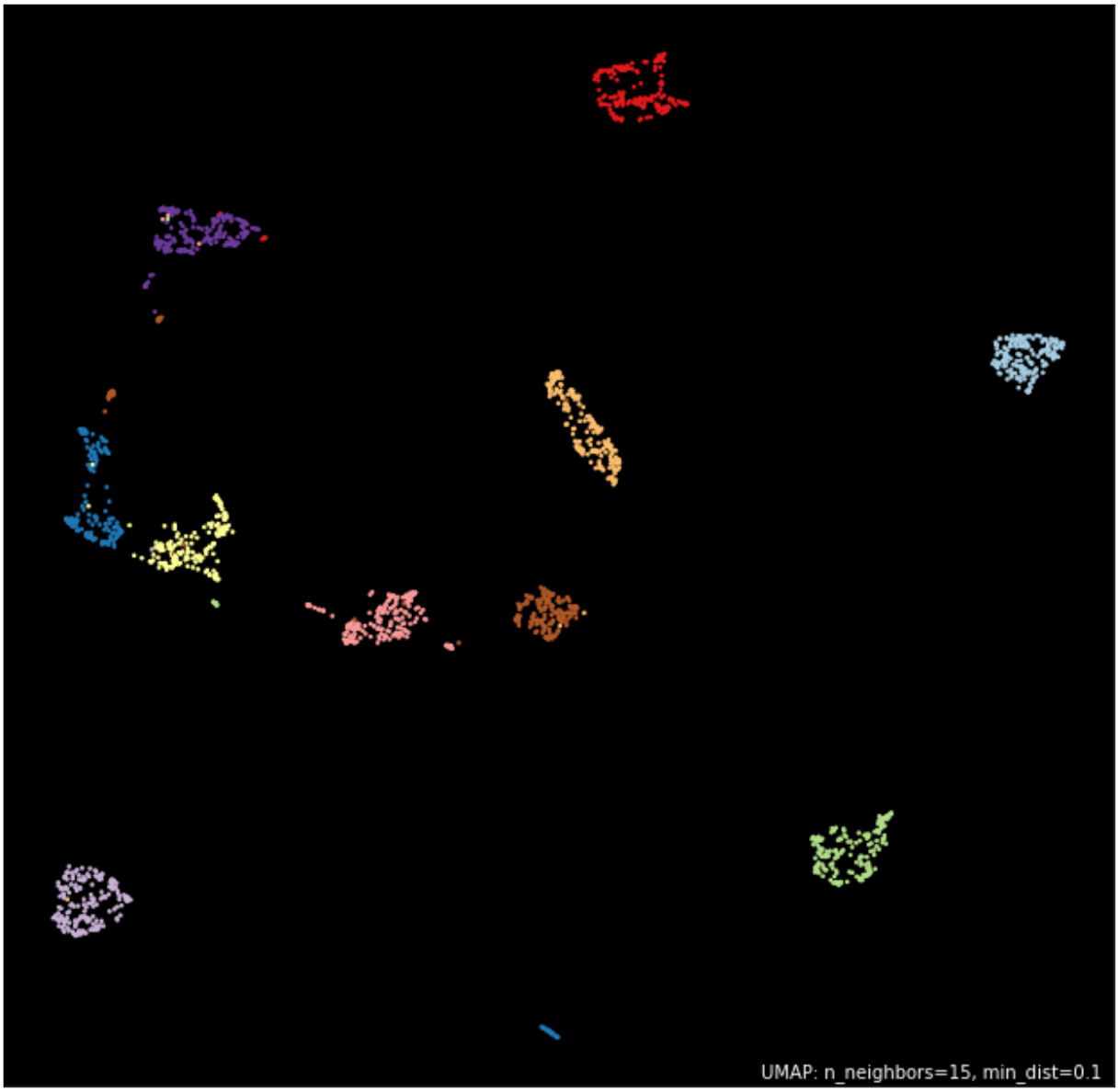
```
umap.plot.points(mapper, values=pendigits.data.mean(axis=1), theme='fire')
```



If you want greater control you can specify exact colormaps and background colors. For example here we want to color the data by label, but use a black background and use the 'Paired' colormap for the categorical coloring (passed

as `color_key_cmap`; the `cmap` keyword defines the continuous colormap).

```
umap.plot.points(mapper, labels=pendigits.target, color_key_cmap='Paired', background=
↪'black')
```



Many more options are available including a `color_key` to specify a dictionary mapping of discrete labels to colors, `cmap` to specify the continous colormap, or the width and height of the resulting plot. Again, this does not provide comprehensive control of the plot aesthetics, but the goal here is to provide a simple to use interface rather than the ability for the user to fine tune all aspects – users seeking such control are far better served making use of the individual underlying packages (matplotlib, datashader, and bokeh) by themselves.

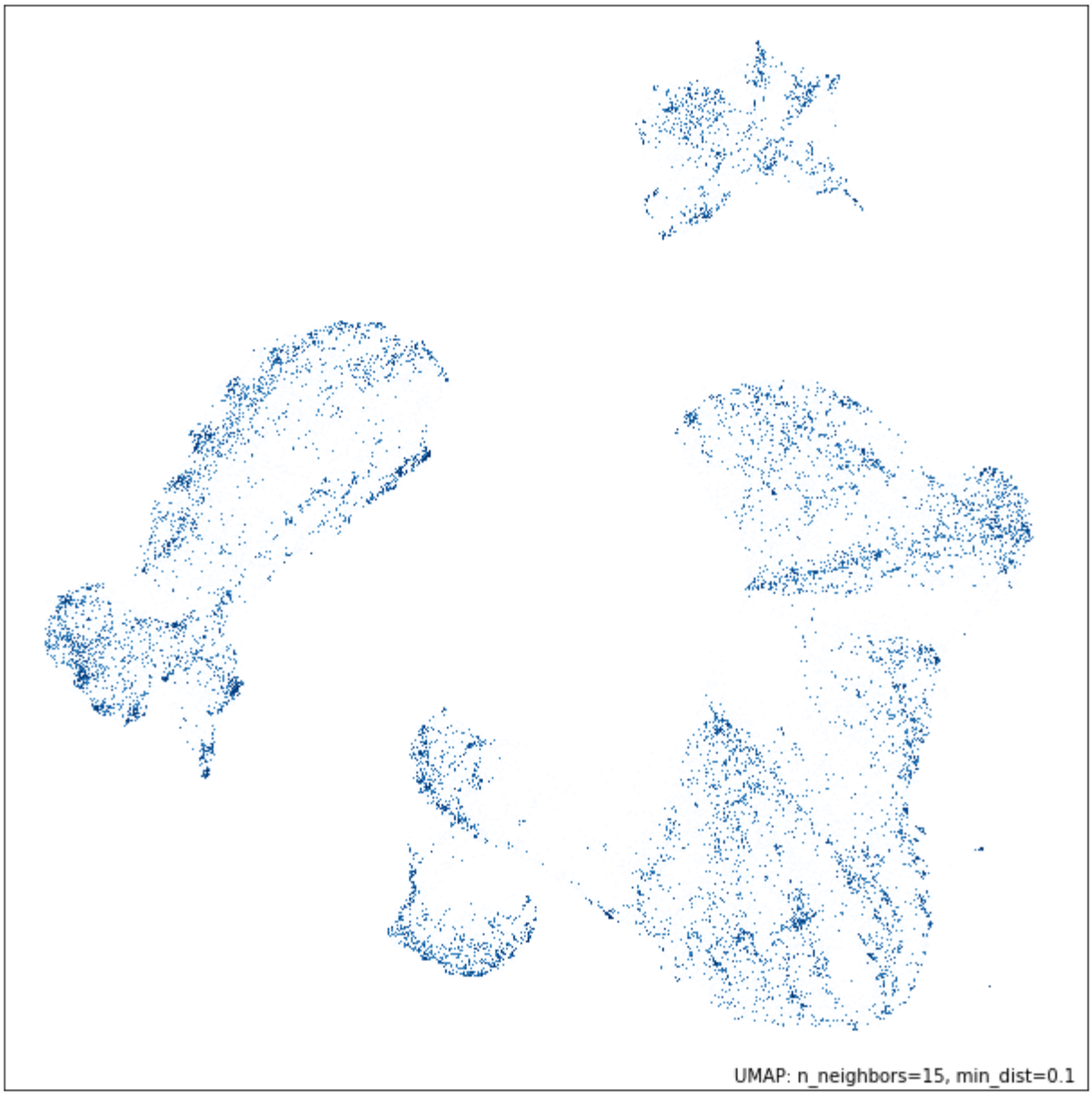
3.1 Plotting larger datasets

Once you have a lot of data it becomes easier for a simple scatter plot to lie to you. Most notably overplotting, where markers for points overlap and pile up on top of each other, can deceive you into thinking that extremely dense clumps may only contain a few points. While there are things that can be done to help remedy this, such as reducing the point size, or adding an alpha channel, few are sufficient to be sure the plot isn't subtly lying to you in some way. This essay in the datashader documentation does an excellent job of describing the issues with overplotting, why the obvious solutions are not quite sufficient, and how to get around the problem. To make life easier for users the `umap.plot` package will automatically switch to using datashader for rendering once your dataset gets large enough. This helps to ensure you don't get fooled by overplotting. We can see this in action by working with one of the larger datasets such as Fashion-MNIST.

```
mapper = umap.UMAP().fit(fmnist.data)
```

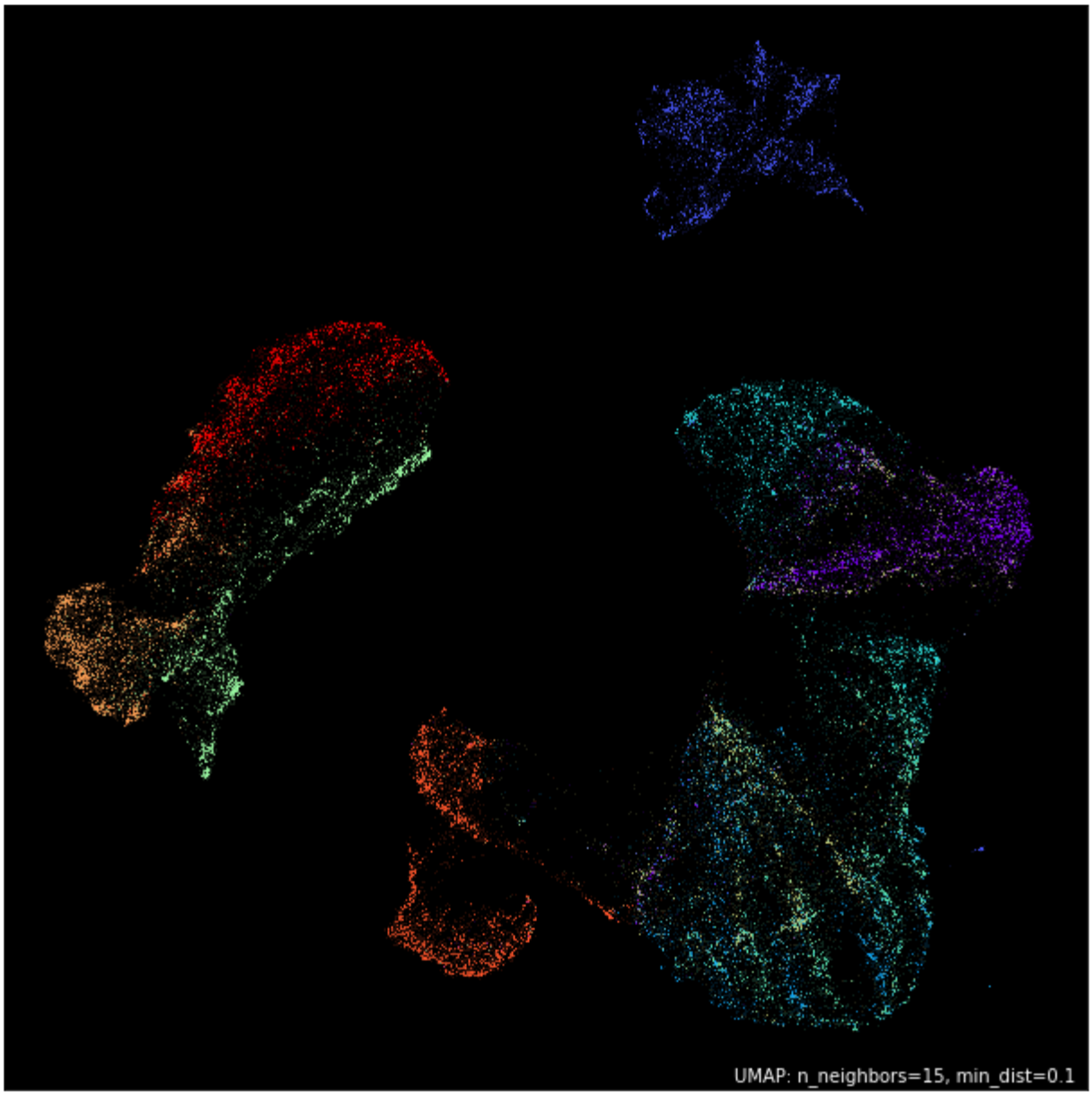
Having fit the data with UMAP we can call `umap.plot.points` exactly as before, but this time, since the data is large enough to have potential overplotting, datashader will be used in the background for rendering.

```
umap.plot.points(mapper)
```



All the same plot options as before hold, so we can color by labels, and apply the same themes, and it will all seamlessly use datashader for the actual rendering. Thus, regardless of how much data you have `umap.plot.points` will render it well with a transparent user interface. You, as a user, don't need to worry about switching to plotting with datashader, or how to convert your plotting to its slightly different API – you can just use the same API and trust the results you get.

```
umap.plot.points(mapper, labels=fmnist.target, theme='fire')
```



3.2 Interactive plotting, and hover tools

Rendering good looking static plots is important, but what if you want to be able to interact with your data – pan around, and zoom in on the clusters to see the finer structure? What if you want to annotate your data with more complex labels than merely colors? Wouldn't it be good to be able to hover over data points and get more information about the individual point? Since this is a very common use case `umap.plot` tries to make it easy to quickly generate such plots, and provide basic utilities to allow you to have annotated hover tools working quickly. Again, the goal is not to provide a comprehensive solution that can do everything, but rather a simple to use and consistent API to get users up and running fast.

To make a good example of this let's use a subset of the Fashion MNIST dataset. We can quickly train a new mapper object on that.

```
mapper = umap.UMAP().fit(fmnist.data[:30000])
```

The goal is to be able to hover over different points and see data associated with the given point (or points) under the cursor. For this simple demonstration we'll just use the target information of the point. To create hover information you need to construct a dataframe of all the data you would like to appear in the hover. Each row should correspond to a source of data points (appearing in the same order), and the columns can provide whatever extra data you would like to display in the hover tooltip. In this case we'll need a dataframe that can include the index of the point, its target number, and the actual name of the type of fashion item that target corresponds to. This is easy to quickly put together using pandas.

```
hover_data = pd.DataFrame({'index':np.arange(30000),
                           'label':fmnist.target[:30000]})
hover_data['item'] = hover_data.label.map(
    {
        '0':'T-shirt/top',
        '1':'Trouser',
        '2':'Pullover',
        '3':'Dress',
        '4':'Coat',
        '5':'Sandal',
        '6':'Shirt',
        '7':'Sneaker',
        '8':'Bag',
        '9':'Ankle Boot',
    }
)
```

For interactive use the `umap.plot` package makes use of bokeh. Bokeh has several output methods, but in the approach we'll be outputting inline in a notebook. We have to enable this using the `output_notebook` function. Alteratively we could use `output_file` or other similar options – see the bokeh documentation for more details.

```
umap.plot.output_notebook()
```

Now we can make an interactive plot using `umap.plot.interactive`. This has a very similar API to the `umap.plot.points` approach, but also supports a `hover_data` keyword which, if passed a suitable dataframe, will provide hover tooltips in the interactive plot. Since bokeh allows different outputs, to display it in the notebook we will have to take the extra stop of calling `show` on the result.

```
p = umap.plot.interactive(mapper, labels=fmnist.target[:30000], hover_data=hover_data,
↪ point_size=2)
umap.plot.show(p)
```

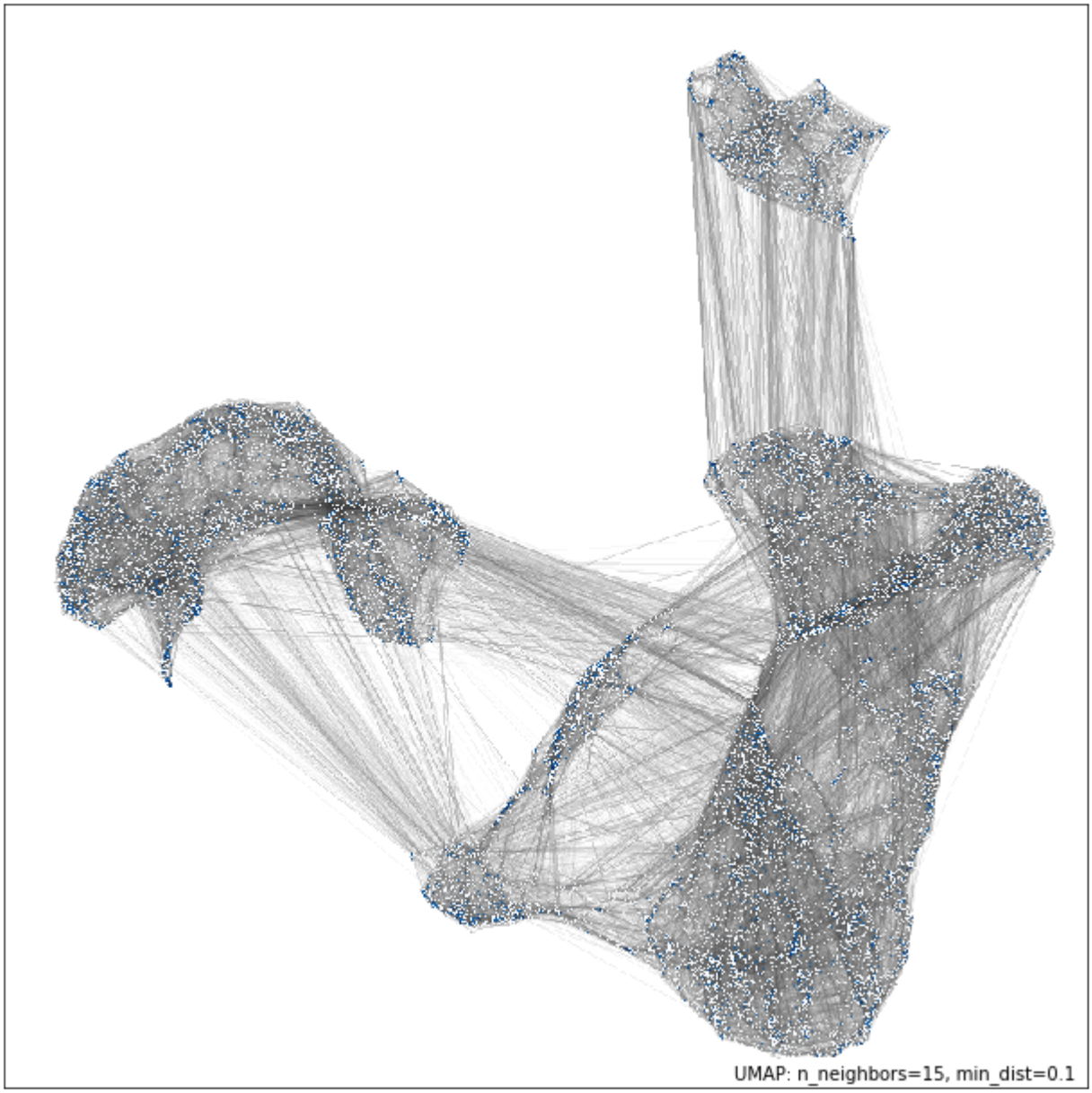
We get the sort of result one would like – a fully interactive plot that can be zoomed in on, and more, but we also now have an interactive hover tool which presents the data from the dataframe we constructed. This allows a quick and easy method to get up and running with a richer interactive exploration of your UMAP plot. `umap.plot.interactive` supports all the same aesthetic parameters as `umap.plot.points` so you can theme your plot, color by label or value, and other similar operations explained above for `umap.plot.points`.

3.3 Plotting connectivity

UMAP works by constructing an intermediate topological representation of the approximate manifold the data may have been sampled from. In practice this structure can be simplified down to a weighted graph. Sometimes it can be beneficial to see how that graph (representing connectivity in the manifold) looks with respect to the resulting embedding. It can be used to better understand the embedding, and for diagnostic purposes. To see the connectivity

you can use the `umap.plot.connectivity` function. It works very similarly to the `umap.plot.points` function, and has the option as to whether to display the embedding point, or just the connectivity. To start let's do a simple plot showing the points:

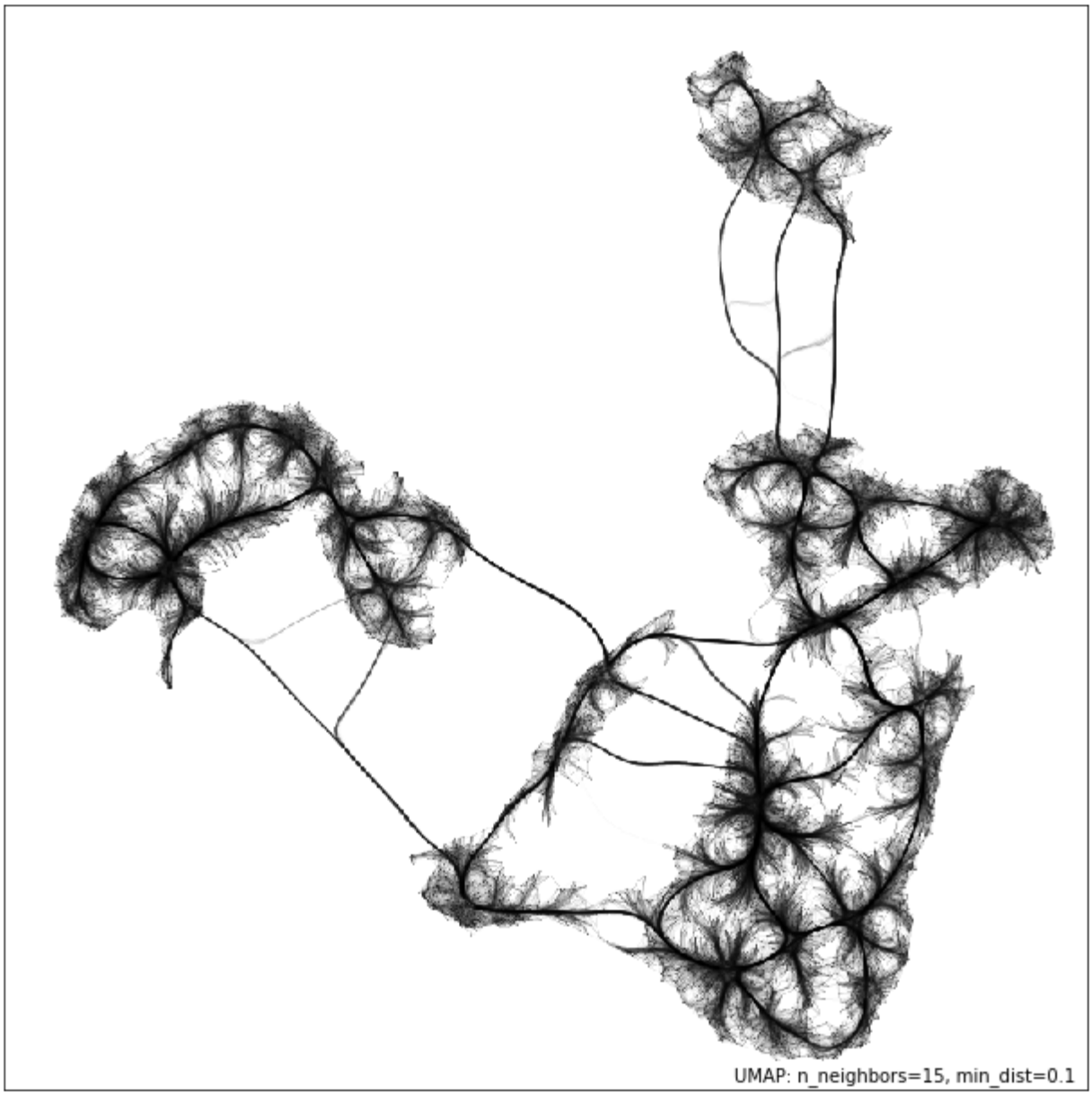
```
umap.plot.connectivity(mapper, show_points=True)
```



As with `umap.plot.points` there are options to control the basic aesthetics, including theme options and an `edge_cmap` keyword argument to specify the colormap used for displaying the edges.

Since this approach already leverages datashader for edge plotting, we can go a step further and make use of the edge-bundling options available in datashader. This can provide a less busy view of connectivity, but can be expensive to compute, particularly for larger datasets.

```
umap.plot.connectivity(mapper, edge_bundling='hammer')
```



3.4 Diagnostic plotting

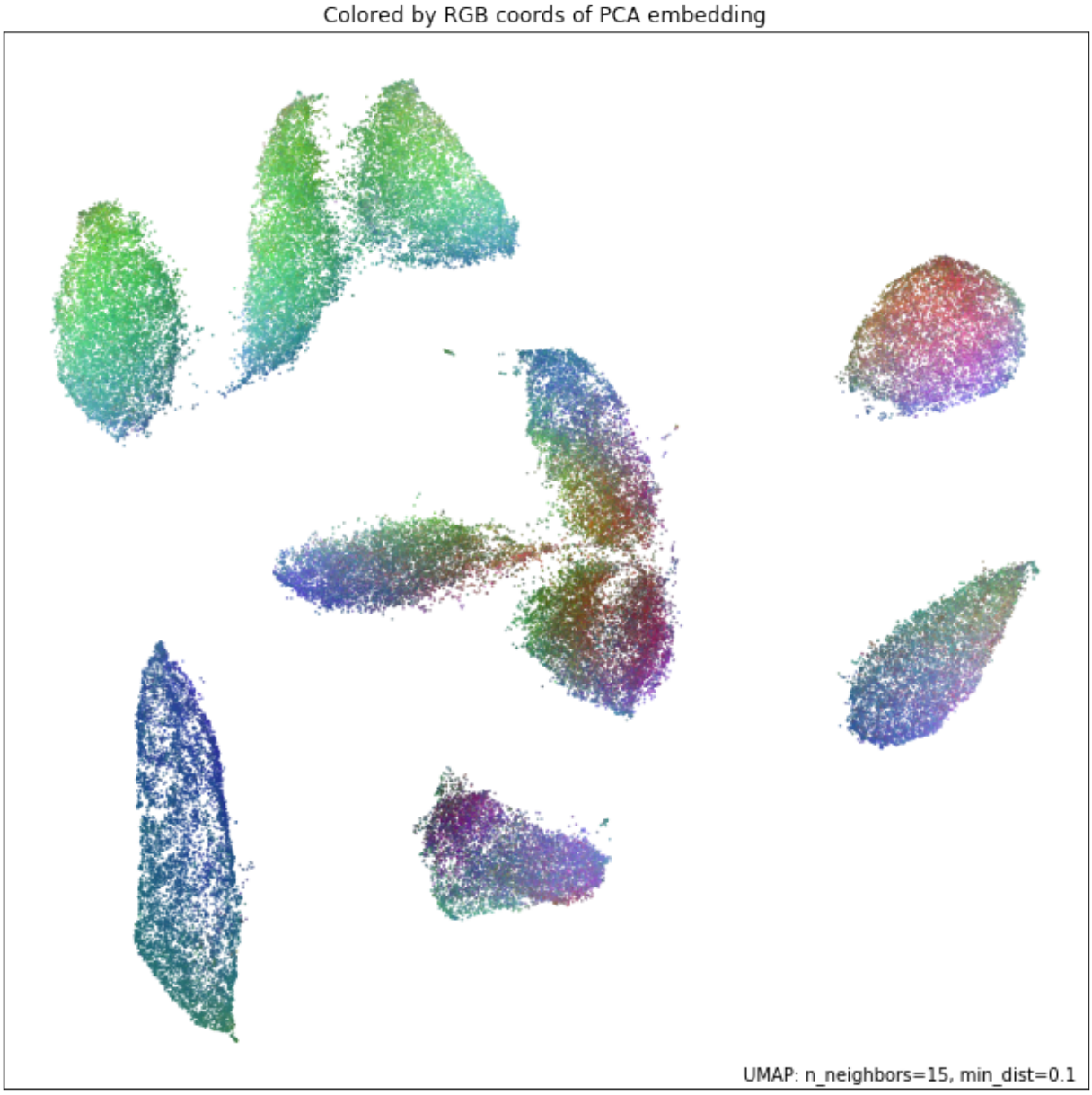
Plotting the connectivity provides at least one basic diagnostic view that helps a user understand what is going on with an embedding. More views on data are better, of course, so `umap.plot` includes a `umap.plot.diagnostic` function that can provide various diagnostic plots. We'll look at a few of them here. To do so we'll use the full MNIST digits data set.

```
mapper = umap.UMAP().fit(mnist.data)
```

The first diagnostic type is a Principal Components Analysis based diagnostic, which you can select with `diagnostic_type='pca'`. The essence of the approach is that we can use PCA, which preserves global structure, to reduce the data to three dimensions. If we scale the results to fit in a 3D cube we can convert the 3D PCA

coordinates of each point into an RGB description of a color. By then coloring the points in the UMAP embedding with the colors induced by the PCA it is possible to get a sense of how some of the more large scale global structure has been represented in the embedding.

```
umap.plot.diagnostic(mapper, diagnostic_type='pca')
```

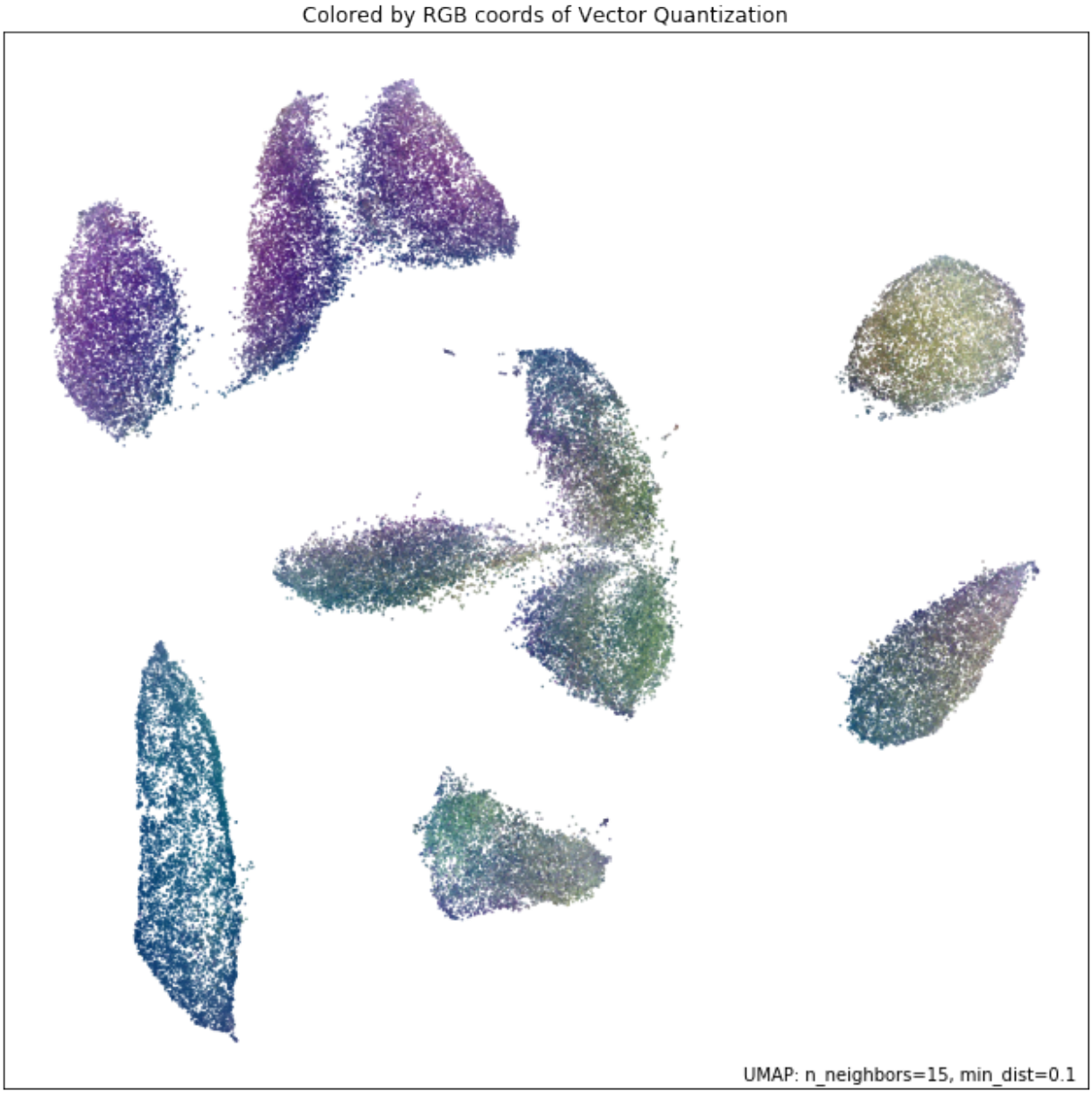


What we are looking for here is a generally smooth transition of colors, and an overall layout that broadly respects the color transitions. In this case the far left has a bottom cluster that transitions from dark green at the bottom to blue at the top, and this matches well with the cluster in the upper right which have a similar shade of blue at the bottom before transitioning to more cyan and blue. In contast in the right of the plot the lower cluster runs from purplish pink to green from top to bottom, while the cluster above it has its bottom edge more purple than green, suggesting that perhaps one or the other of these clusters has been flipped vertically during the optimization process, and this was never quite corrected.

An alternative, but similar, approach is to use vector quantization as the method to generate a 3D embedding to generate

colors. Vector quantization effectively finds 3 representative centers for the data, and then describes each data point in terms of its distance to these centers. Clearly this, again, captures a lot of the broad global structure of the data.

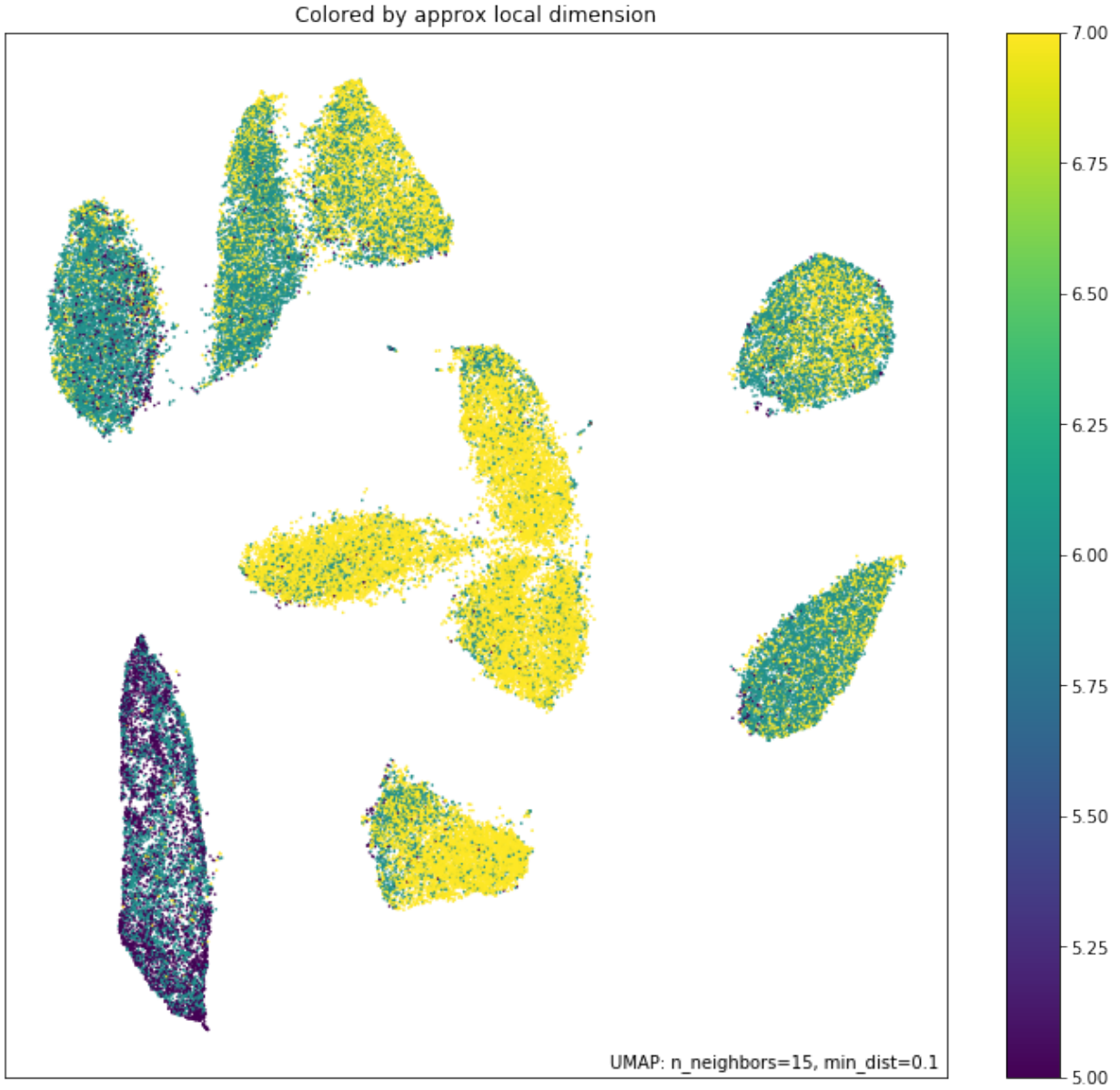
```
umap.plot.diagnostic(mapper, diagnostic_type='vq')
```



Again we are looking for largely smooth transitions, and for related colors to match up between clusters. This view supports the fact that the left hand side of the embedding has worked well, but looking at the right hand side it seems clear that it is the upper two of the clusters that has been inadvertently flipped vertically. By contrasting views like this one can get a better sense of how well the embedding is working.

For a different perspective we can look at approximations of the local dimension around each data point. Ideally the local dimension should match the embedding dimension (although this is often a lot to hope for. In practice when the local dimension is high this represents points (or areas of the space) that UMAP will have a harder time embedding as well. Thus one can trust the embedding to be more accurate in regions where the points have consistently lower local dimension.

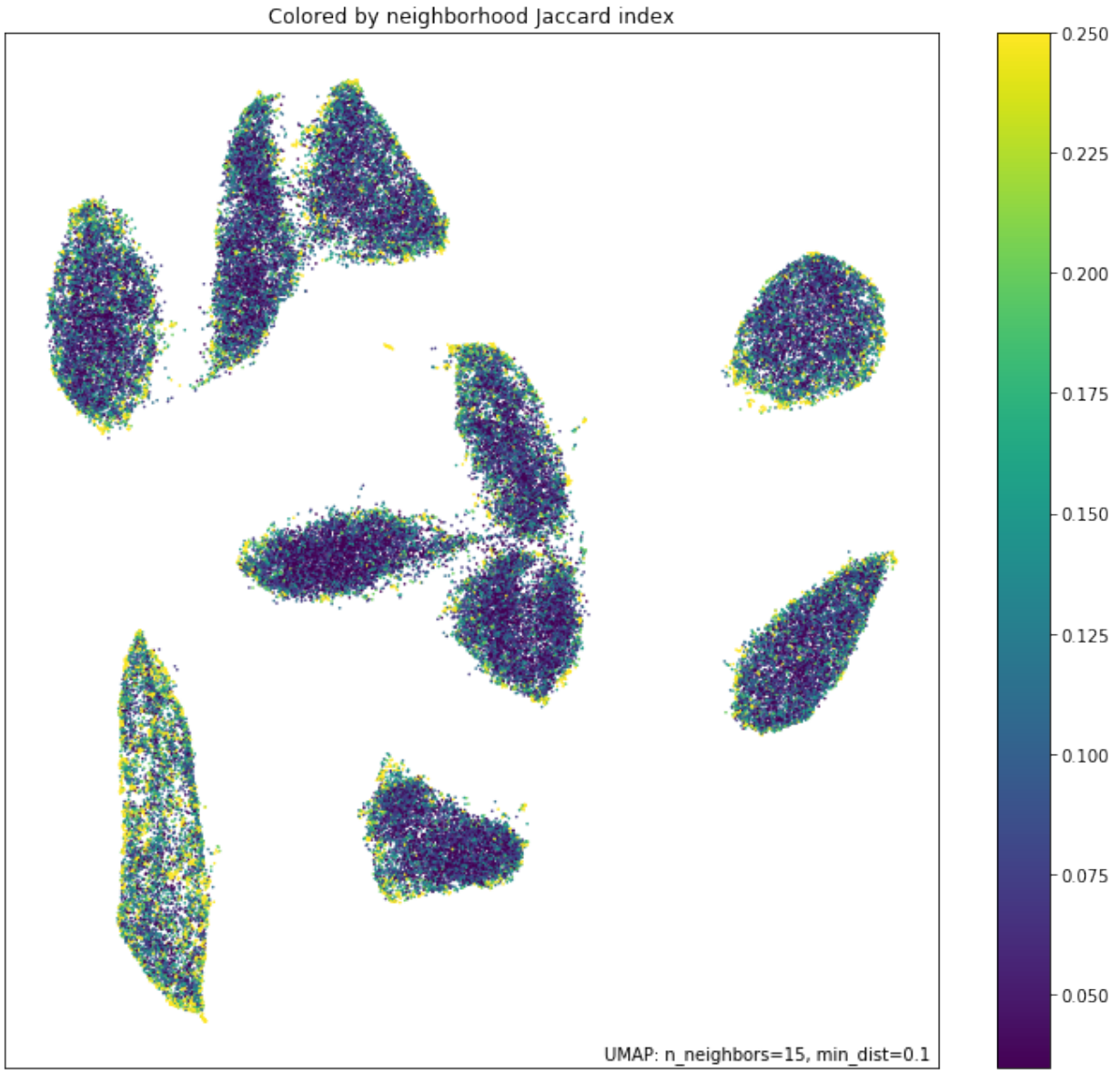
```
local_dims = umap.plot.diagnostic(mapper, diagnostic_type='local_dim')
```



As you can see, the local dimension of the data varies quite widely across the data. In particular the lower left cluster has the lowest local dimension – this is actually unsurprising as this is the cluster corresponding to the digits 1: there are relatively few degrees of freedom over how a person draws a number one, and so the resulting local dimension is lower. In contrast the clusters in the middle have a much higher local dimension. We should expect the embedidng to be a little less accurate in these regions: it is hard to represent seven dimensional data well in only two dimensions, and compromises will need to be made.

The final diagnostic we'll look at is how well local neighborhoods are preserved. We can measure this in terms of the Jaccard index of the local neighborhood in the high dimensional space compared to the equivalent neighborhood in the embedding. The Jaccard index is essentially the ratio of the number of neighbors that the two neighborhoods have in common over the total number of unique neighbors across the two neighborhoods. Higher values mean that the local neighborhood has been more accurately preserved.

```
umap.plot.diagnostic(mapper, diagnostic_type='neighborhood')
```



As one might expect the local neighborhood preservation tends to be a lot better for those points that had a lower local dimension (as seen in the last plot). There is also a tendency for the edges of clusters (where there were clear boundaries to be followed) to have a better preservation of neighborhoods than the centers of the clusters that had higher local dimension. Again, this provides a view on which areas of the embedding you can have greater trust in, and which regions had to make compromises to embed into two dimensions.

CHAPTER 4

UMAP Reproducibility

UMAP is a stochastic algorithm – it makes use of randomness both to speed up approximation steps, and to aid in solving hard optimization problems. This means that different runs of UMAP can produce different results. UMAP is relatively stable – thus the variance between runs should ideally be relatively small – but different runs may have variations none the less. To ensure that results can be reproduced exactly UMAP allows the user to set a random seed state.

Since version 0.4 UMAP also support multi-threading for faster performance; when performing optimization this exploits the fact that race conditions between the threads are acceptable within certain optimization phases. Unfortunately this means that the randomness in UMAP outputs for the multi-threaded case depends not only on the random seed input, but also on race conditions between threads during optimization, over which no control can be had. This means that multi-threaded UMAP results cannot be explicitly reproduced.

In this tutorial we'll look at how UMAP can be used in multi-threaded mode for performance purposes, and alternatively how we can fix random states to ensure exact reproducibility at the cost of some performance. First let's load the relevant libraries and get some data; in this case the MNIST digits dataset.

```
import numpy as np
import sklearn.datasets
import umap
import umap.plot
```

```
data, labels = sklearn.datasets.fetch_openml(
    'mnist_784', version=1, return_X_y=True
)
```

With data in hand let's run UMAP on it, and note how long it takes to run:

```
%%time
mapper1 = umap.UMAP().fit(data)
```

```
CPU times: user 3min 18s, sys: 3.84 s, total: 3min 22s
Wall time: 1min 29s
```

The thing to note here is that the "Wall time" is significantly smaller than the CPU time – this means that multiple CPU cores were used. For this particular demonstration I am making use of the latest version of PyNNDescent for nearest neighbor search (UMAP will use it if you have it installed) which supports multi-threading as well. The result is a very fast fitting to the data that does an effective job of using several cores. If you are on a large server with many cores available and don't wish to use them *all* (which is the default situation) you can currently control the number of cores used by setting the numba environment variable `NUMBA_NUM_THREADS`; see the numba documentation for more details.

Now let's plot our result to see what the embedding looks like:

```
umap.plot.points(mapper1, labels=labels)
```

Now, let's run UMAP again and compare the results to that of our first run.

```
%%time
mapper2 = umap.UMAP().fit(data)
```

```
CPU times: user 2min 53s, sys: 4.16 s, total: 2min 57s
Wall time: 1min 5s
```

You will note that this time we ran *even faster*. This is because during the first run numba was still JIT compiling some of the code in the background. In contrast, this time that work has already been done, so it no longer takes up any of our run-time. We see that we are still making use of mutliple cores well.

Now let's plot the results of this second run and compare to the first:

```
umap.plot.points(mapper2, labels=labels)
```

Qualitatively this looks very similar, but a little closer inspection will quickly show that the results are actually different

between the runs. Note that even in versions of UMAP prior to 0.4 this would have been the case – since we fixed no specific random seed, and were thus using the current random state of the system which will naturally differ between runs. This is the default behaviour, as is standard with sklearn estimators that are stochastic. Rather than having a default random seed the user is required to explicitly provide one should they want a reproducible result. As noted by Vito Zanotelli

> ... setting a random seed is like signing a waiver "I am aware that this is a stochastic algorithm and I have done sufficient tests to confirm that my main conclusions are not affected by this randomness".

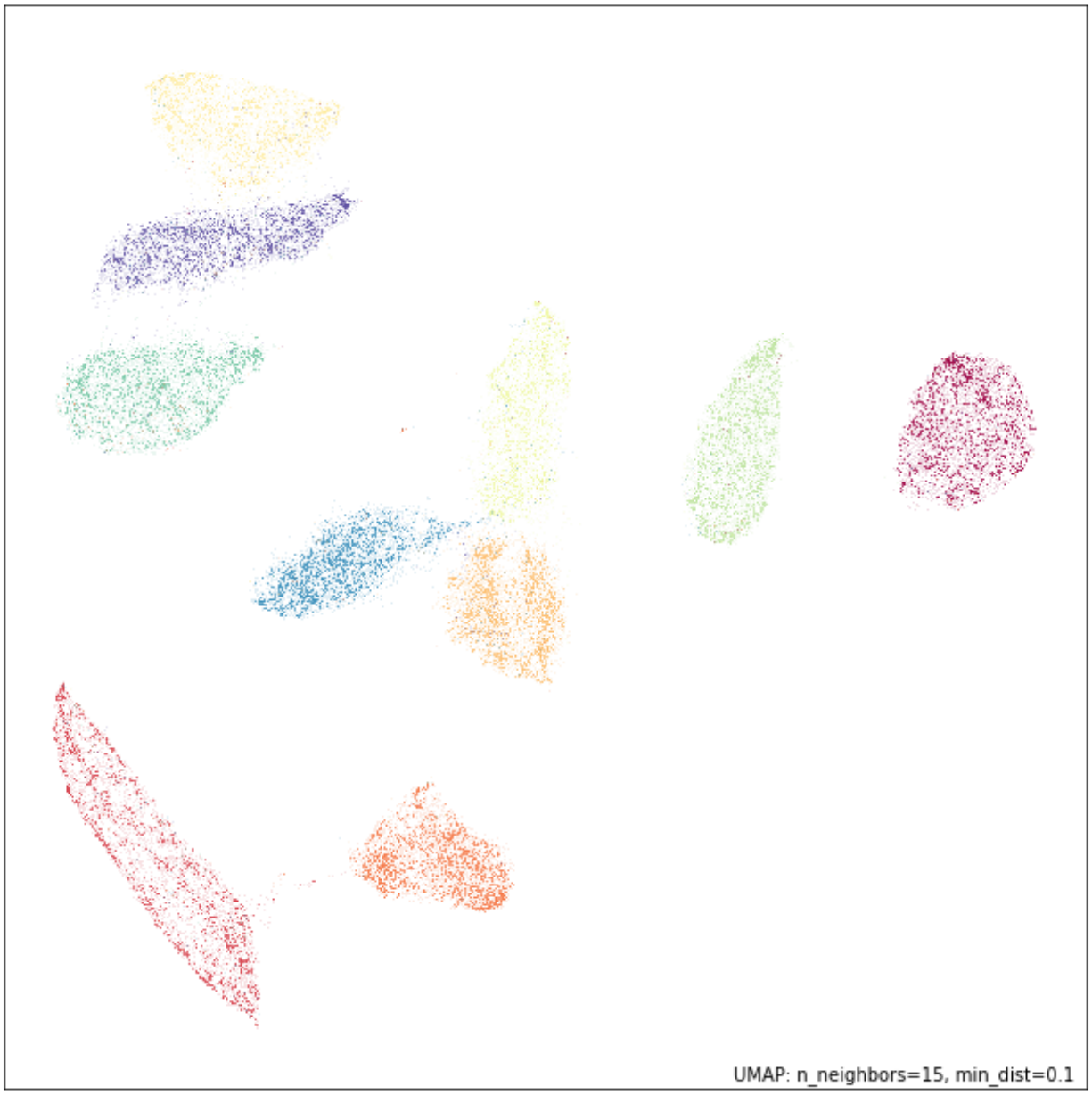
With that in mind, let's see what happens if we set an explicit `random_state` value:

```
%%time
mapper3 = umap.UMAP(random_state=42).fit(data)
```

```
CPU times: user 2min 27s, sys: 4.16 s, total: 2min 31s
Wall time: 1min 56s
```

The first thing to note that that this run took significantly longer (despite having all the functions JIT compiled by numba already). Then note that the Wall time and CPU times are now much closer to each other – we are no longer exploiting multiple cores to anywhere near the same degree. This is because by setting a `random_state` we are effectively turning off any of the multi-threading that does not support explicit reproducibility. Let's plot the results:

```
umap.plot.points(mapper3, labels=labels)
```



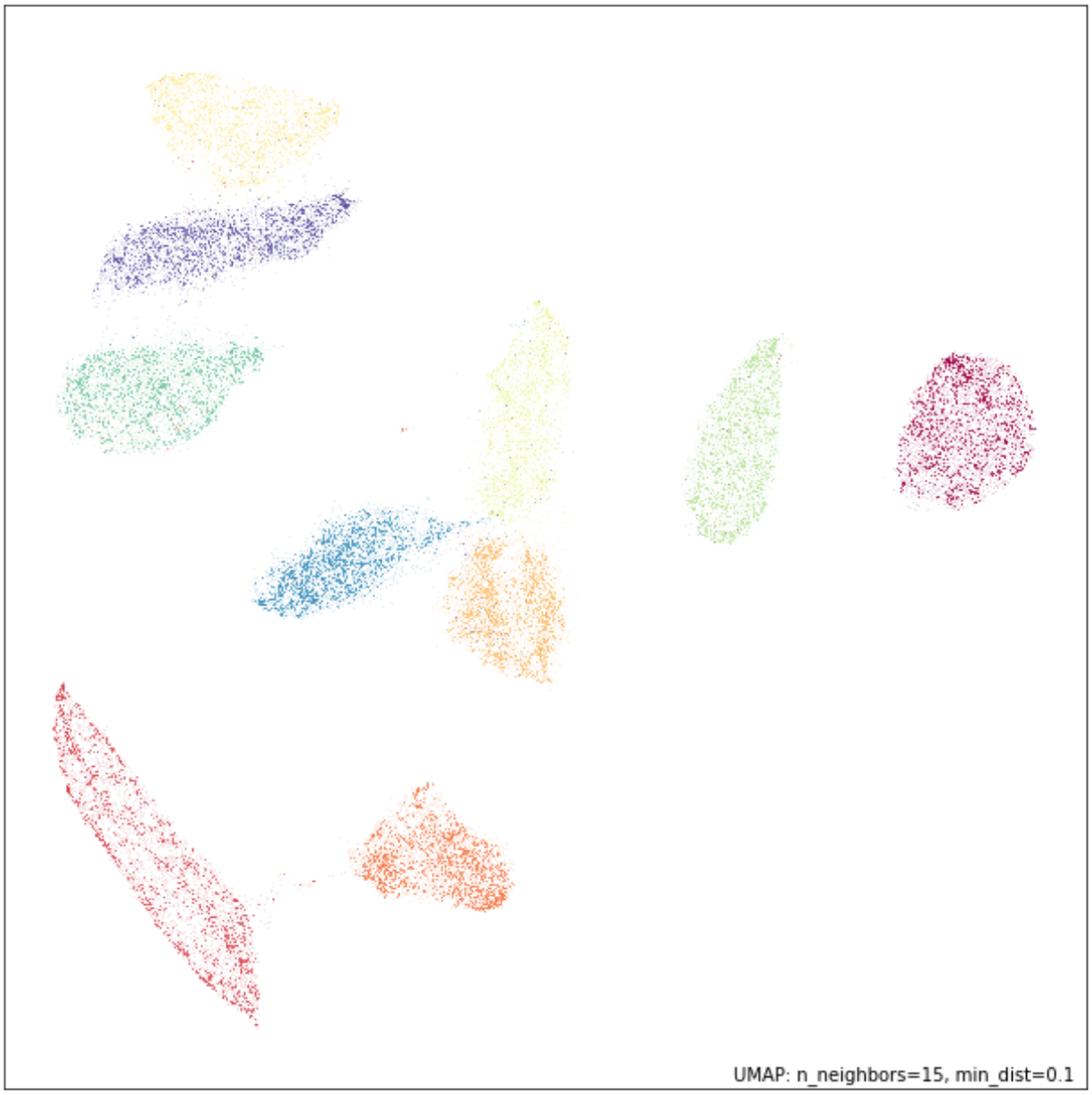
We arrive at much the same results as before from a qualitative point of view, but again inspection will show that there are some differences. More importantly this result should now be reproducible. Thus we can run UMAP again, with the same `random_state` set ...

```
%%time
mapper4 = umap.UMAP(random_state=42).fit(data)
```

```
CPU times: user 2min 26s, sys: 4.13 s, total: 2min 30s
Wall time: 1min 54s
```

Again, this takes longer than the earlier runs with no `random_state` set. However when we plot the results of the second run we see that they look not merely qualitatively similar, but instead appear to be almost identical:

```
umap.plot.points(mapper4, labels=labels)
```



We can, in fact, check that the results are identical by verifying that each and every coordinate of the resulting embeddings match perfectly:

```
np.all(mapper3.embedding_ == mapper4.embedding_)
```

```
True
```

So we have, in fact, reproduced the embedding exactly.

CHAPTER 5

Transforming New Data with UMAP

UMAP is useful for generating visualisations, but if you want to make use of UMAP more generally for machine learning tasks it is important to be be able to train a model and then later pass new data to the model and have it transform that data into the learned space. For example if we use UMAP to learn a latent space and then train a classifier on data transformed into the latent space then the classifier is only useful for prediction if we can transform data for which we want a prediction into the latent space the classifier uses. Fortunately UMAP makes this possible, albeit more slowly than some other transformers that allow this.

To demonstrate this functionality we'll make use of scikit-learn and the digits dataset contained therein (see *How to Use UMAP* for an example of the digits dataset). First let's load all the modules we'll need to get this done.

```
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC

import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

```
sns.set(context='notebook', style='white', rc={'figure.figsize':(14,10)})
```

```
digits = load_digits()
```

To keep everything honest let's use sklearn `train_test_split` to separate out a training and test set (stratified over the different digit types). By default `train_test_split` will carve off 25% of the data for testing, which seems suitable in this case.

```
X_train, X_test, y_train, y_test = train_test_split(digits.data,
                                                    digits.target,
                                                    stratify=digits.target,
                                                    random_state=42)
```

Now to get a benchmark idea of what we are looking at let's train a couple of different classifiers and then see how well they score on the test set. For this example let's try a support vector classifier and a KNN classifier. Ideally we should be tuning hyper-parameters (perhaps a grid search using k-fold cross validation), but for the purposes of this simple demo we will simply use default parameters for both classifiers.

```
svc = SVC().fit(X_train, y_train)
knn = KNeighborsClassifier().fit(X_train, y_train)
```

The next question is how well these classifiers perform on the test set. Conveniently sklearn provides a `score` method that can output the accuracy on the test set.

```
svc.score(X_test, y_test), knn.score(X_test, y_test)
```

```
(0.62, 0.9844444444444445)
```

The result is that the support vector classifier apparently had poor hyper-parameters for this case (I expect with some tuning we could build a much more accurate model) and the KNN classifier is doing very well.

The goal now is to make use of UMAP as a preprocessing step that one could potentially fit into a pipeline. We will therefore obviously need the `umap` module loaded.

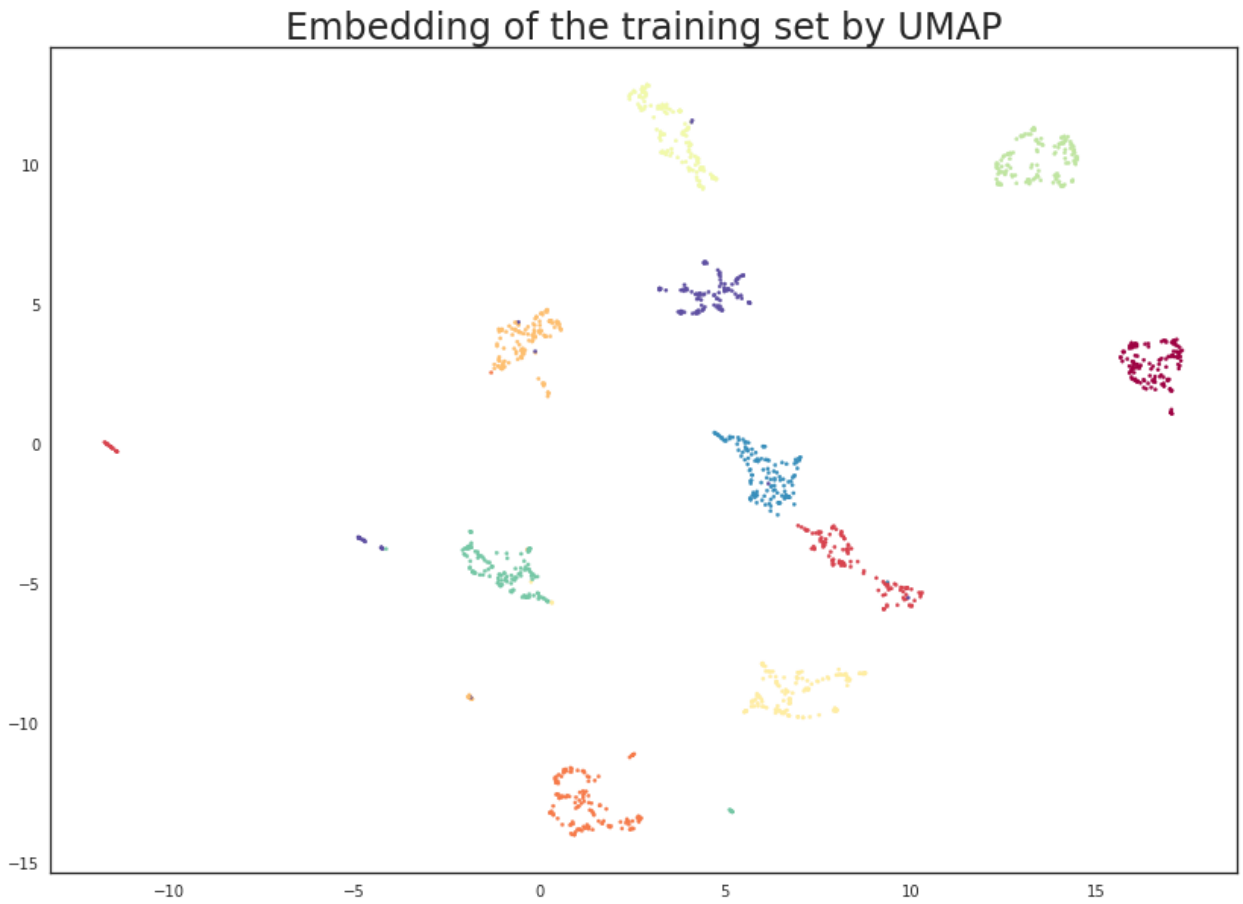
```
import umap
```

To make use of UMAP as a data transformer we first need to fit the model with the training data. This works exactly as in the *How to Use UMAP* example using the fit method. In this case we simply hand it the training data and it will learn an appropriate (two dimensional by default) embedding.

```
trans = umap.UMAP(n_neighbors=5, random_state=42).fit(X_train)
```

Since we embedded to two dimensions we can visualise the results to ensure that we are getting a potential benefit out of this approach. This is simply a matter of generating a scatterplot with data points colored by the class they come from. Note that the embedded training data can be accessed as the `.embedding_` attribute of the UMAP model once we have fit the model to some data.

```
plt.scatter(trans.embedding_[:, 0], trans.embedding_[:, 1], s= 5, c=y_train, cmap=
↪'Spectral')
plt.title('Embedding of the training set by UMAP', fontsize=24);
```



This looks very promising! Most of the classes got very cleanly separated, and that gives us some hope that it could help with classifier performance. It is worth noting that this was a completely unsupervised data transform; we could have used the training label information, but that is the subject of *a later tutorial*.

We can now train some new models (again an SVC and a KNN classifier) on the embedded training data. This looks exactly as before but now we pass it the embedded data. Note that calling `transform` on input identical to what the model was trained on will simply return the `embedding_` attribute, so sklearn pipelines will work as expected.

```
svc = SVC().fit(trans.embedding_, y_train)
knn = KNeighborsClassifier().fit(trans.embedding_, y_train)
```

Now we want to work with the test data which none of the models (UMAP or the classifiers) have seen. To do this we use the standard sklearn API and make use of the `transform` method, this time handing it the new unseen test data. We will assign this to `test_embedding` so that we can take a closer look at the result of applying an existing UMAP model to new data.

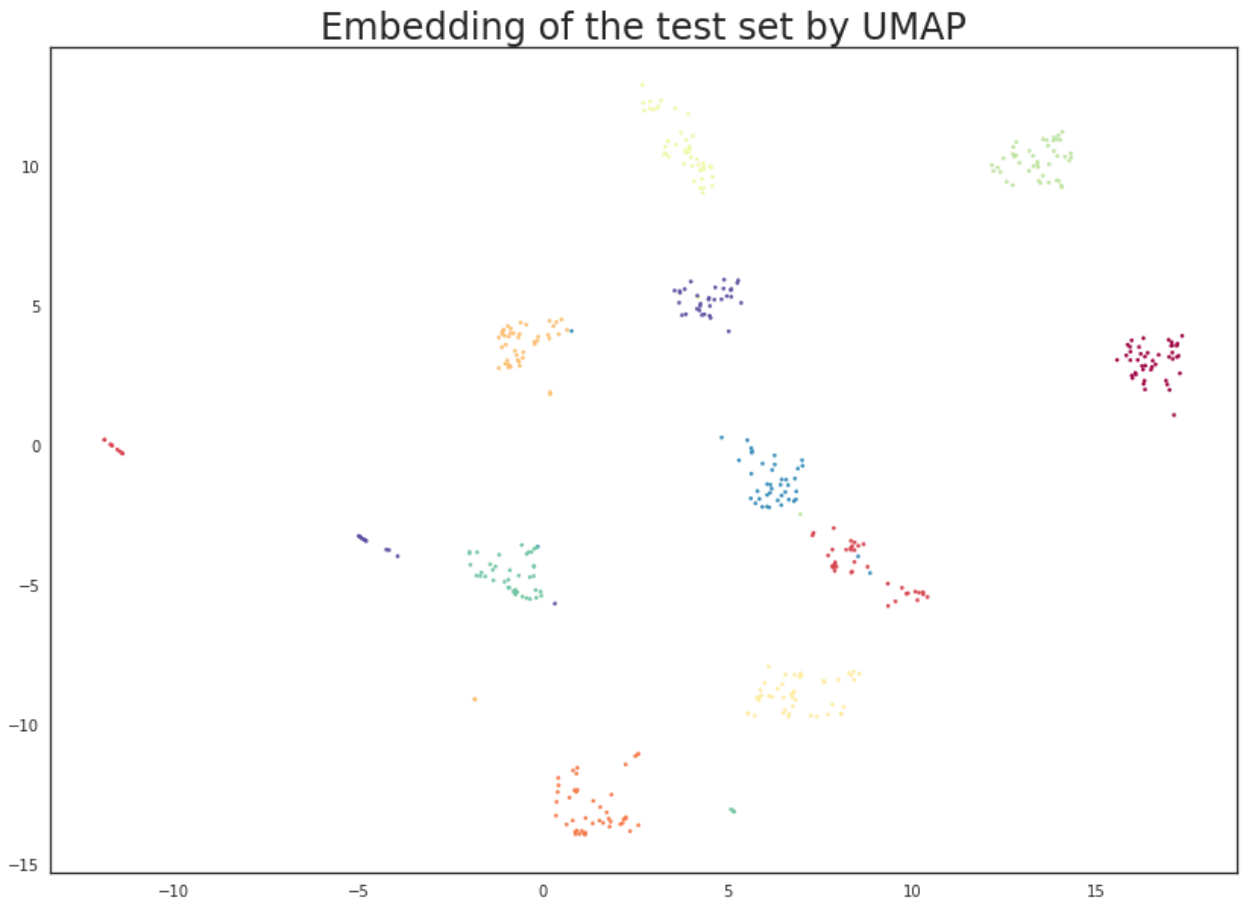
```
%time test_embedding = trans.transform(X_test)
```

```
CPU times: user 867 ms, sys: 70.7 ms, total: 938 ms
Wall time: 335 ms
```

Note that the transform operations works very efficiently – taking less than half a second. Compared to some other transformers this is a little on the slow side, but it is fast enough for many uses. Note that as the size of the training and/or test sets increase the performance will slow proportionally. It's also worth noting that the first call to transform may be slow due to Numba JIT overhead – further runs will be very fast.

The next important question is what the transform did to our test data. In principle we have a new two dimensional representation of the test-set, and ideally this should be based on the existing embedding of the training set. We can check this by visualising the data (since we are in two dimensions) to see if this is true. A simple scatterplot as before will suffice.

```
plt.scatter(test_embedding[:, 0], test_embedding[:, 1], s= 5, c=y_test, cmap='Spectral
↪')
plt.title('Embedding of the test set by UMAP', fontsize=24);
```



The results look like what we should expect; the test data has been embedded into two dimensions in exactly the locations we should expect (by class) given the embedding of the training data visualised above. This means we can now try out models that were trained on the embedded training data by handing them the newly transformed test set.

```
svc.score(trans.transform(X_test), y_test), knn.score(trans.transform(X_test), y_test)
```

```
(0.9844444444444445, 0.9844444444444445)
```

The results are pretty good. While the accuracy of the KNN classifier did not improve there was not a lot of scope for improvement given the data. On the other hand the SVC has improved to have equal accuracy to the KNN classifier. Of course we could probably have achieved this level of accuracy by better setting SVC hyper-parameters, but the point here is that we can use UMAP as if it were a standard sklearn transformer as part of an sklearn machine learning pipeline.

Just for fun we can run the same experiments, but this time reduce to ten dimensions (where we can no longer visualise). In practice this will have little gain in this case – for the digits dataset two dimensions is plenty for UMAP and more dimensions won't help. On the other hand for more complex datasets where more dimensions may allow for

a much more faithful embedding it is worth noting that we are not restricted to only two dimension.

```
trans = umap.UMAP(n_neighbors=5, n_components=10, random_state=42).fit(X_train)
```

```
svc = SVC().fit(trans.embedding_, y_train)
knn = KNeighborsClassifier().fit(trans.embedding_, y_train)
```

```
svc.score(trans.transform(X_test), y_test), knn.score(trans.transform(X_test), y_test)
```

```
(0.9822222222222222, 0.9822222222222222)
```

And we see that in this case we actually marginally lowered our accuracy scores (within the potential noise in such scoring mind you). However for more interesting datasets the larger dimensional embedding might have been a significant gain – it is certainly worth exploring as one of the parameters in a grid search across a pipeline that includes UMAP.

CHAPTER 6

Inverse transforms

UMAP has some support for inverse transforms – generating a high dimensional data sample given a location in the low dimensional embedding space. To start let's load all the relevant libraries.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec
import seaborn as sns
import sklearn.datasets
import umap
import umap.plot
```

We will need some data to test with. To start we'll use the MNIST digits dataset. This is a dataset of 70000 handwritten digits encoded as grayscale 28x28 pixel images. Our goal is to use UMAP to reduce the dimension of this dataset to something small, and then see if we can generate new digits by sampling points from the embedding space. To load the MNIST dataset we'll make use of sklearn's `fetch_openml` function.

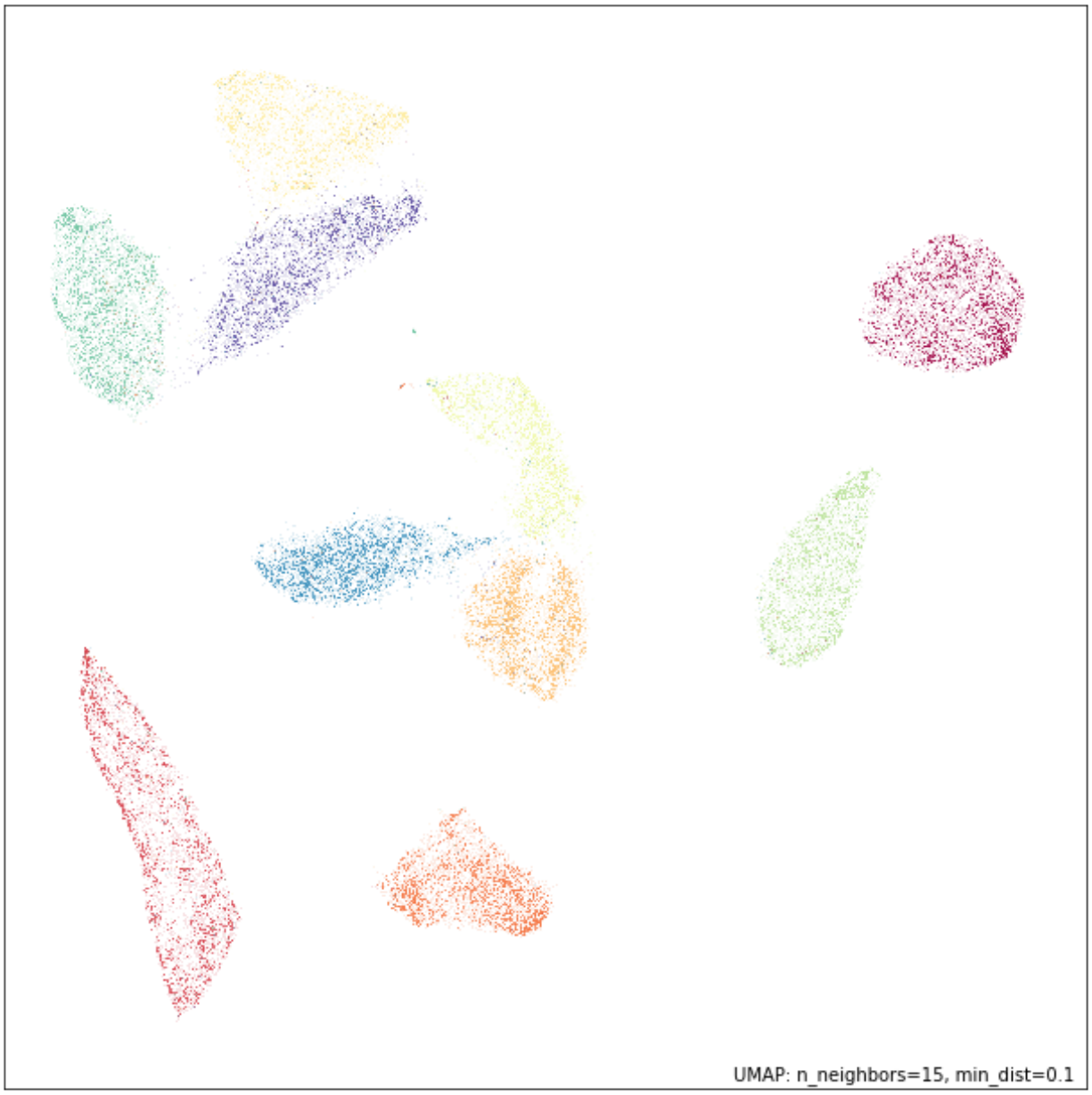
```
data, labels = sklearn.datasets.fetch_openml('mnist_784', version=1, return_X_y=True)
```

Now we need to generate a reduced dimension representation of this data. This is straightforward with UMAP, but in this case rather than using `fit_transform` we'll use the fit method so that we can retain the trained model for later generating new digits based on samples from the embedding space.

```
mapper = umap.UMAP(random_state=42).fit(data)
```

To ensure that things worked correctly we can plot the data (since we reduced it to two dimensions). We'll use the `umap.plot` functionality to do this.

```
umap.plot.points(mapper, labels=labels)
```



This looks much like we would expect. The different digit classes have been decently separated. Now we need to create a set of samples in the embedding space to apply the `inverse_transform` operation to. To do this we'll generate a grid of samples linearly interpolating between four corner points. To make our selection interesting we'll carefully choose the corners to span over the dataset, and sample different digits so that we can better see the transitions.

```
corners = np.array([
    [-5, -10],  # 1
    [-7, 6],  # 7
    [2, -8],  # 2
    [12, 4],  # 0
])

test_pts = np.array([
    (corners[0]*(1-x) + corners[1]*x)*(1-y) +
```

(continues on next page)

(continued from previous page)

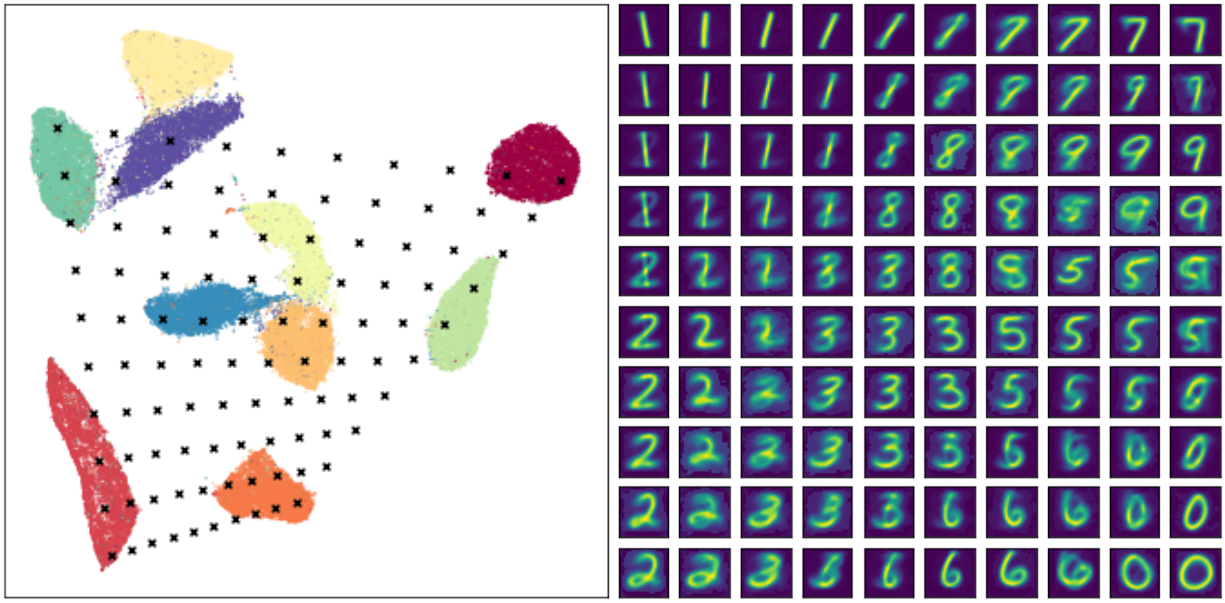
```
    (corners[2]*(1-x) + corners[3]*x)*y
    for y in np.linspace(0, 1, 10)
    for x in np.linspace(0, 1, 10)
])
```

Now we can apply the `inverse_transform` method to this set of test points. Each test point is a two dimensional point lying somewhere in the embedding space. The `inverse_transform` method will convert this into an approximation of the high dimensional representation that would have been embedded into such a location. Following the sklearn API this is as simple to use as calling the `inverse_transform` method of the trained model and passing it the set of test points that we want to convert into high dimensional representations. Be warned that this can be quite expensive computationally.

```
inv_transformed_points = mapper.inverse_transform(test_pts)
```

Now the goal is to visualize how well we have done. Effectively what we would like to do is show the test points in the embedding space, and then show a grid of the corresponding images generated by the inverse transform. To get all of this in a single matplotlib figure takes a little setting up, but is quite manageable – mostly it is just a matter of managing `GridSpec` formatting. Once we have that setup we just need a scatterplot of the embedding, a scatterplot of the test points, and finally a grid of the images we generated (converting the inverse transformed vectors into images is just a matter of reshaping them back to 28 by 28 pixel grids and using `imshow`).

```
# Set up the grid
fig = plt.figure(figsize=(12,6))
gs = GridSpec(10, 20, fig)
scatter_ax = fig.add_subplot(gs[:, :10])
digit_axes = np.zeros((10, 10), dtype=object)
for i in range(10):
    for j in range(10):
        digit_axes[i, j] = fig.add_subplot(gs[i, 10 + j])

# Use umap.plot to plot to the major axis
# umap.plot.points(mapper, labels=labels, ax=scatter_ax)
scatter_ax.scatter(mapper.embedding_[:, 0], mapper.embedding_[:, 1],
                   c=labels.astype(np.int32), cmap='Spectral', s=0.1)
scatter_ax.set(xticks=[], yticks=[])

# Plot the locations of the text points
scatter_ax.scatter(test_pts[:, 0], test_pts[:, 1], marker='x', c='k', s=15)

# Plot each of the generated digit images
for i in range(10):
    for j in range(10):
        digit_axes[i, j].imshow(inv_transformed_points[i*10 + j].reshape(28, 28))
        digit_axes[i, j].set(xticks=[], yticks=[])
```



The end result looks pretty good – we did indeed generate plausible looking digit images, and many of the transitions (from 1 to 7 across the top row for example) seem pretty natural and make sense. This can help you to understand the structure of the cluster of 1s (it transitions on the angle, sloping toward what will eventually be 7s), and why 7s and 9s are close together in the embedding. Of course there are also some stranger transitions, especially where the test points fell into large gaps between clusters in the embedding – in some sense it is hard to interpret what should go in some of those gaps as they don't really represent anything resembling a smooth transition).

A further note: None of the test points chosen fall outside the convex hull of the embedding. This is deliberate – the inverse transform function operates poorly outside the bounds of that convex hull. Be warned that if you select points to inverse transform that are outside the bounds about the embedding you will likely get strange results (often simply snapping to a particular source high dimensional vector).

Let's continue the demonstration by looking at the Fashion MNIST dataset. As before we can load this through sklearn.

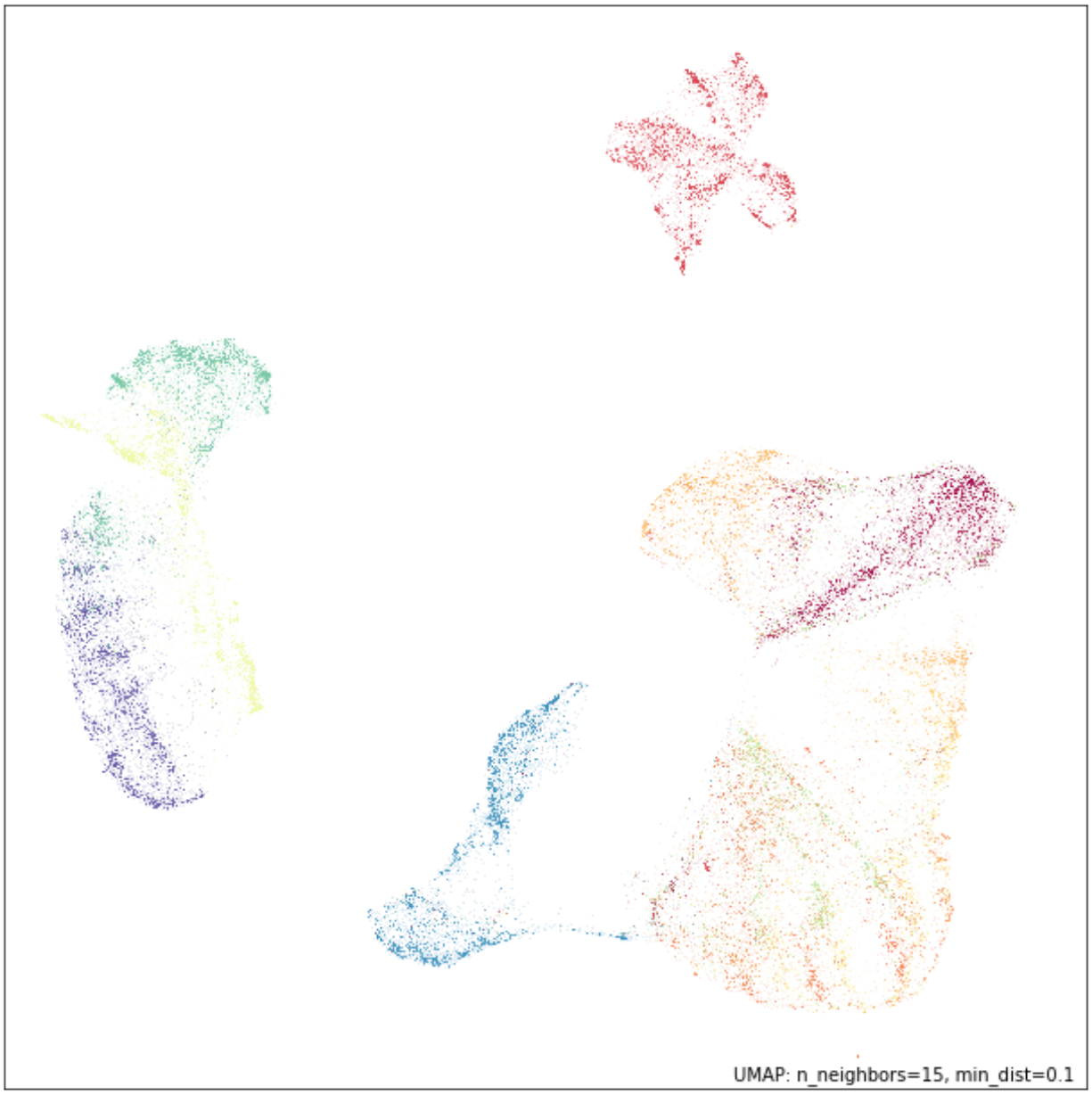
```
data, labels = sklearn.datasets.fetch_openml('Fashion-MNIST', version=1, return_X_
↪y=True)
```

Again we can fit this data with UMAP and get a mapper object.

```
mapper = umap.UMAP(random_state=42).fit(data)
```

Let's plot the embedding to see what we got as a result:

```
umap.plot.points(mapper, labels=labels)
```



Again we'll generate a set of test points by making a grid interpolating between four corners. As before we'll select the corners so that we can stay within the convex hull of the embedding points and ensure nothing too strange happens with the inverse transforms.

```
corners = np.array([
    [-2, -6],  # bags
    [-9, 3],  # boots?
    [7, -5],  # shirts/tops/dresses
    [4, 10],  # pants
])

test_pts = np.array([
    (corners[0]*(1-x) + corners[1]*x)*(1-y) +
    (corners[2]*(1-x) + corners[3]*x)*y
```

(continues on next page)

(continued from previous page)

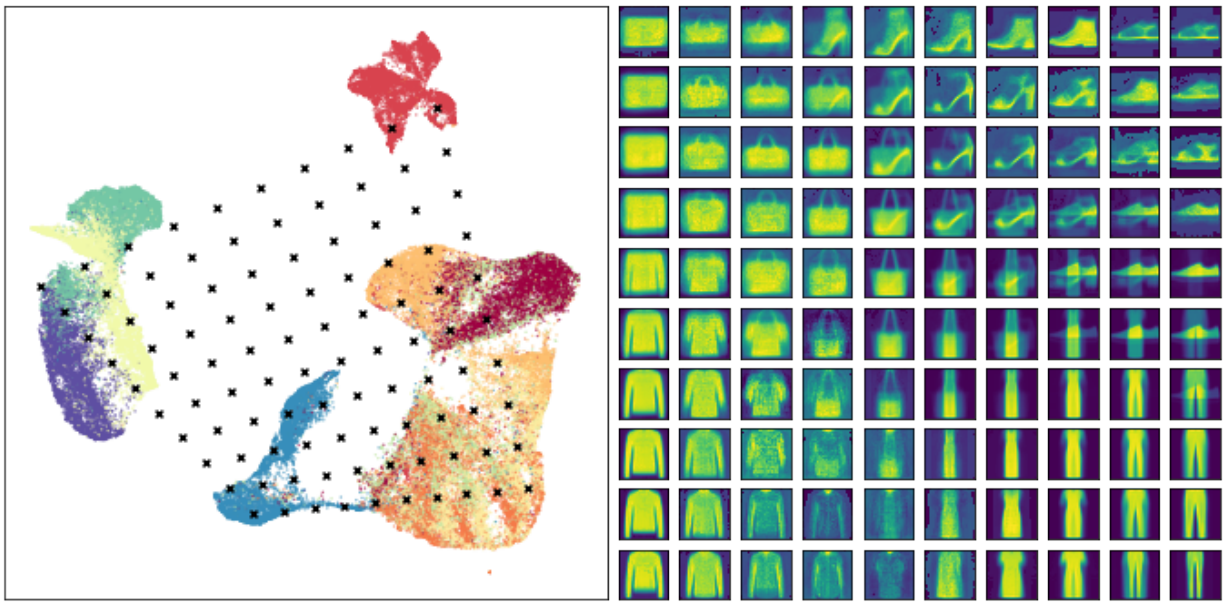
```
    for y in np.linspace(0, 1, 10)
    for x in np.linspace(0, 1, 10)
])
```

Now we simply apply the inverse transform just as before. Again, be warned, this is quite expensive computationally and may take some time to complete.

```
inv_transformed_points = mapper.inverse_transform(test_pts)
```

And now we can use similar code as above to set up our plot of the embedding with test points overlaid, and the generated images.

```
# Set up the grid
fig = plt.figure(figsize=(12,6))
gs = GridSpec(10, 20, fig)
scatter_ax = fig.add_subplot(gs[:, :10])
digit_axes = np.zeros((10, 10), dtype=object)
for i in range(10):
    for j in range(10):
        digit_axes[i, j] = fig.add_subplot(gs[i, 10 + j])

# Use umap.plot to plot to the major axis
# umap.plot.points(mapper, labels=labels, ax=scatter_ax)
scatter_ax.scatter(mapper.embedding_[:, 0], mapper.embedding_[:, 1],
                   c=labels.astype(np.int32), cmap='Spectral', s=0.1)
scatter_ax.set(xticks=[], yticks=[])

# Plot the locations of the text points
scatter_ax.scatter(test_pts[:, 0], test_pts[:, 1], marker='x', c='k', s=15)

# Plot each of the generated digit images
for i in range(10):
    for j in range(10):
        digit_axes[i, j].imshow(inv_transformed_points[i*10 + j].reshape(28, 28))
        digit_axes[i, j].set(xticks=[], yticks=[])
```

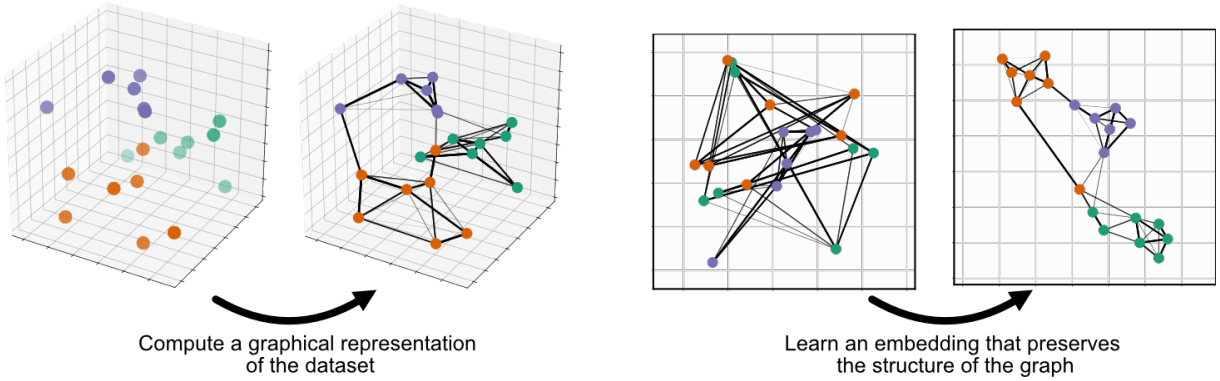


This time we see some of the interpolations between items looking rather strange – particularly the points that lie somewhere between shoes and pants – ultimately it is doing the best it can with a difficult problem. At the same time many of the other transitions seem to work pretty well, so it is, indeed, providing useful information about how the embedding is structured.

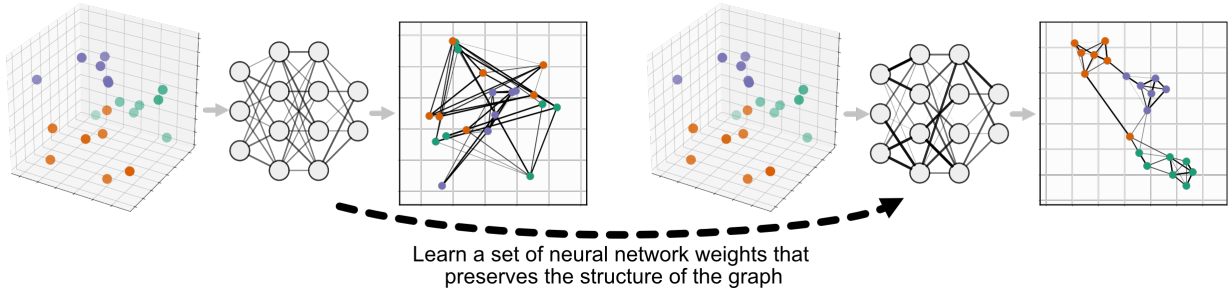
CHAPTER 7

Parametric Embedding

UMAP is comprised of two steps: First, compute a graph representing your data, second, learn an embedding for that graph:



Parametric UMAP replaces the second step, minimizing the same objective function as UMAP (we'll call it non-parametric UMAP here), but learning the relationship between the data and embedding using a neural network, rather than learning the embeddings directly:



Parametric UMAP is simply a subclass of UMAP, so it can be used just like nonparametric UMAP, replacing `umap.UMAP` with `parametric_umap.ParametricUMAP`. The most basic usage of parametric UMAP would be to simply replace UMAP with ParametricUMAP in your code:

```
from umap.parametric_umap import ParametricUMAP
embedder = ParametricUMAP()
embedding = embedder.fit_transform(my_data)
```

In this implementation, we use Keras and Tensorflow as a backend to train that neural network. The added complexity of a learned embedding presents a number of configurable settings available in addition to those in non-parametric UMAP. A set of Jupyter notebooks walking you through these parameters are available on the GitHub repository

7.1 Defining your own network

By default, parametric UMAP uses 3-layer 100-neuron fully-connected neural network. To extend Parametric UMAP to use a more complex architecture, like a convolutional neural network, we simply need to define the network and pass it in as an argument to ParametricUMAP. This can be done easliy, using tf.keras.Sequential. Here's an example for MNIST:

```
# define the network
import tensorflow as tf
dims = (28, 28, 1)
n_components = 2
encoder = tf.keras.Sequential([
    tf.keras.layers.InputLayer(input_shape=dims),
    tf.keras.layers.Conv2D(
        filters=32, kernel_size=3, strides=(2, 2), activation="relu", padding="same"
    ),
    tf.keras.layers.Conv2D(
        filters=64, kernel_size=3, strides=(2, 2), activation="relu", padding="same"
    ),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(units=256, activation="relu"),
    tf.keras.layers.Dense(units=256, activation="relu"),
    tf.keras.layers.Dense(units=n_components),
])
encoder.summary()
```

To load pass the data into ParametricUMAP, we first need to flatten it from 28x28x1 images to a 784-dimensional vector.

```
from tensorflow.keras.datasets import mnist
(train_images, Y_train), (test_images, Y_test) = mnist.load_data()
train_images = train_images.reshape((train_images.shape[0], -1))/255.
test_images = test_images.reshape((test_images.shape[0], -1))/255.
```

We can then the network into ParametricUMAP and train:

```
# pass encoder network to ParametricUMAP
embedder = ParametricUMAP(encoder=encoder, dims=dims)
embedding = embedder.fit_transform(train_images)
```

If you are unfamilar with Tensorflow/Keras and want to train your own model, we reccomend that you take a look at the Tensorflow documentation.

7.2 Saving and loading your model

Unlike non-parametric UMAP Parametric UMAP cannot be saved simply by pickling the UMAP object because of the Keras networks it contains. To save Parametric UMAP, there is a build in function:

```
embedder.save('/your/path/here')
```

You can then load parametric UMAP elsewhere:

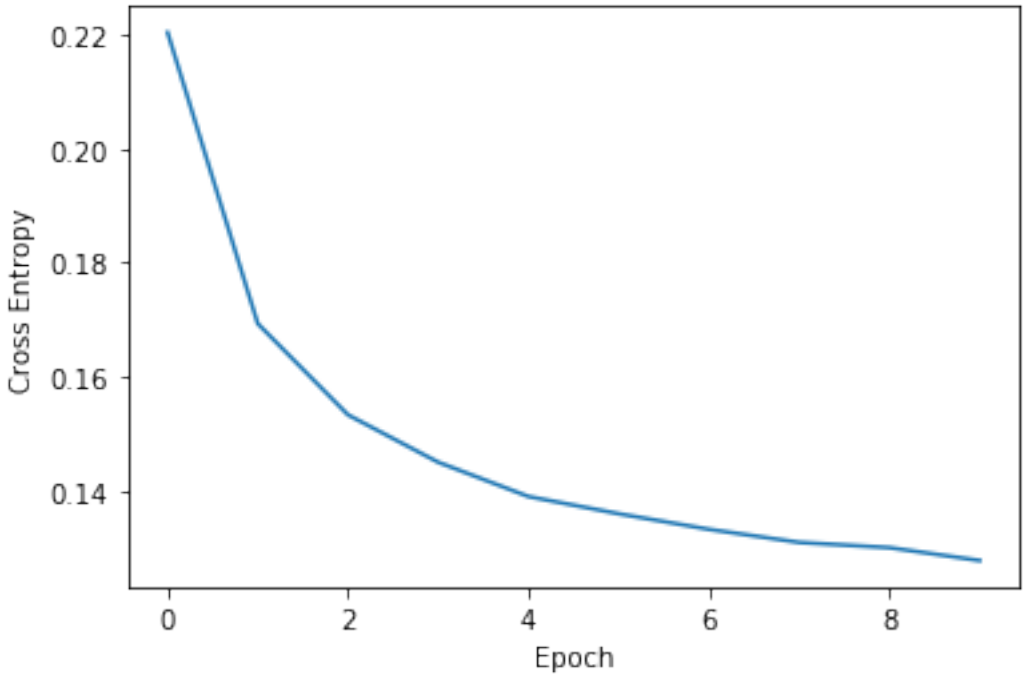
```
from umap.parametric_umap import load_ParametricUMAP
embedder = load_ParametricUMAP('/your/path/here')
```

This loads both the UMAP object and the parametric networks it contains.

7.3 Plotting loss

Parametric UMAP monitors loss during training using Keras. That loss will be printed after each epoch during training. This loss is saved in `embedder.history`, and can be plotted:

```
print(embedder._history)
fig, ax = plt.subplots()
ax.plot(embedder._history['loss'])
ax.set_ylabel('Cross Entropy')
ax.set_xlabel('Epoch')
```



7.4 Parametric inverse_transform (reconstruction)

To use a second neural network to learn an inverse mapping between data and embeddings, we simply need to pass *parametric_reconstruction= True* to the ParametricUMAP.

Like the encoder, a custom decoder can also be passed to ParametricUMAP, e.g.

```
decoder = tf.keras.Sequential([
    tf.keras.layers.InputLayer(input_shape=(n_components)),
    tf.keras.layers.Dense(units=256, activation="relu"),
    tf.keras.layers.Dense(units=7 * 7 * 256, activation="relu"),
    tf.keras.layers.Reshape(target_shape=(7, 7, 256)),
    tf.keras.layers.UpSampling2D((2)),
    tf.keras.layers.Conv2D(
        filters=64, kernel_size=3, padding="same", activation="relu"
    ),
    tf.keras.layers.UpSampling2D((2)),
    tf.keras.layers.Conv2D(
        filters=32, kernel_size=3, padding="same", activation="relu"
    ),

])
```

In addition, validation data can be used to test reconstruction loss on out-of-dataset samples:

```
validation_images = test_images.reshape((test_images.shape[0], -1))/255.
```

Finally, we can pass the validation data and the networks to ParametricUMAP and train:

```
embedder = ParametricUMAP(
    encoder=encoder,
    decoder=decoder,
    dims=dims,
    parametric_reconstruction= True,
    reconstruction_validation=validation_images,
    verbose=True,
)
embedding = embedder.fit_transform(train_images)
```

7.5 Autoencoding UMAP

In the example above, the encoder is trained to minimize UMAP loss, and the decoder is trained to minimize reconstruction loss. To train the encoder jointly on both UMAP loss and reconstruction loss, pass `autoencoder_loss = True` into the ParametricUMAP.

```
embedder = ParametricUMAP(
    encoder=encoder,
    decoder=decoder,
    dims=dims,
    parametric_reconstruction= True,
    reconstruction_validation=validation_images,
    autoencoder_loss = True,
    verbose=True,
)
```

7.6 Early stopping and Keras callbacks

It can sometimes be useful to train the embedder until some plateau in training loss is met. In deep learning, early stopping is one way to do this. Keras provides custom callbacks that allow you to implement checks during training, such as early stopping. We can use callbacks, such as early stopping, with ParametricUMAP to stop training early based on a predefined training threshold, using the `keras_fit_kwargs` argument:

```
keras_fit_kwargs = {"callbacks": [
    tf.keras.callbacks.EarlyStopping(
        monitor='loss',
        min_delta=10**-2,
        patience=10,
        verbose=1,
    )
]}

embedder = ParametricUMAP(
    verbose=True,
    keras_fit_kwargs = keras_fit_kwargs,
    n_training_epochs=20
)
```

We also passed in `n_training_epochs = 20`, allowing early stopping to end training before 20 epochs are reached.

7.7 Additional important parameters

- **batch_size:** ParametricUMAP in trained over batches of edges randomly sampled from the UMAP graph, and then trained via gradient descent. ParametricUMAP defaults to a batch size of 1000 edges, but can be adjusted to a value that fits better on your GPU or CPU.
- **loss_report_frequency:** If set to 1, an epoch in in the Keras embedding refers to a single iteration over the graph computed in UMAP. Setting `loss_report_frequency` to 10, would split up that epoch into 10 seperate epochs, for more frequent reporting.
- **n_training_epochs:** The number of epochs over the UMAP graph to train for (irrespective of `loss_report_frequency`). Training the network for multiple epochs will result in better embeddings, but take longer. This parameter is different than `n_epochs` in the base UMAP class, which corresponds to the maximum number of times an edge is trained in a single ParametricUMAP epoch.
- **optimizer:** The optimizer used to train the neural network. by default Adam (`tf.keras.optimizers.Adam(1e-3)`) is used. You might be able to speed up or improve training by using a different optimizer.
- **parametric_embedding:** If set to false, a non-parametric embedding is learned, using the same code as the parametric embedding, which can serve as a direct comparison between parametric and non-parametric embedding using the same optimizer. The parametric embeddings are performed over the entire dataset simultaneously.

7.8 Extending the model

You may want to customize parametric UMAP beyond what we have implemented in this package. To make it as easy as possible to tinker around with Parametric UMAP, we made a few Jupyter notebooks that show you how to extend Parametric UMAP to your own use-cases.

- [Link coming soon] [Colab link]

7.9 Citing our work

If you use Parametric UMAP in your work, please cite our paper:

[link coming soon]

CHAPTER 8

UMAP on sparse data

Sometimes datasets get very large, and potentially very very high dimensional. In many such cases, however, the data itself is sparse – that is, while there are many many features, any given sample has only a small number of non-zero features observed. In such cases the data can be represented much more efficiently in terms of memory usage by a sparse matrix data structure. It can be hard to find dimension reduction techniques that work directly on such sparse data – often one applies a basic linear technique such as `TruncatedSVD` from sklearn (which does accept sparse matrix input) to get the data in a format amenable to other more advanced dimension reduction techniques. In the case of UMAP this is not necessary – UMAP can run directly on sparse matrix input. This tutorial will walk through a couple of examples of doing this. First we'll need some libraries loaded. We need `numpy` obviously, but we'll also make use of `scipy.sparse` which provides sparse matrix data structures. One of our examples will be purely mathematical, and we'll make use of `sympy` for that; the other example is test based and we'll use sklearn for that (specifically `sklearn.feature_extraction.text`). Beyond that we'll need umap, and plotting tools.

```
import numpy as np
import scipy.sparse
import sympy
import sklearn.datasets
import sklearn.feature_extraction.text
import umap
import umap.plot
import matplotlib.pyplot as plt
%matplotlib inline
```

8.1 A mathematical example

Our first example constructs a sparse matrix of data out of pure math. This example is inspired by the work of John Williamson, and if you haven't looked at that work you are strongly encouraged to do so. The dataset under consideration will be the integers. We will represent each integer by a vector of its divisibility by distinct primes. Thus our feature space is the space of prime numbers (less than or equal to the largest integer we will be considering) – potentially very high dimensional. In practice a given integer is divisible by only a small number of distinct primes, so each sample will be mostly made up of zeros (all the primes that the number is not divisible by), and thus we will have a very sparse dataset.

To get started we'll need a list of all the primes. Fortunately we have `sympy` at our disposal and we can quickly get that information with a single call to `primerange`. We'll also need a dictionary mapping the different primes to the column number they correspond to in our data structure; effectively we'll just be enumerating the primes.

```
primes = list(sympy.primerange(2, 110000))
prime_to_column = {p:i for i, p in enumerate(primes)}
```

Now we need to construct our data in a format we can put into a sparse matrix easily. At this point a little background on sparse matrix data structures is useful. For this purpose we'll be using the so called "LIL" format. LIL is short for "List of Lists", since that is how the data is internally stored. There is a list of all the rows, and each row is stored as a list giving the column indices of the non-zero entries. To store the data values there is a parallel structure containing the value of the entry corresponding to a given row and column.

To put the data together in this sort of format we need to construct such a list of lists. We can do that by iterating over all the integers up to a fixed bound, and for each integer (i.e. each row in our dataset) generating the list of column indices which will be non-zero. The column indices will simply be the indices corresponding to the primes that divide the number. Since `sympy` has a function `primefactors` which returns a list of the unique prime factors of any integer we simply need to map those through our dictionary to covert the primes into column numbers.

Parallel to that we'll construct the corresponding structure of values to insert into a matrix. Since we are only concerned with divisibility this will simply be a one in every non-zero entry, so we can just add a list of ones of the appropriate length for each row.

```
%%time
lil_matrix_rows = []
lil_matrix_data = []
for n in range(100000):
    prime_factors = sympy.primefactors(n)
    lil_matrix_rows.append([prime_to_column[p] for p in prime_factors])
    lil_matrix_data.append([1] * len(prime_factors))
```

```
CPU times: user 2.07 s, sys: 26.4 ms, total: 2.1 s
Wall time: 2.1 s
```

Now we need to get that into a sparse matrix. Fortunately the `scipy.sparse` package makes this easy, and we've already built the data in a fairly useful structure. First we create a sparse matrix of the correct format (LIL) and the right shape (as many rows as we have generated, and as many columns as there are primes). This is essentially just an empty matrix however. We can fix that by setting the `rows` attribute to be the rows we have generated, and the `data` attribute to be the corresponding structure of values (all ones). The result is a sparse matrix data structure which can then be easily manipulated and converted into other sparse matrix formats easily.

```
factor_matrix = scipy.sparse.lil_matrix((len(lil_matrix_rows), len(primes)), dtype=np.
↪float32)
factor_matrix.rows = np.array(lil_matrix_rows)
factor_matrix.data = np.array(lil_matrix_data)
factor_matrix
```

```
<100000x10453 sparse matrix of type '<class 'numpy.float32'>'
    with 266398 stored elements in LInked List format>
```

As you can see we have a matrix with 100000 rows and over 10000 columns. If we were storing that as a numpy array it would take a great deal of memory. In practice, however, there are only 260000 or so entries that are not zero, and that's all we really need to store, making it much more compact.

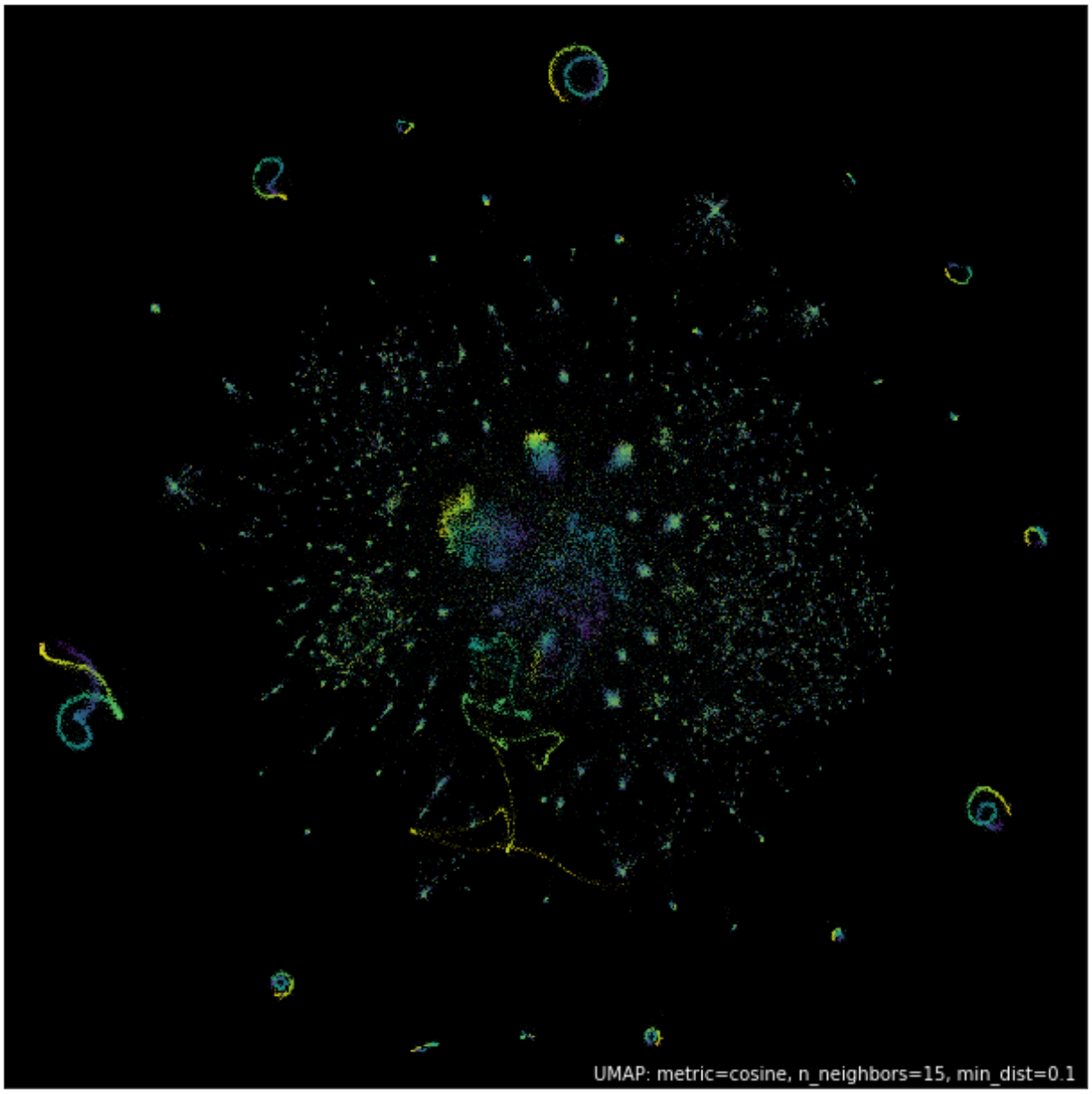
The question now is how can we feed that sparse matrix structure into UMAP to have it learn an embedding. The answer is surprisingly straightforward – we just hand it directly to the fit method. Just like other sklearn estimators that can handle sparse input UMAP will detect the sparse matrix and just do the right thing.

```
%%time
mapper = umap.UMAP(metric='cosine', random_state=42, low_memory=True).fit(factor_
↪matrix)
```

```
CPU times: user 9min 36s, sys: 6.76 s, total: 9min 43s
Wall time: 9min 7s
```

That was easy! But is it really working? We can easily plot the results:

```
umap.plot.points(mapper, values=np.arange(100000), theme='viridis')
```



And this looks very much in line with the results John Williamson got with the proviso that we only used 100,000 integers instead of 1,000,000 to ensure that most users should be able to run this example (the full million may require a large memory compute node). So it seems like this is working well. The next question is whether we can use the

`transform` functionality to map new data into this space. To test that we'll need some more data. Fortunately there are more integers. We'll grab the next 10,000 and put them in a sparse matrix, much as we did for the first 100,000.

```
%%time
lil_matrix_rows = []
lil_matrix_data = []
for n in range(100000, 110000):
    prime_factors = sympy.primefactors(n)
    lil_matrix_rows.append([prime_to_column[p] for p in prime_factors])
    lil_matrix_data.append([1] * len(prime_factors))
```

```
CPU times: user 214 ms, sys: 1.99 ms, total: 216 ms
Wall time: 222 ms
```

```
new_data = scipy.sparse.lil_matrix((len(lil_matrix_rows), len(primes)), dtype=np.
↪float32)
new_data.rows = np.array(lil_matrix_rows)
new_data.data = np.array(lil_matrix_data)
new_data
```

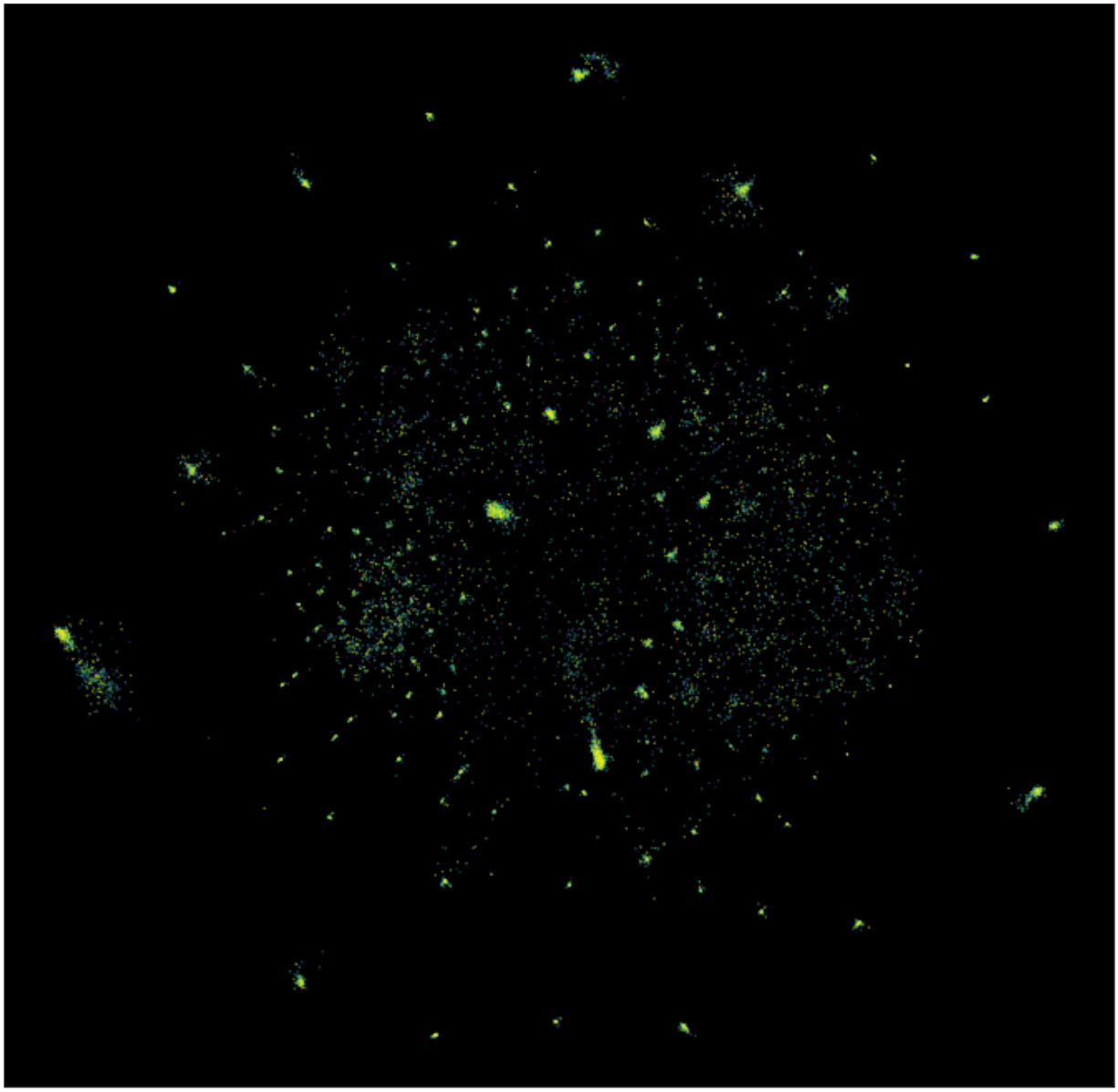
```
<10000x10453 sparse matrix of type '<class 'numpy.float32'>'
    with 27592 stored elements in LInked List format>
```

To map the new data we generated we can simply hand it to the `transform` method of our trained model. This is a little slow, but it does work.

```
new_data_embedding = mapper.transform(new_data)
```

And we can plot the results. Since we just got the locations of the points this time (rather than a model) we'll have to resort to matplotlib for plotting.

```
fig = plt.figure(figsize=(12,12))
ax = fig.add_subplot(111)
plt.scatter(new_data_embedding[:, 0], new_data_embedding[:, 1], s=0.1, c=np.
↪arange(10000), cmap='viridis')
ax.set(xticks=[], yticks=[], facecolor='black');
```



The color scale is different in this case, but you can see that the data has been mapped into locations corresponding to the various structures seen in the original embedding. Thus, even with large sparse data we can embed the data, and even add new data to the embedding.

8.2 A text analysis example

Let's look at a more classical machine learning example of working with sparse high dimensional data – working with text documents. Machine learning on text is hard, and there is a great deal of literature on the subject, but for now we'll just consider a basic approach. Part of the difficulty of machine learning with text is turning language into numbers, since numbers are really all most machine learning algorithms understand (at heart anyway). One of the most straightforward ways to do this for documents is what is known as the "bag-of-words" model. In this model we view a document as simply a multi-set of the words contained in it – we completely ignore word order. The result can be viewed as a matrix of data by setting the feature space to be the set of all words that appear in any document, and a

document is represented by a vector where the value of the *i*th entry is the number of times the *i*th word occurs in that document. This is a very common approach, and is what you will get if you apply sklearn's `CountVectorizer` to a text dataset for example. The catch with this approach is that the feature space is often *very* large, since we have a feature for each and every word that ever occurs in the entire corpus of documents. The data is sparse however, since most documents only use a small portion of the total possible vocabulary. Thus the default output format of `CountVectorizer` (and other similar feature extraction tools in sklearn) is a `scipy.sparse` format matrix.

For this example we'll make use of the classic 20newsgroups dataset, a sampling of newsgroup messages from the old NNTP newsgroup system covering 20 different newsgroups. The `sklearn.datasets` module can easily fetch the data, and, in fact, we can fetch a pre-vectorized version to save us the trouble of running `CountVectorizer` ourselves. We'll grab both the training set, and the test set for later use.

```
news_train = sklearn.datasets.fetch_20newsgroups_vectorized(subset='train')
news_test = sklearn.datasets.fetch_20newsgroups_vectorized(subset='test')
```

If we look at the actual data we have pulled back, we'll see that sklearn has run a `CountVectorizer` and produced the data in the sparse matrix format.

```
news_train.data
```

```
<11314x130107 sparse matrix of type '<class 'numpy.float64'>'
    with 1787565 stored elements in Compressed Sparse Row format>
```

The value of the sparse matrix format is immediately obvious in this case; while there are only 11,000 samples there are 130,000 features! If the data were stored in a standard `numpy` array we would be using up 10GB of memory! And most of that memory would simply be storing the number zero, over and over again. In the sparse matrix format it easily fits in memory on most machines. This sort of dimensionality of data is very common with text workloads.

The raw counts are, however, not ideal since common words such as "the" and "and" will dominate the counts for most documents, while contributing very little information about the actual content of the document. We can correct for this by using a `TfidfTransformer` from sklearn, which will convert the data into TF-IDF format. There are lots of ways to think about the transformation done by TF-IDF, but I like to think of it intuitively as follows. The information content of a word can be thought of as (roughly) proportional to the negative log of the frequency of the word; the more often a word is used, the less information it tends to carry, and infrequent words carry more information. What TF-IDF is going to do can be thought of as akin to re-weighting the columns according to the information content of the word associated to that column. Thus the common words like "the" and "and" will get down-weighted, as carrying less information about the document, while infrequent words will be deemed more imporant and have their associated columns up-weighted. We can apply this transformation to both the train and test sets (using the same transformer trained on the training set).

```
tfidf = sklearn.feature_extraction.text.TfidfTransformer(norm='l1').fit(news_train.
↪data)
train_data = tfidf.transform(news_train.data)
test_data = tfidf.transform(news_test.data)
```

The result is still a sparse matrix, since TF-IDF doesn't change the zero elements at all, nor the number of features.

```
train_data
```

```
<11314x130107 sparse matrix of type '<class 'numpy.float64'>'
    with 1787565 stored elements in Compressed Sparse Row format>
```

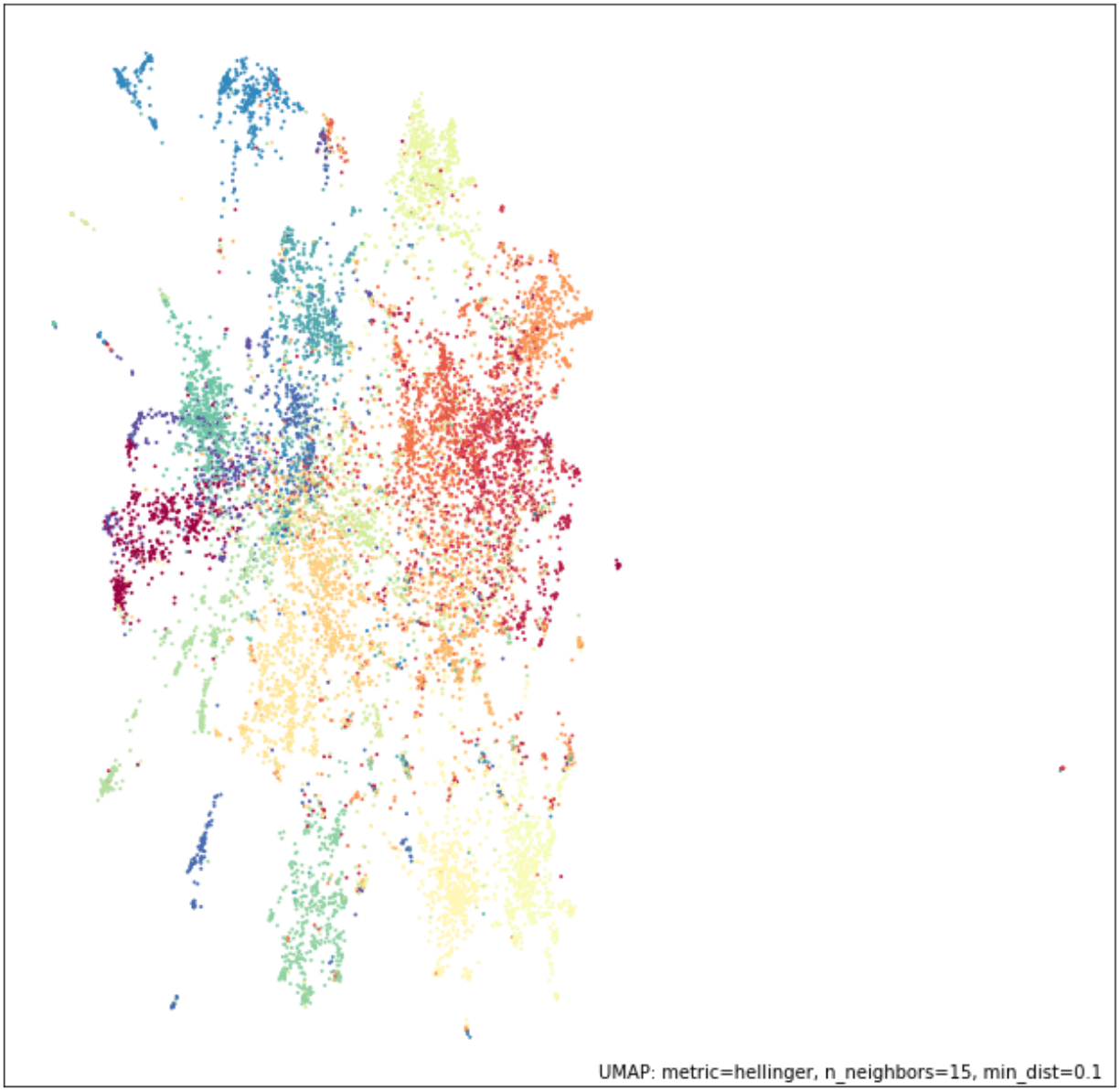
Now we need to pass this very high dimensional data to UMAP. Unlike some other non-linear dimension reduction techniques we don't need to apply PCA first to get the data down to a reasonable dimensionality; nor do we need to use other techniques to reduce the data to be able to be represented as a dense `numpy` array; we can work directly on the 130,000 dimensional sparse matrix.

```
%%time
mapper = umap.UMAP(metric='hellinger', random_state=42).fit(train_data)
```

```
CPU times: user 8min 40s, sys: 3.07 s, total: 8min 44s
Wall time: 8min 43s
```

Now we can plot the results, with labels according to the target variable of the data – which newsgroup the posting was drawn from.

```
umap.plot.points(mapper, labels=news_train.target)
```



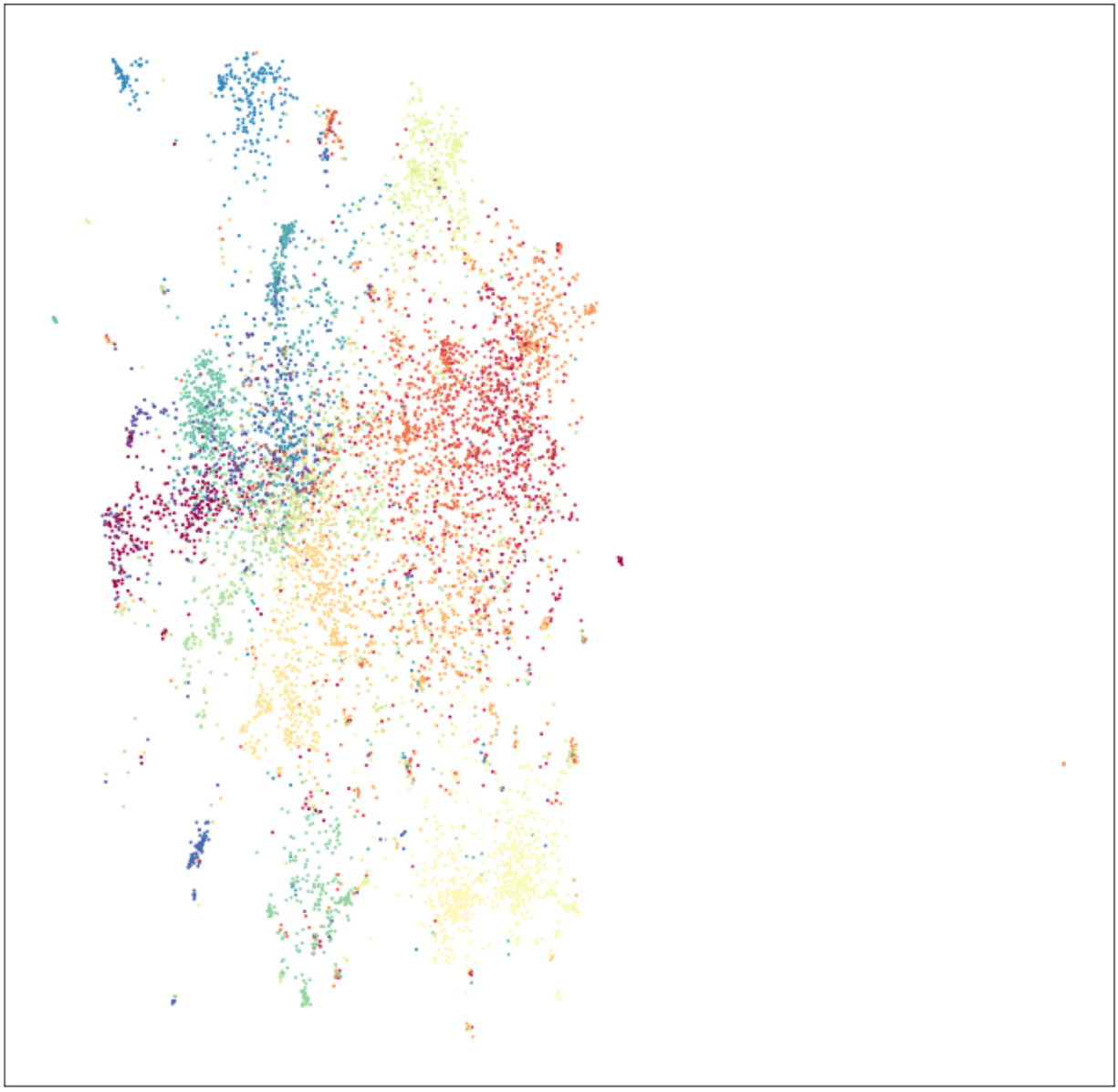
We can see that even going directly from a 130,000 dimensional space down to only 2 dimensions UMAP has done a decent job of separating out many of the different newsgroups.

We can now attempt to add the test data to the same space using the `transform` method.

```
test_embedding = mapper.transform(test_data)
```

While this is somewhat expensive computationally, it does work, and we can plot the end result:

```
fig = plt.figure(figsize=(12,12))
ax = fig.add_subplot(111)
plt.scatter(test_embedding[:, 0], test_embedding[:, 1], s=1, c=news_test.target, cmap=
↪'Spectral')
ax.set(xticks=[], yticks=[]);
```



CHAPTER 9

UMAP for Supervised Dimension Reduction and Metric Learning

While UMAP can be used for standard unsupervised dimension reduction the algorithm offers significant flexibility allowing it to be extended to perform other tasks, including making use of categorical label information to do supervised dimension reduction, and even metric learning. We'll look at some examples of how to do that below.

First we will need to load some base libraries – `numpy`, obviously, but also `mnist` to read in the Fashion-MNIST data, and matplotlib and seaborn for plotting.

```
import numpy as np
from mnist.loader import MNIST
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
sns.set(style='white', context='poster')
```

Our example dataset for this exploration will be the Fashion-MNIST dataset from Zalando Research. It is designed to be a drop-in replacement for the classic MNIST digits dataset, but uses images of fashion items (dresses, coats, shoes, bags, etc.) instead of handwritten digits. Since the images are more complex it provides a greater challenge than MNIST digits. We can load it in (after downloading the dataset) using the mnist library. We can then package up the train and test sets into one large dataset, normalise the values (to be in the range [0,1]), and set up labels for the 10 classes.

```
mndata = MNIST('fashion-mnist/data/fashion')
train, train_labels = mndata.load_training()
test, test_labels = mndata.load_testing()
data = np.array(np.vstack([train, test]), dtype=np.float64) / 255.0
target = np.hstack([train_labels, test_labels])
classes = [
    'T-shirt/top',
    'Trouser',
    'Pullover',
    'Dress',
    'Coat',
    'Sandal',
    'Shirt',
```

(continues on next page)

(continued from previous page)

```
    'Sneaker',
    'Bag',
    'Ankle boot']
```

Next we'll load the `umap` library so we can perform dimension reduction on this dataset.

```
import umap
```

9.1 UMAP on Fashion MNIST

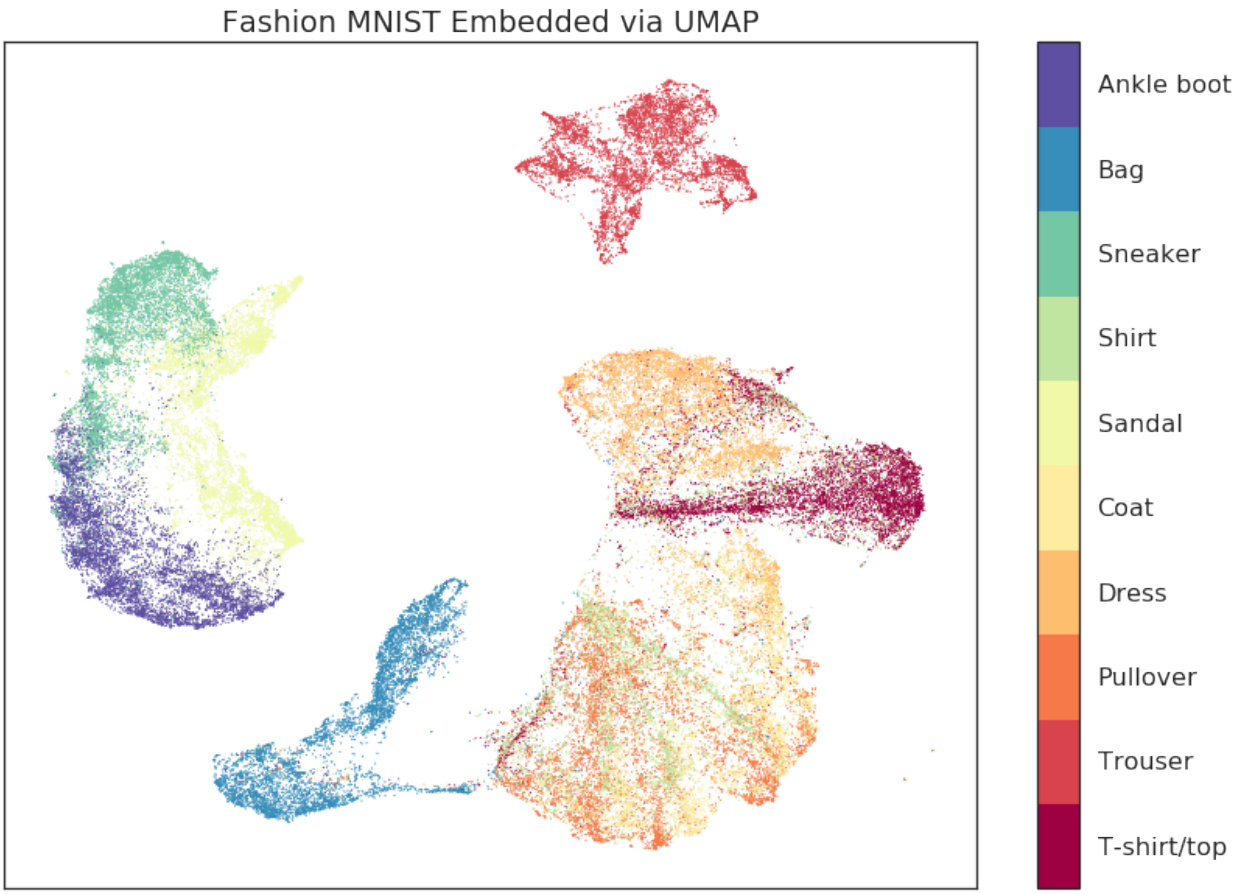
First we'll just do standard unsupervised dimension reduction using UMAP so we have a baseline of what the results look like for later comparison. This is simply a matter of instantiating a *UMAP* object (in this case setting the `n_neighbors` parameter to be 5 – we are interested mostly in very local information), then calling the *fit_transform()* method with the data we wish to reduce. By default UMAP reduces to two dimensions, so we'll be able to view the results as a scatterplot.

```
%%time
embedding = umap.UMAP(n_neighbors=5).fit_transform(data)
```

```
CPU times: user 1min 45s, sys: 7.22 s, total: 1min 52s
Wall time: 1min 26s
```

That took a little time, but not all that long considering it is 70,000 data points in 784 dimensional space. We can simply plot the results as a scatterplot, colored by the class of the fashion item. We can use matplotlib's colorbar with suitable tick-labels to give us the color key.

```
fig, ax = plt.subplots(1, figsize=(14, 10))
plt.scatter(*embedding.T, s=0.3, c=target, cmap='Spectral', alpha=1.0)
plt.setp(ax, xticks=[], yticks=[])
cbar = plt.colorbar(boundaries=np.arange(11)-0.5)
cbar.set_ticks(np.arange(10))
cbar.set_ticklabels(classes)
plt.title('Fashion MNIST Embedded via UMAP');
```



The result is fairly good. We successfully separated a number of the classes, and the global structure (separating pants and footwear from shirts, coats and dresses) is well preserved as well. Unlike results for MNIST digits, however, there were a number of classes that did not separate quite so cleanly. In particular T-shirts, shirts, dresses, pullovers, and coats are all a little mixed. At the very least the dresses are largely separated, and the T-shirts are mostly in one large clump, but they are not well distinguished from the others. Worse still are the coats, shirts, and pullovers (somewhat unsurprisingly as these can certainly look very similar) which all have significant overlap with one another. Ideally we would like much better class separation. Since we have the label information we can actually give that to UMAP to use!

9.2 Using Labels to Separate Classes (Supervised UMAP)

How do we go about coercing UMAP to make use of target labels? If you are familiar with the sklearn API you'll know that the *fit()* method takes a target parameter `y` that specifies supervised target information (for example when training a supervised classification model). We can simply pass the *UMAP* model that target data when fitting and it will make use of it to perform supervised dimension reduction!

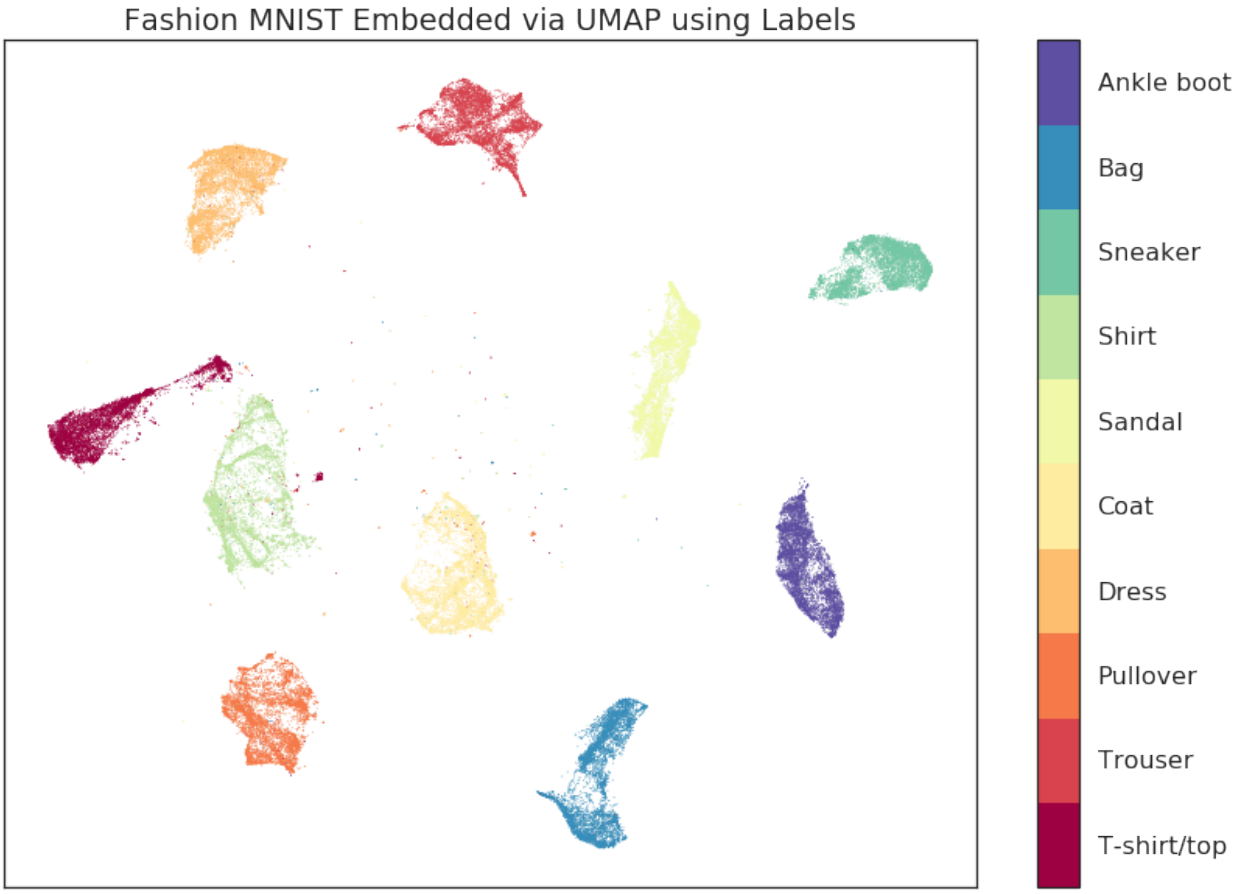
```
%%time
embedding = umap.UMAP().fit_transform(data, y=target)
```

```
CPU times: user 3min 28s, sys: 9.17 s, total: 3min 37s
Wall time: 2min 45s
```

This took a little longer – both because we are using a larger `n_neighbors` value (which is suggested when doing supervised dimension reduction; here we are using the default value of 15), and because we need to condition on the

label data. As before we have reduced the data down to two dimensions so we can again visualize the data with a scatterplot, coloring by class.

```
fig, ax = plt.subplots(1, figsize=(14, 10))
plt.scatter(*embedding.T, s=0.1, c=target, cmap='Spectral', alpha=1.0)
plt.setp(ax, xticks=[], yticks=[])
cbar = plt.colorbar(boundaries=np.arange(11)-0.5)
cbar.set_ticks(np.arange(10))
cbar.set_ticklabels(classes)
plt.title('Fashion MNIST Embedded via UMAP using Labels');
```



The result is a cleanly separated set of classes (and a little bit of stray noise – points that were sufficiently different from their class as to not be grouped with the rest). Aside from the clear class separation however (which is expected – we gave the algorithm all the class information), there are a couple of important points to note. The first point to note is that we have retained the internal structure of the individual classes. Both the shirts and pullovers still have the distinct banding pattern that was visible in the original unsupervised case; the pants, t-shirts and bags both retained their shape and internal structure; etc. The second point to note is that we have also retained the global structure. While the individual classes have been cleanly separated from one another, the inter-relationships among the classes have been preserved: footwear classes are all near one another; trousers and bags are at opposite sides of the plot; and the arc of pullover, shirts, t-shirts and dresses is still in place.

The key point is this: the important structural properties of the data have been retained while the known classes have been cleanly pulled apart and isolated. If you have data with known classes and want to separate them while still having a meaningful embedding of individual points then supervised UMAP can provide exactly what you need.

9.3 Using Partial Labelling (Semi-Supervised UMAP)

What if we only have some of our data labelled, however, and a number of items are without labels. Can we still make use of the label information we do have? This is now a semi-supervised learning problem, and yes, we can work with those cases too. To set up the example we'll mask some of the target information – we'll do this by using the sklearn standard of giving unlabelled points a label of -1 (such as, for example, the noise points from a DBSCAN clustering).

```
masked_target = target.copy().astype(np.int8)
masked_target[np.random.choice(70000, size=10000, replace=False)] = -1
```

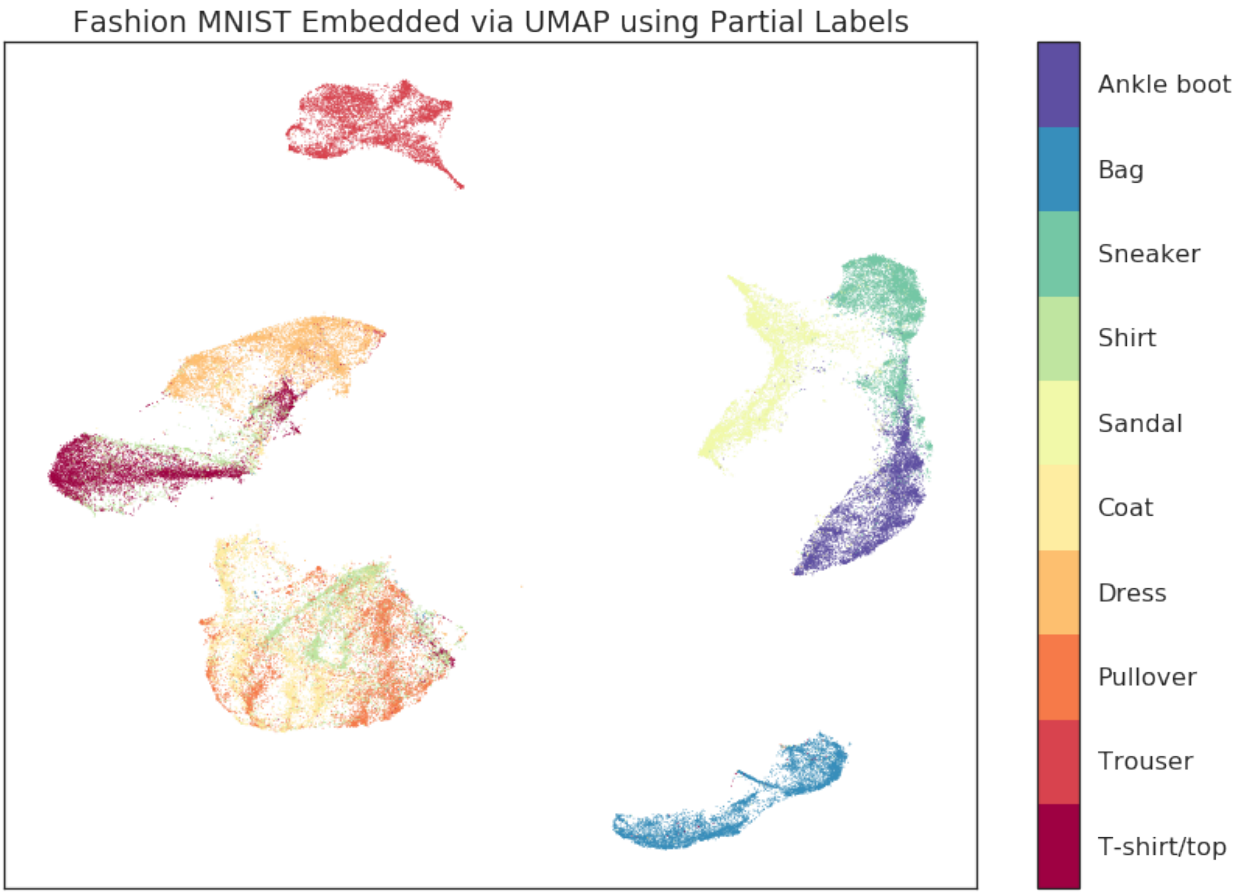
Now that we have randomly masked some of the labels we can try to perform supervised learning again. Everything works as before, but UMAP will interpret the -1 label as being an unlabelled point and learn accordingly.

```
%%time
fitter = umap.UMAP().fit(data, y=masked_target)
embedding = fitter.embedding_
```

```
CPU times: user 3min 8s, sys: 7.85 s, total: 3min 16s
Wall time: 2min 40s
```

Again we can look at a scatterplot of the data colored by class.

```
fig, ax = plt.subplots(1, figsize=(14, 10))
plt.scatter(*embedding.T, s=0.1, c=target, cmap='Spectral', alpha=1.0)
plt.setp(ax, xticks=[], yticks=[])
cbar = plt.colorbar(boundaries=np.arange(11)-0.5)
cbar.set_ticks(np.arange(10))
cbar.set_ticklabels(classes)
plt.title('Fashion MNIST Embedded via UMAP using Partial Labels');
```



The result is much as we would expect – while we haven't cleanly separated the data as we did in the totally supervised case, the classes have been made cleaner and more distinct. This semi-supervised approach provides a powerful tool when labelling is potentially expensive, or when you have more data than labels, but want to make use of that extra data.

9.4 Training with Labels and Embedding Unlabelled Test Data (Metric Learning with UMAP)

If we have learned a supervised embedding, can we use that to embed new previously unseen (and now unlabelled) points into the space? This would provide an algorithm for metric learning, where we can use a labelled set of points to learn a metric on data, and then use that learned metric as a measure of distance between new unlabelled points. This can be particularly useful as part of a machine learning pipline where we learn a supervised embedding as a form of supervised feature engineering, and then build a classifier on that new space – this is viable as long as we can pass new data to the embedding model to be transformed to the new space.

To try this out with UMAP let's use the train/test split provided by Fashion MNIST:

```
train_data = np.array(train)
test_data = np.array(test)
```

Now we can fit a model to the training data, making use of the training labels to learn a supervised embedding.

```
%%time
mapper = umap.UMAP(n_neighbors=10).fit(train_data, np.array(train_labels))
```

```
CPU times: user 2min 18s, sys: 7.53 s, total: 2min 26s
Wall time: 1min 52s
```

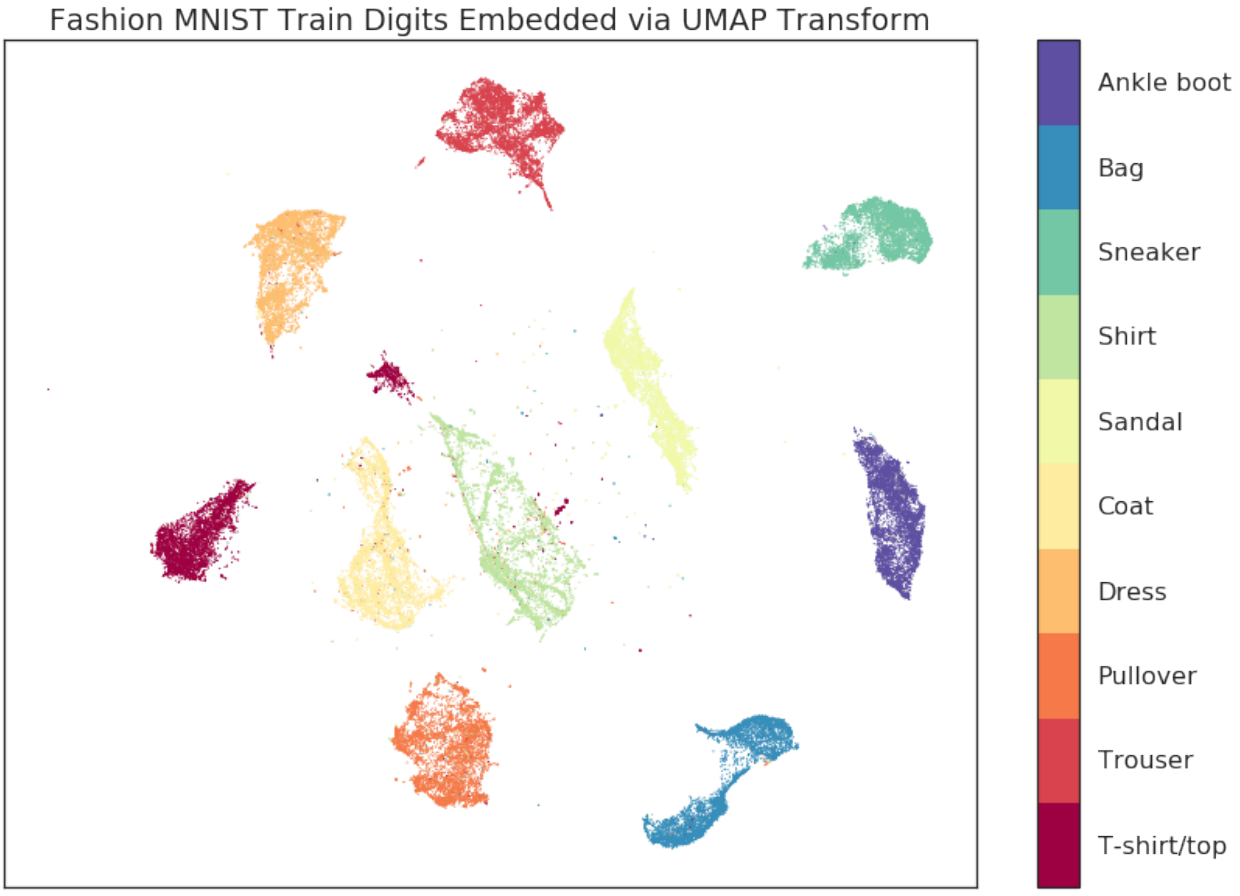
Next we can use the *transform()* method on that model to transform the test set into the learned space. This time we won't pass the label information and let the model attempt to place the data correctly.

```
%%time
test_embedding = mapper.transform(test_data)
```

```
CPU times: user 17.3 s, sys: 986 ms, total: 18.3 s
Wall time: 15.4 s
```

UMAP transforms are not as fast as some approaches, but as you can see this was still fairly efficient. The important question is how well we managed to embed the test data into the existing learned space. To start let's visualise the embedding of the training data so we can get a sense of where things *should* go.

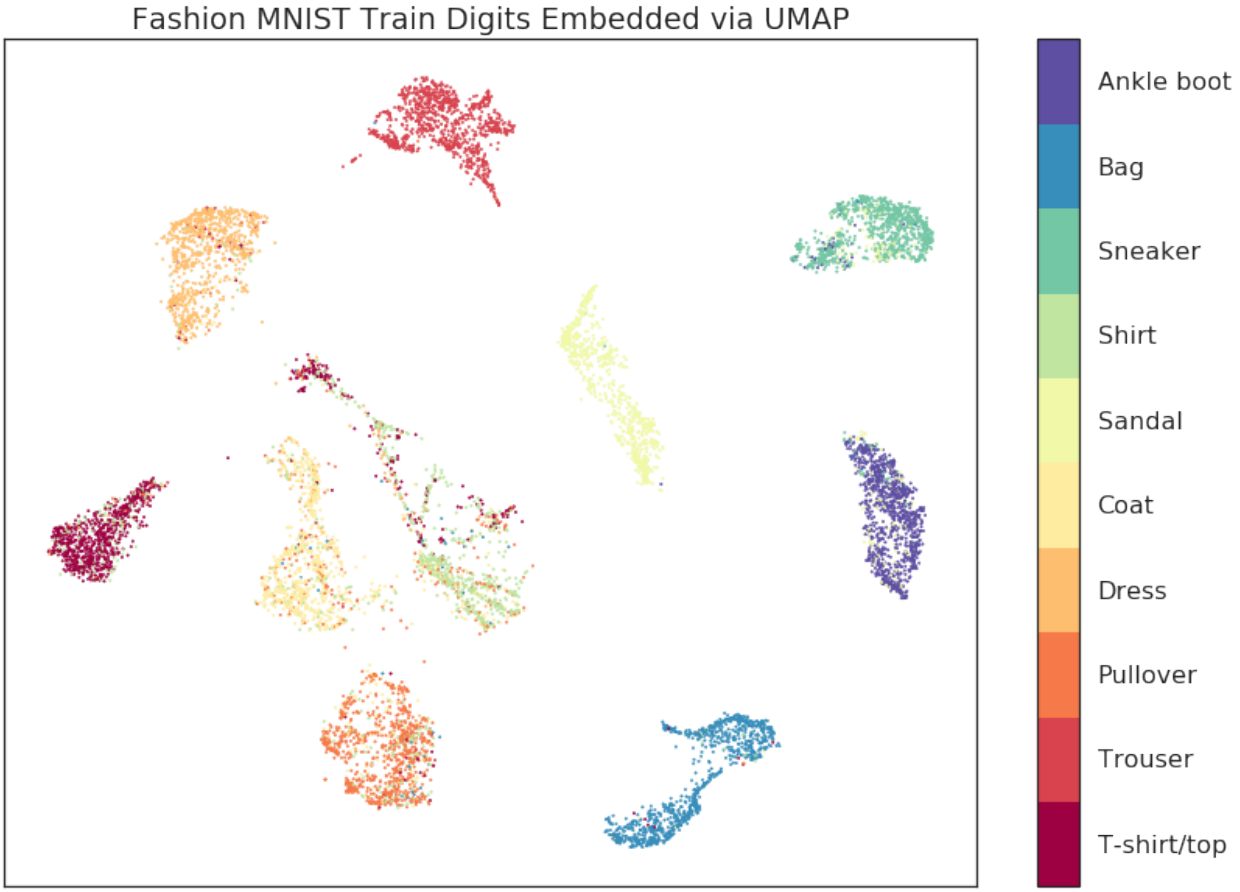
```
fig, ax = plt.subplots(1, figsize=(14, 10))
plt.scatter(*mapper.embedding_.T, s=0.3, c=np.array(train_labels), cmap='Spectral',␣
↪alpha=1.0)
plt.setp(ax, xticks=[], yticks=[])
cbar = plt.colorbar(boundaries=np.arange(11)-0.5)
cbar.set_ticks(np.arange(10))
cbar.set_ticklabels(classes)
plt.title('Fashion MNIST Train Digits Embedded via UMAP Transform');
```



As you can see this has done a similar job as before, successfully embedding the separate classes while retaining both

the internal structure and the overall global structure. We can now look at how the test set, for which we provided no label information, was embedded via the *transform()* method.

```
fig, ax = plt.subplots(1, figsize=(14, 10))
plt.scatter(*test_embedding.T, s=2, c=np.array(test_labels), cmap='Spectral', alpha=1.
↪0)
plt.setp(ax, xticks=[], yticks=[])
cbar = plt.colorbar(boundaries=np.arange(11)-0.5)
cbar.set_ticks(np.arange(10))
cbar.set_ticklabels(classes)
plt.title('Fashion MNIST Train Digits Embedded via UMAP');
```



As you can see we have replicated the layout of the training data, including much of the internal structure of the classes. For the most part assignment of new points follows the classes well. The greatest source of confusion are some t-shirts that ended up mixed with the shirts, and some pullovers which are confused with the coats. Given the difficulty of the problem this is a good result, particularly when compared with current state-of-the-art approaches such as siamese and triplet networks.

CHAPTER 10

Using UMAP for Clustering

UMAP can be used as an effective preprocessing step to boost the performance of density based clustering. This is somewhat controversial, and should be attempted with care. For a good discussion of some of the issues involved in this, please see the various answers in this stackoverflow thread on clustering the results of t-SNE. Many of the points of concern raised there are salient for clustering the results of UMAP. The most notable is that UMAP, like t-SNE, does not completely preserve density. UMAP, like t-SNE, can also create false tears in clusters, resulting in a finer clustering than is necessarily present in the data. Despite these concerns there are still valid reasons to use UMAP as a preprocessing step for clustering. As with any clustering approach one will want to do some exploration and evaluation of the clusters that come out to try to validate them if possible.

With all of that said, let's work through an example to demonstrate the difficulties that can face clustering approaches and how UMAP can provide a powerful tool to help overcome them.

First we'll need a selection of libraries loaded up. Obviously we'll need data, and we can use sklearn's `fetch_openml` to get it. We'll also need the usual tools of numpy, and plotting. Next we'll need umap, and some clustering options. Finally, since we'll be working with labeled data, we can make use of strong cluster evaluation metrics Adjusted Rand Index and Adjusted Mutual Information.

```
from sklearn.datasets import fetch_openml
from sklearn.decomposition import PCA
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

# Dimension reduction and clustering libraries
import umap
import hdbscan
import sklearn.cluster as cluster
from sklearn.metrics import adjusted_rand_score, adjusted_mutual_info_score
```

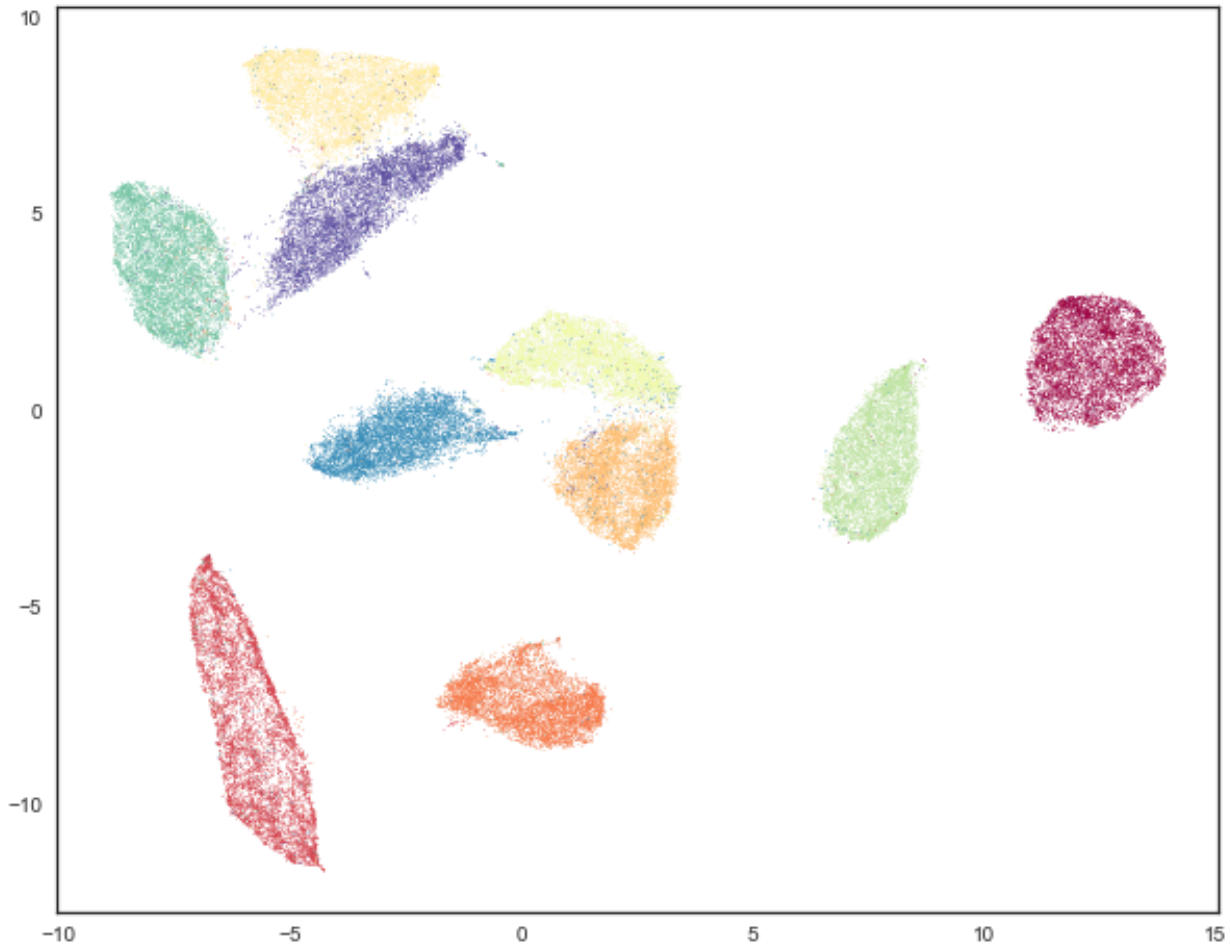
Now let's set up the plotting and grab the data we'll be using – in this case the MNIST handwritten digits dataset. MNIST consists of 28x28 pixel grayscale images of handwritten digits (0 through 9). These can be unraveled such that each digit is described by a 784 dimensional vector (the gray scale value of each pixel in the image). Ideally we would like the clustering to recover the digit structure.

```
sns.set(style='white', rc={'figure.figsize':(10,8)})
```

```
mnist = fetch_openml('Fashion-MNIST', version=1)
mnist.target = mnist.target.astype(int)
```

For visualization purposes we can reduce the data to 2-dimensions using UMAP. When we cluster the data in high dimensions we can visualize the result of that clustering. First, however, we'll view the data colored by the digit that each data point represents – we'll use a different color for each digit. This will help frame what follows.

```
standard_embedding = umap.UMAP(random_state=42).fit_transform(mnist.data)
plt.scatter(standard_embedding[:, 0], standard_embedding[:, 1], c=mnist.target.
↪astype(int), s=0.1, cmap='Spectral');
```



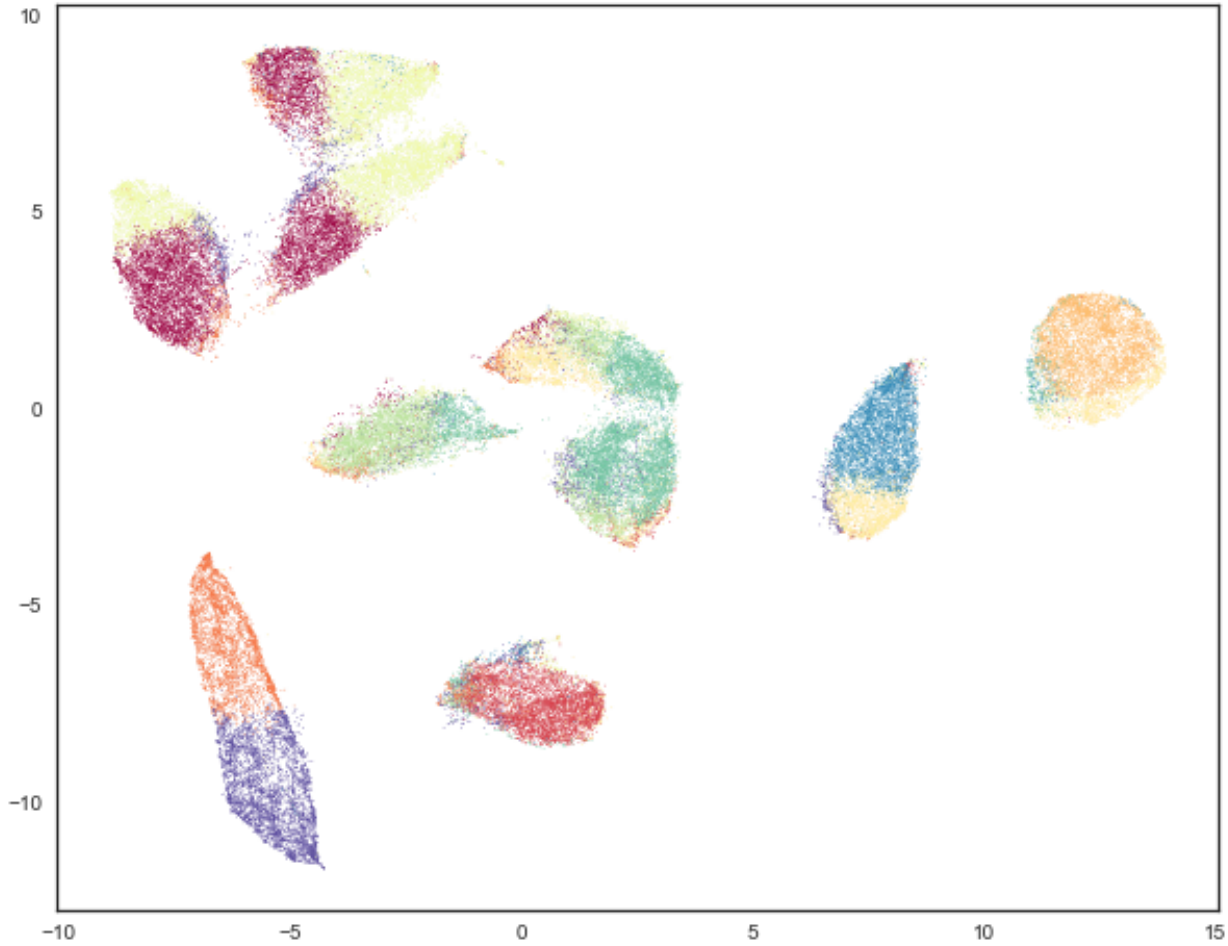
10.1 Traditional clustering

Now we would like to cluster the data. As a first attempt let's try the traditional approach: K-Means. In this case we can solve one of the hard problems for K-Means clustering – choosing the right k value, giving the number of clusters we are looking for. In this case we know the answer is exactly 10. We will use sklearns K-Means implementation looking for 10 clusters in the original 784 dimensional data.

```
kmeans_labels = cluster.KMeans(n_clusters=10).fit_predict(mnist.data)
```

And how did the clustering do? We can look at the results by coloring out UMAP embedded data by cluster membership.

```
plt.scatter(standard_embedding[:, 0], standard_embedding[:, 1], c=kmeans_labels, s=0.
↪1, cmap='Spectral');
```



This is not really the result we were looking for (though it does expose interesting properties of how K-Means chooses clusters in high dimensional space, and how UMAP unwraps manifolds by finding manifold boundaries). While K-Means gets some cases correct, such as the two clusters on the right side which are mostly correct, most of the rest of the data looks somewhat arbitrarily carved up among the remaining clusters. We can put this impression to the test by evaluating the adjusted Rand score and adjusted mutual information for this clustering as compared with the true labels.

```
(
    adjusted_rand_score(mnist.target, kmeans_labels),
    adjusted_mutual_info_score(mnist.target, kmeans_labels)
)
```

```
(0.36675295135972552, 0.49614118437750965)
```

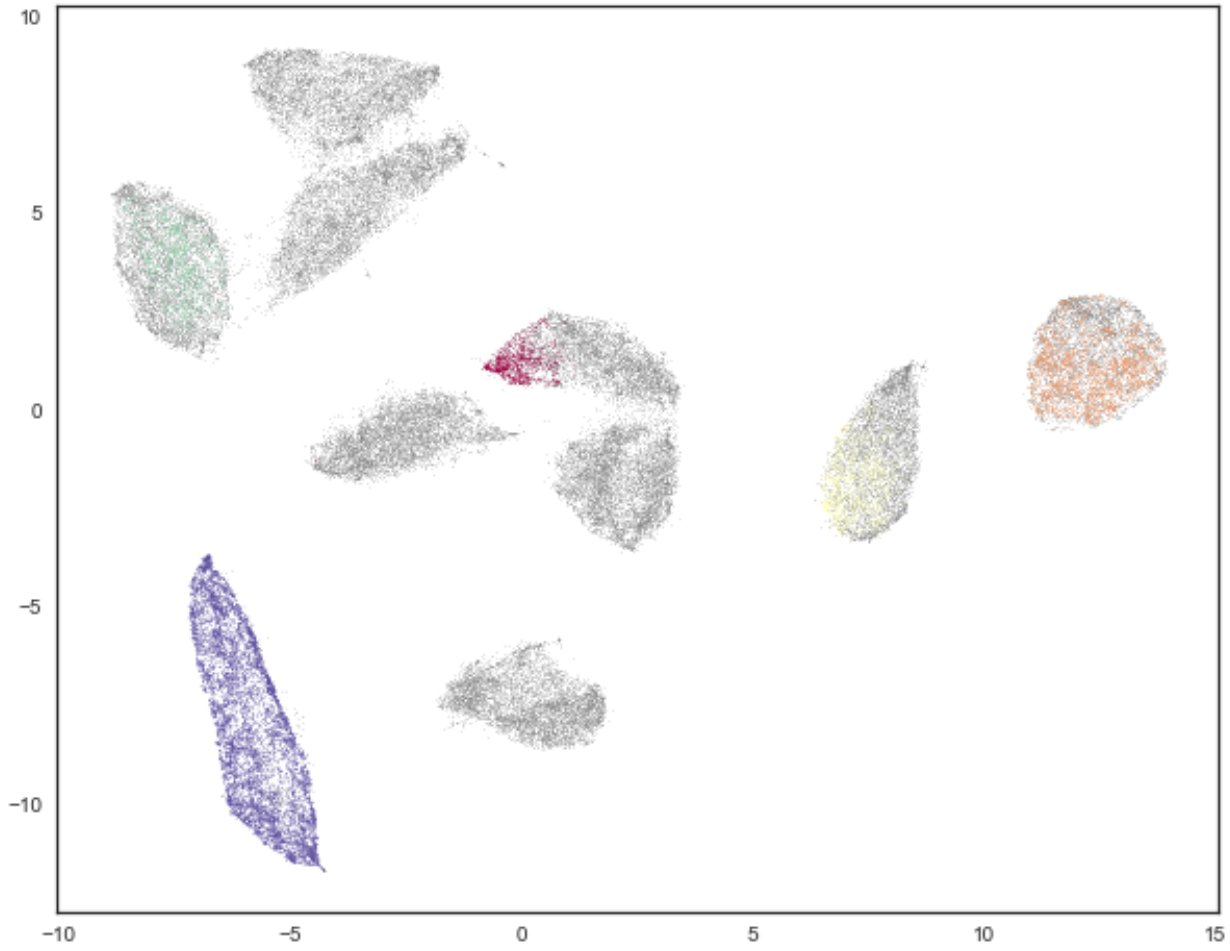
As might be expected, we have not done a particularly good job – both scores take values in the range 0 to 1, with 0

representing a bad (essentially random) clustering and 1 representing perfectly recovering the true labels. K-Means definitely was not random, but it was also quite a long way from perfectly recovering the true labels. Part of the problem is the way K-Means works, based on centroids with an assumption of largely spherical clusters – this is responsible for some of the sharp divides that K-Means puts across digit classes. We can potentially improve on this by using a smarter density based algorithm. In this case we've chosen to try HDBSCAN, which we believe to be among the most advanced density based techniques. For the sake of performance we'll reduce the dimensionality of the data down to 50 dimensions via PCA (this recovers most of the variance), since HDBSCAN scales somewhat poorly with the dimensionality of the data it will work on.

```
lowd_mnist = PCA(n_components=50).fit_transform(mnist.data)
hdbscan_labels = hdbscan.HDBSCAN(min_samples=10, min_cluster_size=500).fit_
↪predict(lowd_mnist)
```

We can now inspect the results. Before we do, however, it should be noted that one of the features of HDBSCAN is that it can refuse to cluster some points and classify them as "noise". To visualize this aspect we will color points that were classified as noise gray, and then color the remaining points according to the cluster membership.

```
clustered = (hdbscan_labels >= 0)
plt.scatter(standard_embedding[~clustered, 0],
            standard_embedding[~clustered, 1],
            c=(0.5, 0.5, 0.5),
            s=0.1,
            alpha=0.5)
plt.scatter(standard_embedding[clustered, 0],
            standard_embedding[clustered, 1],
            c=hdbscan_labels[clustered],
            s=0.1,
            cmap='Spectral');
```



This looks somewhat underwhelming. It meets HDBSCAN's approach of "not being wrong" by simply refusing to classify the majority of the data. The result is a clustering that almost certainly fails to recover all the labels. We can verify this by looking at the clustering validation scores.

```
(
    adjusted_rand_score(mnist.target, hdbscan_labels),
    adjusted_mutual_info_score(mnist.target, hdbscan_labels)
)
```

```
(0.053830107882840102, 0.19756104096566332)
```

These scores are far worse than K-Means! Partially this is due to the fact that these scores assume that the noise points are simply an extra cluster. We can instead only look at the subset of the data that HDBSCAN was actually confident enough to assign to clusters – a simple sub-selection will let us recompute the scores for only that data.

```
clustered = (hdbscan_labels >= 0)
(
    adjusted_rand_score(mnist.target[clustered], hdbscan_labels[clustered]),
    adjusted_mutual_info_score(mnist.target[clustered], hdbscan_labels[clustered])
)
```

```
(0.99843407988303912, 0.99405521087764015)
```

And here we see that where HDBSCAN was willing to cluster it got things almost entirely correct. This is what it was

designed to do – be right for what it can, and defer on anything that it couldn't have sufficient confidence in. Of course the catch here is that it deferred clustering a lot of the data. How much of the data did HDBSCAN actually assign to clusters? We can compute that easily enough.

```
np.sum(clustered) / mnist.data.shape[0]
```

```
0.17081428571428572
```

It seems that less than 18% of the data was clustered. While HDBSCAN did a great job on the data it could cluster it did a poor job of actually managing to cluster the data. The problem here is that, as a density based clustering algorithm, HDBSCAN tends to suffer from the curse of dimensionality: high dimensional data requires more observed samples to produce much density. If we could reduce the dimensionality of the data more we would make the density more evident and make it far easier for HDBSCAN to cluster the data. The problem is that trying to use PCA to do this is going to become problematic. While reducing the 50 dimensions still explained a lot of the variance of the data, reducing further is going to quickly do a lot worse. This is due to the linear nature of PCA. What we need is strong manifold learning, and this is where UMAP can come into play.

10.2 UMAP enhanced clustering

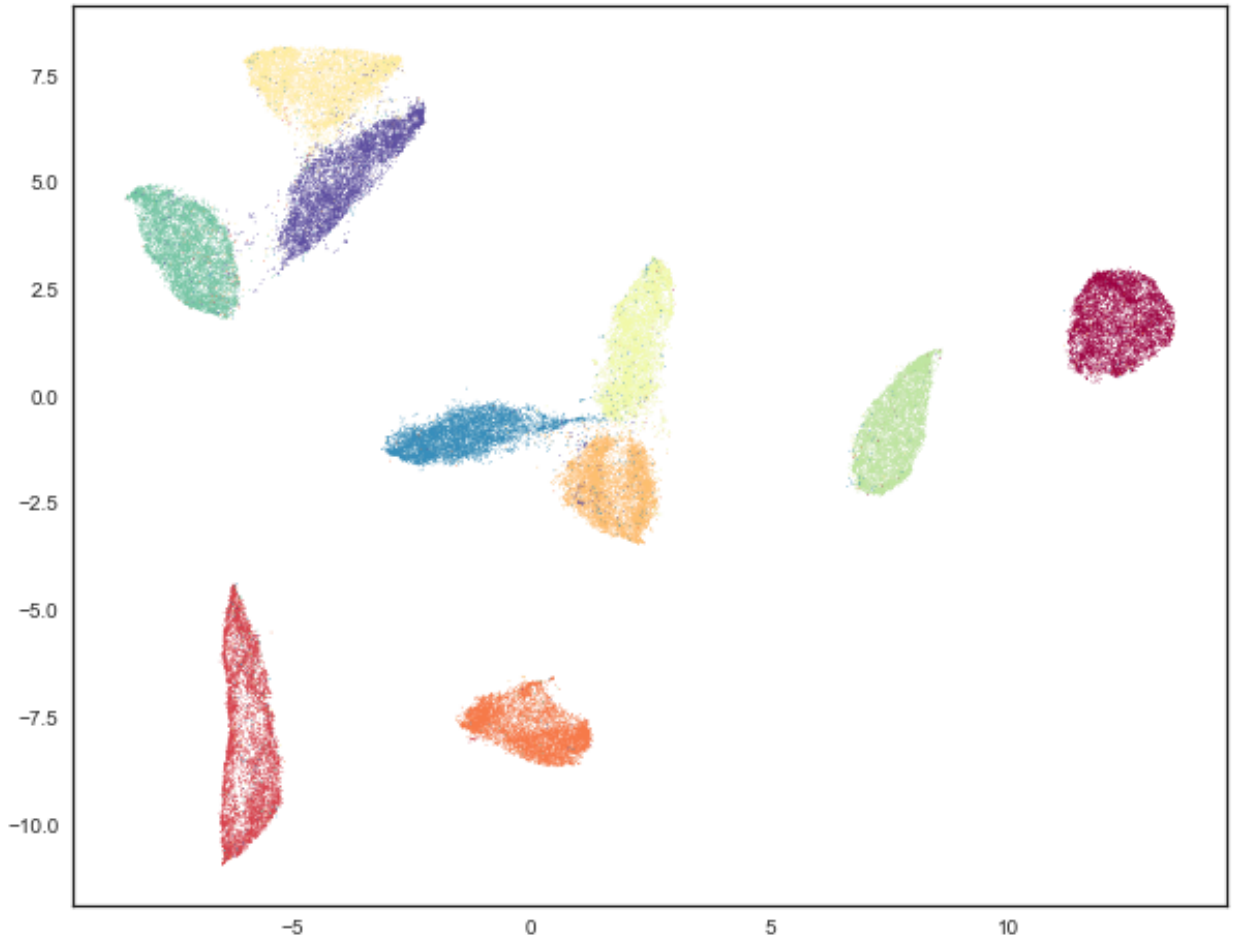
Our goal is to make use of UMAP to perform non-linear manifold aware dimension reduction so we can get the dataset down to a number of dimensions small enough for a density based clustering algorithm to make progress. One advantage of UMAP for this is that it doesn't require you to reduce to only two dimensions – you can reduce to 10 dimensions instead since the goal is to cluster, not visualize, and the performance cost with UMAP is minimal. As it happens MNIST is such a simple dataset that we really can push it all the way down to only two dimensions, but in general you should explore different embedding dimension options.

The next thing to be aware of is that when using UMAP for dimension reduction you will want to select different parameters than if you were using it for visualization. First of all we will want a larger `n_neighbors` value – small values will focus more on very local structure and are more prone to producing fine grained cluster structure that may be more a result of patterns of noise in the data than actual clusters. In this case we'll double it from the default 15 up to 30. Second it is beneficial to set `min_dist` to a very low value. Since we actually want to pack points together densely (density is what we want after all) a low value will help, as well as making cleaner separations between clusters. In this case we will simply set `min_dist` to be 0.

```
clusterable_embedding = umap.UMAP(
    n_neighbors=30,
    min_dist=0.0,
    n_components=2,
    random_state=42,
).fit_transform(mnist.data)
```

We can visualize the results of this so see how it compares with more visualization attuned parameters:

```
plt.scatter(clusterable_embedding[:, 0], clusterable_embedding[:, 1],
            c=mnist.target, s=0.1, cmap='Spectral');
```



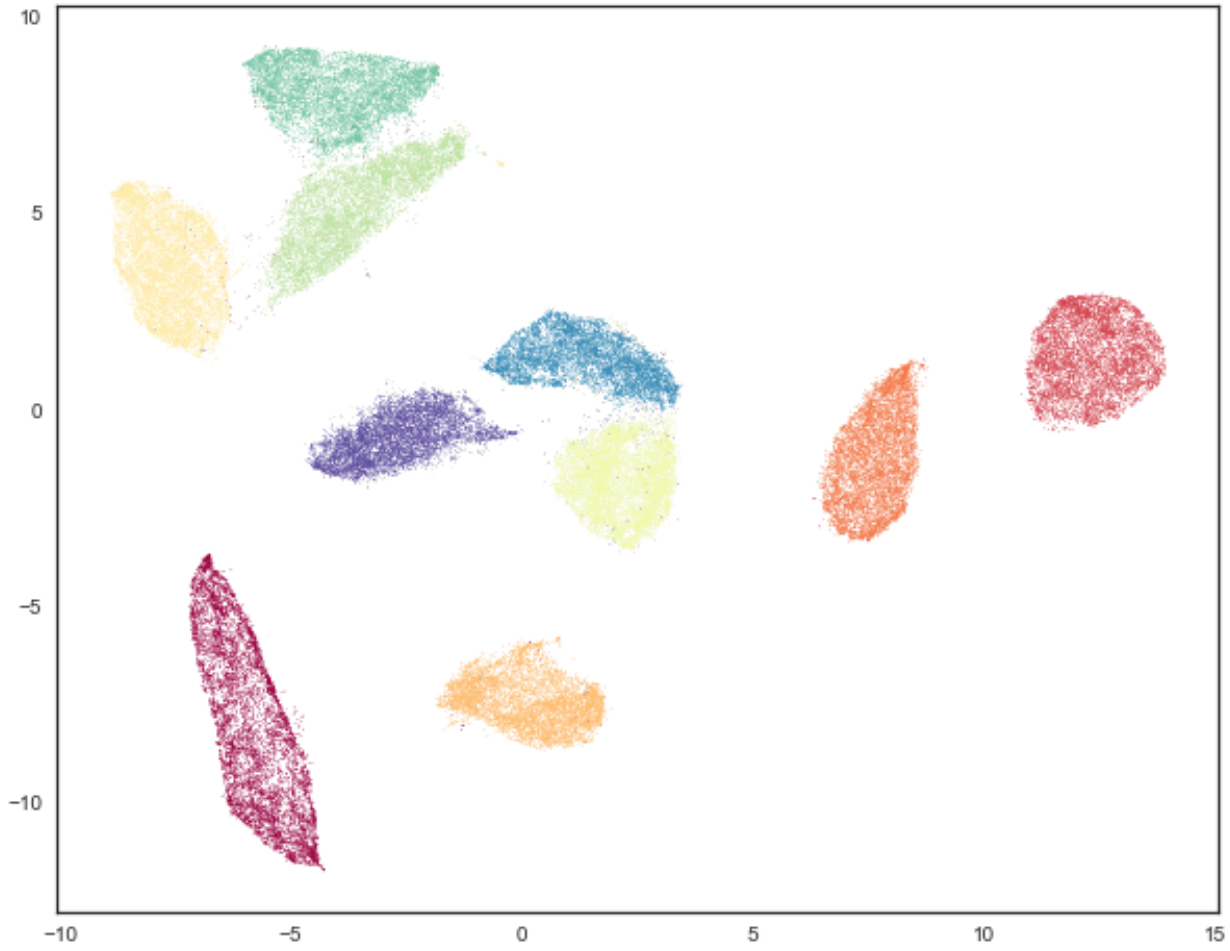
As you can see we still have the general global structure, but we are packing points together more tightly within clusters, and consequently we can see larger gaps between the clusters. Ultimately this embedding was for clustering purposes only, and we will go back to the original embedding for visualization purposes from here on out.

The next step is to cluster this data. We'll use HDBSCAN again, with the same parameter setting as before.

```
labels = hdbscan.HDBSCAN(
    min_samples=10,
    min_cluster_size=500,
).fit_predict(clusterable_embedding)
```

And now we can visualize the results, just as before.

```
clustered = (labels >= 0)
plt.scatter(standard_embedding[~clustered, 0],
            standard_embedding[~clustered, 1],
            c=(0.5, 0.5, 0.5),
            s=0.1,
            alpha=0.5)
plt.scatter(standard_embedding[clustered, 0],
            standard_embedding[clustered, 1],
            c=labels[clustered],
            s=0.1,
            cmap='Spectral');
```



We can see that we have done a much better job of finding clusters rather than merely assigning the majority of data as noise. This is because we no longer have to try to cope with the relative lack of density in 50 dimensional space and now HDBSCAN can more cleanly discern the clusters.

We can also make a quantitative assessment by using the clustering quality measures as before.

```
adjusted_rand_score(mnist.target, labels), adjusted_mutual_info_score(mnist.target,␣
↪labels)
```

```
(0.9239306564265013, 0.90302671641133736)
```

Where before HDBSCAN performed very poorly, we now have scores of 0.9 or better. This is because we actually clustered far more of the data. As before we can also look at how the clustering did on just the data that HDBSCAN was confident in clustering.

```
clustered = (labels >= 0)
(
    adjusted_rand_score(mnist.target[clustered], labels[clustered]),
    adjusted_mutual_info_score(mnist.target[clustered], labels[clustered])
)
```

```
(0.93240371696811541, 0.91912906363537572)
```

This is a little worse than the original HDBSCAN, but it is unsurprising that you are going to be wrong more often if

you make more predictions. The question is how much more of the data is HDBSCAN actually clustering? Previously we were clustering only 17% of the data.

```
np.sum(clustered) / mnist.data.shape[0]
```

```
0.99164285714285716
```

Now we are clustering over 99% of the data! And our results in terms of adjusted Rand score and adjusted mutual information are in line with the current state of the art techniques using convolutional autoencoder techniques. That's not bad for an approach that is simply viewing the data as arbitrary 784 dimensional vectors.

Hopefully this has outlined how UMAP can be beneficial for clustering. As with all things care must be taken, but clearly UMAP can provide significantly better clustering results when used judiciously.

CHAPTER 11

Outlier detection using UMAP

While an earlier tutorial looked at using UMAP for clustering, it can also be used for outlier detection, providing that some care is taken. This tutorial will look at how to use UMAP in this manner, and what to look out for, by finding anomalous digits in the MNIST handwritten digits dataset. To start with let's load the relevant libraries:

```
import numpy as np
import sklearn.datasets
import sklearn.neighbors
import umap
import umap.plot
import matplotlib.pyplot as plt
%matplotlib inline
```

With this in hand, let's grab the MNIST digits dataset from the internet, using the new `fetch_ml` loader in sklearn.

```
data, labels = sklearn.datasets.fetch_openml('mnist_784', version=1, return_X_y=True)
```

Before we get started we should try looking for outliers in terms of the native 784 dimensional space that MNIST digits live in. To do this we will make use of the Local Outlier Factor (LOF) method for determining outliers since sklearn has an easy to use implementation. The essential intuition of LOF is to look for points that have a (locally approximated) density that differs significantly from the average density of their neighbors. In our case the actual details are not so important – it is enough to know that the algorithm is reasonably robust and effective on vector space data. We can apply it using the `fit_predict` method of the sklearn class. The LOF class take a parameter `contamination` which specifies the percentage of data that the user expects to be noise. For this use case we will set it to 0.001428 since, given the 70,000 samples in MNIST, this will result in 100 outliers, which we can then look at in more detail.

```
%%time
outlier_scores = sklearn.neighbors.LocalOutlierFactor(contamination=0.001428).fit_
↪predict(data)
```

```
CPU times: user 1h 29min 10s, sys: 12.4 s, total: 1h 29min 22s
Wall time: 1h 29min 53s
```

It is worth noting how long that took. Over an hour and a half! Why did it take so long? Because LOF requires a notion of density, which in turn relies on a nearest neighbor type computation – which is expensive in sklearn for high dimensional data. This alone is potentially a reason to look at reducing the dimension of the data – it makes it more amenable to existing techniques like LOF.

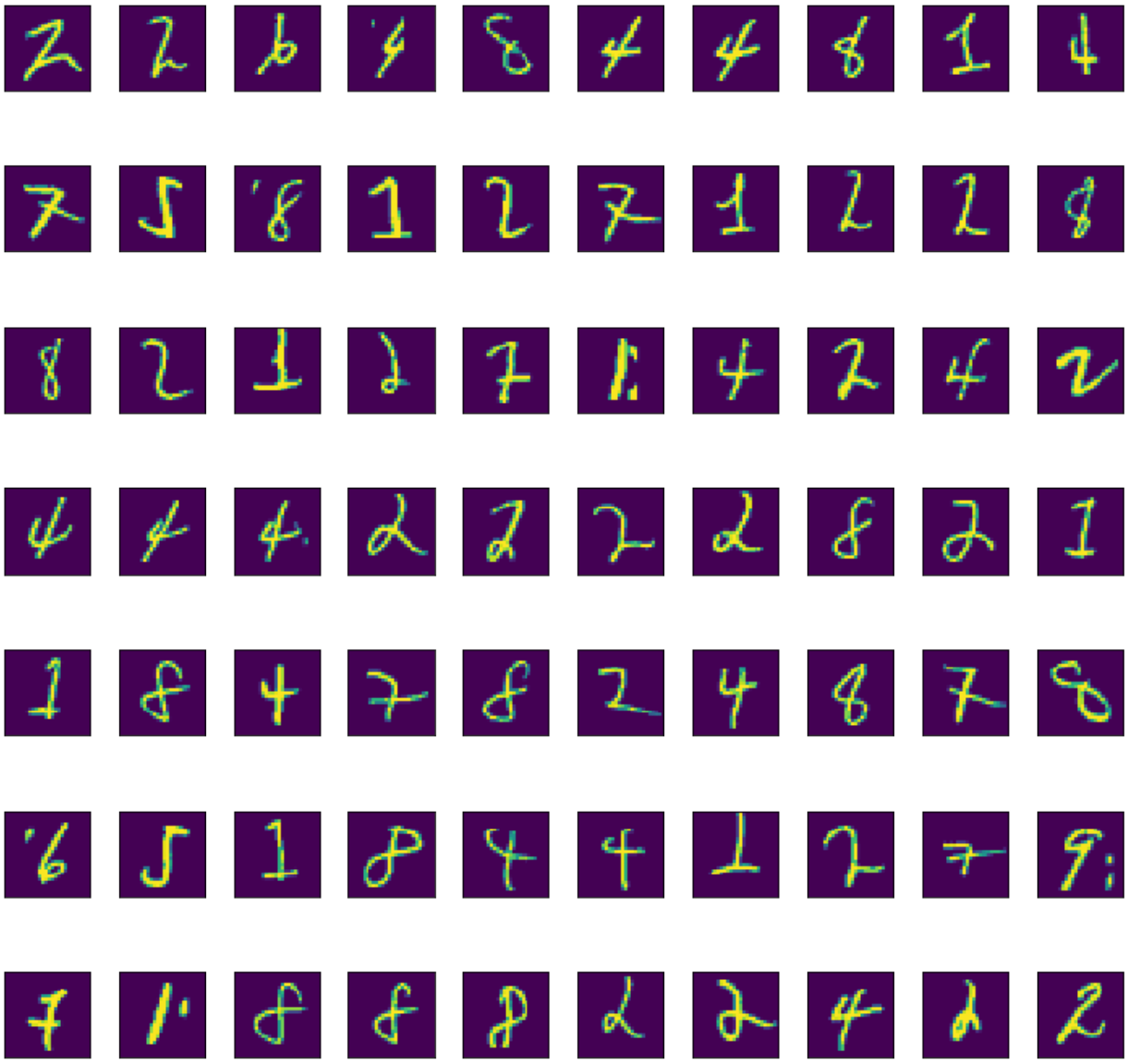
Now that we have a set of outlier scores we can find the actual outlying digit images – these are the ones with scores equal to -1. Let's extract that data, and check that we got 100 different digit images.

```
outlying_digits = data[outlier_scores == -1]
outlying_digits.shape
```

```
(100, 784)
```

Now that we have the outlying digit images the first question we should be asking is "what do they look like?". Fortunately for us we can convert the 784 dimensional vectors back into image and plot them, making it easier to look at. Since we extracted the 100 most outlying digit images we can just display a 10x10 grid of them.

```
fig, axes = plt.subplots(7, 10, figsize=(10,10))
for i, ax in enumerate(axes.flatten()):
    ax.imshow(outlying_digits[i].reshape((28,28)))
    plt.setp(ax, xticks=[], yticks=[])
plt.tight_layout()
```



These do certainly look like somewhat strange looking handwritten digits, so our outlier detection seems to be working to some extent.

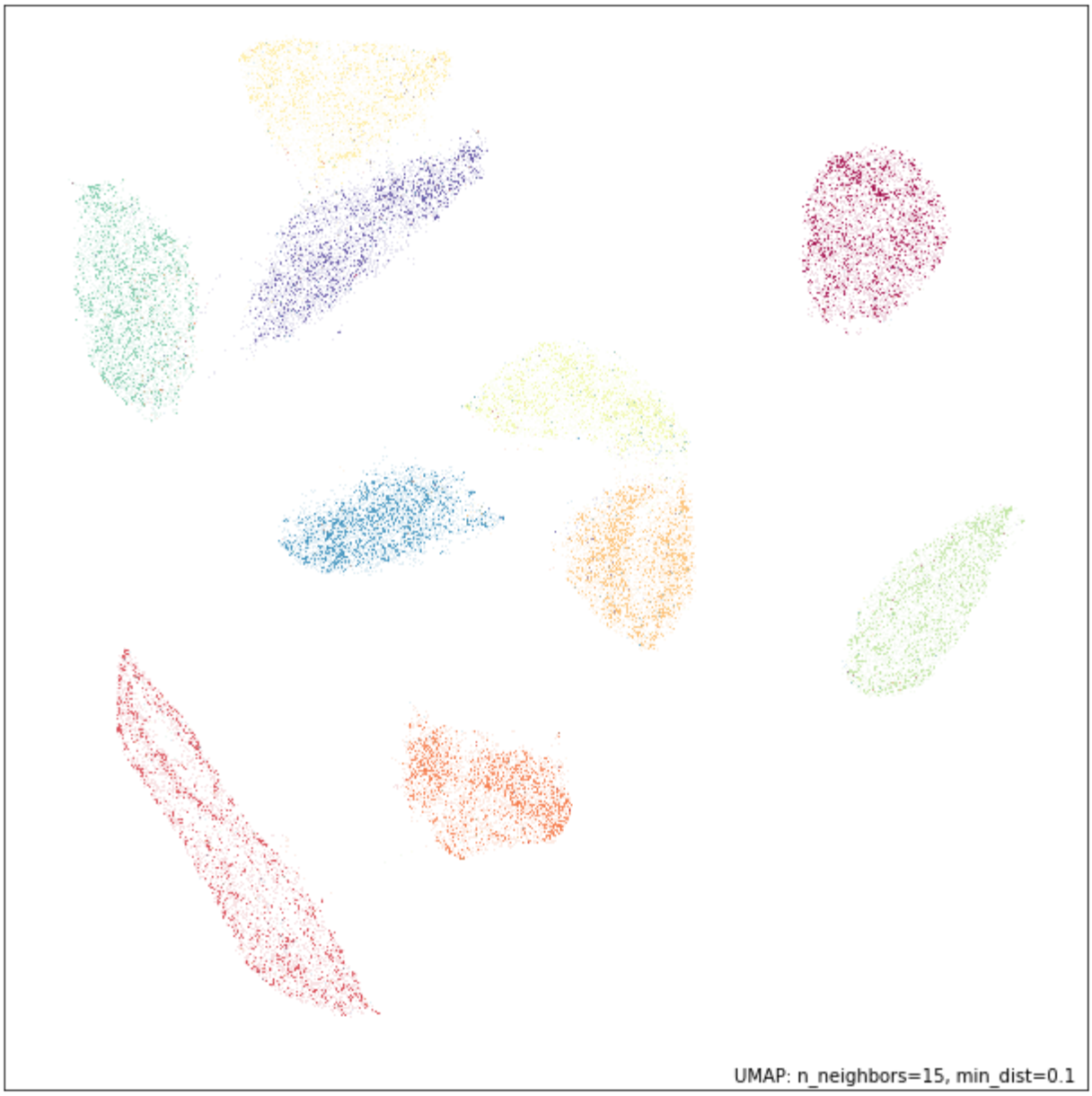
Now let's try a naive approach using UMAP and see how far that gets us. First let's just apply UMAP directly with default parameters to the MNIST data.

```
mapper = umap.UMAP().fit(data)
```

Now we can see what we got using the new plotting tools in umap.plot.

```
umap.plot.points(mapper, labels=labels)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1c3db71358>
```



That looks like what we have come to expect from a UMAP embedding of MNIST. The question is have we managed to preserve outliers well enough that LOF can still find the bizarre digit images, or has the embedding lost that information and contracted the outliers into the individual digit clusters? We can simply apply LOF to the embedding and see what that returns.

```
%%time
outlier_scores = sklearn.neighbors.LocalOutlierFactor(contamination=0.001428).fit_
↪predict(mapper.embedding_)
```

This was obviously much faster since we are operating in a much lower dimensional space that is more amenable to the spatial indexing methods that sklearn uses to find nearest neighbors. As before we extract the outlying digit images, and verify that we got 100 of them,

```
outlying_digits = data[outlier_scores == -1]
```

(continues on next page)

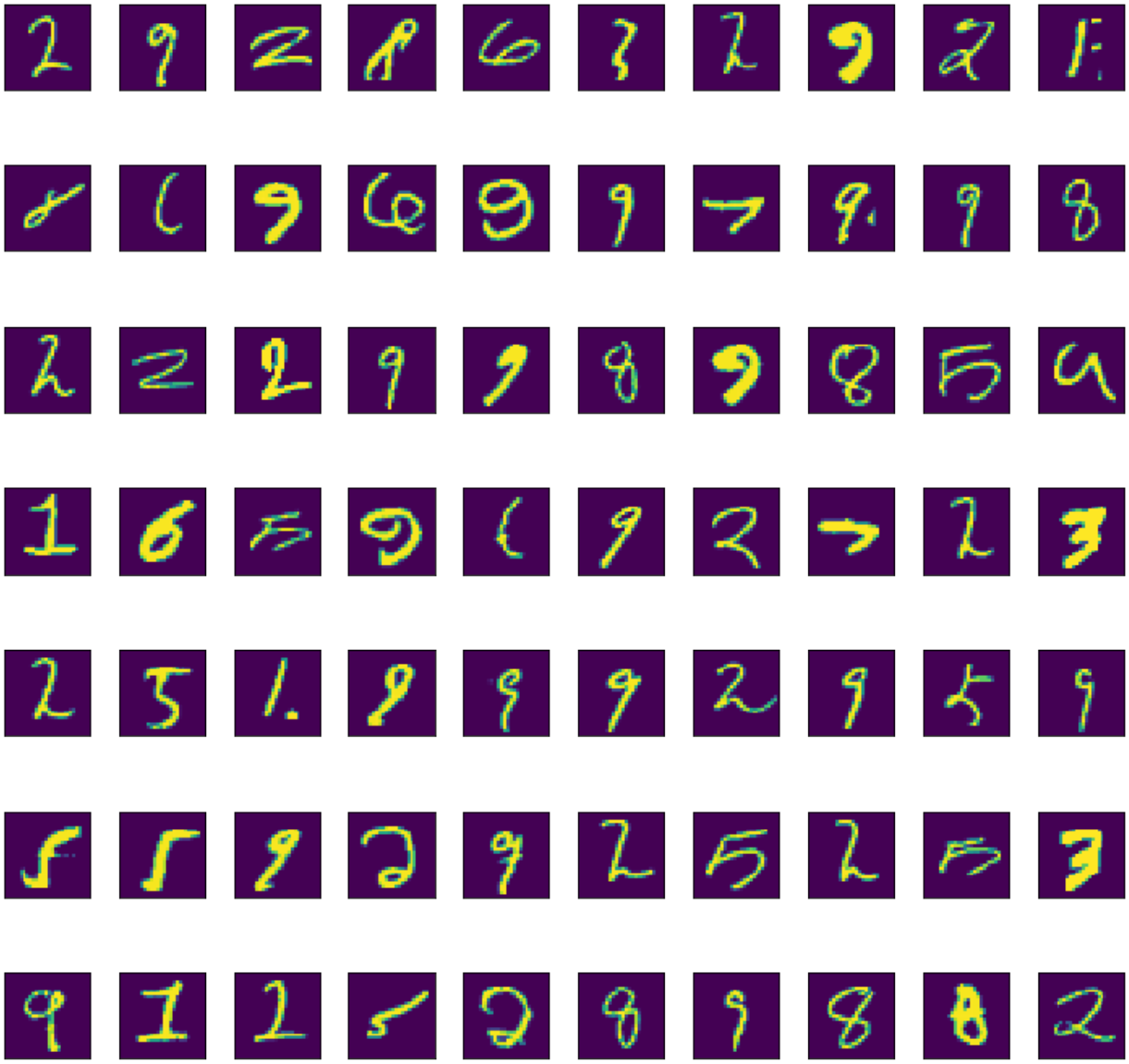
(continued from previous page)

```
outlying_digits.shape
```

```
(100, 784)
```

Now we need to plot the outlying digit images to see what kinds of digit images this approach found to be particularly strange.

```
fig, axes = plt.subplots(7, 10, figsize=(10,10))
for i, ax in enumerate(axes.flatten()):
    ax.imshow(outlying_digits[i].reshape((28,28)))
    plt.setp(ax, xticks=[], yticks=[])
plt.tight_layout()
```



In many ways this looks to be a *better* result than the original LOF in the high dimensional space. While the digit images that the high dimensional LOF found to be strange were indeed somewhat odd looking, many of these digit images are considerably stranger – significantly odd line thickness, warped shapes, and images that are hard to even

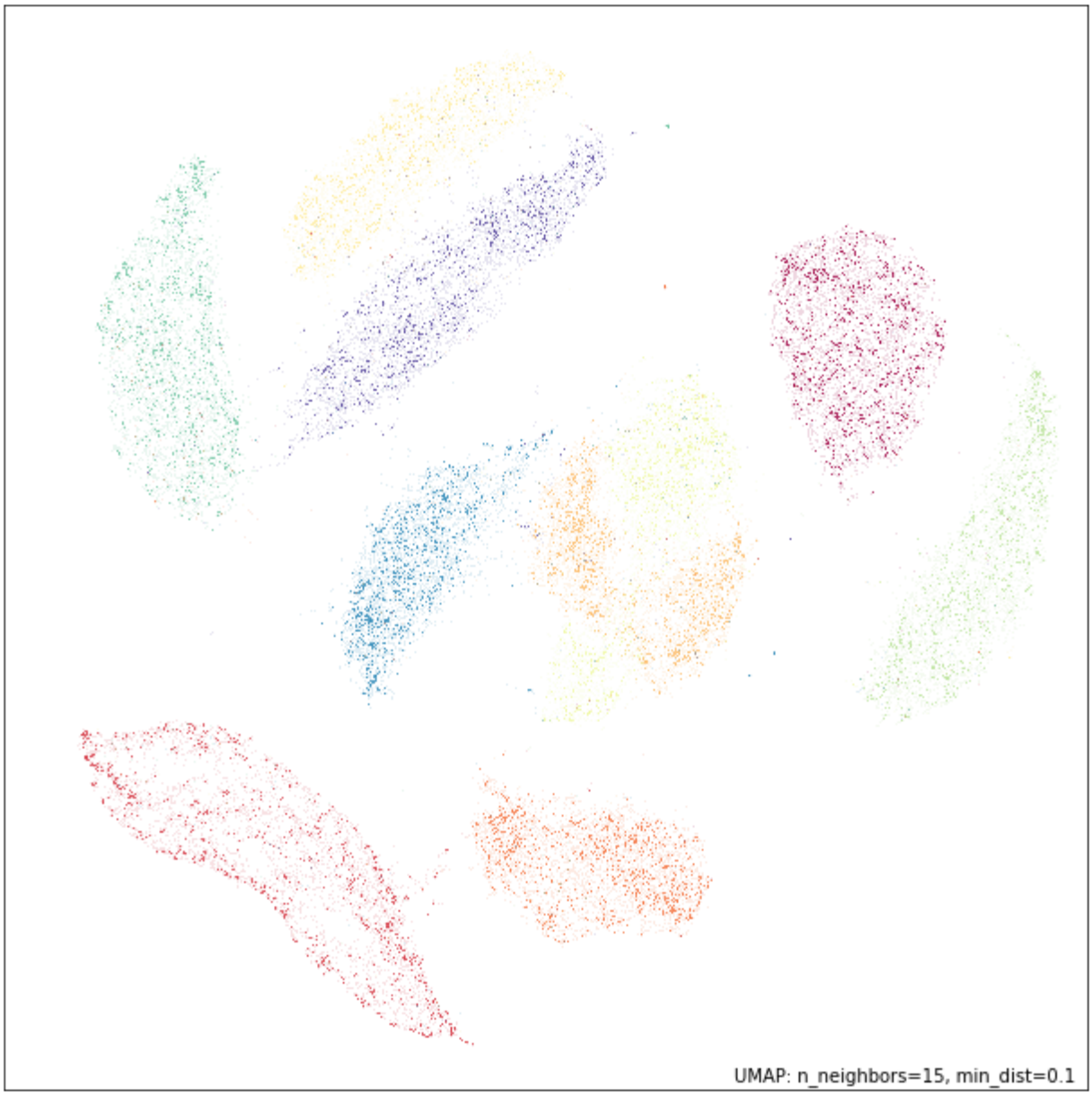
recognise as digits. This helps to demonstrate a certain amount of confirmation bias when examining outliers: since we expect things tagged as outliers to be strange we tend to find aspects of them that justify that classification, potentially unaware of how much stranger some of the data may in fact be. This should make us wary of even this outlier set: what else might lurk in the dataset?

We can, in fact, potentially improve on this result by tuning the UMAP embedding a little for the task of finding outliers. When UMAP combines together the different local simplicial sets (see *How UMAP Works* for more details) the standard approach uses a union, but we could instead take an intersection. An intersection ensures that outliers remain disconnected, which is certainly beneficial when seeking to find outliers. A downside of the intersection is that it tends to break up the resulting simplicial set into many disconnected components and a lot of the more non-local and global structure is lost, resulting in a lot lower quality of the resulting embedding. We can, however, interpolate between the union and intersection. In UMAP this is given by the `set_op_mix_ratio`, where a value of 0.0 represents an intersection, and a value of 1.0 represents a union (the default value is 1.0). By setting this to a lower value, say 0.25, we can encourage the embedding to do a better job of preserving outliers as outlying, while still retaining the benefits of a union operation.

```
mapper = umap.UMAP(set_op_mix_ratio=0.25).fit(data)
```

```
umap.plot.points(mapper, labels=labels)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1c3f496908>
```



As you can see the embedding is not as well structured overall as when we had a `set_op_mix_ratio` of 1.0, but we have potentially done a better job of ensuring that outliers remain outlying. We can test that hypothesis by running LOF on this embedding and looking at the resulting digit images we get out. Ideally we should expect to find some potentially even stranger results.

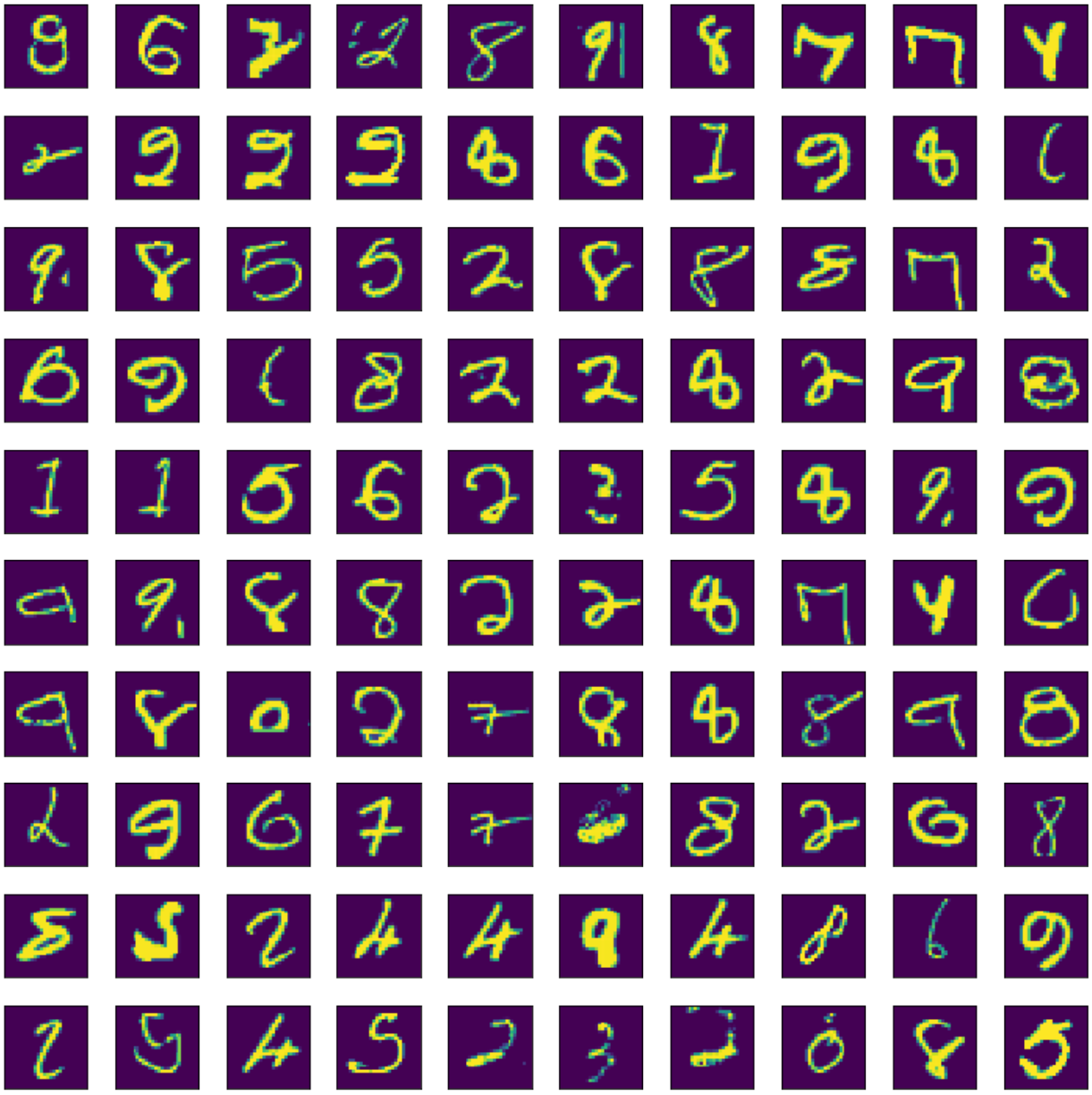
```
%%time
outlier_scores = sklearn.neighbors.LocalOutlierFactor(contamination=0.001428).fit_
↪predict(mapper.embedding_)
```

```
outlying_digits = data[outlier_scores == -1]
outlying_digits.shape
```

```
(100, 784)
```

We have the expected 100 most outlying digit images, so let's visualise the results and see if they really are particularly strange.

```
fig, axes = plt.subplots(10, 10, figsize=(10,10))
for i, ax in enumerate(axes.flatten()):
    ax.imshow(outlying_digits[i].reshape((28,28)))
    plt.setp(ax, xticks=[], yticks=[])
plt.tight_layout()
```



Here we see that the line thickness variation (particularly "fat" digits, or particularly "fine" lines) that the original embedding helped surface come through even more strongly here. We also see a number of clearly corrupted images with extra lines, dots, or strange blurring occurring.

So, in summary, using UMAP to reduce dimension prior to running classical outlier detection methods such as LOF can improve both the speed with which the algorithm runs, and the quality of results the outlier detection can find. Fur-

thermore we have introduced the `set_op_mix_ratio` parameter, and explained how it can be used to potentially improve the performance of outlier detection approaches applied to UMAP embeddings.

CHAPTER 12

Combining multiple UMAP models

It is possible to combine together multiple UMAP models, assuming that they are operating on the same underlying data. To get an idea of how this works recall that UMAP uses an intermediate fuzzy topological representation (see *How UMAP Works*). Given different views of the same underlying data this will generate different fuzzy topological representations. We can apply intersections or unions to these representations to get a new composite fuzzy topological representation which we can then embed into low dimensional space in the standard UMAP way. The key is that, to be able to sensibly intersect or union these representations, there must be one-to-one correspondences between the data samples from the two different views.

To get an idea of how this might work it is useful to see it in practice. Let's load some libraries and get started.

```
import sklearn.datasets
from sklearn.preprocessing import RobustScaler
import seaborn as sns
import pandas as pd
import numpy as np
import umap
import umap.plot
```

12.1 MNIST digits example

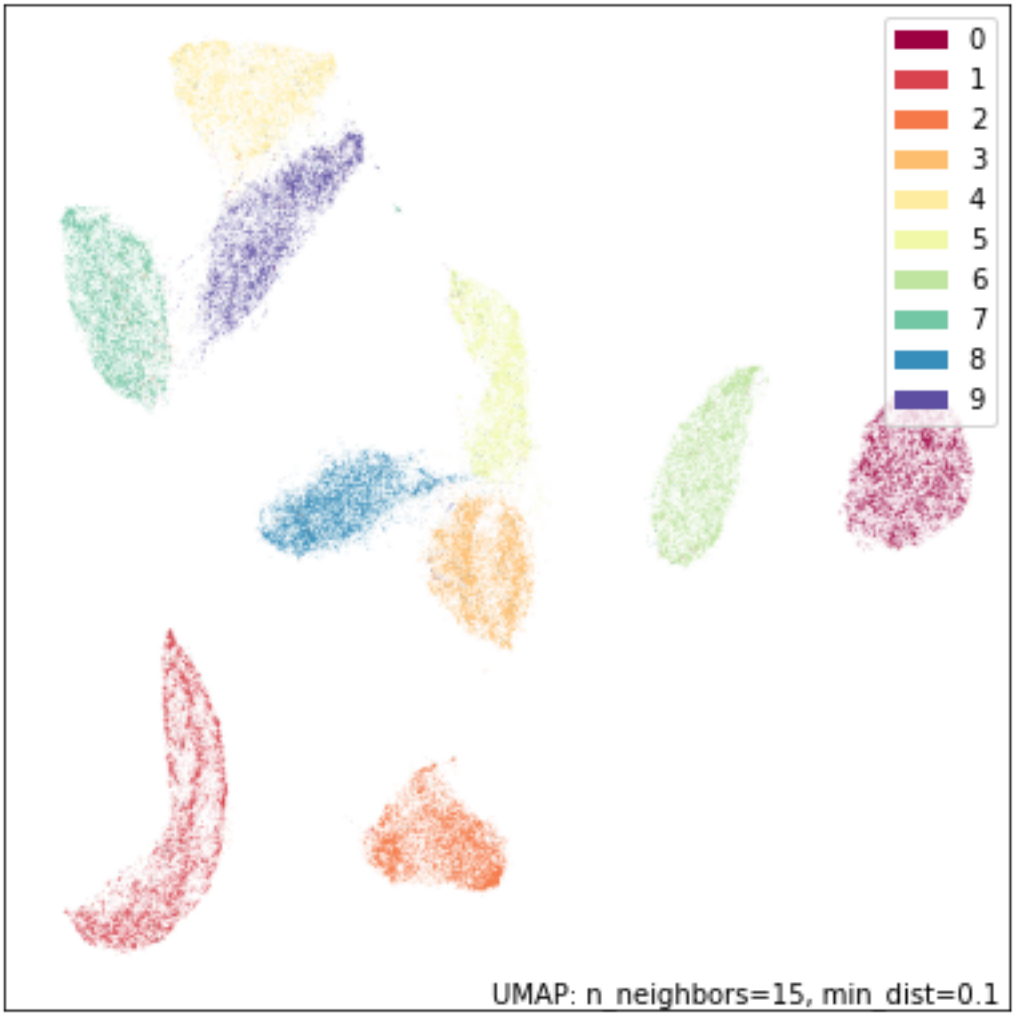
To begin with let's use a relatively familiar dataset – the MNIST digits dataset that we've used in other sections of this tutorial. The data is (grayscale) 28x28 pixel images of handwritten digits (0 through 9); in total there are 70,000 such images, and each image is unrolled into a 784 element vector.

```
mnist = sklearn.datasets.fetch_openml("mnist_784")
```

To ensure we have an idea of what this dataset looks like through the lens of UMAP we can run UMAP on the full dataset.

```
mapper = umap.UMAP(random_state=42).fit(mnist.data)
```

```
umap.plot.points(mapper, labels=mnist.target, width=500, height=500)
```



To make the problem more interesting let's carve the dataset in two – not into two sets of 35,000 samples, but instead carve each image in half. That is, we'll end up with 70,000 samples each of which is the top half of the image of the handwritten digit, and another 70,000 samples each of which is the bottom half of the image of the handwritten digit.

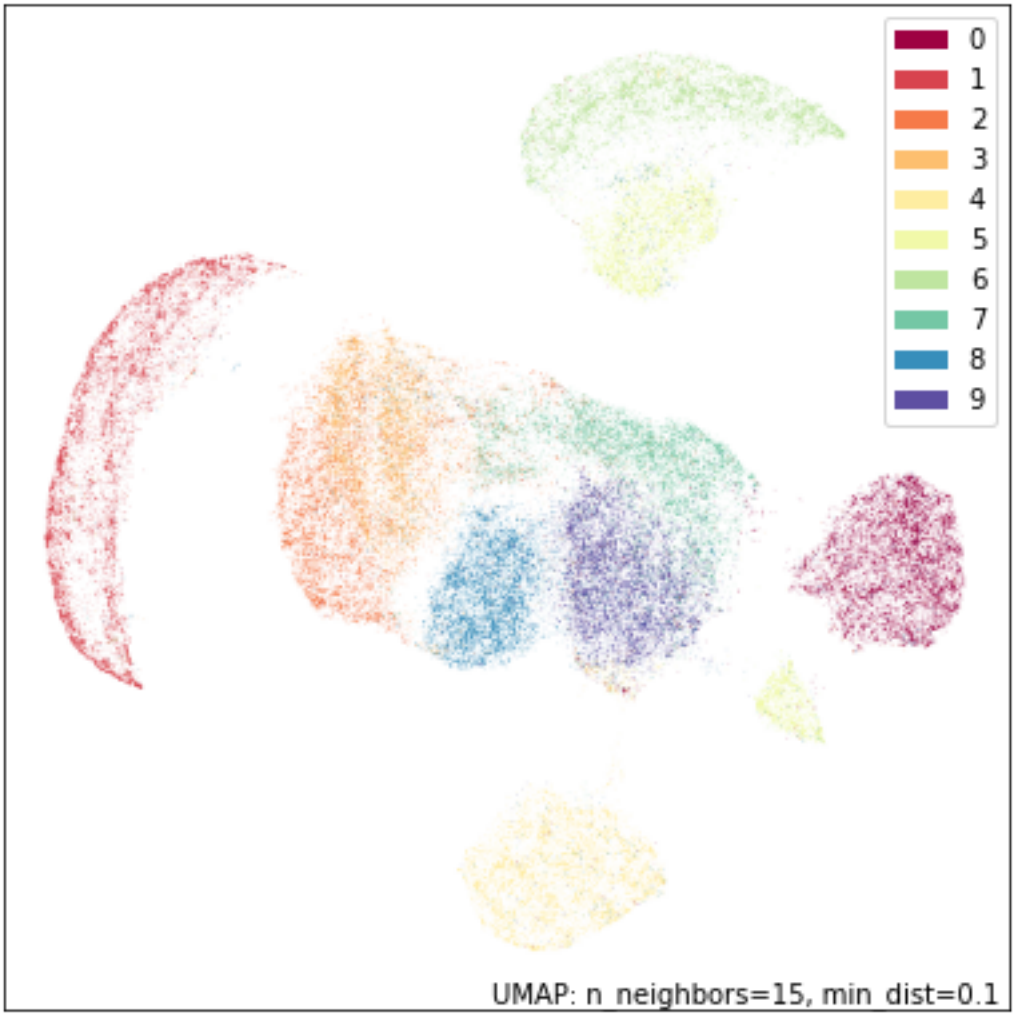
```
top = mnist.data[:, :28 * 14]
bottom = mnist.data[:, 28 * 14:]
```

This is a little artificial, but it provides us with an example dataset where we have two distinct views of the data which we can still well understand. In practice this situation would be more likely to arise when there are two different data collection processes sampling from the same underlying population. In our case we could simply glue the data back together (hstack the numpy arrays for example), but potentially this isn't feasible as the different data views may have different scales or modalities. So, despite the fact that we could glue things back together in this case, we will proceed as if we can't – as may be the case for many real world problems.

Let's first look at what UMAP does individually on each dataset. We'll start with the top halves of the digits:

```
top_mapper = umap.UMAP(random_state=42).fit(top)
```

```
umap.plot.points(top_mapper, labels=mnist.target, width=500, height=500)
```

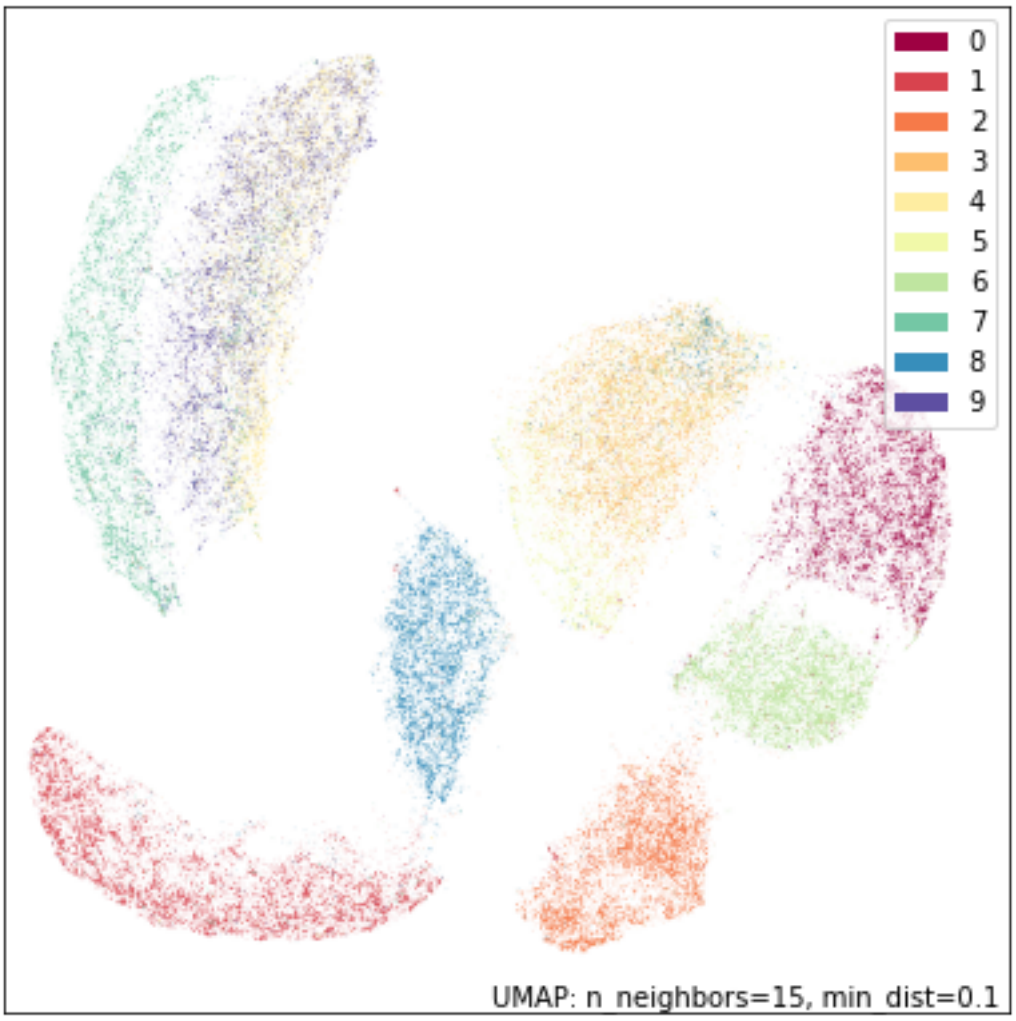


While UMAP still manages to mostly separate the different digit classes we can see the results are quite different from UMAP on the full standard MNIST dataset. The twos and threes are blurred together (as we would expect given that we don't have the bottom half of the image wich would let us tell them apart); The twos and threes are also in a large grouping that pulls together all of the eights, sevens and nines (again, what we would expect given only the top half of the digit), while the fives and sixes are somewhat distinct, but clearly are similar to each other. It is only the ones, fours and zeros that are very clearly discernible.

Now let's see what sorts of results we get with the bottom halves of the digits:

```
bot_mapper = umap.UMAP(random_state=42).fit(bottom)
```

```
umap.plot.points(bot_mapper, labels=mnist.target, width=500, height=500)
```



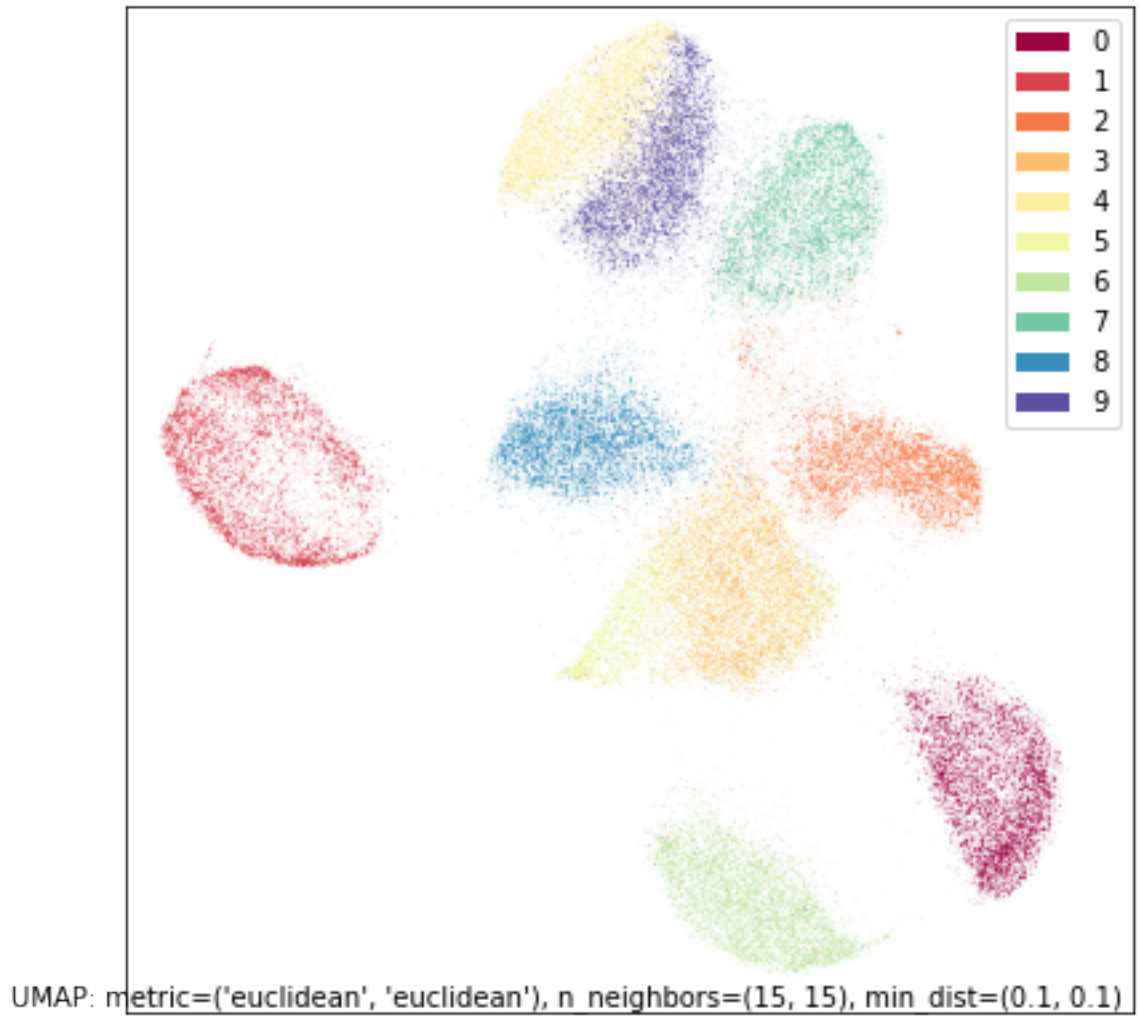
This is clearly a very different view of the data. Now it is the fours and nines that blur together (presumably many of the nines are drawn with straight rather than curved stems), with sevens nearby. The twos and the threes are very distinct from each other, but the threes and the fives are combined (as one might expect given that the bottom halves *should* look similar). Zeros and sixes are distinct, but close to each other. Ones, eights and twos are the most distinctive digits in this view.

So, assuming we can't just glue the raw data together and stick a reasonable metric on it, what can we do? We can perform intersections or unions on the fuzzy topological representations. There is also some work to be done re-asserting UMAP's theoretical assumptions (local connectivity, approximately uniform distributions). Fortunately UMAP makes this relatively easy as long as you have a copy of fitted UMAP models on hand (which we do in this case). To intersect two models simply use the $\star$ operator; to union them use the + operator. Note that this will actually take some time since we need to compute the 2D embedding of the combined model.

```
intersection_mapper = top_mapper * bot_mapper
union_mapper = top_mapper + bot_mapper
```

With that complete we can visualize the results. First let's look at the intersection:

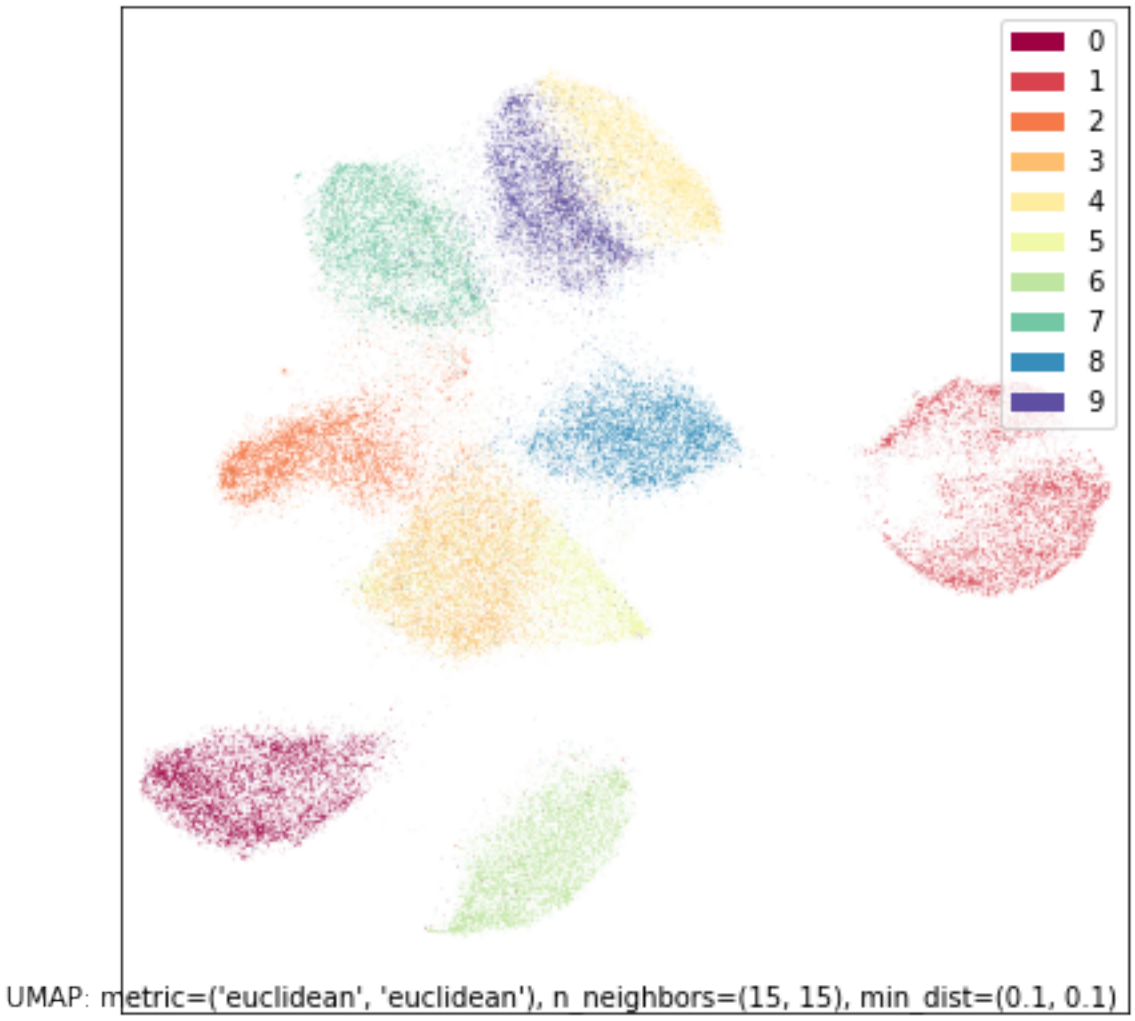
```
umap.plot.points(intersection_mapper, labels=mnist.target, width=500, height=500)
```



As you can see, while this isn't as good as a UMAP plot for the full MNIST dataset it has recovered the individual digits quite well. The worst of the remaining overlap is between the threes and fives in the center, which is it still struggling to fully distinguish. But note, also, that we have recovered more of the overall structure than either of the two different individual views, with the layout of different digit classes more closely resembling that of the UMAP run on the full dataset.

Now let's look at the union.

```
umap.plot.points(union_mapper, labels=mnist.target, width=500, height=500)
```

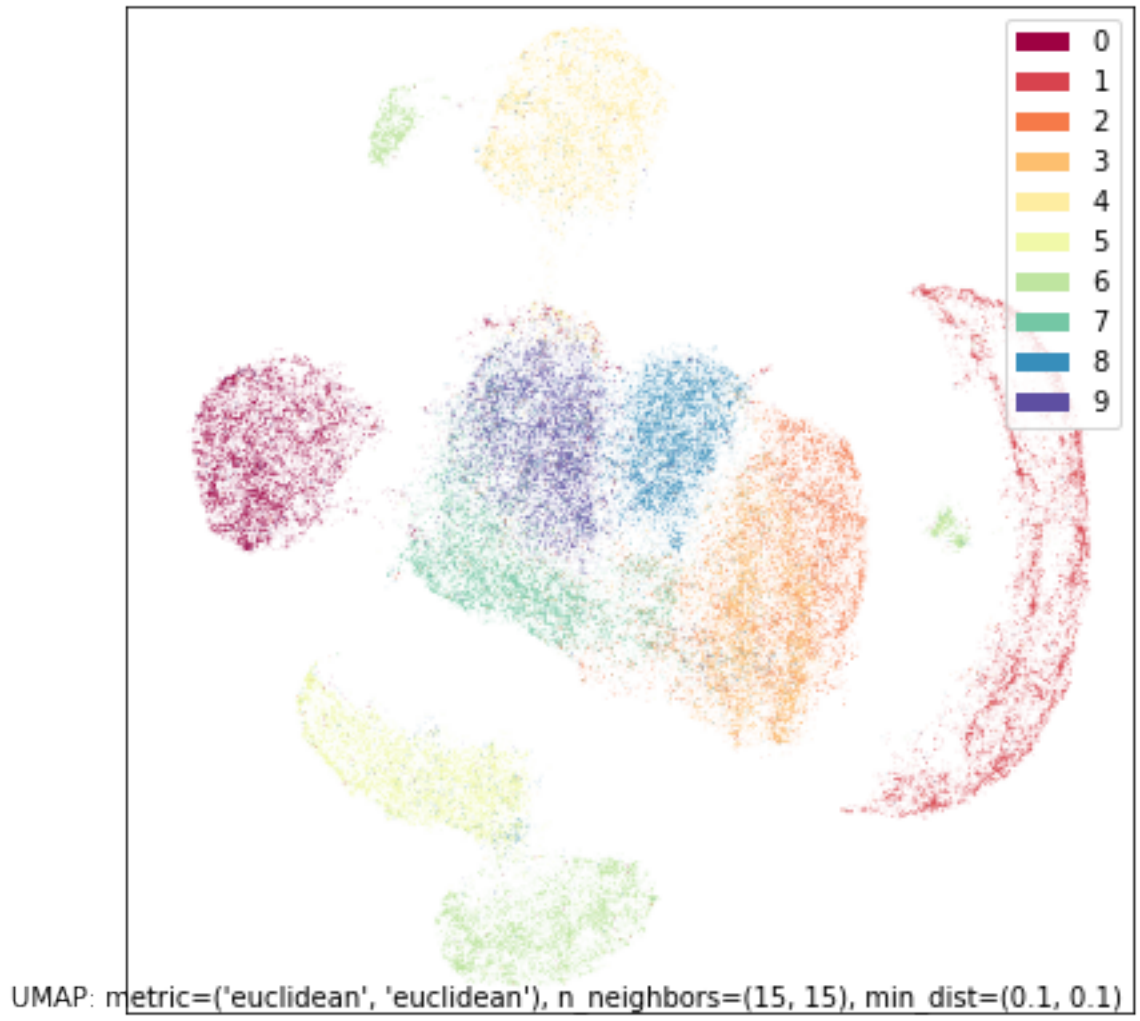


Given that UMAP is agnostic to rotation or reflection of the final layout, this is essentially the same result as the intersection since it is almost the reflection of it in the y-axis. This sort of result (intersection and union being similar) is not always the case (in fact it is not that common), but since the underlying structure of the digits dataset is so clear we find that either way of piecing it together from the two half datasets manage to find the same core underlying structure.

If you are willing to try something a little more experimental there is also a third option using the − operator which effectively intersects with the fuzzy set complement (and is thus not commutative, just as − implies). The goal here is to try to provide a sense of what the data looks like when we contrast it against a second view.

```
contrast_mapper = top_mapper - bot_mapper
```

```
umap.plot.points(contrast_mapper, labels=mnist.target, width=500, height=500)
```



In this case the result is not overly dissimilar from the embedding of just the top half, so the contrast has perhaps not shown is as much as we might have hoped.

12.2 Diamonds dataset example

Now let's try the same approach on a different dataset where the option of just running UMAP on the full dataset is not available. For this we'll use the diamonds dataset. In this dataset each row represents a different diamond and provides details on the weight (carat), cut, color, clarity, size (depth, table, x, y, z) and price of the given diamond. How these different factors interplay is somewhat complicated.

```
diamonds = sns.load_dataset('diamonds')
diamonds.head()
```

For our purposes let's take "price" as a "target" variable (as is often the case when the dataset is used in machine learning contexts). What we would like to do is provide a UMAP embedding of the data using the remaining features. This is tricky since we can't exactly use a euclidean metric over the whole thing. What we can do, however, is split the data into two distinct types: the purely numeric features relating to size and weight, and the categorical features of color, cut and clarity. Let's pull each of those feature sets out so we can work with them independently.

```
numeric = diamonds[["carat", "table", "x", "y", "z"]].copy()
ordinal = diamonds[["cut", "color", "clarity"]].copy()
```

Now we have a new problem: the numeric features are not at all on the same scales, so any sort of standard distance metric across them will be dominated by those features with the largest ranges. We can correct for that by performing feature scaling. To do that we'll make use of sklearn's `RobustScaler` which uses robust statistics (such as the median and interquartile range) to center and rescale the data feature by feature. If we look at the results on the first five rows we see that the different features are all now reasonably comparable, and it is reasonable to apply something like euclidean distance across them.

```
scaled_numeric = RobustScaler().fit_transform(numeric)
scaled_numeric[:5]
```

```
array([[-0.734375  , -0.66666667, -0.95628415, -0.95054945, -0.97345133],
       [-0.765625  ,  1.33333333, -0.98907104, -1.02747253, -1.07964602],
       [-0.734375  ,  2.66666667, -0.90163934, -0.9010989 , -1.07964602],
       [-0.640625  ,  0.33333333, -0.81967213, -0.81318681, -0.79646018],
       [-0.609375  ,  0.33333333, -0.7431694 , -0.74725275, -0.69026549]])
```

What is the best way to handle the categorical features? If they are purely categorical it would make sense to one-hot encode the categories and use "dice" distance between them. A downside of that is that, with so few categories, it is a very coarse metric which will fail to provide much differentiation. For the diamonds dataset, however, the categories come with a strict order: Ideal cut is better than Premium cut, which is better than Very Good cut and so on. Color grades work similarly, and there is a distinct grading scheme for clarity as well. We can use an ordinal encoding on these categories. Now, while the *ranges* of values may vary, the differences between them are all comparable – a difference of 1 for each grade level. That means we don't need to rescale this data after the ordinal coding.

```
ordinal["cut"] = ordinal.cut.map({"Fair":0, "Good":1, "Very Good":2, "Premium":3,
↪"Ideal":4})
ordinal["color"] = ordinal.color.map({"D":0, "E":1, "F":2, "G":3, "H":4, "I":5, "J":6}
↪)
ordinal["clarity"] = ordinal.clarity.map({"I1":0, "SI2":1, "SI1":2, "VS2":3, "VS1":4,
↪"VVS2":5, "VVS1":6, "IF":7})
```

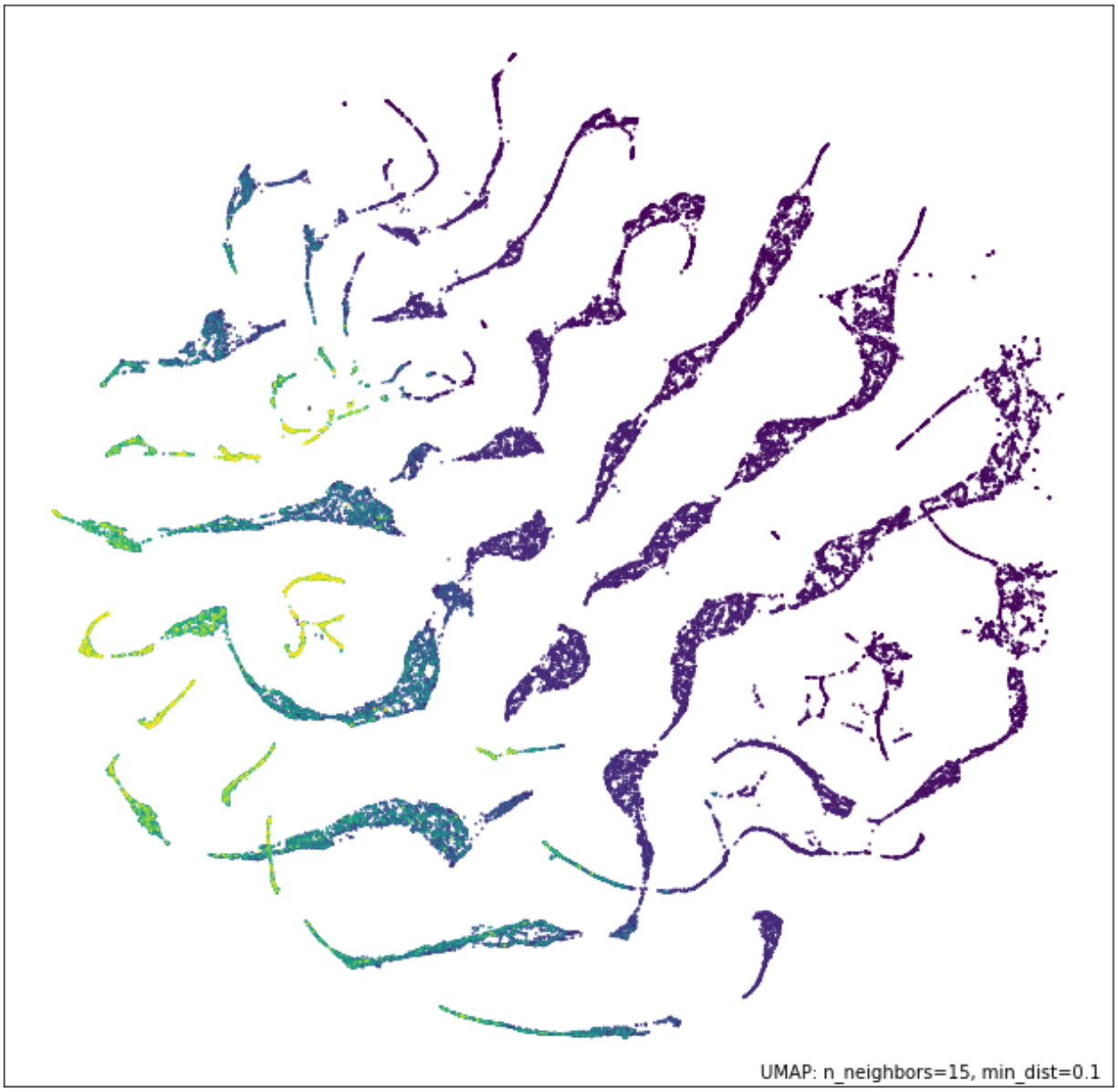
```
ordinal
```

As noted we can use euclidean as a sensible distance on the rescaled numeric data. On the other hand since the different ordinal categories are entirelty independent of each other, and we have a strict ordinal codin, the socalled "manhattan" metric makes more sense here – it is simply the sum of the absolute differences in each category. As before we can now train UMAP models on each dataset – this time, however, since the datasets are different we need different metrics and even different values of `n_neighbors`.

```
numeric_mapper = umap.UMAP(n_neighbors=15, random_state=42).fit(scaled_numeric)
ordinal_mapper = umap.UMAP(metric="manhattan", n_neighbors=150, random_state=42).
↪fit(ordinal.values)
```

We can look at the results of each of these independent views of the dataset reduced to 2D using UMAP. Let's first look at the numeric data on size and weight of the diamonds. We can colour by the price to get some idea of how the dataset fits together.

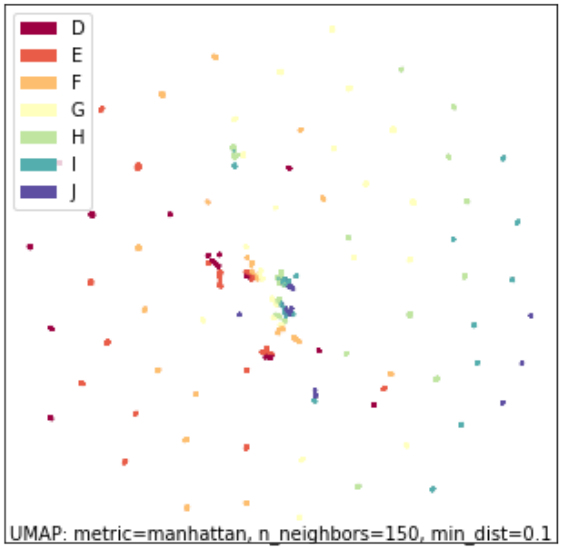
```
umap.plot.points(numeric_mapper, values=diamonds["price"], cmap="viridis")
```

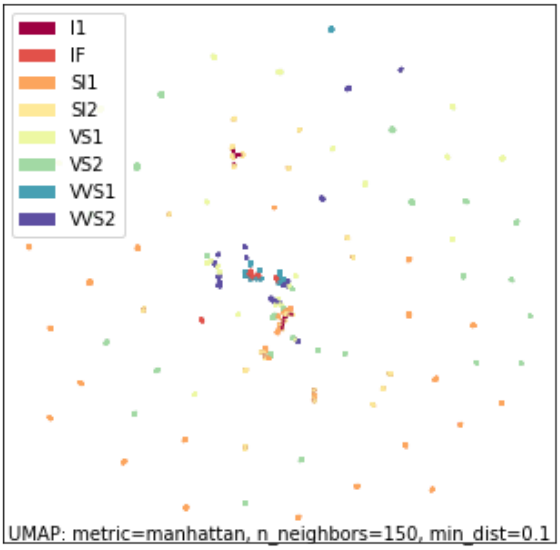


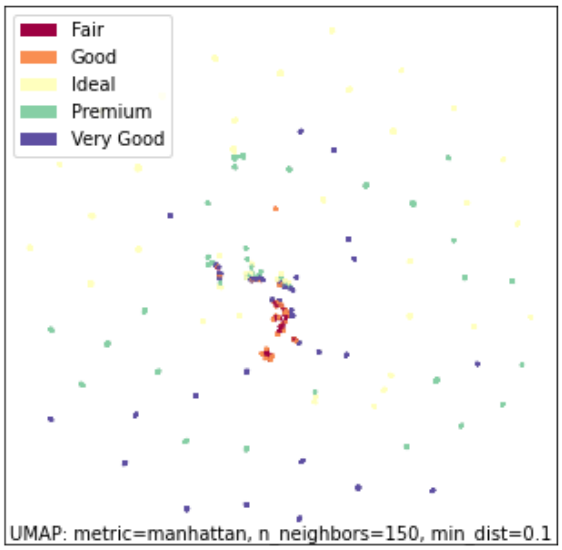
We see that while the data generally correlates somewhat with the price of the diamonds there are distinctly different threads in the data, presumably corresponding to different styles of cut, and how that results in different sizing of diamonds in the various dimensions, depending on the weight.

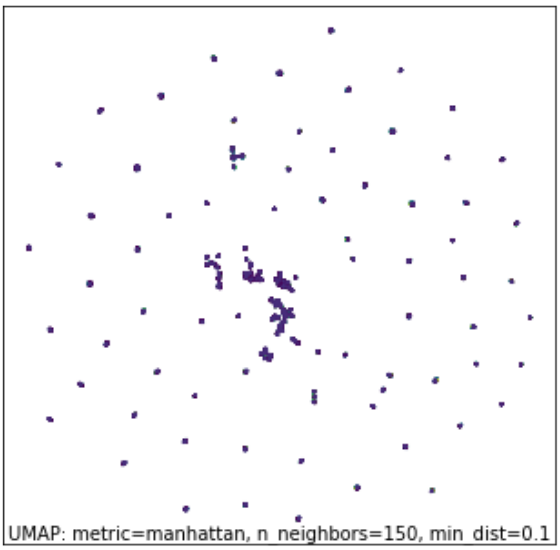
In contrast we ca look at the ordinal data. In this case we'll colour it by the different categories as well as by price.

```
fig, ax = umap.plot.plt.subplots(2, 2, figsize=(12,12))
umap.plot.points(ordinal_mapper, labels=diamonds["color"], ax=ax[0,0])
umap.plot.points(ordinal_mapper, labels=diamonds["clarity"], ax=ax[0,1])
umap.plot.points(ordinal_mapper, labels=diamonds["cut"], ax=ax[1,0])
umap.plot.points(ordinal_mapper, values=diamonds["price"], cmap="viridis", ax=ax[1,1])
```









As you can see this is a markedly different result! The ordinal data has a relatively coarse metric, since the different categories can only take on a small range of discrete values. This means that, with respect to the trio of color, cut, and clarity, diamonds are largely either almost identical, or quite distinct. The result is very tight groupings which have very high density. You can see a gradient of color from left to right in the plot; colouring by cut or clarity show different stratifications. The combination of these very distinct statifications results in this highly clustered embedding. It is exactly for this reason that we need such a high `n_neighbors` value: the very local structure of the data is merely clusters of identical categories; we need to see wider to learn more structure.

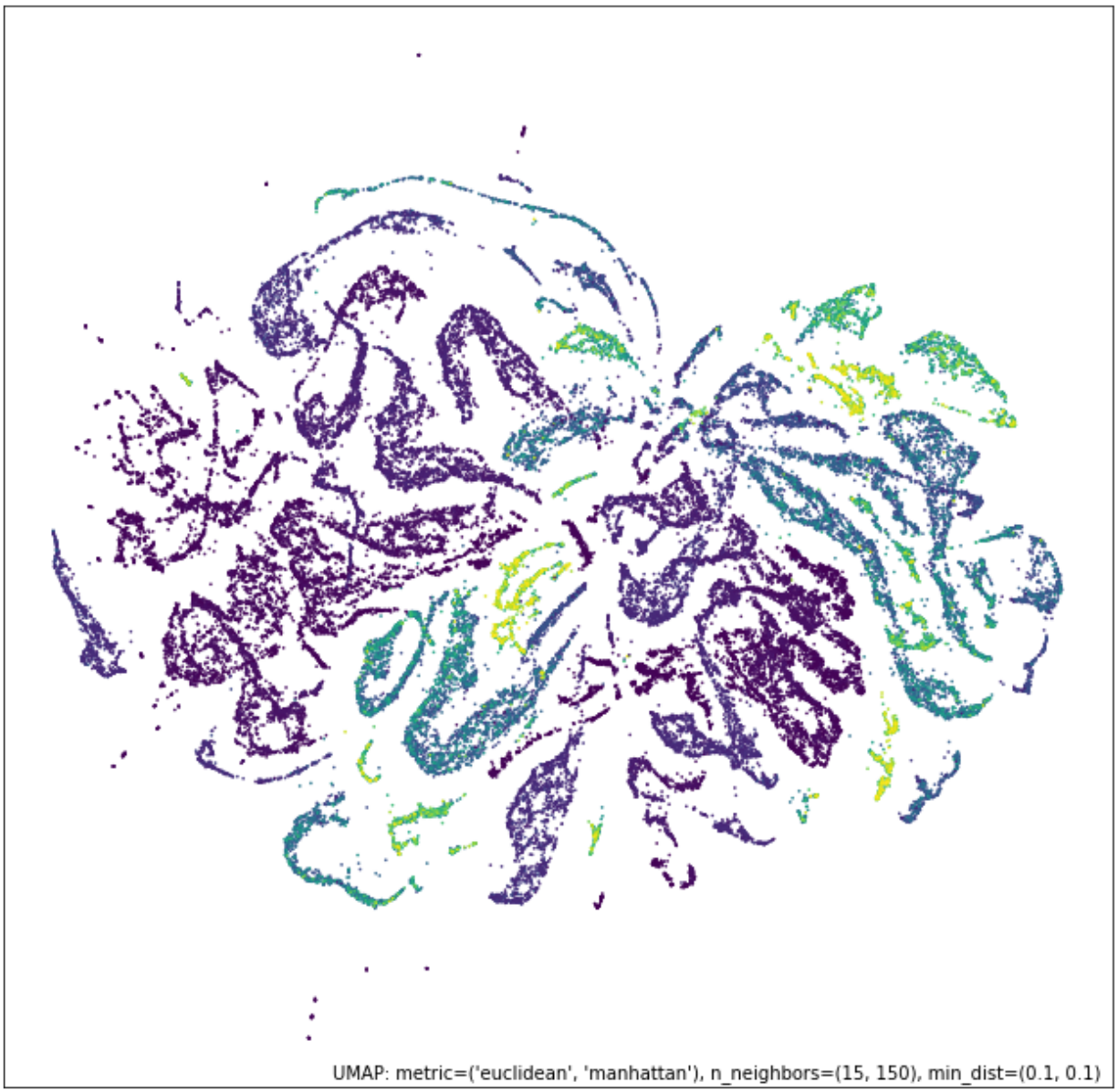
Given these radically different views of the data, what do we get if we try to integrate them together? As before we can use the intersection and union operators to simply combine the models. As noted before this is a somewhat time-consuming operation as a new 2D representation for the combined models needs to be optimized.

```
intersection_mapper = numeric_mapper * ordinal_mapper
union_mapper = numeric_mapper + ordinal_mapper
```

Let's start by looking at the intersection; here we are only really decreasing connectivity since edges are assigned the

probability of existing in *both* data views (before re-asserting local connectivity and uniform distribution assumptions).

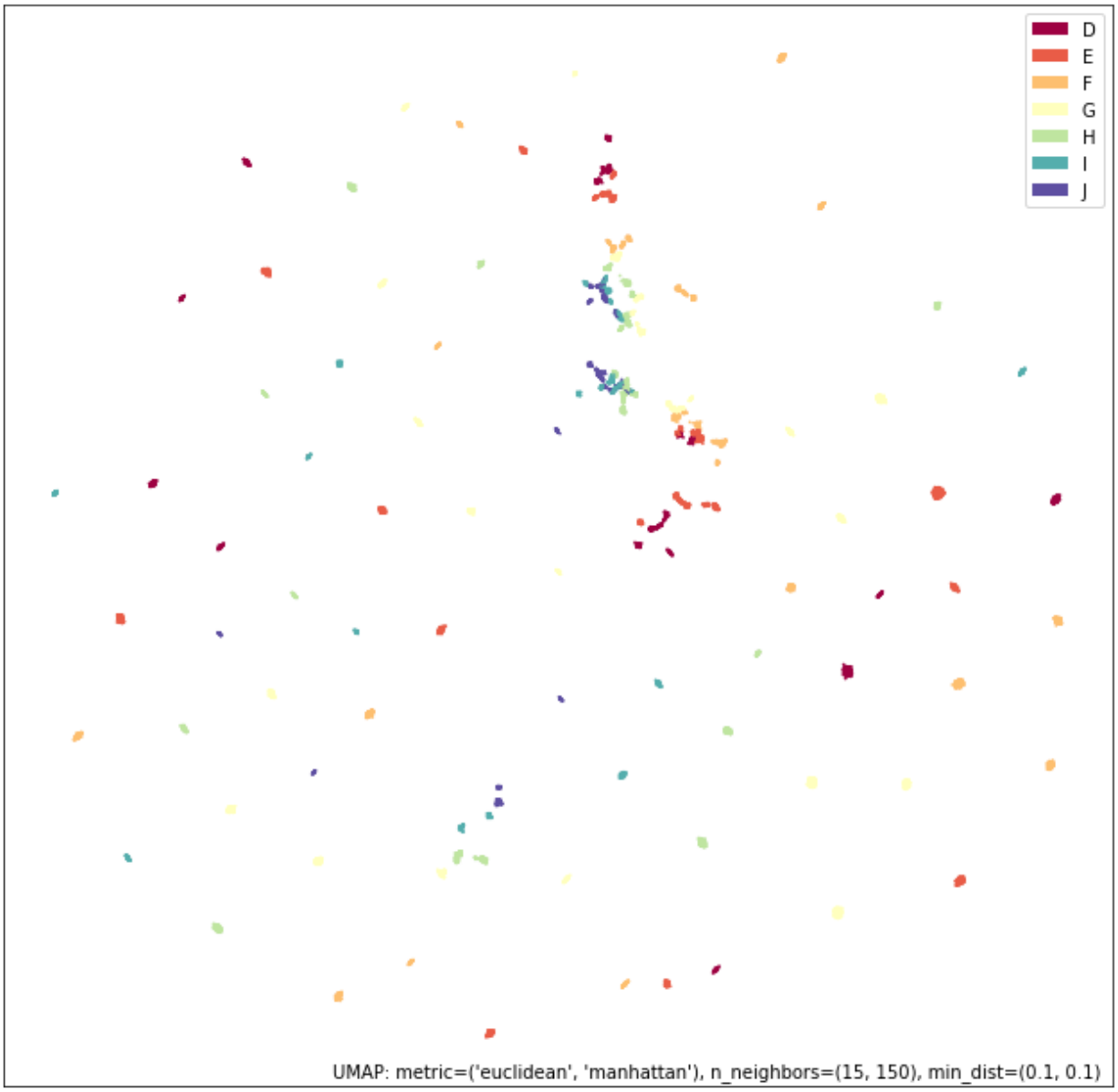
```
umap.plot.points(intersection_mapper, values=diamonds["price"], cmap="viridis")
```



What we get most closely represents the numeric data view. Why is this? Because the categorical data view has points either connected with certainty (because they are, or are nearly, identical) or very loosely. The points connected with near certainty are very dense clusters – almost points in the plot – and mostly what we are doing with the intersection is breaking up those clusters with the more fine-grained and variable connectivity provided by the numerical data. At th esame time we have shifted the result significantly from the numerical data view on its own; the categorical information has made each cluster more uniform (rather than being a gradient) in its price.

Given this result, what would you expect of the union?

```
umap.plot.points(union_mapper, labels=diamonds["color"])
```

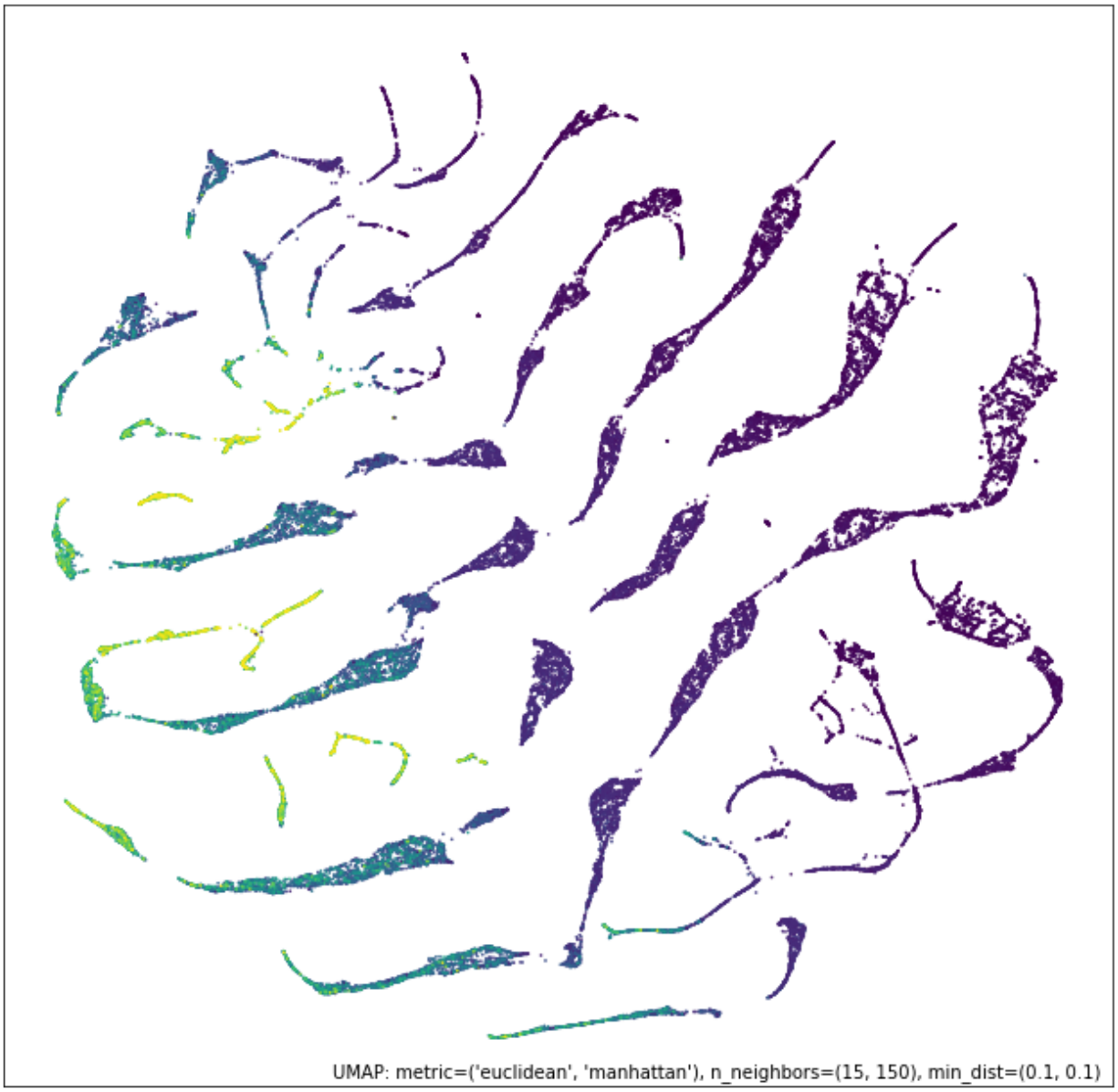


What we get in practice looks a lot more like the categorical view of the data. This time we are only *increasing* the connectivity (prior to re-asserting local connectivity and uniform distribution assumptions); thus we retain most of the structure of the high-connectivity categorical view. Note, however, that we have created more connected and coherent clusters in the center of the plot, showing a range of diamond colors, and the introduction of the numerical size and weight information has induced a rearrangement of the individual clusters around the fringes.

We can go a step further and experiment with the contrast composition method.

```
contrast_mapper = numeric_mapper - ordinal_mapper
```

```
umap.plot.points(contrast_mapper, values=diamonds["price"], cmap="viridis")
```



Here we see that we've retained a lot of the structure of the numeric data view, but have refined and broken it down further into clear clusters with price gradients running through each of them.

To further demonstrate the power of this approach we can go a step further and intersect a higher `n_neighbors` based embedding of the numeric data view with our existing union of numeric and categorical data – providing a model that is a composition of three simpler models.

```
intersect_union_mapper = umap.UMAP(random_state=42, n_neighbors=60).fit(numeric) *␣
↪union_mapper
```

```
umap.plot.points(intersect_union_mapper, values=diamonds["price"], cmap="viridis")
```

Here the greater global structure from the larger `n_neighbors` value glues together longer strands and we get an interesting result out. In this case it is not necessarily particularly informative, but it is included as a demonstration that even composed models can be composed with each other, stacking together potentially many different views.

CHAPTER 13

Better Preserving Local Density with DensMAP

A notable assumption in UMAP is that the data is uniformly distributed on some manifold and that it is ultimately this manifold that we would like to present. This is highly effective for many use cases, but it can be the case that one would like to preserve more information about the relative local density of data. A recent paper presented a technique called DensMAP the computes estimates of the local density and uses those estimates as a regularizer in the optimization of the low dimensional representation. The details are well explained in the paper and we encourage those curious about the details to read it. The result is a low dimensional representation that preserves information about the relative local density of the data. To see what this means in practice let's load some modules and try it out on some familiar data.

```
import sklearn.datasets
import umap
import umap.plot
```

For test data we will make use of the now familiar (see earlier tutorial sections) MNIST and Fashion-MNIST datasets. MNIST is a collection of 70,000 gray-scale images of hand-written digits. Fashion-MNIST is a collection of 70,000 gray-scale images of fashion items.

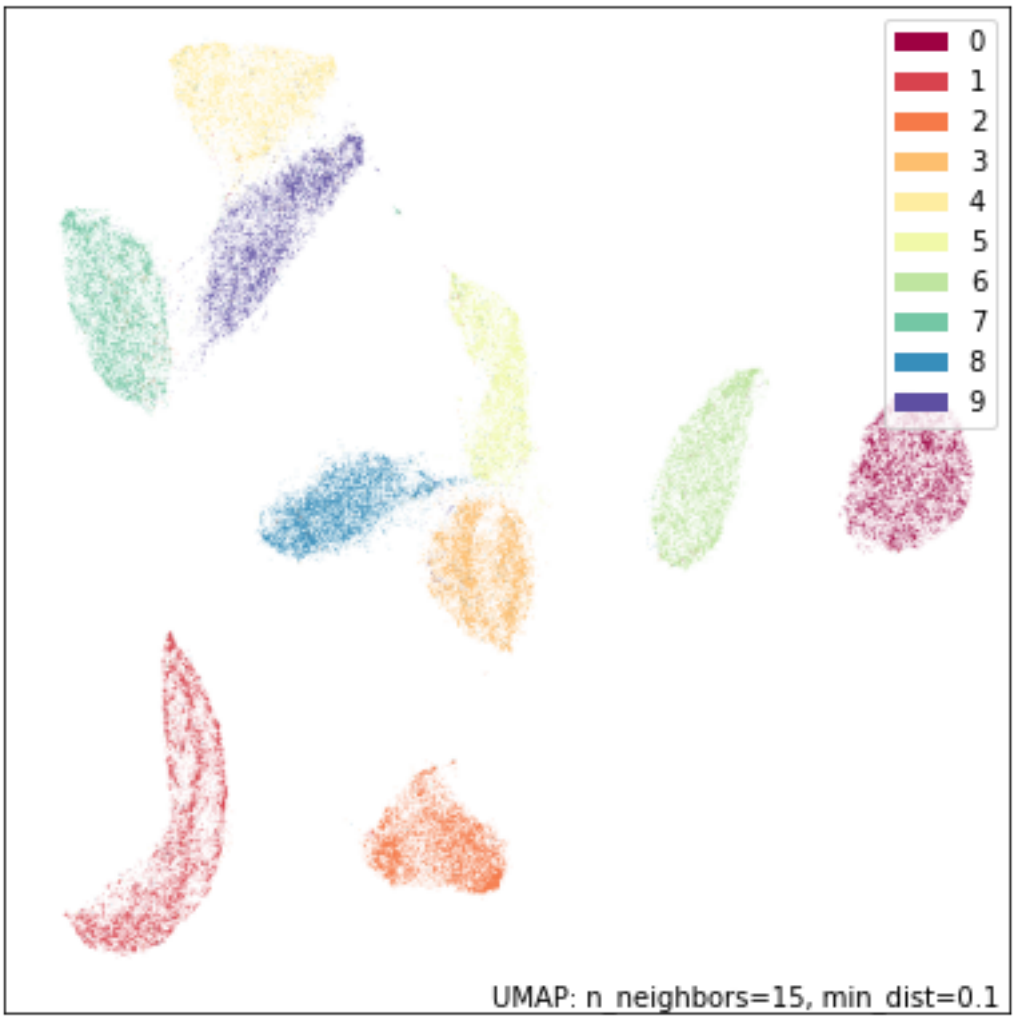
```
mnist = sklearn.datasets.fetch_openml("mnist_784")
fmnist = sklearn.datasets.fetch_openml("Fashion-MNIST")
```

Before we try out DensMAP let's run standard UMAP so we have a baseline to compare to. We'll start with MNIST digits.

```
%%time
mapper = umap.UMAP(random_state=42).fit(mnist.data)
```

```
CPU times: user 2min, sys: 15 s, total: 2min 15s
Wall time: 1min 43s
```

```
umap.plot.points(mapper, labels=mnist.target, width=500, height=500)
```



Now let's try running DensMAP instead. In practice this is as easy as adding the parameter `densmap=True` to the UMAP constructor – this will cause UMAP use use DensMAP regularization with the default DensMAP parameters.

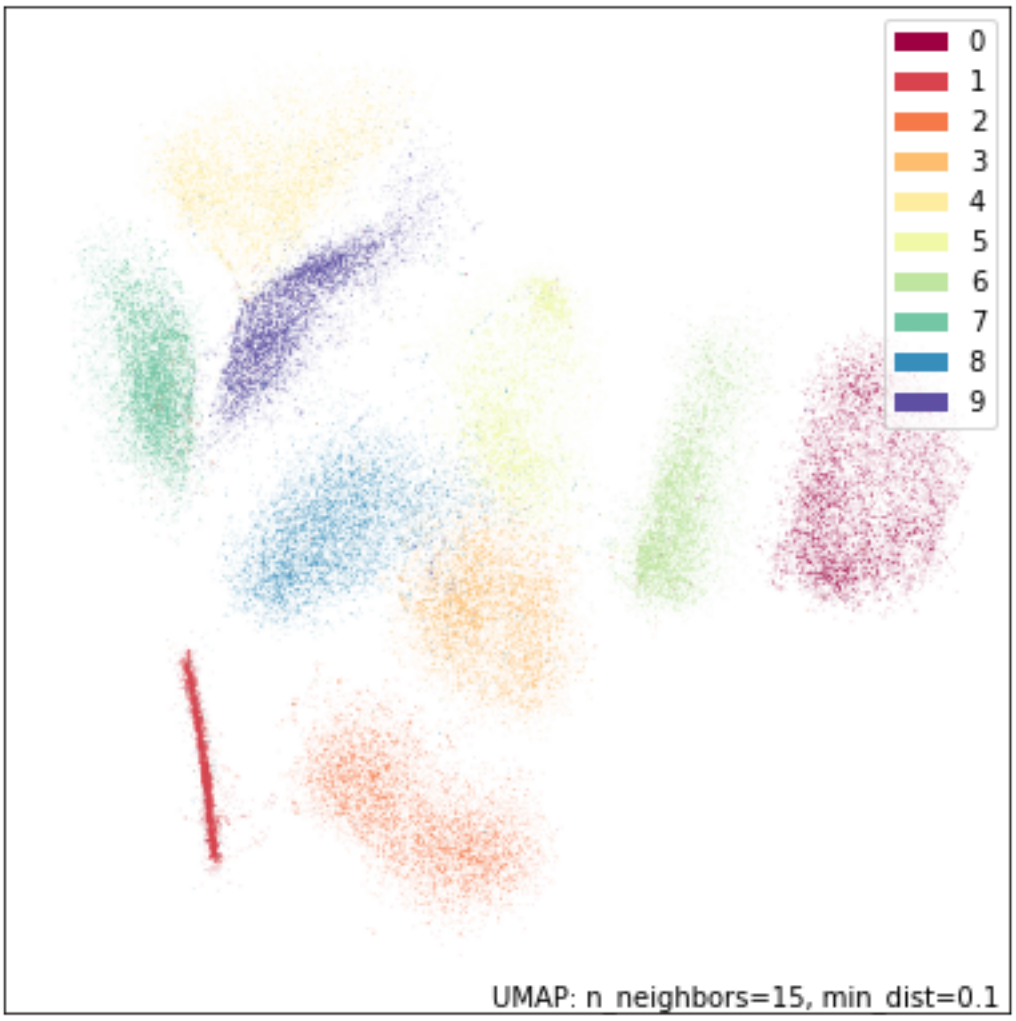
```
%%time
dens_mapper = umap.UMAP(densmap=True, random_state=42).fit(mnist.data)
```

```
CPU times: user 3min 42s, sys: 12.9 s, total: 3min 55s
Wall time: 2min 20s
```

Note that this is a little slower than standard UMAP – there is a little more work to be done. It is worth noting, however, that the DensMAP overhead is relatively constant, so the difference in runtime won't increase much as you scale out DensMAP to larger datasets.

Now let's see what sort of results we get:

```
umap.plot.points(dens_mapper, labels=mnist.target, width=500, height=500)
```



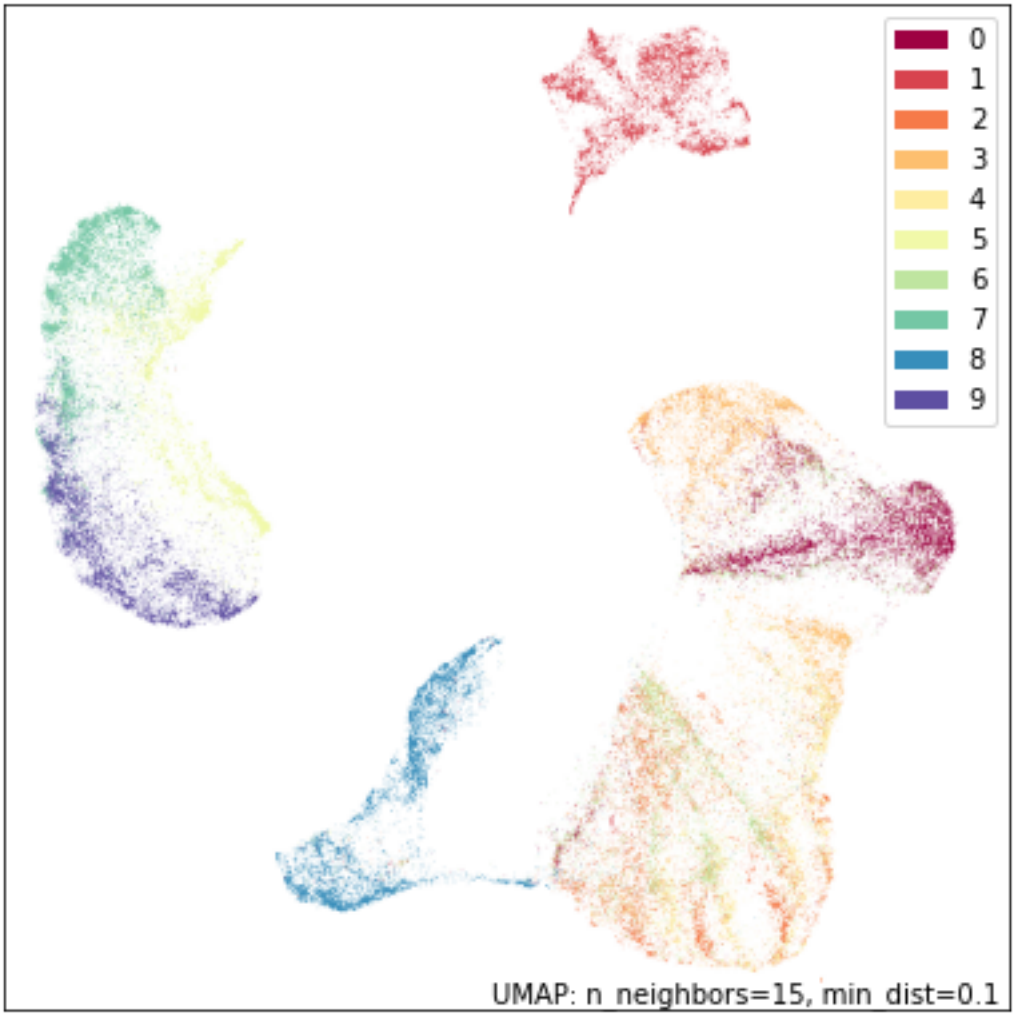
This is a significantly different result – although notably the same groupings of digits and overall structure have resulted. The most striking aspects are that the ones cluster has be compressed into a very narrow and dense stripe, while other digit clusters, most notably the zeros and the twos have expanded out to fill more space in the plot. This is due to the fact that in the high dimensional space the ones are indeed more densely packed together, with largely only variation along one dimension (the angle with which the stroke of the one is drawn). In contrast a digit like the zero has a lot more variation (rounder, narrower, taller, shorter, sloping one way or another); this results in less local density in high dimensional space, and this lack of local density has been preserved by DensMAP.

Let's no look at the Fashion-MNIST dataset; as before we'll start by reminding ourselves what the default UMAP results look like:

```
%%time
mapper = umap.UMAP(random_state=42).fit(fmnist.data)
```

```
CPU times: user 1min 6s, sys: 8.66 s, total: 1min 15s
Wall time: 49.8 s
```

```
umap.plot.points(mapper, labels=fmnist.target, width=500, height=500)
```

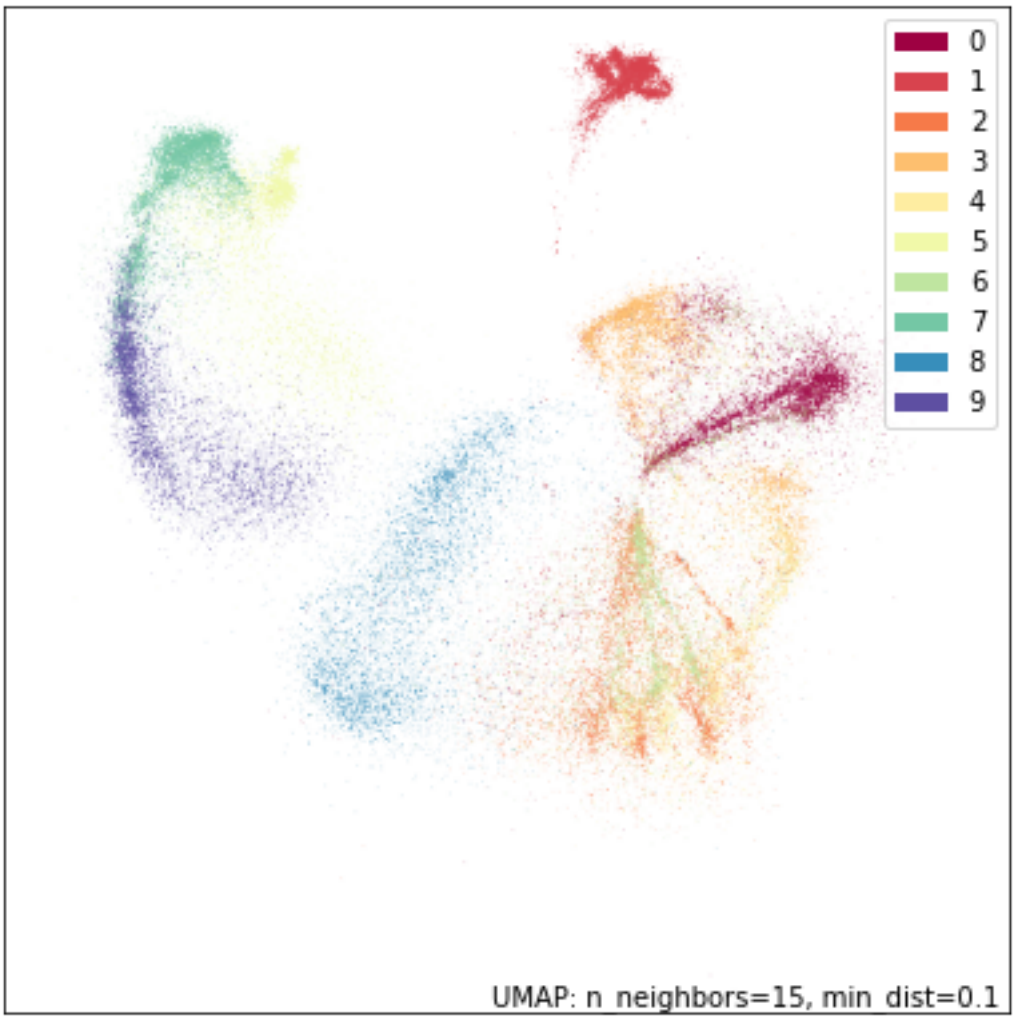


Now let's try running DensMAP. As before that is as simple as setting the `densmap=True` flag.

```
%%time
dens_mapper = umap.UMAP(densmap=True, random_state=42).fit(fmnist.data)
```

```
CPU times: user 3min 48s, sys: 8.07 s, total: 3min 56s
Wall time: 2min 21s
```

```
umap.plot.points(dens_mapper, labels=fmnist.target, width=500, height=500)
```



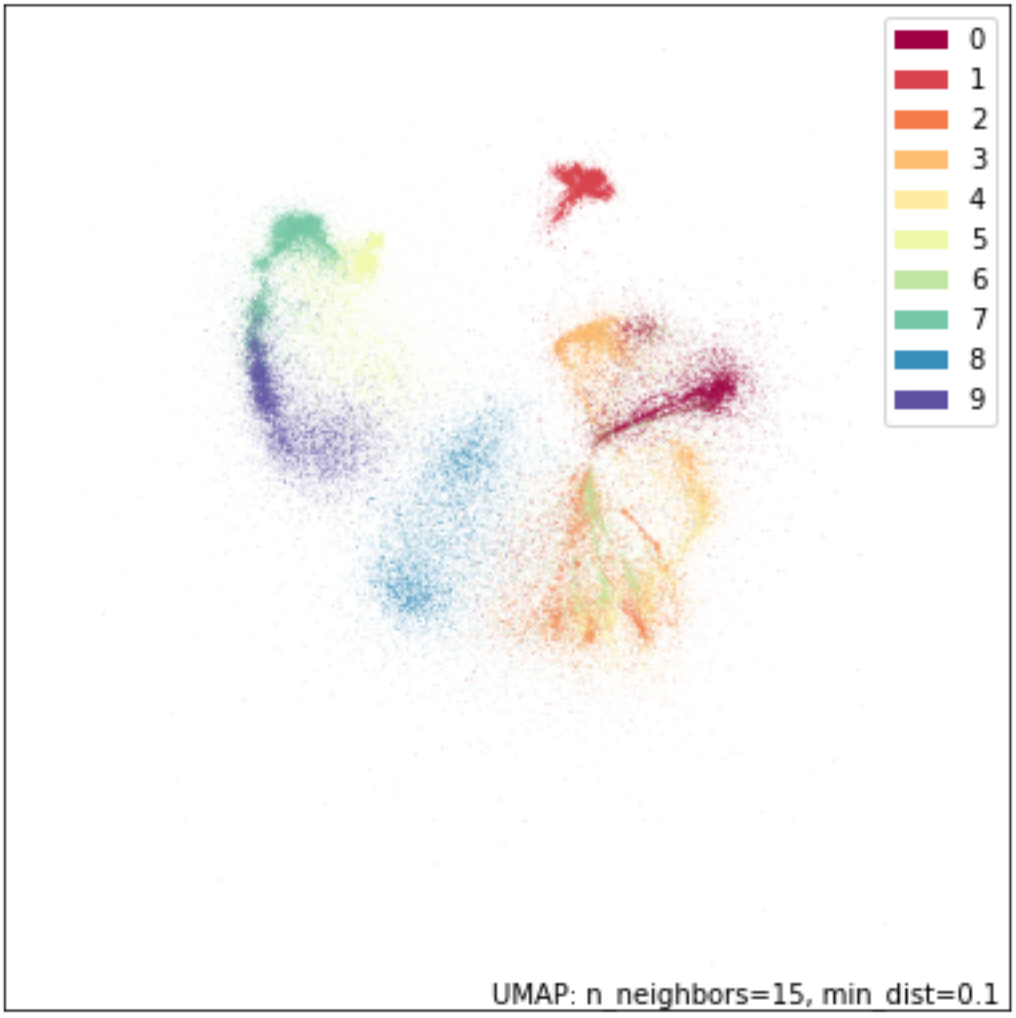
Again we see that DensMAP provides a plot similar to UMAP broadly, but with striking differences. Here we get to see that the cluster of bags (label 8 in blue) is actually quite sparse, while the cluster of pants (label 1 in red) is actually quite dense with little variation compared to other categories. We even see information internal to clusters. Consider the cluster of boots (label 9 in violet): at the top end it is quite dense, but it fades out into a much sparse region.

So far we have used DensMAP with default parameters, but the implementation provides several parameters for adjusting exactly how the local density regularisation is handled. We encourage readers to consult the paper for the details of the many parameters available. For general use the main parameter of interest is called `dens_lambda` and it controls how strongly the local density regularisation acts. Larger values of `dens_lambda` with make preserving the local density a priority over the the standard UMAP objective, while smaller values lean more towards classical UMAP. The default value is 2.0. Let's play with it a little so we can see the effects of varying it. To start we'll use a higher `dens_lambda` of 5.0:

```
%%time
dens_mapper = umap.UMAP(densmap=True, dens_lambda=5.0, random_state=42).fit(fmnist.
↪data)
```

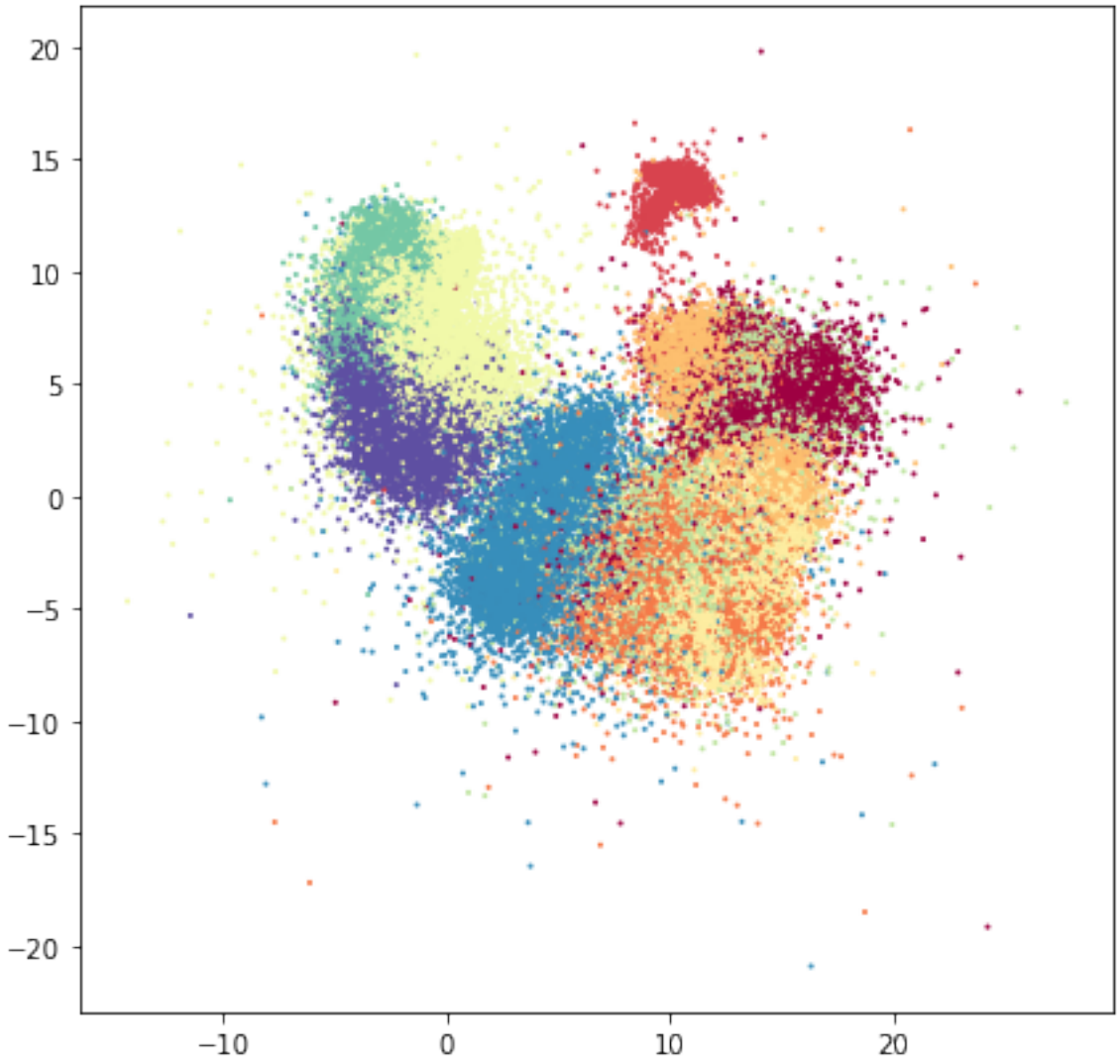
```
CPU times: user 3min 47s, sys: 5.04 s, total: 3min 52s
Wall time: 2min 18s
```

```
umap.plot.points(dens_mapper, labels=fmnist.target, width=500, height=500)
```



This looks kind of like what we had before, but blurrier. And also … smaller? The plot bounds are set by the data, so the fact that it is smaller represents the fact that there are some points right out to the edges of the plot. These are likely points that are in locally very sparse regions of the high dimensional space and are thus pushed well away from everything else. We can see this better if we use raw matplotlib and a scatter plot with larger point size:

```
fig, ax = umap.plot.plt.subplots(figsize=(7,7))
ax.scatter(*dens_mapper.embedding_.T, c=fmnist.target.astype('int8'), cmap="Spectral",
↪ s=1)
```



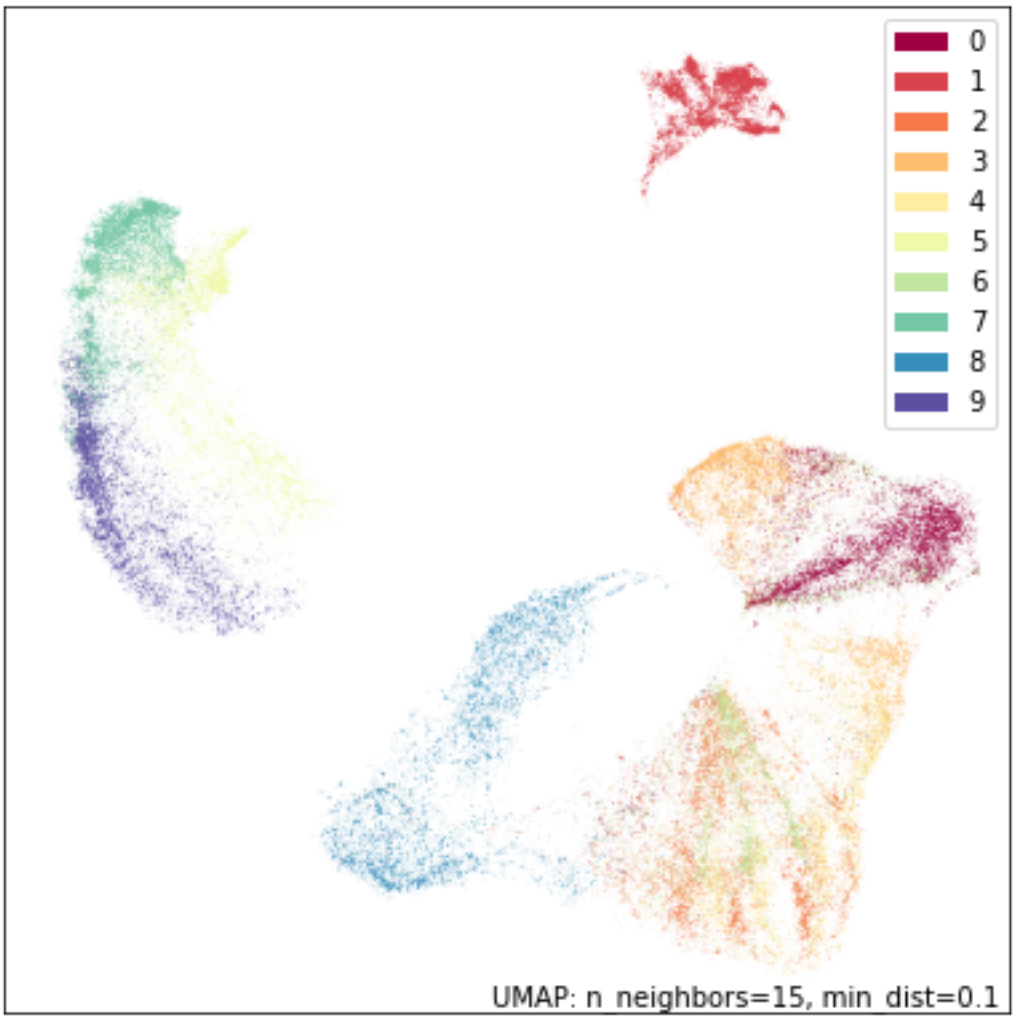
Aside from seeing the issues with overplotting we can see that there are, in fact, quite a few points that create a very soft halo of of sparse points around the fringes.

Now let's try going the other way and reduce `dens_lambda` to a small value, so that in principle we can recover something quite close to the default UMAP plot, with just a hint of local density information encoded.

```
%%time
dens_mapper = umap.UMAP(densmap=True, dens_lambda=0.1, random_state=42).fit(fmnist.
↪data)
```

```
CPU times: user 3min 47s, sys: 3.78 s, total: 3min 51s
Wall time: 2min 16s
```

```
umap.plot.points(dens_mapper, labels=fmnist.target, width=500, height=500)
```



And indeed, this looks very much like the original plot, but the bags (label 8 in blue) are slightly more diffused, and the pants (label 1 in red) are a little denser. This is very much the default UMAP with just a tweak to better reflect some notion of local density.

CHAPTER 14

Document embedding using UMAP

This is a tutorial of using UMAP to embed text (but this can be extended to any collection of tokens). We are going to use the 20 newsgroups dataset which is a collection of forum posts labelled by topic. We are going to embed these documents and see that similar documents (i.e. posts in the same subforum) will end up close together. You can use this embedding for other downstream tasks, such as visualizing your corpus, or run a clustering algorithm (e.g. HDBSCAN). We will use a bag of words model and use UMAP on the count vectors as well as the TF-IDF vectors.

To start with let's load the relevant libraries. **This requires UMAP version >= 0.4.0.**

```
import pandas as pd
import umap
import umap.plot

# Used to get the data
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer

# Some plotting libraries
import matplotlib.pyplot as plt
%matplotlib notebook
from bokeh.plotting import show, save, output_notebook, output_file
from bokeh.resources import INLINE
output_notebook(resources=INLINE)
```

Next let's download and explore the 20 newsgroups dataset.

```
%%time
dataset = fetch_20newsgroups(subset='all',
                             shuffle=True, random_state=42)
```

```
CPU times: user 280 ms, sys: 52 ms, total: 332 ms
Wall time: 460 ms
```

Let's see the size of the corpus:

```
print(f'{len(dataset.data)} documents')
print(f'{len(dataset.target_names)} categories')
```

```
18846 documents
20 categories
```

Here are the categories of documents. As you can see many are related to one another (e.g. 'comp.sys.ibm.pc.hardware' and 'comp.sys.mac.hardware') but they are not all correlated (e.g. 'sci.med' and 'rec.sport.baseball').

```
dataset.target_names
```

```
['alt.atheism',
 'comp.graphics',
 'comp.os.ms-windows.misc',
 'comp.sys.ibm.pc.hardware',
 'comp.sys.mac.hardware',
 'comp.windows.x',
 'misc.forsale',
 'rec.autos',
 'rec.motorcycles',
 'rec.sport.baseball',
 'rec.sport.hockey',
 'sci.crypt',
 'sci.electronics',
 'sci.med',
 'sci.space',
 'soc.religion.christian',
 'talk.politics.guns',
 'talk.politics.mideast',
 'talk.politics.misc',
 'talk.religion.misc']
```

Let's look at a couple of sample documents:

```
for idx, document in enumerate(dataset.data[:3]):
    category = dataset.target_names[dataset.target[idx]]

    print(f'Category: {category}')
    print('----------------------------')
    # Print the first 500 characters of the post
    print(document[:500])
    print('----------------------------')
```

```
Category: rec.sport.hockey
----------------------------
From: Mamatha Devineni Ratnam <mr47+@andrew.cmu.edu>
Subject: Pens fans reactions
Organization: Post Office, Carnegie Mellon, Pittsburgh, PA
Lines: 12
NNTP-Posting-Host: po4.andrew.cmu.edu



I am sure some bashers of Pens fans are pretty confused about the lack
of any kind of posts about the recent Pens massacre of the Devils. Actually,
```

(continues on next page)

(continued from previous page)

```
I am  bit puzzled too and a bit relieved. However, I am going to put an end
to non-PIttsburghers' relief with a bit of praise for the Pens. Man, they
are killin
----------------------------
Category: comp.sys.ibm.pc.hardware
----------------------------
From: mblawson@midway.ecn.uoknor.edu (Matthew B Lawson)
Subject: Which high-performance VLB video card?
Summary: Seek recommendations for VLB video card
Nntp-Posting-Host: midway.ecn.uoknor.edu
Organization: Engineering Computer Network, University of Oklahoma, Norman, OK, USA
Keywords: orchid, stealth, vlb
Lines: 21

  My brother is in the market for a high-performance video card that supports
VESA local bus with 1-2MB RAM.  Does anyone have suggestions/ideas on:

  - Diamond Stealth Pro Local
----------------------------
Category: talk.politics.mideast
----------------------------
From: hilmi-er@dsv.su.se (Hilmi Eren)
Subject: Re: ARMENIA SAYS IT COULD SHOOT DOWN TURKISH PLANES (Henrik)
Lines: 95
Nntp-Posting-Host: viktoria.dsv.su.se
Reply-To: hilmi-er@dsv.su.se (Hilmi Eren)
Organization: Dept. of Computer and Systems Sciences, Stockholm University




|>The student of "regional killings" alias Davidian (not the Davidian religios sect)␣
↪writes:


|>Greater Armenia would stretch from Karabakh, to the Black Sea, to the
|>Mediterranean, so if you use the term "Greater Armenia
----------------------------
```

Now we will create a dataframe with the target labels to be used in plotting. This will allow us to see the newsgroup when we hover over the plotted points (if using interactive plotting). This will help us evaluate (by eye) how good the embedding looks.

```
category_labels = [dataset.target_names[x] for x in dataset.target]
hover_df = pd.DataFrame(category_labels, columns=['category'])
```

14.1 Using raw counts

Next, we are going to use a bag-of-words approach (i.e. word order doesn't matter) and construct a word document matrix. In this matrix the rows will correspond to a document (i.e. post) and each column will correspond to a particular word. The values will be the counts of how many times a given word appeared in a particular document.

We will use sklearns CountVectorizer function to do this for us along with a couple other preprocessing steps:

1) Split the text into tokens (i.e. words) by splitting on whitespace

2) Remove english stopwords (the, and, etc)

3) Remove all words which occur less than 5 times in the entire corpus (via the min_df parameter)

```
vectorizer = CountVectorizer(min_df=5, stop_words='english')
word_doc_matrix = vectorizer.fit_transform(dataset.data)
```

This gives us a 18846x34880 matrix where there are 18846 documents (same as above) and 34880 unique tokens. This matrix is sparse since most words do not appear in most documents.

```
word_doc_matrix
```

```
<18846x34880 sparse matrix of type '<class 'numpy.int64'>'
    with 1939023 stored elements in Compressed Sparse Row format>
```

Now we are going to do dimension reduction using UMAP to reduce the matrix from 34880 dimensions to 2 dimensions (since n_components=2). We need a distance metric and will use Hellinger distance which measures the similarity between two probability distributions. Each document has a set of counts generated by a multinomial distribution where we can use Hellinger distance to measure the similarity of these distributions.

```
%%time
embedding = umap.UMAP(n_components=2, metric='hellinger').fit(word_doc_matrix)
```

```
CPU times: user 2min 24s, sys: 1.18 s, total: 2min 25s
Wall time: 2min 3s
```

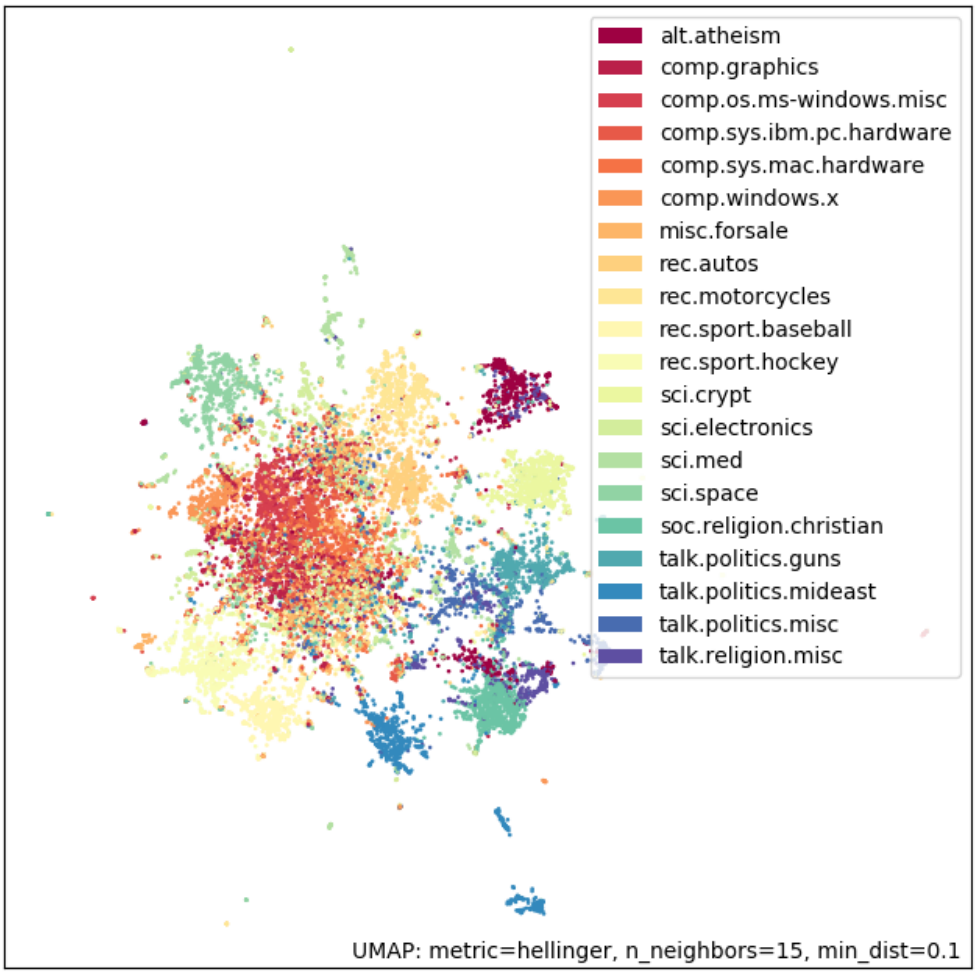
Now we have an embedding of 18846x2.

```
embedding.embedding_.shape
```

```
(18846, 2)
```

Let's plot the embedding. If you are running this in a notebook, you should use the interactive plotting method as it lets you hover over your points and see what category they belong to.

```
# For interactive plotting use
# f = umap.plot.interactive(embedding, labels=dataset.target, hover_data=hover_df,␣
↪point_size=1)
# show(f)
f = umap.plot.points(embedding, labels=hover_df['category'])
```



As you can see this does reasonably well. There is some separation and groups that you would expect to be similar (such as 'rec.sport.baseball' and 'rec.sport.hockey') are close together. The big clump in the middle corresponds to a lot of extremely similar newsgroups like 'comp.sys.ibm.pc.hardware' and 'comp.sys.mac.hardware'.

14.2 Using TF-IDF

We will now do the same pipeline with the only change being the use of TF-IDF weighting. TF-IDF gives less weight to words that appear frequently across a large number of documents since they are more popular in general. It asserts a higher weight to words that appear frequently in a smaller subset of documents since they are probably important words for those documents.

To do the TF-IDF weighting we will use sklearns TfidfVectorizer with the same parameters as CountVectorizer above.

```
tfidf_vectorizer = TfidfVectorizer(min_df=5, stop_words='english')
tfidf_word_doc_matrix = tfidf_vectorizer.fit_transform(dataset.data)
```

We get a matrix of the same size as before:

```
tfidf_word_doc_matrix
```

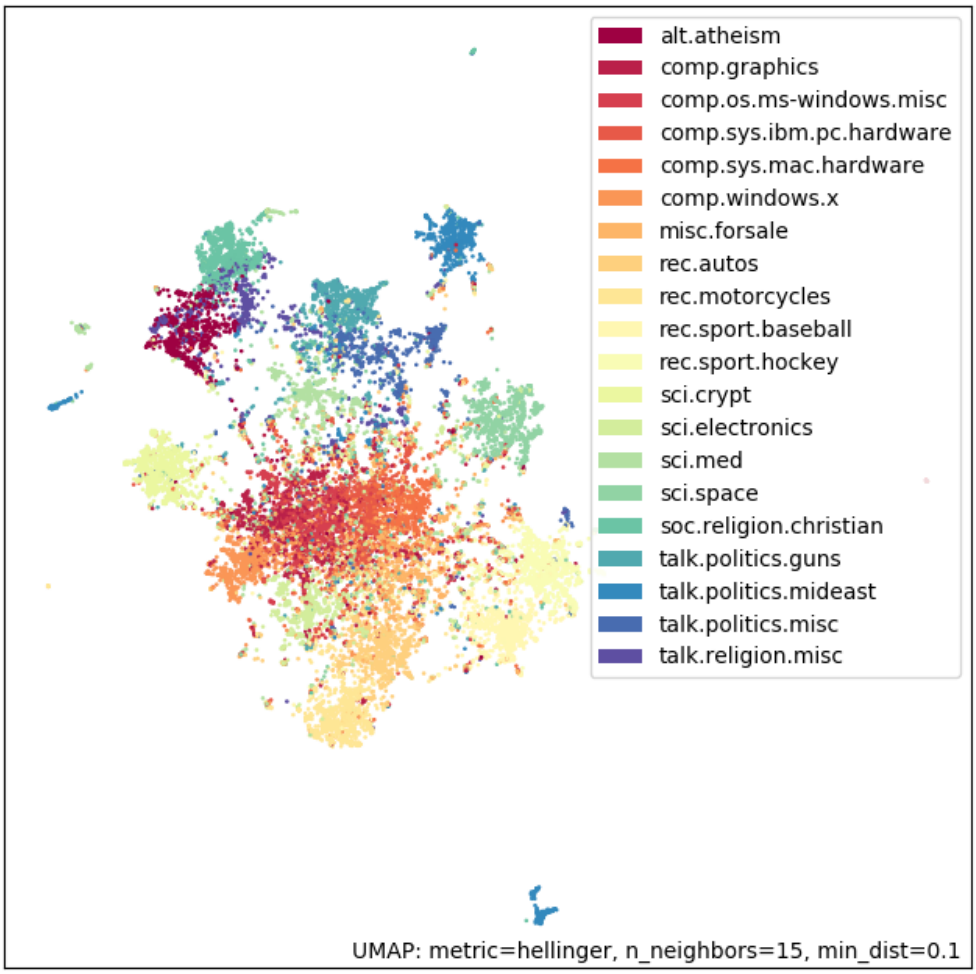
```
<18846x34880 sparse matrix of type '<class 'numpy.float64'>'
    with 1939023 stored elements in Compressed Sparse Row format>
```

Again we use Hellinger distance and UMAP to embed the documents

```
%%time
tfidf_embedding = umap.UMAP(metric='hellinger').fit(tfidf_word_doc_matrix)
```

```
CPU times: user 2min 19s, sys: 1.27 s, total: 2min 20s
Wall time: 1min 57s
```

```
# For interactive plotting use
# fig = umap.plot.interactive(tfidf_embedding, labels=dataset.target, hover_
↪data=hover_df, point_size=1)
# show(fig)
fig = umap.plot.points(tfidf_embedding, labels=hover_df['category'])
```



The results look fairly similar to before but this can be a useful trick to have in your toolbox.

14.3 Potential applications

Now that we have an embedding, what can we do with it?

- Explore/visualize your corpus to identify topics/trends
- Cluster the embedding to find groups of related documents
- Look for nearest neighbours to find related documents
- Look for anomalous documents

CHAPTER 15

Embedding to non-Euclidean spaces

By default UMAP embeds data into Euclidean space. For 2D visualization that means that data is embedded into a 2D plane suitable for a scatterplot. In practice, however, there aren't really any major constraints that prevent the algorithm from working with other more interesting embedding spaces. In this tutorial we'll look at how to get UMAP to embed into other spaces, how to embed into your own custom space, and why this sort of approach might be useful.

To start we'll load the usual selection of libraries. In this case we will not be using the `umap.plot` functionality, but working with matplotlib directly since we'll be generating some custom visualizations for some of the more unique embedding spaces.

```
import numpy as np
import numba
import sklearn.datasets
import matplotlib.pyplot as plt
import seaborn as sns
from mpl_toolkits.mplot3d import Axes3D
import umap
%matplotlib inline
```

```
sns.set(style='white', rc={'figure.figsize':(10,10)})
```

As a test dataset we'll use the PenDigits dataset from sklearn – embedding into exotic spaces can be considerably more computationally taxing, so a simple relatively small dataset is going to be useful.

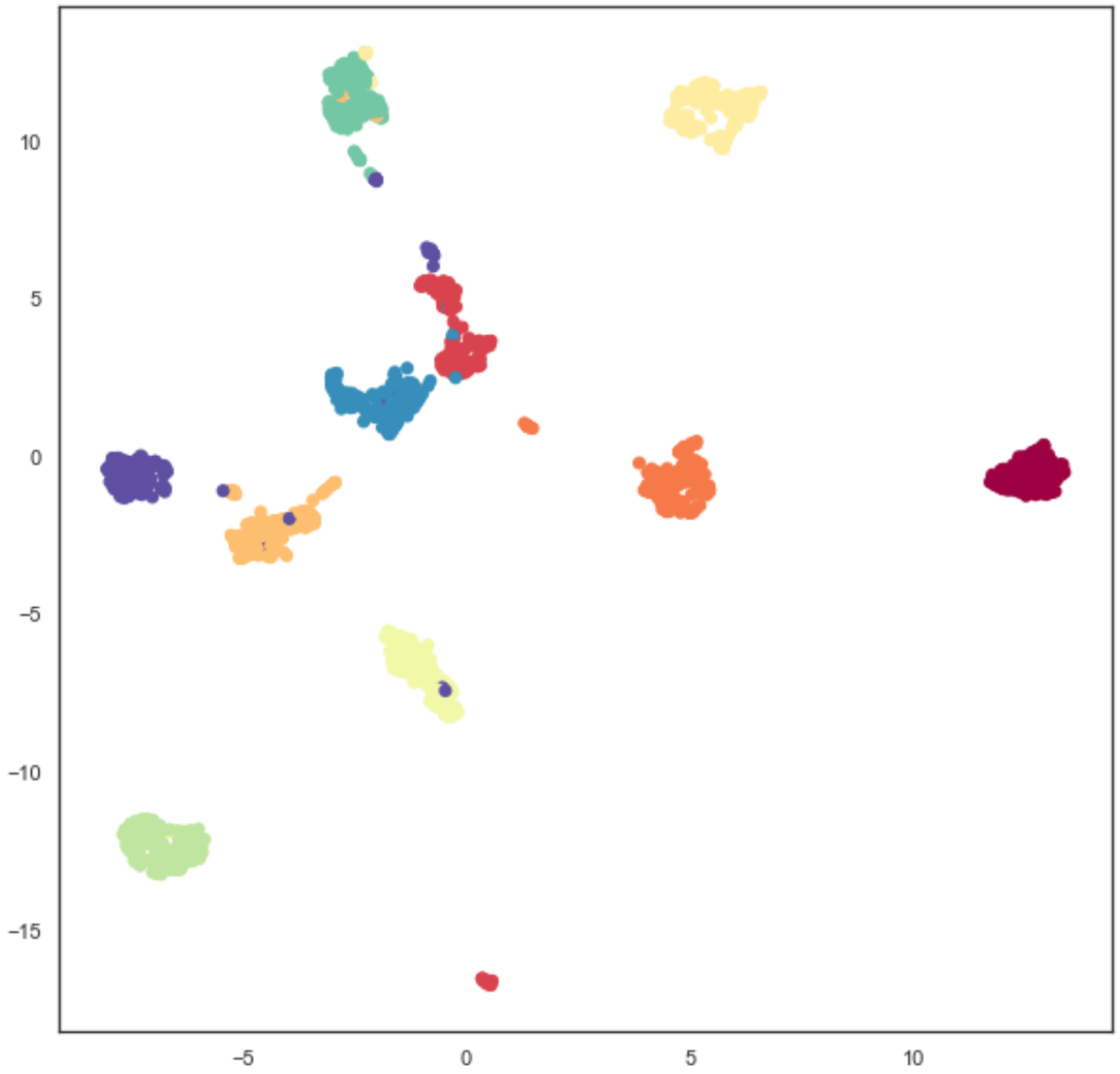
```
digits = sklearn.datasets.load_digits()
```

15.1 Plane embeddings

Plain old plane embeddings are simple enough – it is the default for UMAP. Here we'll run through the example again, just to ensure you are familiar with how this works, and what the result of a UMAP embedding of the PenDigits dataset looks like in the simple case of embedding in the plane.

```
plane_mapper = umap.UMAP(random_state=42).fit(digits.data)
```

```
plt.scatter(plane_mapper.embedding_.T[0], plane_mapper.embedding_.T[1], c=digits.
↪target, cmap='Spectral')
```



15.2 Spherical embeddings

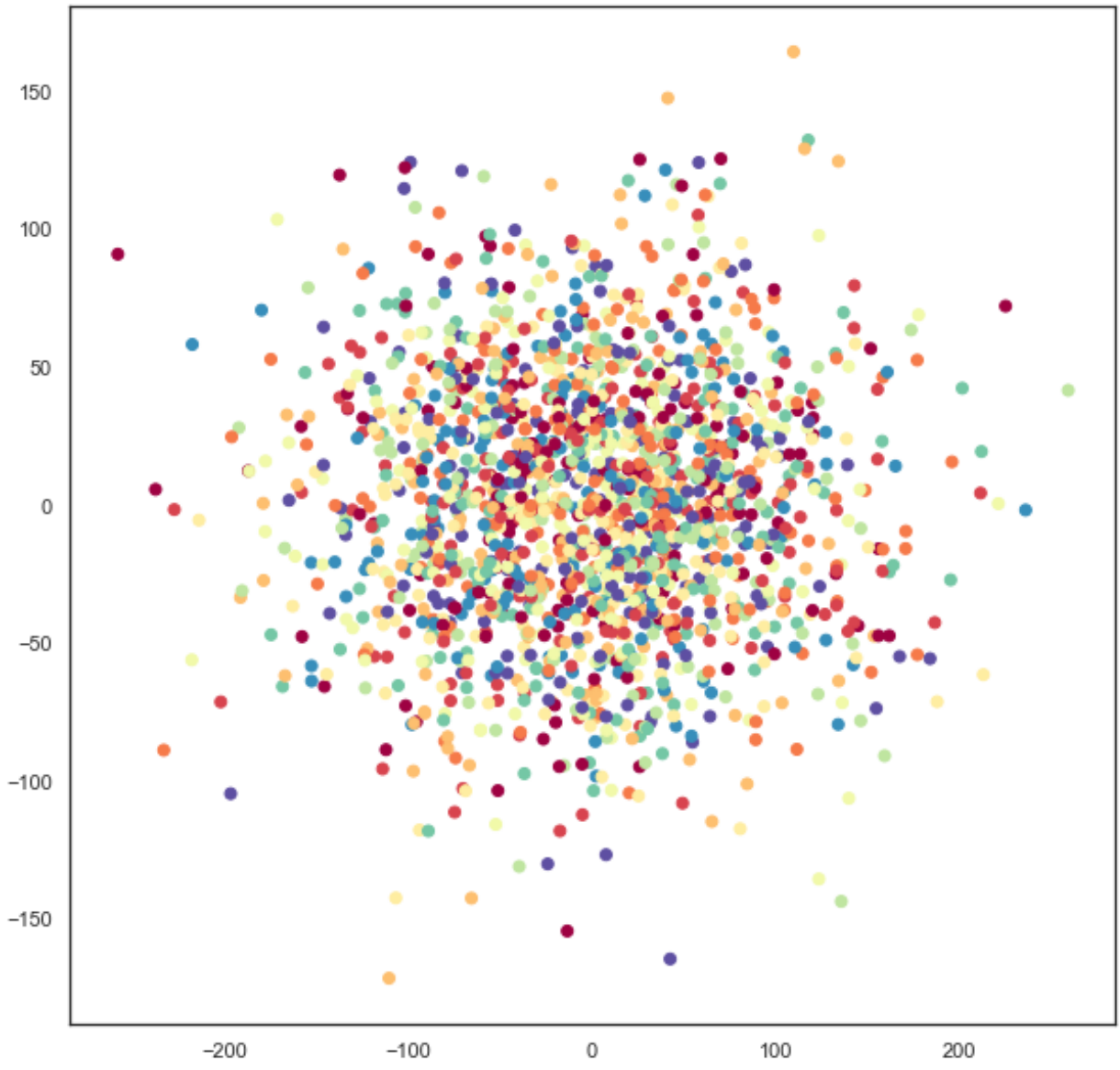
What if we wanted to embed data onto a sphere rather than a plane? This might make sense, for example, if we have reason to expect some sort of periodic behaviour or other reasons to expect that no point can be infinitely far from any other. To make UMAP embed onto a sphere we need to make use of the `output_metric` parameter, which specifies what metric to use for the **output** space. By default UMAP uses a Euclidean `output_metric` (and even has a special faster code-path for this case), but you can pass in other metrics. Among the metrics UMAP supports is the Haversine metric, used for measuring distances on a sphere, given in latitude and longitude (in radians). If we set

the `output_metric` to `"haversine"` then UMAP will use that to measure distance in the embedding space.

```
sphere_mapper = umap.UMAP(output_metric='haversine', random_state=42).fit(digits.data)
```

The result is the pendigits data embedded with respect to haversine distance on a sphere. The catch is that if we visualize this naively then we will get nonsense.

```
plt.scatter(sphere_mapper.embedding_.T[0], sphere_mapper.embedding_.T[1], c=digits.
↪target, cmap='Spectral')
```



What has gone astray is that under the embedding distance metric a point at $(0, \pi)$ is distance zero from a point at $(0, 3\pi)$ since that will wrap all the way around the equator. You'll note that the scales on the x and y axes of the above plot go well outside the ranges $(-\pi, \pi)$ and $(0, 2\pi)$, so this isn't the right representation of the data. We can, however, use straightforward formulas to map this data onto a sphere embedded in 3d-space.

```
x = np.sin(sphere_mapper.embedding_[:, 0]) * np.cos(sphere_mapper.embedding_[:, 1])
```

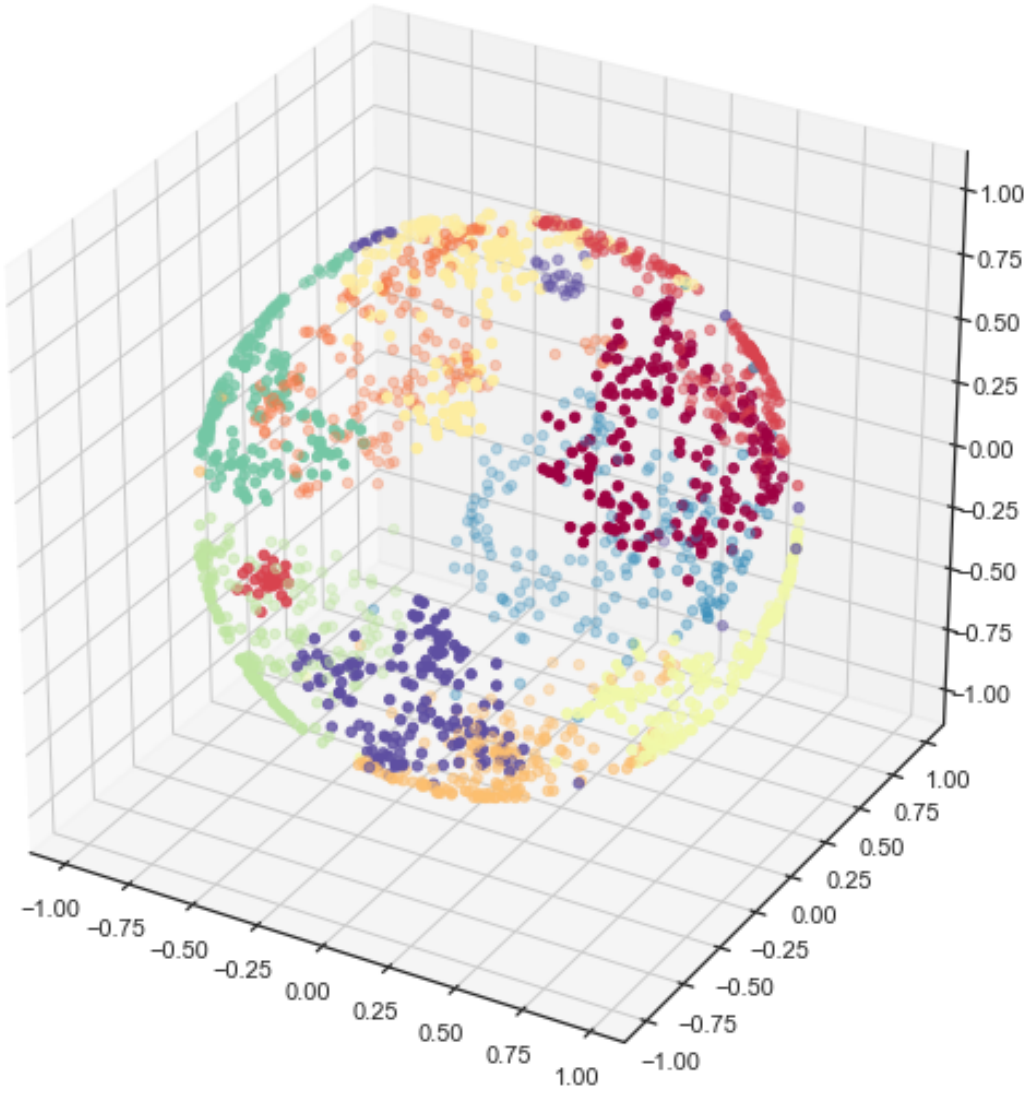
(continues on next page)

(continued from previous page)

```
y = np.sin(sphere_mapper.embedding_[:, 0]) * np.sin(sphere_mapper.embedding_[:, 1])
z = np.cos(sphere_mapper.embedding_[:, 0])
```

Now `x`, `y`, and `z` give 3d coordinates for each embedding point that lies on the surface of a sphere. We can visualize this using matplotlib's 3d plotting capabilities, and see that we have in fact induced a quite reasonable embedding of the data onto the surface of a sphere.

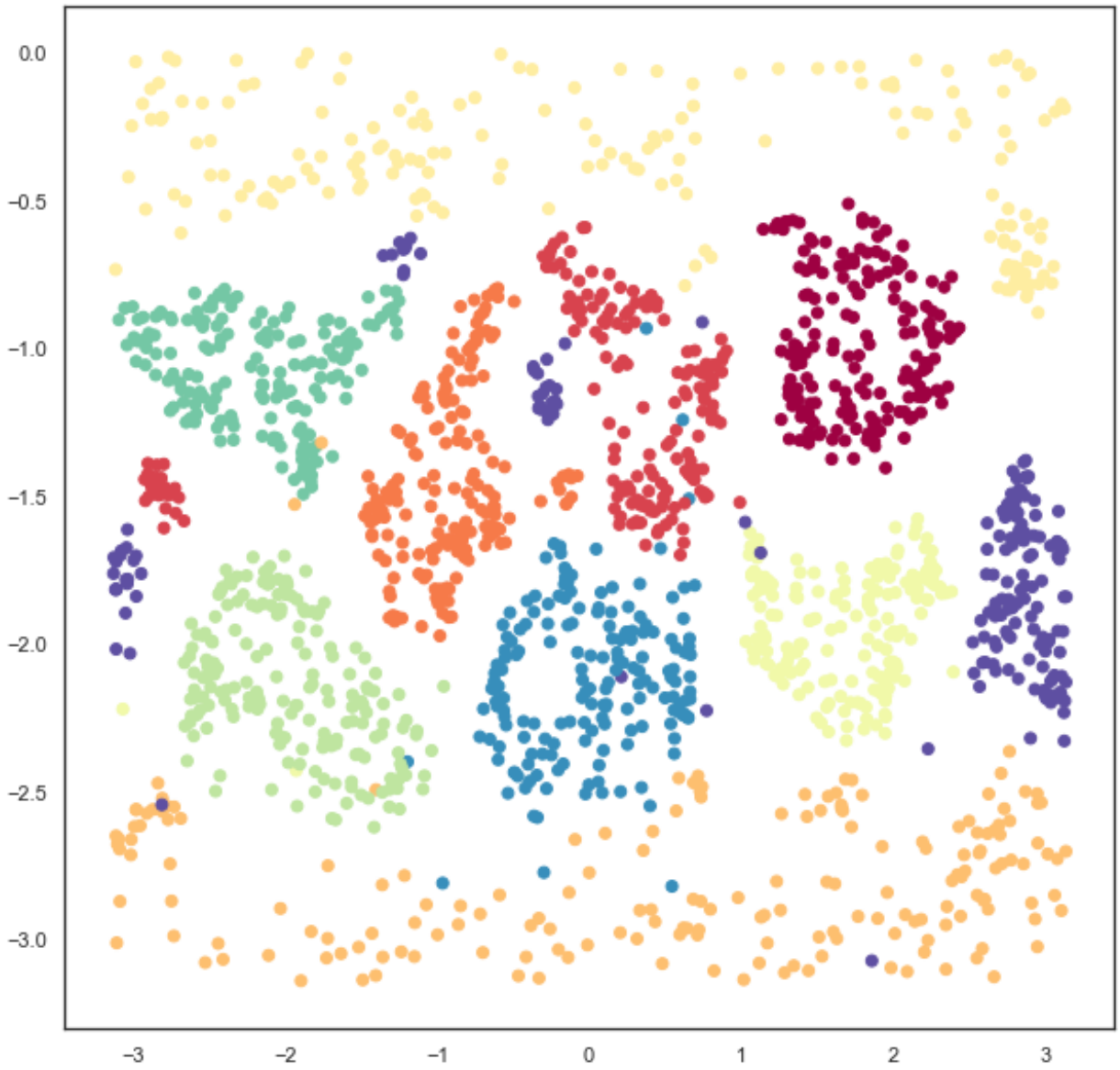
```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(x, y, z, c=digits.target, cmap='Spectral')
```



If you prefer a 2d plot we can convert these into lat/long coordinates in the appropriate ranges and get the equivalent of a map projection of the sphere data.

```
x = np.arctan2(x, y)
y = -np.arccos(z)
```

```
plt.scatter(x, y, c=digits.target.astype(np.int32), cmap='Spectral')
```



15.3 Embedding on a Custom Metric Space

What if you have some other custom notion of a metric space that you would like to embed data into? In the same way that UMAP can support custom written distance metrics for the input data (as long as they can be compiled with numba), the `output_metric` parameter can accept custom distance functions. One catch is that, to support gradient descent optimization, the distance function needs to return both the distance, and a vector for the gradient of the distance. This latter point may require a little bit of calculus on the users part. A second catch is that it is highly beneficial to parameterize the embedding space in a way that has no coordinate constraints – otherwise the gradient

descent may step a point outside the embedding space, resulting in bad things happening. This is why, for example, the sphere example simply has points wrap around rather than constraining coordinates to be in the appropriate ranges.

Let's work through an example where we construct a distance metric and gradient for a different sort of space: a torus. A torus is essentially just the outer surface of a donut. We can parameterize the torus in terms of x, y coordinates with the caveat that we can "wrap around" (similar to the sphere). In such a model distances are mostly just euclidean distances, we just have to check for which is the shorter direction – across or wrapping around – and ensure we account for the equivalence of wrapping around several times. We can write a simple function to calculate that.

```
@numba.njit(fastmath=True)
def torus_euclidean_grad(x, y, torus_dimensions=(2*np.pi,2*np.pi)):
    """Standard euclidean distance.

    ..math::
        D(x, y) = \sqrt{\sum_i (x_i - y_i)^2}
    """
    distance_sqr = 0.0
    g = np.zeros_like(x)
    for i in range(x.shape[0]):
        a = abs(x[i] - y[i])
        if 2*a < torus_dimensions[i]:
            distance_sqr += a ** 2
            g[i] = (x[i] - y[i])
        else:
            distance_sqr += (torus_dimensions[i]-a) ** 2
            g[i] = (x[i] - y[i]) * (a - torus_dimensions[i]) / a
    distance = np.sqrt(distance_sqr)
    return distance, g/(1e-6 + distance)
```

Note that the gradient just derives from the standard euclidean gradient, we just have to check the direction according to the way we've wrapped around to compute the distance. We can now plug that function directly in to the `output_metric` parameter and end up embedding data on a torus.

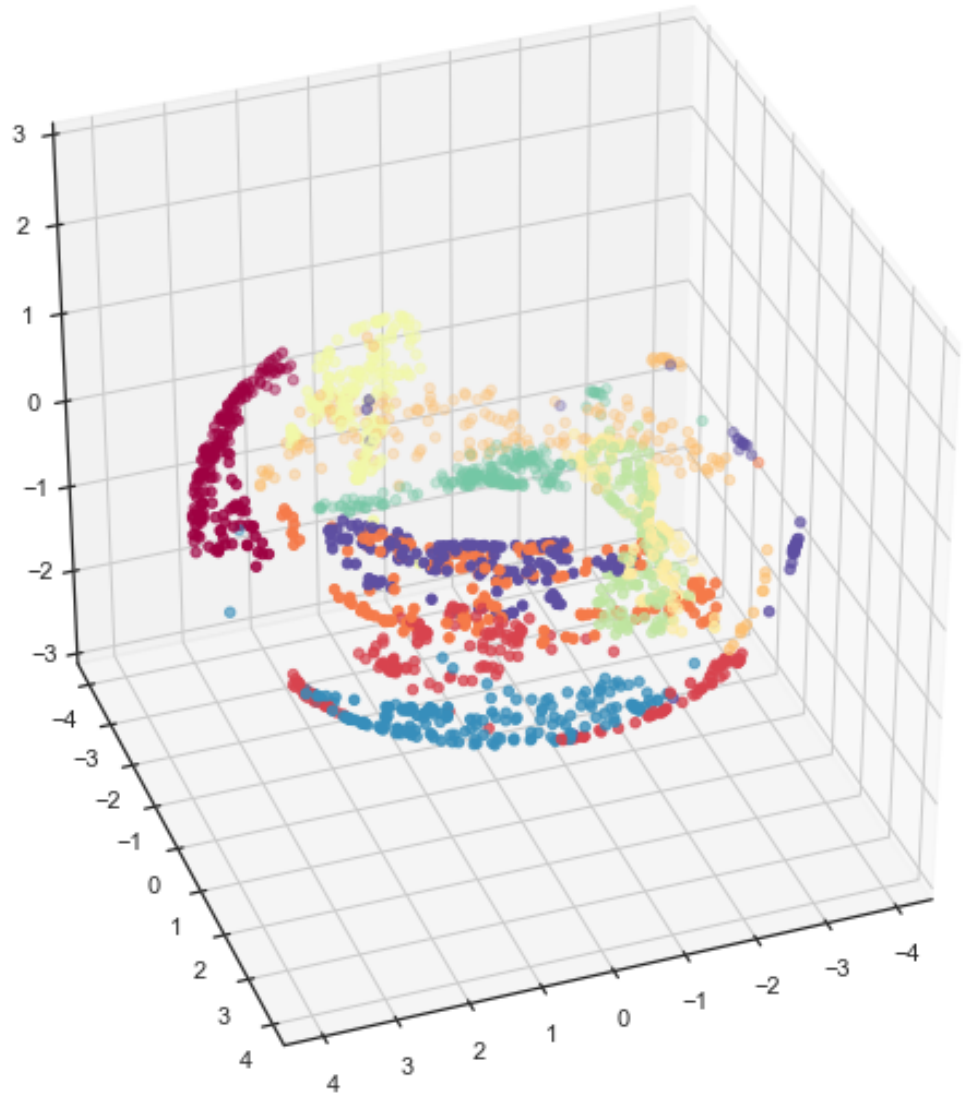
```
torus_mapper = umap.UMAP(output_metric=torus_euclidean_grad, random_state=42).
↪fit(digits.data)
```

As with the sphere case, a naive visualisation will look strange, due the the wrapping around and equivalence of looping several times. But, also just like the torus, we can construct a suitable visualization by computing the 3d coordinates for the points using a little bit of straightforward geometry (yes, I still had to look it up to check).

```
R = 3 # Size of the doughnut circle
r = 1 # Size of the doughnut cross-section

x = (R + r * np.cos(torus_mapper.embedding_[:, 0])) * np.cos(torus_mapper.embedding_
↪[:, 1])
y = (R + r * np.cos(torus_mapper.embedding_[:, 0])) * np.sin(torus_mapper.embedding_
↪[:, 1])
z = r * np.sin(torus_mapper.embedding_[:, 0])
```

Now we can visualize the result using matplotlib and see that, indeed, the data has been suitably embedded onto a torus.

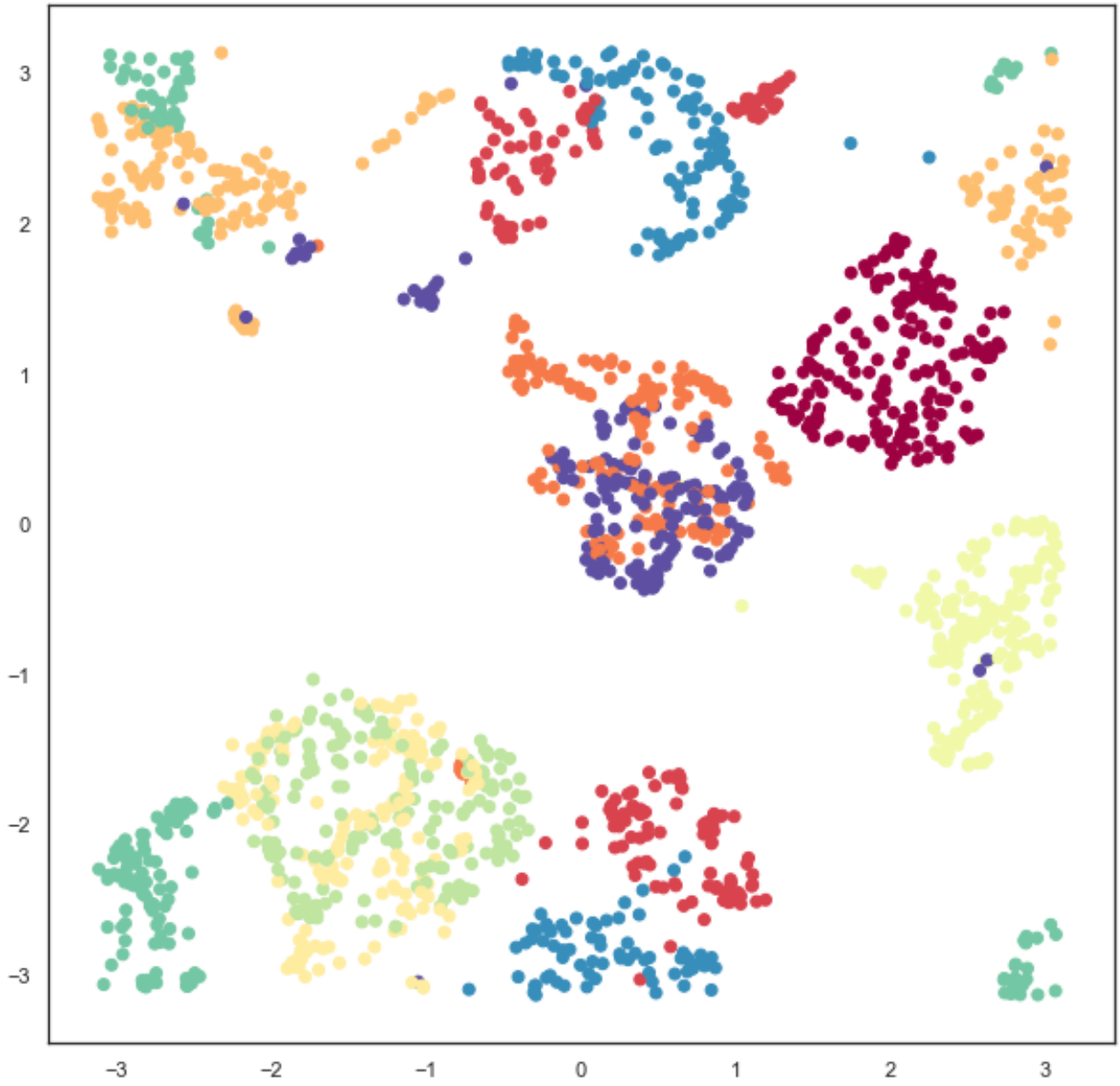
```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(x, y, z, c=digits.target, cmap='Spectral')
ax.set_zlim3d(-3, 3)
ax.view_init(35, 70)
```



And as with the torus we can do a little geometry and unwrap the torus into a flat plane with the appropriate bounds.

```
u = np.arctan2(x,y)
v = np.arctan2(np.sqrt(x**2 + y**2) - R, z)
```

```
plt.scatter(u, v, c=digits.target, cmap='Spectral')
```



15.4 A Practical Example

While the examples given so far may have some use (because some data does have suitable periodic or looping structures that we expect will be better represented in a sphere or a torus), most data doesn't really fall in the realm of something that a user can, apriori, expect to lie on an exotic manifold. Are there more practical uses for the ability to embed in other spaces? It turns out that there are. One interesting example to consider is the space formed by 2d-Gaussian distributions. We can measure the distance between two Gaussians (parameterized by a 2d vector for the mean, and 2x2 matrix giving the covariance) by the negative log of the inner product between the PDFs (since this has a nice closed form solution, and is reasonably computable). That gives us a metric space to embed into where samples are represented not as points in 2d, but as Gaussian distributions in 2d, encoding some uncertainty in how each sample in the high dimensional space is to be embedded.

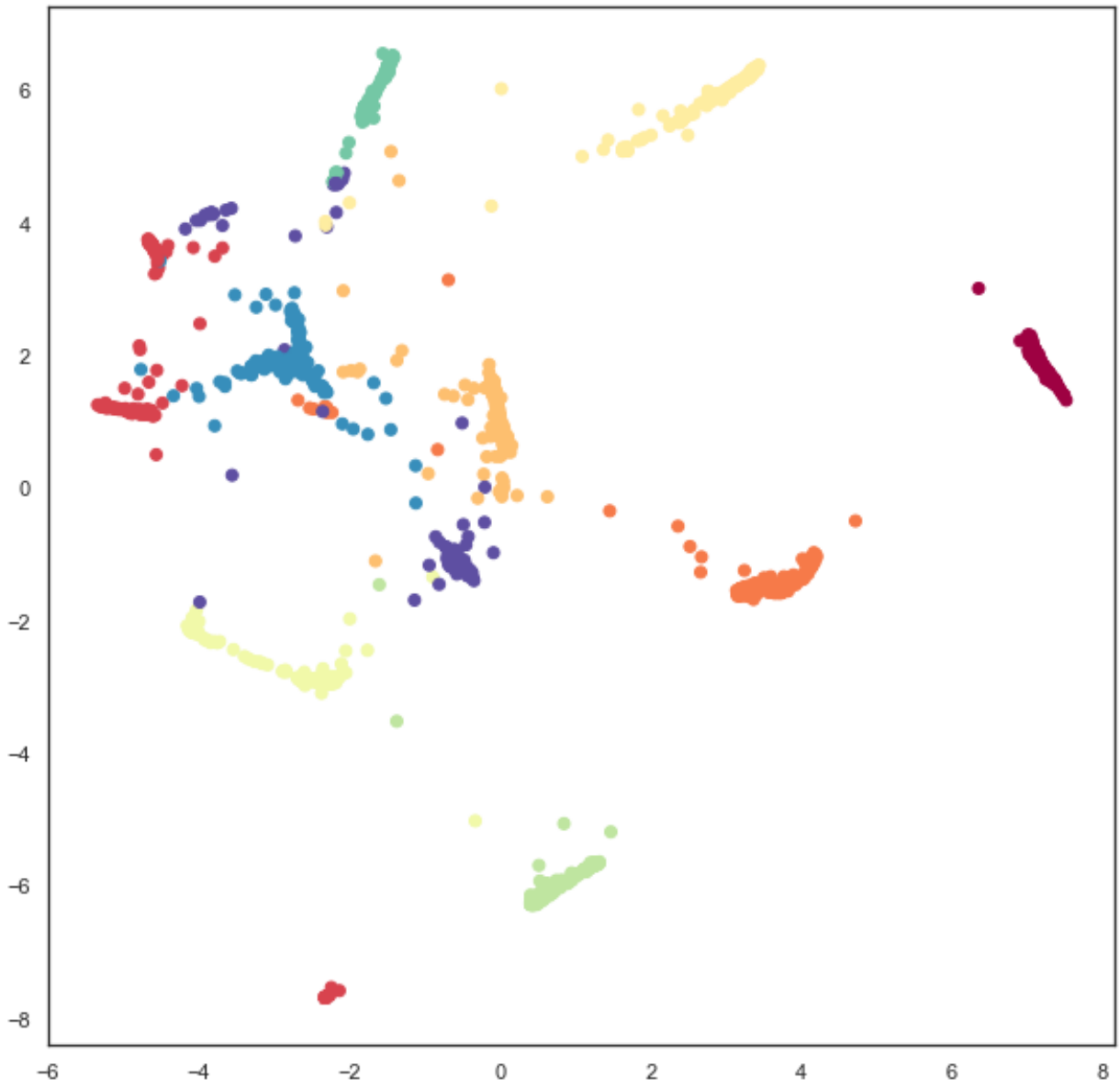
Of course we still have the issues of parameterizations that are suitable for SGD – requiring that the covariance matrix be symmetric and positive definite is challenging. Instead we can parameterize the covariance in terms of a width,

height and angle, and recover the covariance matrix from these if required. That gives us a total of 5 components to embed into (two for the mean, 3 for parameters describing the covariance). We can simply do this since the appropriate metric is defined already. Note that we have to specifically pass `n_components=5` since we need to explicitly embed into a 5 dimensional space to support all the covariance parameters associated to 2d Gaussians.

```
gaussian_mapper = umap.UMAP(output_metric='gaussian_energy',
                            n_components=5,
                            random_state=42).fit(digits.data)
```

Since we have embedded the data into a 5 dimensional space visualization is not as trivial as it was earlier. We can get a start on visualizing the results by looking at just the means, which are the 2d locations of the modes of the Gaussians. A traditional scatter plot will suffice for this.

```
plt.scatter(gaussian_mapper.embedding_.T[0], gaussian_mapper.embedding_.T[1],␣
↪c=digits.target, cmap='Spectral')
```

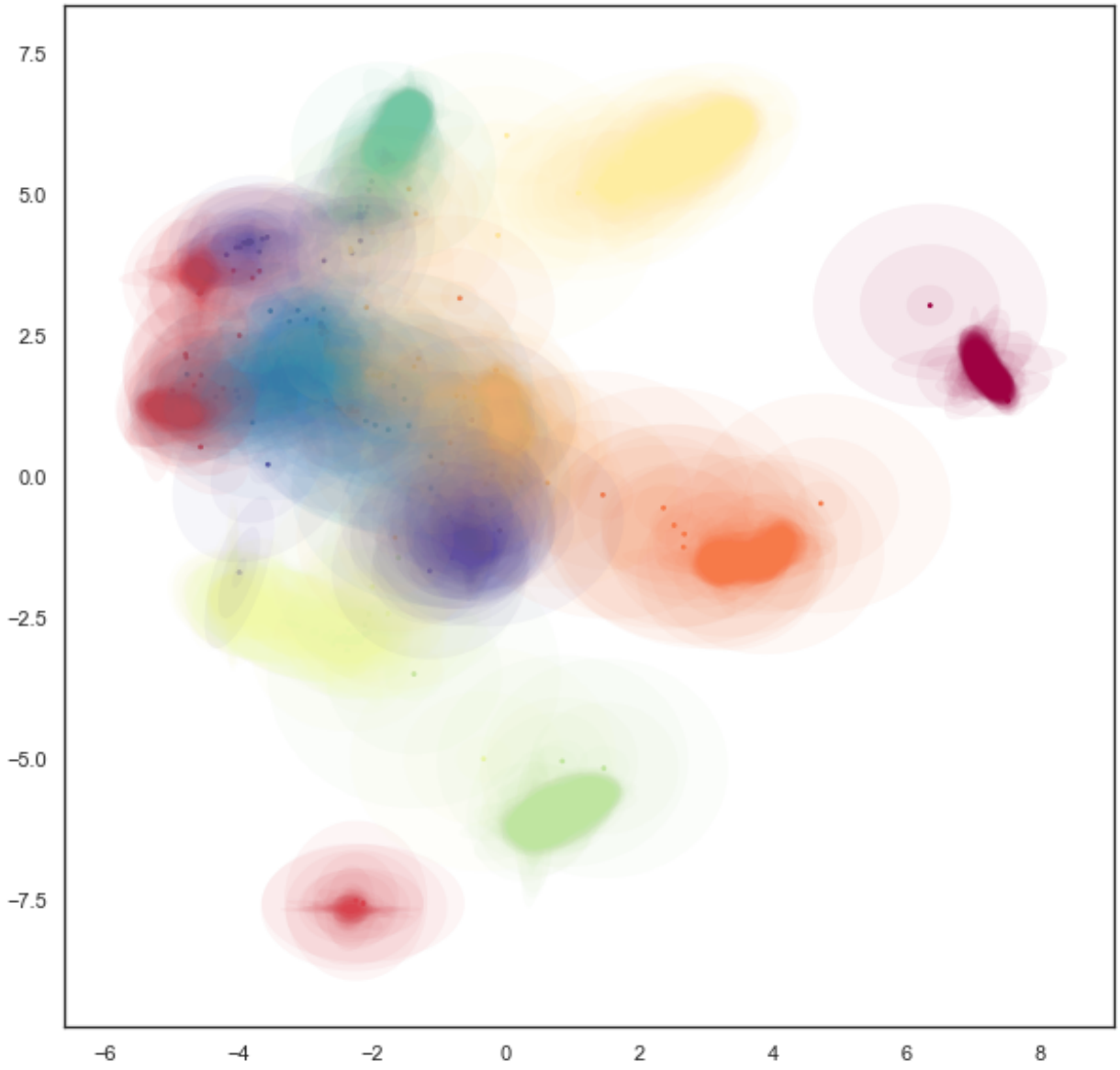


We see that we have gotten a result similar to a standard embedding into euclidean space, but with less clear clustering, and more points between clusters. To get a clearer idea of what is going on it will be necessary to devise a means to display some of the extra information contained in the extra 3 dimensions providing covariance data. To do this it will be helpful to be able to draw ellipses corresponding to super-level sets of the PDF of the 2d Gaussian. We can start on this by writing a simple function to draw ellipses on a plot accoriding to a position, a width, a height, and an angle (since this is the format the embedding computed the data).

```
from matplotlib.patches import Ellipse

def draw_simple_ellipse(position, width, height, angle,
                        ax=None, from_size=0.1, to_size=0.5, n_ellipses=3,
                        alpha=0.1, color=None,
                        **kwargs):
    ax = ax or plt.gca()
    angle = (angle / np.pi) * 180
    width, height = np.sqrt(width), np.sqrt(height)
    # Draw the Ellipse
    for nsig in np.linspace(from_size, to_size, n_ellipses):
        ax.add_patch(Ellipse(position, nsig * width, nsig * height,
                             angle, alpha=alpha, lw=0, color=color, **kwargs))
```

Now we can plot the data by providing a scatterplot of the centers (as before), but overlaying that over a super-level-set ellipses of the associated Gaussians. The obvious catch is that this will induce a lot of over-plotting, but it will at least provide a way to start understanding the embedding we have produced.

```
fig = plt.figure(figsize=(10,10))
ax = fig.add_subplot(111)
colors = plt.get_cmap('Spectral')(np.linspace(0, 1, 10))
for i in range(gaussian_mapper.embedding_.shape[0]):
    pos = gaussian_mapper.embedding_[i, :2]
    draw_simple_ellipse(pos, gaussian_mapper.embedding_[i, 2],
                        gaussian_mapper.embedding_[i, 3],
                        gaussian_mapper.embedding_[i, 4],
                        ax, color=colors[digits.target[i]],
                        from_size=0.2, to_size=1.0, alpha=0.05)
ax.scatter(gaussian_mapper.embedding_.T[0],
           gaussian_mapper.embedding_.T[1],
           c=digits.target, cmap='Spectral', s=3)
```



Now we can see that the covariance structure for the points can vary greatly, both in absolute size, and in shape. We note that many of the points falling between clusters have much larger variances, in a sense representing the greater uncertainty of the location of the embedding. It is also worth noting that the shape of the ellipses can vary significantly – there are several very stretched ellipses, quite distinct from many of the very round ellipses; in a sense this represents where the uncertainty falls more along a single line for example.

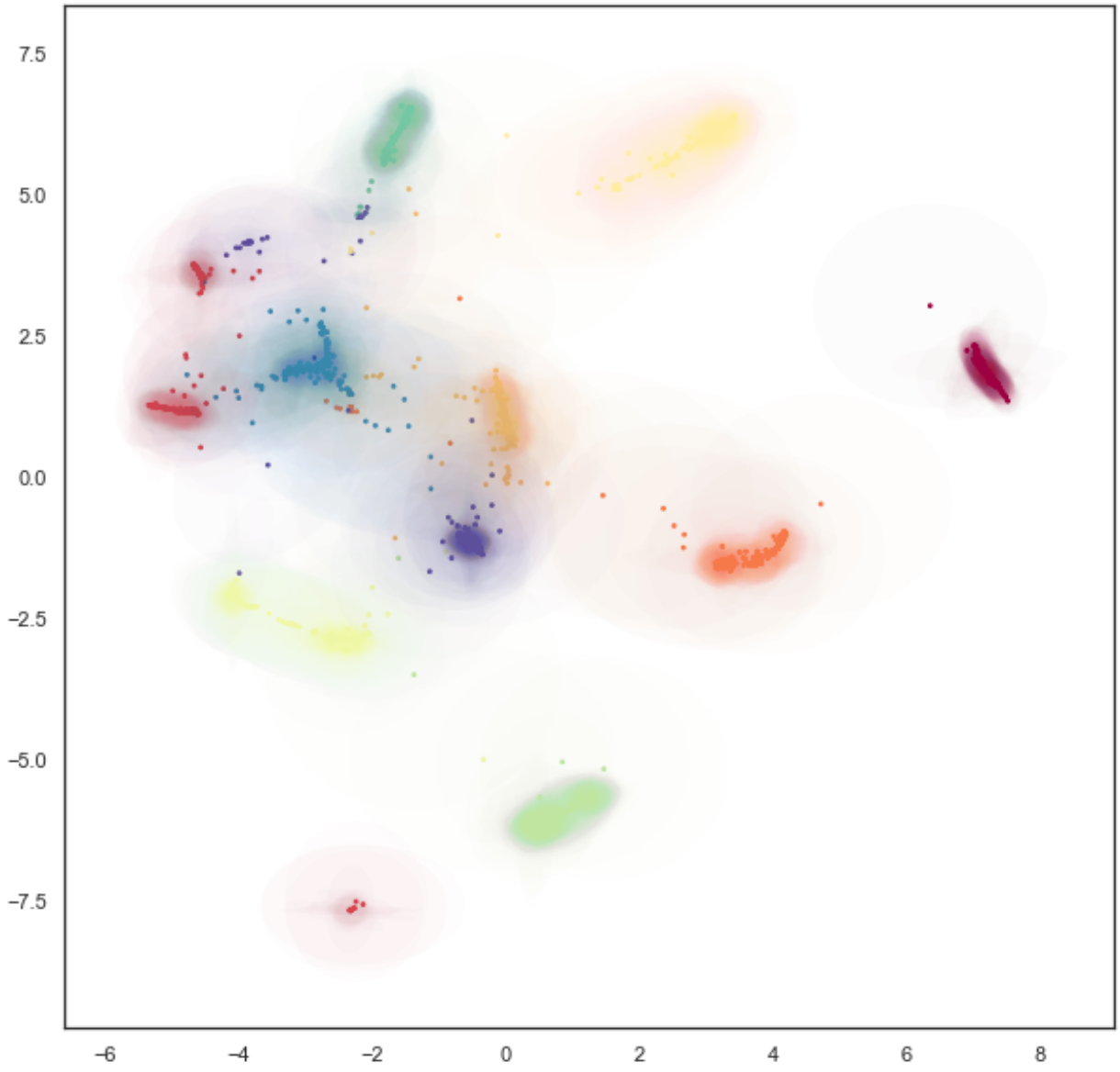
While this plot highlights some of the covariance structure in the outlying points, in practice the overplotting here obscures a lot of the more interesting structure in the clusters themselves. We can try to see this structure better by plotting only a single ellipse per point and using a lower alpha channel value for the ellipses, making them more translucent.

```
fig = plt.figure(figsize=(10,10))
ax = fig.add_subplot(111)
for i in range(gaussian_mapper.embedding_.shape[0]):
    pos = gaussian_mapper.embedding_[i, :2]
    draw_simple_ellipse(pos, gaussian_mapper.embedding_[i, 2],
```

(continues on next page)

(continued from previous page)

```
                        gaussian_mapper.embedding_[i, 3],
                        gaussian_mapper.embedding_[i, 4],
                        ax, n_ellipses=1,
                        color=colors[digits.target[i]],
                        from_size=1.0, to_size=1.0, alpha=0.01)
ax.scatter(gaussian_mapper.embedding_.T[0],
           gaussian_mapper.embedding_.T[1],
           c=digits.target, cmap='Spectral', s=3)
```



This lets us see the variation of density of clusters with respect to the covariance structure – some clusters have consistently very tight covariance, while others are more spread out (and hence have, in a sense, greater associated uncertainty. Of course we still have a degree of overplotting even here, and it will become increasingly difficult to tune alpha channels to make things visible. Instead what we would want is an actual density plot, showing the the density of the sum over all of these Gaussians.

To do this we'll need to define some functions, whose execution will be accelerated using numba: the evaluation of the

density of a 2d Gaussian at a given point; an evaluation of the density of a given point summing over a set of several Gaussians; and a function to generate the density for each point in some grid (summing only over nearby Gaussians to make this naive approach more computable).

```
from sklearn.neighbors import KDTree

@numba.njit(fastmath=True)
def eval_gaussian(x, pos=np.array([0, 0]), cov=np.eye(2, dtype=np.float32)):
    det = cov[0,0] * cov[1,1] - cov[0,1] * cov[1,0]
    if det > 1e-16:
        cov_inv = np.array([[cov[1,1], -cov[0,1]], [-cov[1,0], cov[0,0]]]) * 1.0 / det
        diff = x - pos
        m_dist = cov_inv[0,0] * diff[0]**2 - \
            (cov_inv[0,1] + cov_inv[1,0]) * diff[0] * diff[1] + \
            cov_inv[1,1] * diff[1]**2
        return (np.exp(-0.5 * m_dist)) / (2 * np.pi * np.sqrt(np.abs(det)))
    else:
        return 0.0

@numba.njit(fastmath=True)
def eval_density_at_point(x, embedding):
    result = 0.0
    for i in range(embedding.shape[0]):
        pos = embedding[i, :2]
        t = embedding[i, 4]
        U = np.array([[np.cos(t), np.sin(t)], [np.sin(t), -np.cos(t)]])
        cov = U @ np.diag(embedding[i, 2:4]) @ U
        result += eval_gaussian(x, pos=pos, cov=cov)
    return result

def create_density_plot(X, Y, embedding):
    Z = np.zeros_like(X)
    tree = KDTree(embedding[:, :2])
    for i in range(X.shape[0]):
        for j in range(X.shape[1]):
            nearby_points = embedding[tree.query_radius([[X[i,j],Y[i,j]]], r=2)[0]]
            Z[i, j] = eval_density_at_point(np.array([X[i,j],Y[i,j]]), nearby_points)
    return Z / Z.sum()
```

Now we simply need an appropriate grid of points. We can use the plot bounds seen above, and a grid size selected for the sake of computability. The numpy `meshgrid` function can supply the actual grid.

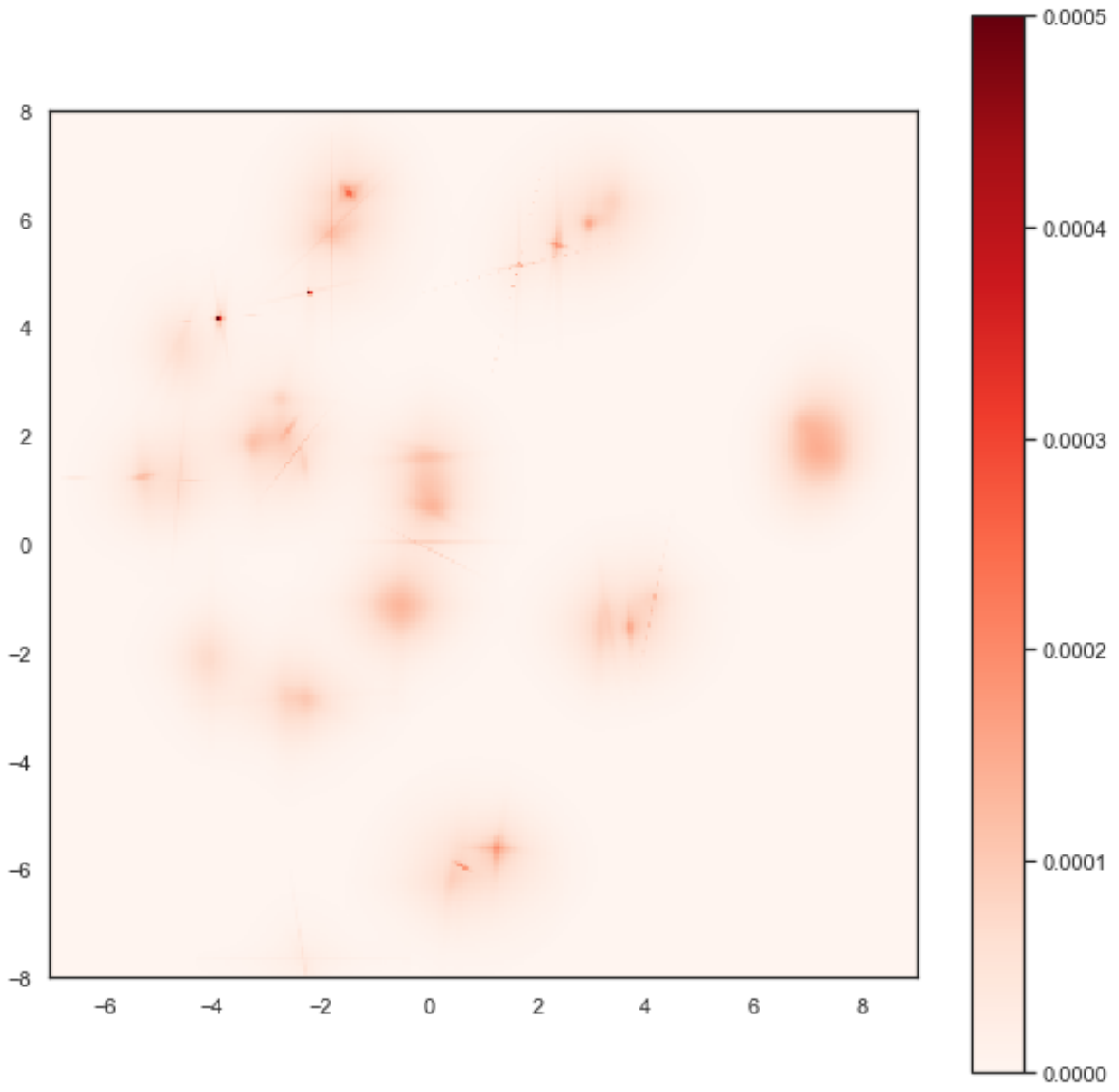
```
X, Y = np.meshgrid(np.linspace(-7, 9, 300), np.linspace(-8, 8, 300))
```

Now we can use the function defined above to compute the density at each point in the grid, given the Gaussians produced by the embedding.

```
Z = create_density_plot(X, Y, gaussian_mapper.embedding_)
```

Now we can view the result as a density plot using `imshow`.

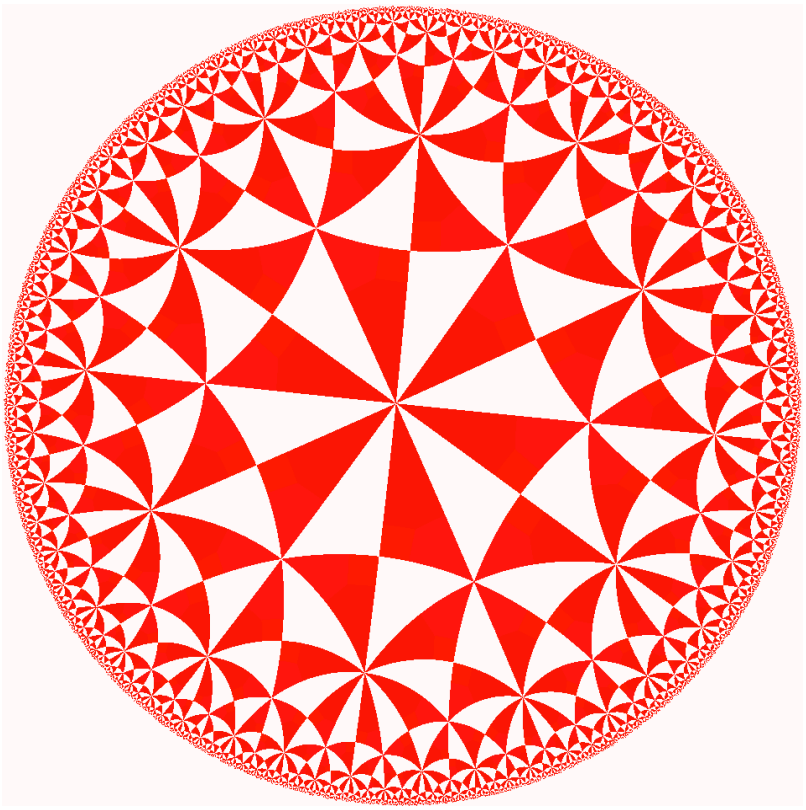
```
plt.imshow(Z, origin='lower', cmap='Reds', extent=(-7, 9, -8, 8), vmax=0.0005)
plt.colorbar()
```



Here we see the finer structure within the various clusters, including some of the interesting linear structures, demonstrating that this Gaussian uncertainty based embedding has captured quite detailed and useful information about the inter-relationships among the PenDigits dataset.

15.5 Bonus: Embedding in Hyperbolic space

As a bonus example let's look at embedding data into hyperbolic space. The most popular model for this for visualization is Poincare's disk model. An example of a regular tiling of hyperbolic space in Poincare's disk model is shown below; you may note it is similar to famous images by M.C. Escher.

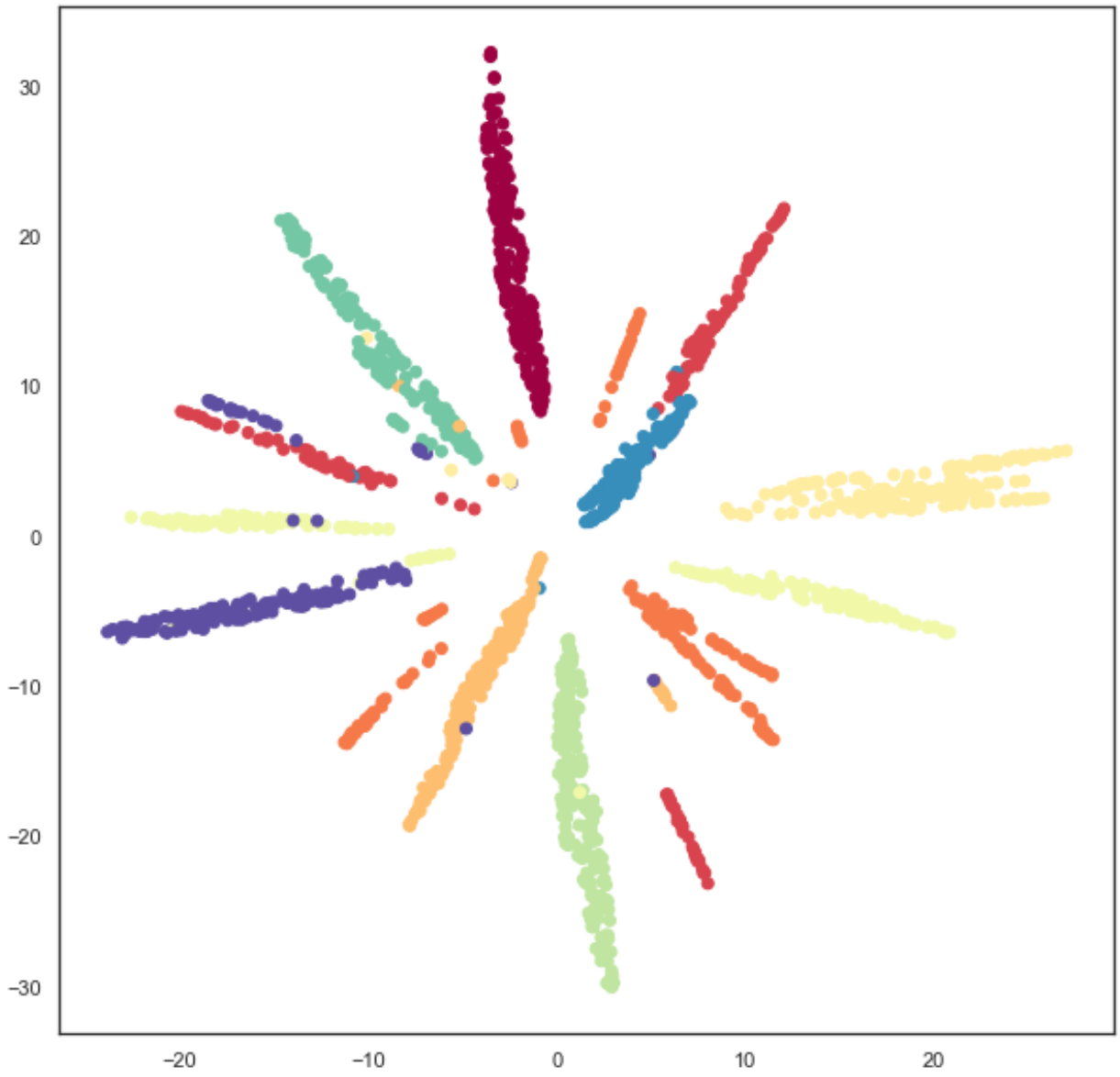


Ideally we would be able to embed directly into this Poincare disk model, but in practice this proves to be very difficult. The issue is that the disk has a "line at infinity" in a circle of radius one bounding the disk. Outside of that circle things are not well defined. As you may recall from the discussion of embedding onto spheres and toruses it is best if we can have a parameterisation of the embedding space that it is hard to move out of. The Poincare disk model is almost the opposite of this – as soon as we move outside the unit circle we have moved off the manifold and further updates will be badly defined. We therefore instead need a different parameterisation of hyperbolic space that is less constrained. One option is the Poincare half-plane model, but this, again, has a boundary that it is easy to move beyond. The simplest option is the hyperboloid model. Under this model we can simply move in x and y coordinates, and solve for the corresponding z coordinate when we need to compute distances. This model has been implemented under the distance metric `"hyperboloid"` so we can simply use it out-of-the-box.

```
hyperbolic_mapper = umap.UMAP(output_metric='hyperboloid',
                              random_state=42).fit(digits.data)
```

A straightforward visualization option is to simply view the x and y coordinates we have arrived at:

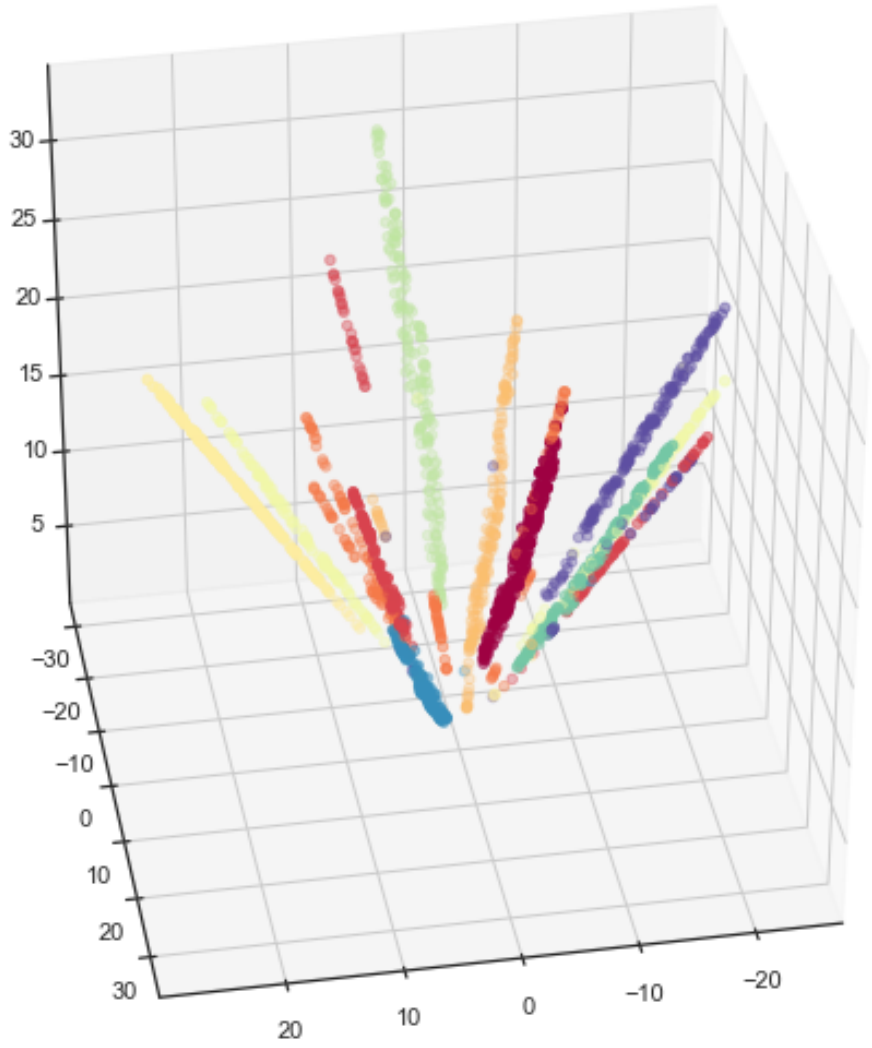
```
plt.scatter(hyperbolic_mapper.embedding_.T[0],
            hyperbolic_mapper.embedding_.T[1],
            c=digits.target, cmap='Spectral')
```



We can also solve for the z coordinate and view the data lying on a hyperboloid in 3d space.

```
x = hyperbolic_mapper.embedding_[:, 0]
y = hyperbolic_mapper.embedding_[:, 1]
z = np.sqrt(1 + np.sum(hyperbolic_mapper.embedding_**2, axis=1))
```

```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(x, y, z, c=digits.target, cmap='Spectral')
ax.view_init(35, 80)
```

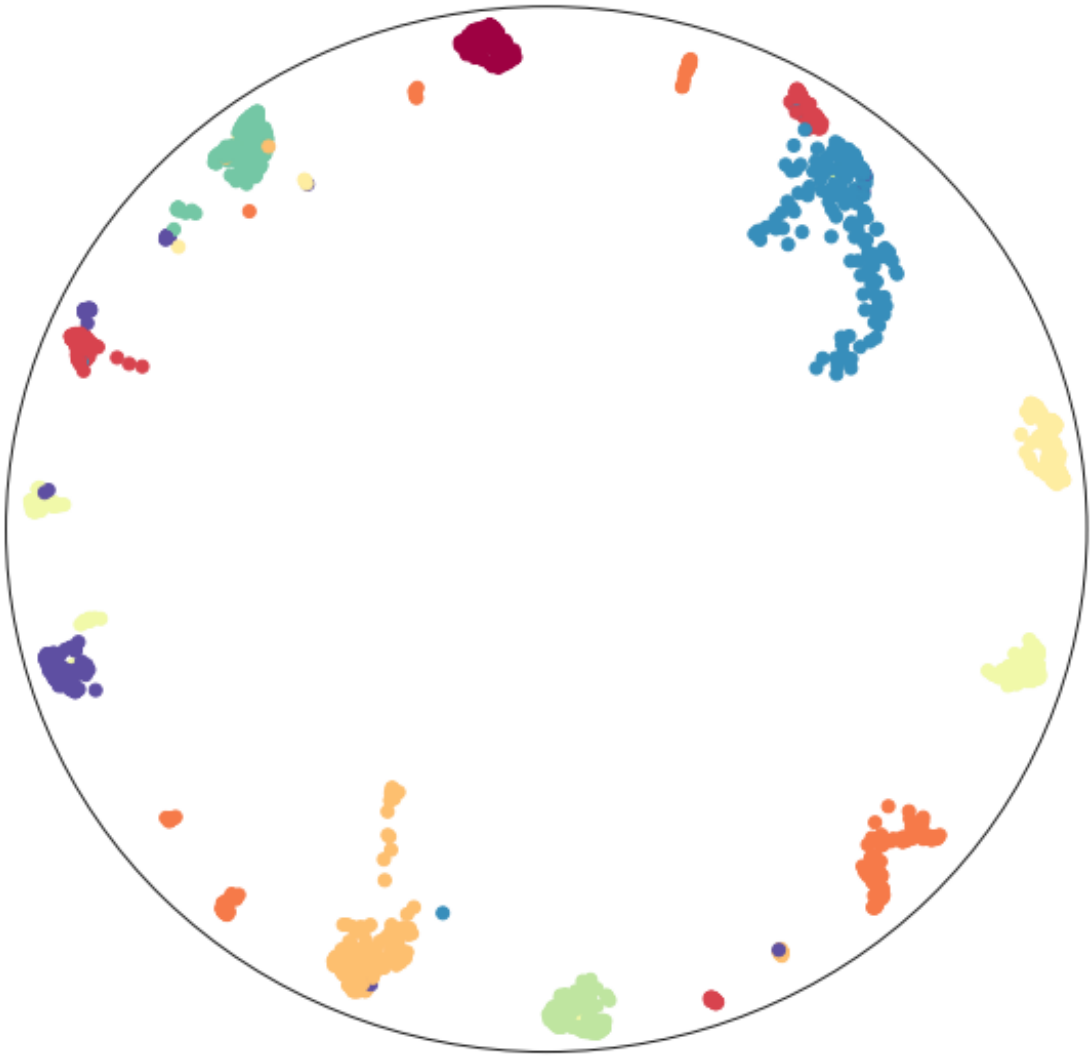


But we can do more – since we have embedded the data successfully in hyperbolic space we can map the data into the Poincare disk model. This is, in fact, a straightforward computation.

```
disk_x = x / (1 + z)
disk_y = y / (1 + z)
```

Now we can visualize the data in a Poincare disk model embedding as we first wanted. For this we simply generate a scatterplot of the data, and then draw in the bounding circle of the line at infinity.

```
fig = plt.figure()
ax = fig.add_subplot(111)
ax.scatter(disk_x, disk_y, c=digits.target, cmap='Spectral')
boundary = plt.Circle((0,0), 1, fc='none', ec='k')
ax.add_artist(boundary)
ax.axis('off');
```



Hopefully this has provided a useful example of how to go about embedding into non-euclidean spaces. This last example ideally highlights the limitations of this approach (we really need a suitable parameterisation), and some potential approaches to get around this: we can use an alternative parameterisation for the embedding, and then transform the data into the desired representation.

CHAPTER 16

How to use AlignedUMAP

It may happen that it would be beneficial to have different UMAP embeddings aligned with each other. There are several ways to go about doing this. One simple approach is to simply embed each dataset with UMAP independently and then solve for a Procrustes transformation on shared points. An alternative approach is to embed the first dataset and then construct an initial embedding for the second dataset based on locations of shared points in the first embedding and then go from there. A third approach, which will provide better alignments in general, is to optimize both embeddings at the same time with some form of constraint as to how far shared points can take different locations in different embeddings *during* the optimization. This last option is possible, but is not easily tractable to implement yourself (unlike the first two options). To remedy this issue it has been implemented as a separate model class in `umap-learn` called `AlignedUMAP`. The resulting class is quite flexible, but here we will walk through simple usage on some basic (and somewhat contrived) data just to demonstrate how to get it running on data.

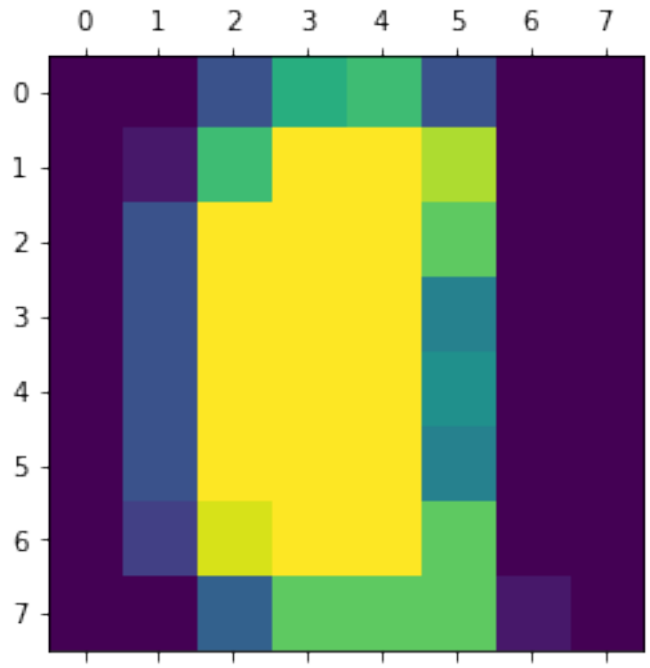
```
import numpy as np
import sklearn.datasets
import umap
import umap.plot
import umap.utils as utils
import umap.aligned_umap
import matplotlib.pyplot as plt
```

For our demonstration we'll just use the pendigits dataset from sklearn.

```
digits = sklearn.datasets.load_digits()
```

To make a sequence of datasets with some shared points between each different dataset we'll first sort the data so we have some vaguely sensible progression. In this case we'll sort by the total amount of "ink" in the handwritten digit. This isn't meant to be meaningful, it is merely meant to provide something useful to slicing into overlapping chunks that we will want to embed separately and yet keep aligned.

```
ordered_digits = digits.data[np.argsort(digits.data.sum(axis=1))]
ordered_target = digits.target[np.argsort(digits.data.sum(axis=1))]
plt.matshow(ordered_digits[-1].reshape((8,8)))
```



We can then divide up the dataset into slices of 400 samples, moving along in chunks of 150 to ensure that there are overlaps between consecutive slices. This will give us a list of ten different datasets that we can embed, with the goal being to ensure that the positions of points in the embeddings are relatively consistent.

```
slices = [ordered_digits[150 * i:min(ordered_digits.shape[0], 150 * i + 400)] for i␣
↪in range(10)]
```

To ensure that consistency `AlignedUMAP` will need more information than *just* the datasets – we also need some information about how the datasets relate to one another. These take the form of dictionaries that relate the indices of one dataset to the indices of another. Currently `AlignedUMAP` only supports sequences of datasets with relations between each consecutive pair in the sequence. To construct the relations for this dataset we note that the last 250 samples of one dataset are going to be the same samples as the first 250 samples of the next dataset – this makes it easy to construct the dictionary: it is mapping

```
150 --> 0
151 --> 1
...
398 --> 248
399 --> 249
```

which we can construct easily using a dictionary comprehension. We will have the same relation between each consecutive pair, so to make the list of relations between pairs we can just duplicate the constructed relation the requisite number of times.

```
relation_dict = {i+150:i for i in range(400-150)}
relation_dicts = [relation_dict.copy() for i in range(len(slices) - 1)]
```

Note that while in this case the relation defines a map between identical samples in different datasets it can be much more general – see the politics example later for a case where the relation is constructed from external information (representatives names and states).

Now that we have both a list of data slices and a list of relations between the consecutive pairs we can use the `AlignedUMAP` class to generate a list of embeddings. The `AlignedUMAP` class takes most of the parameters that UMAP accepts. The major difference is that the fit method requires a *list* of datasets, and a keyword argument

`relations` that specifies the relation dictionaries between consecutive pairs of datasets. Other than that things are essentially push-button.

```
%%time
aligned_mapper = umap.AlignedUMAP().fit(slices, relations=relation_dicts)
```

```
CPU times: user 57.4 s, sys: 8.43 s, total: 1min 5s
Wall time: 57.4 s
```

You will note that this took a non-trivial amount of time to run, despite being on the relatively small pendigits dataset. This is because we are completing 10 different UMAP embeddings at once, so on average we are taking about five seconds per embedding, which is more reasonable – the alignment does have overhead cost however.

The next step is to look at the results. To ensure that the plots we produce have a consistent x and y axis we'll use a small function to compute a set of axis bounds for plotting.

```
def axis_bounds(embedding):
    left, right = embedding.T[0].min(), embedding.T[0].max()
    bottom, top = embedding.T[1].min(), embedding.T[1].max()
    adj_h, adj_v = (right - left) * 0.1, (top - bottom) * 0.1
    return [left - adj_h, right + adj_h, bottom - adj_v, top + adj_v]
```

Now it is just a matter of plotting the results in ten different scatter plots. We can do this most easily with matplotlib directly, setting up a grid of plots. Note that the progression proceeds by row then column, so read the progression as if you were reading a page of text (across, then down).

```
fig, axs = plt.subplots(5,2, figsize=(10, 20))
ax_bound = axis_bounds(np.vstack(aligned_mapper.embeddings_))
for i, ax in enumerate(axs.flatten()):
    current_target = ordered_target[150 * i:min(ordered_target.shape[0], 150 * i +␣
↪400)]
    ax.scatter(*aligned_mapper.embeddings_[i].T, s=2, c=current_target, cmap="Spectral
↪")
    ax.axis(ax_bound)
    ax.set(xticks=[], yticks=[])
plt.tight_layout()
```

So despite being different embeddings on different datasets, the clusters keep their general alignment – the top left plot and bottom right plot have the same rough positions for specific digit clusters. We can also, to a degree, see how the structure changes over the course of the different slices. Thus we are keeping the various embeddings aligned, but allowing the changes dictated by the differing structures of each different slice of data.

16.1 Online updating of aligned embeddings

It may be the case that we have incoming temporal data and would like to have embeddings of time-windows that, ideally, align with the embeddings of prior time-windows. As long as we overlap the time-windows we use to allow for relations between time windows then this is possible – except that the previous code required all the time-windows to be input *at once* for fitting. We would instead like to train an initial model and then update it as we go. This is possible via the `update` method which we'll demonstrate below.

First we need to fit a base `AlignedUMAP` model; we'll use the first two slices and the first relation dict to do so.

```
%%time
updating_mapper = umap.AlignedUMAP().fit(slices[:2], relations=relation_dicts[:1])
```

```
CPU times: user 9.32 s, sys: 1.47 s, total: 10.8 s
Wall time: 9.17 s
```

Note that this is fairly quick, since we are only fitting two slices. Given the trained model the update method requires a new slice of data to add, along with a relation dictionary (passed in with the `relations` keyword argument as with `fit`). This will append a new embedding to the `embeddings_` attribute of the model for the new data, aligned with what has been seen so far.

```
for i in range(2,len(slices)):
    %time updating_mapper.update(slices[i], relations={v:k for k,v in relation_
↪dicts[i-1].items()})
```

```
CPU times: user 7.78 s, sys: 1.15 s, total: 8.93 s
Wall time: 7.92 s
CPU times: user 6.64 s, sys: 1.17 s, total: 7.81 s
Wall time: 6.6 s
CPU times: user 6.94 s, sys: 1.17 s, total: 8.11 s
Wall time: 6.81 s
CPU times: user 6.45 s, sys: 1.51 s, total: 7.96 s
Wall time: 6.45 s
CPU times: user 7.44 s, sys: 1.32 s, total: 8.76 s
Wall time: 7.16 s
CPU times: user 7.68 s, sys: 1.73 s, total: 9.41 s
Wall time: 7.59 s
CPU times: user 7.88 s, sys: 1.65 s, total: 9.54 s
Wall time: 7.39 s
CPU times: user 7.82 s, sys: 1.98 s, total: 9.8 s
Wall time: 7.7 s
```

Note that each new slice takes a relatively short period of time, as we might hope. The downside of this, as you can imagine, is that we have no "forward" relations – the windows over slices only look backward. This means the results are less good, but we are trading that for the ability to quickly and easily update as we go.

We can look at how we did using essentially the same code as before.

```
fig, axs = plt.subplots(5,2, figsize=(10, 20))
ax_bound = axis_bounds(np.vstack(updating_mapper.embeddings_))
```

(continues on next page)

(continued from previous page)

```
for i, ax in enumerate(axs.flatten()):
    current_target = ordered_target[150 * i:min(ordered_target.shape[0], 150 * i +␣
↪400)]
    ax.scatter(*updating_mapper.embeddings_[i].T, s=2, c=current_target, cmap=
↪"Spectral")
    ax.axis(ax_bound)
    ax.set(xticks=[], yticks=[])
plt.tight_layout()
```

Wee see that the alignment is indeed working, so new slices remain comparable with previously trained slices. As noted the overall alignments and progression is not as nice as the previous version, but it does have the significant benefit of allowing an update as you go approach.

Note that right now this model keeps all the previous data, so it will only really work in a batch streaming approach where occasionally a fresh model is trained, dropping some of the historical data before continuing with updates.

16.2 Aligning varying parameters

It is possible to align UMAP embedding that vary in the parameters used instead of the data. To demonstrate how this can work we'll continue to use the pendigits dataset, but instead of slicing the data as we did before, we'll use the full dataset. That means that our relations between datasets are simply constant relations. We can construct those ahead of time:

```
constant_dict = {i:i for i in range(digits.data.shape[0])}
constant_relations = [constant_dict for i in range(9)]
```

To run AlignedUMAP over a range of parameters you simply need to pass in a *list* of the sequence of parameters you wish to use. You can do this for several different parameters – just ensure that all the lists are the same length! In this case we'll try looking at how the embeddings change if we change `n_neighbors` and `min_dist`. This means that when we create the AlignedUMAP object we pass a list, instead of a single value, to each of those parameters. To make the visualization a little more interesting we'll also vary some of the alignment parameters (there are only two of major consequence). Specifically we'll adjust the `alignment_window_size`, which controls how far forward and backward across the datasets we look when doing alignment, and the `alignment_regularisation` which controls how heavily we weight the alignment aspect versus the UMAP layout. Larger values of `alignment_regularisation` will work harder to keep points aligned across embeddings (at the cost of the embedding quality at each slice), while smaller values will allow the optimisation to focus more on the individual embeddings and put less emphasis on aligning the embeddings with each other.

Given a model we can then fit it. As before we need to hand it a list of datasets, and a list of relations. Since we are using the same data each time (and varying the parameters) we can just repeat the full pendigits dataset. Note that the number of datasets needs to match the number of parameter values being used. The same goes for the number of relations (one less than the number of parameter values).

```
neighbors_mapper = umap.AlignedUMAP(
    n_neighbors=[3,4,5,7,11,16,22,29,37,45,54],
    min_dist=[0.01,0.05,0.1,0.15,0.2,0.25,0.3,0.35,0.4,0.45],
    alignment_window_size=2,
    alignment_regularisation=1e-3,
).fit(
    [digits.data for i in range(10)], relations=constant_relations
)
```

As before we can look at the results by plotting each of the embeddings.

```
fig, axs = plt.subplots(5,2, figsize=(10, 20))
ax_bound = axis_bounds(np.vstack(neighbors_mapper.embeddings_))
for i, ax in enumerate(axs.flatten()):
    ax.scatter(*neighbors_mapper.embeddings_[i].T, s=2, c=digits.target, cmap=
↪"Spectral")
    ax.axis(ax_bound)
    ax.set(xticks=[], yticks=[])
plt.tight_layout()
```

To get a better feel for the evolution of the embedding over the change in parameter values we can plot the data in three dimensions, with the third dimension being the parameter value chosen. To better show how data points in the embedding *move* with respect to the changing parameters we can plot them not as points, but as *curves* connecting the same point in each sequential embedding. For three dimensional plots like this we'll make use of the plotly plotting library.

```
import plotly.graph_objects as go
import plotly.express as px
import pandas as pd
```

The first thing we'll have to do is wrangle the data into a suitable format for plotly. That's the reason we loaded up pandas as well – plotly likes dataframes. This involves stacking all the embeddings together, and then assigning an extra `z` value according to which embedding we are in. For the purposes of visualization we'll just have a linear scale from 0 to 1 of the appropriate length for the z coordinates.

```
n_embeddings = len(neighbors_mapper.embeddings_)
es = neighbors_mapper.embeddings_
embedding_df = pd.DataFrame(np.vstack(es), columns=('x', 'y'))
embedding_df['z'] = np.repeat(np.linspace(0, 1.0, n_embeddings), es[0].shape[0])
embedding_df['id'] = np.tile(np.arange(es[0].shape[0]), n_embeddings)
embedding_df['digit'] = np.tile(digits.target, n_embeddings)
```

The next thing we can do to improve the visualization is to smooth out the curves rather than leaving them as piecewise linear lines. To to this we can use the `scipy.interpolate` functionality to create smooth cubic splines that pass through all the points of the curve we wish to create.

```
import scipy.interpolate
```

The interpolate module has a function `interp1d` that generates a (vector of) smooth function given a one dimensional set of datapoints that it needs to pass through. We can generate separate functions for the x and y coordinates for each pendigit sample, allowing us to generate smooth curves in three dimensions.

```
fx = scipy.interpolate.interp1d(
    embedding_df.z[embedding_df.id == 0], embedding_df.x.values.reshape(n_embeddings,␣
↪digits.data.shape[0]).T, kind="cubic"
)
fy = scipy.interpolate.interp1d(
    embedding_df.z[embedding_df.id == 0], embedding_df.y.values.reshape(n_embeddings,␣
↪digits.data.shape[0]).T, kind="cubic"
)
z = np.linspace(0, 1.0, 100)
```

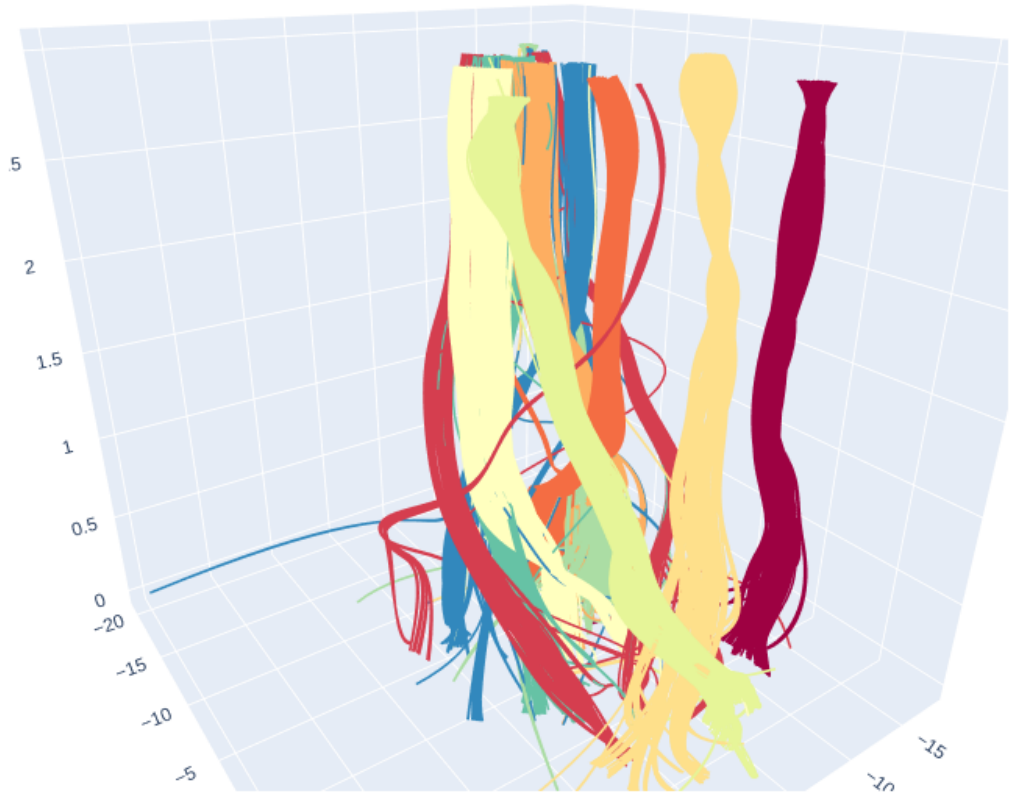
With that is hand it is just a matter of plotting all the curves. In plotly parlance each curve is a "trace" and we generate each one separately (along with a suitable colour given by the digit the sample represents). We then add all the traces to a figure, and display the figure.

```
palette = px.colors.diverging.Spectral
interpolated_traces = [fx(z), fy(z)]
traces = [
    go.Scatter3d(
        x=interpolated_traces[0][i],
        y=interpolated_traces[1][i],
        z=z*3.0,
        mode="lines",
        line=dict(
            color=palette[digits.target[i]],
            width=3.0
```

(continues on next page)

(continued from previous page)

```
        ),
        opacity=1.0,
    )
    for i in range(digits.data.shape[0])
]
fig = go.Figure(data=traces)
fig.update_layout(
    width=800,
    height=700,
    autosize=False,
    showlegend=False,
)
fig.show()
```



Since it is tricky to get the interactive plotly figure embedded in documentation we have a static image here, but if you run this yourself you will have a fully interactive view of the data.

CHAPTER 17

AlignedUMAP for Time Varying Data

It is not uncommon to have datasets that can be partitioned into segments, often with respect to time, where we want to understand not only the structure of each segment, but how that structure changes over the different segments. An example of this is the relative political leanings of the US congress over time. In determining the relative political leanings we can look at the representatives voting record on roll call votes, with the presumption that representatives with similar political principles will have similar voting records. We can, of course, look at such data for any given congress, but since representatives are commonly re-elected we can also consider how their relative position in congress changes with time – an ideal use case for AlignedUMAP.

First we'll need a selection of libraries. Aside from UMAP we will need to do a little bit of data wrangling; for that we'll need pandas, and also for matching up names of representatives we'll make use of the library `fuzzywuzzy` which provides easy to use fuzzy string matching.

```
import umap
import umap.utils as utils
import umap.aligned_umap
import sklearn.decomposition

import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
import seaborn as sns

from fuzzywuzzy import fuzz, process
import re
```

```
sns.set(style="darkgrid", color_codes=True)
```

Next we'll need to voting records for the representatives, along with the associated metadata from the roll call vote records. You can obtain the data https://clerk.house.gov; a notebook demonstrating how to pull down the data and parse it into the csv files used here is available here.

17.1 Processing Congressional Voting Records

The voting records provide a row for each representative with a -1 for "No", 0 for "Present" or "Not Voting", and 1 for "Aye". A separate csv file contains the raw data of all the votes with a row for each legislators vote on each roll-call item. We really just need some metadata – which state they represent and the party they represent so we can decorate the results with this kind of information later. For that we just need to extra the names, states, and parties for each year. We can grab those columns and then drop duplicates. A catch: the party is very occasionally entered incorrectly, and occasionally representatives switch parties, making duplicated rows. We'll just take the first entry of such duplciates for now.

```
votes = [pd.read_csv(f"house_votes/{year}_voting_record.csv", index_col=0).sort_
↪index()
         for year in range(1990,2021)]
metadata = [pd.read_csv(
    f"house_votes/{year}_full.csv",
    index_col=0
)[["legislator", "state", "party"]].drop_duplicates(["legislator", "state"]).sort_
↪values('legislator')
            for year in range(1990,2021)]
```

Let's take a look at the voting record for a single year to see what sort of data we are looking at:

```
votes[5]
```

We can examine the associated metadata for the same year.

```
metadata[5]
```

You may note that sometimes representatives names list a state in parenthesis afterwards. This is to provide disambiguation for representatives that happen to have the last name. This actually complicates matters for us since the disambiguation is only applied in those cases where there is a name collision in that sitting of congress. That means that for several years a representative may have simply their last name, but then switch to being disambiguated, before potentially switching back again. This would make it much harder to consistently treck representatives over their entire career in congress. To fix this up we'll simply re-index by a unique representative ID that has their last name, party, and state all listed over all the voting dataframes. We'll need a function to generate those from the metadata, and then we'll need to apply it to all the reocrds. Importantly we'll have to finesse those situations where representatives are listed twice (under un-ambiguous and disambiguated names) with some groupby tricks.

```
def unique_legislator(row):
    name, state, party = row.legislator, row.state, row.party
    # Strip of disambiguating state designators
    if re.search(r'(\w+) \([A-Z]{2}\)', name) is not None:
        name = name[:-5]
    return f"{name} ({party}, {state})"
```

```
for i, _ in enumerate(votes):
    votes[i].index = pd.Index(metadata[i].apply(unique_legislator, axis=1), name=
↪"legislator_index")
    votes[i] = votes[i].groupby(level=0).sum()
    metadata[i].index = pd.Index(metadata[i].apply(unique_legislator, axis=1), name=
↪"legislator_index")
    metadata[i] = metadata[i].groupby(level=0).first()
```

Now that we have the data at least a little wrangled into order there is the question of ensuring some degree of continuity fore representatives. To make this a little easier we'll use voting records over *four year spans* instead of over single years. Equally importantly we'll do this in a sliding window fashion so that we consider the record for

1990-1994 and then the record for 1991-1995 and so on. By overlapping the windows in this way we can ensure a little greater continuity of political stance through the years. To make this happen we just have to merge data frames in a sliding set of pairs, and then merge the pairs via the same approach:

```
votes = [
    pd.merge(
        v1, v2, how="outer", on="legislator_index"
    ).fillna(0.0).sort_index()
    for v1, v2 in zip(votes[:-1], votes[1:])
] + votes[-1:]

metadata = [
    pd.concat([m1, m2]).groupby("legislator_index").first().sort_index()
    for m1, m2 in zip(metadata[:-1], metadata[1:])
] + metadata[-1:]
```

That's the pairs of years; now we merge these pairwise to get sets of four years worth of votes.

```
votes = [
    pd.merge(
        v1, v2, how="outer", on="legislator_index"
    ).fillna(0.0).sort_index()
    for v1, v2 in zip(votes[:-1], votes[1:])
] + votes[-1:]

metadata = [
    pd.concat([m1, m2]).groupby(level=0).first().sort_index()
    for m1, m2 in zip(metadata[:-1], metadata[1:])
] + metadata[-1:]
```

17.2 Applying AlignedUMAP

To make use of AlignedUMAP we need to generate relations between consecutive dataset slices. In this case that means we need to have a relation describing row from one four year slice corresponds to a row from the following four year slice for the same representative. For AlignedUMAP to work this should be formatted as a list of dictionaries; each dictionary gives a mapping from indices of one slice to indices of the next. Importantly this mapping can be partial – it only has to relate indices for which there is a match between the two slices.

The vote dataframes that we are using for slices are already indexed with unique identifiers for representatives, so to make relations we simply have to match them up, creating a dictionary of indices from one to the other. In practice we can do this relatively efficiently by using pandas to merge dataframes on the pandas indexes of the two vote dataframes with the data being simply the numeric indices of the rows. The resulting dictionary is then just the dictionary of pairs given by the inner join.

```
def make_relation(from_df, to_df):
    left = pd.DataFrame(data=np.arange(len(from_df)), index=from_df.index)
    right = pd.DataFrame(data=np.arange(len(to_df)), index=to_df.index)
    merge = pd.merge(left, right, left_index=True, right_index=True)
    return dict(merge.values)
```

With a function for relation creation in place we simply need to apply it to each consecutive pair of vote dataframes.

```
relations = [make_relation(x,y) for x, y in zip(votes[:-1], votes[1:])]
```

If you are still unsure of what these relations are it might be beneficial to look at a few of the dictionaries, along with the corresponding pairs of vote dataframes. Here is (part of) the first relation dictionary:

```
relations[0]
```

```
{0: 0,
 1: 1,
 3: 2,
 4: 3,
 5: 4,
 6: 5,
 7: 6,
 8: 7,
 9: 8,
 10: 9,
 11: 10,
 12: 11,
 13: 12,
 14: 13,
 15: 14,
 ...
 475: 547,
 476: 549,
 477: 550,
 478: 552,
 479: 553,
 480: 554,
 481: 555,
 482: 556,
 483: 557,
 484: 559}
```

Now we are finally in a position to run AlignedUMAP. Most of the standard UMAP parameters are available for use, including choosing a metric and a number of neighbors. Here we will also make use of the extra AlignedUMAP parameters `alignment_regularisation` and `alignment_window_size`. The first is a value that weights how important retaining alignment is. Typically the value is much smaller than this (the default is 0.01), but given the relatively high volatility in voting records we are going to increase it here. The second parameter, `alignment_window_size` determines how far out on either side AlignedUMAP will look when aligning embeddings – even though the relations are specified only between consecutive slices it will chain them together to construct relations reaching further. In this case we'll have it look as far out as 5 slices either side.

```
%%time
aligned_mapper = umap.aligned_umap.AlignedUMAP(
    metric="cosine",
    n_neighbors=20,
    alignment_regularisation=0.1,
    alignment_window_size=5,
    n_epochs=200,
    random_state=42,
).fit(votes, relations=relations)
embeddings = aligned_mapper.embeddings_
```

```
CPU times: user 6min 7s, sys: 30.6 s, total: 6min 37s
Wall time: 5min 57s
```

17.3 Visualizing the Results

Now we need to plot the data somehow. To make the visualization interesting it would be beneficial to have some colour variation – ideally showing a different view of the relative political stance. For that we want to attempt to get an idea of the position of each candidate from an alternative source. To do this we can try to extract the vote margin that the representative won by. The catch here is that while the election data can be collected and processed, the names don't match perfectly as they come from a different source. That means we need to do our best to get a name match for each candidate. We'll use fuzzy string matching restricted to the relevant year and state to try to get a good match. A notebook providing details for obtaining and processing the election winners data can be found here.

```
election_winners = pd.read_csv('election_winners_1976-2018.csv', index_col=0)
election_winners.head()
```

Now we need to simply go through the metadata and fill it out with the extra information we can glean from the election winners data. Since we can't do exact name matching (the data for both is somewhat messy when it comes to text fields like names) we can't simply perform a join, but must instead process things year by year and representative by representative, finding the best string match on name that we can for the given year and state election. In practice we are undoubtedly going to get some of these wrong, and if the goal was a rigorous analysis based on this data a lot more care would need to be taken. Since this is just a demonstration and we'll only be using this extra information as a colour channel in plots we can excuise a few errors here and there from in-exact data processing.

```
n_name_misses = 0
for year, df in enumerate(metadata, 1990):
    df["partisan_lean"] = 0.5
    df["district"] = np.full(len(df), -1, dtype=np.int8)
    for idx, (loc, row) in enumerate(df.iterrows()):
        name, state, party = row.legislator, row.state, row.party
        # Strip of disambiguating state designators
        if re.search(r'(\w+) \([A-Z]{2}\)', name) is not None:
            name = name[:-5]
        # Get a party designator matching the election_winners data
        party = "republican" if party == "R" else "democrat"
        # Restrict to the right state and time-frame
        state_election_winners = election_winners[(election_winners.state == state)
                                                  & (election_winners.year <= year +␣
↪4)
                                                  & (election_winners.year >= year -␣
↪4)]
        # Try to match a name; and fail "gracefully"
        try:
            matched_name = process.extractOne(
                name,
                state_election_winners.winner.tolist(),
                scorer=fuzz.partial_token_sort_ratio,
                score_cutoff=50,
            )
        except:
            matched_name = None

        # If we got a unique match, get the election data
        if matched_name is not None:
            winner = state_election_winners[state_election_winners.winner == matched_
↪name[0]]
        else:
            winner = []
```

(continues on next page)

(continued from previous page)

```
        # We either have none, one, or *several* match elections. Take a best guess.
        if len(winner) < 1:
            df.loc[loc, ["partisan_lean"]] = 0.25 if party == "republican" else 0.75
            n_name_misses += 1
        elif len(winner) > 1:
            df.iloc[idx, 4] = int(winner.district.values[-1])
            df.iloc[idx, 3] = float(winner.winning_ratio.values[-1])
        else:
            df.iloc[idx, 4] = int(winner.district.values)
            df.iloc[idx, 3] = float(winner.winning_ratio.values[0])

print(f"Failed to match a name {n_name_misses} times")
```

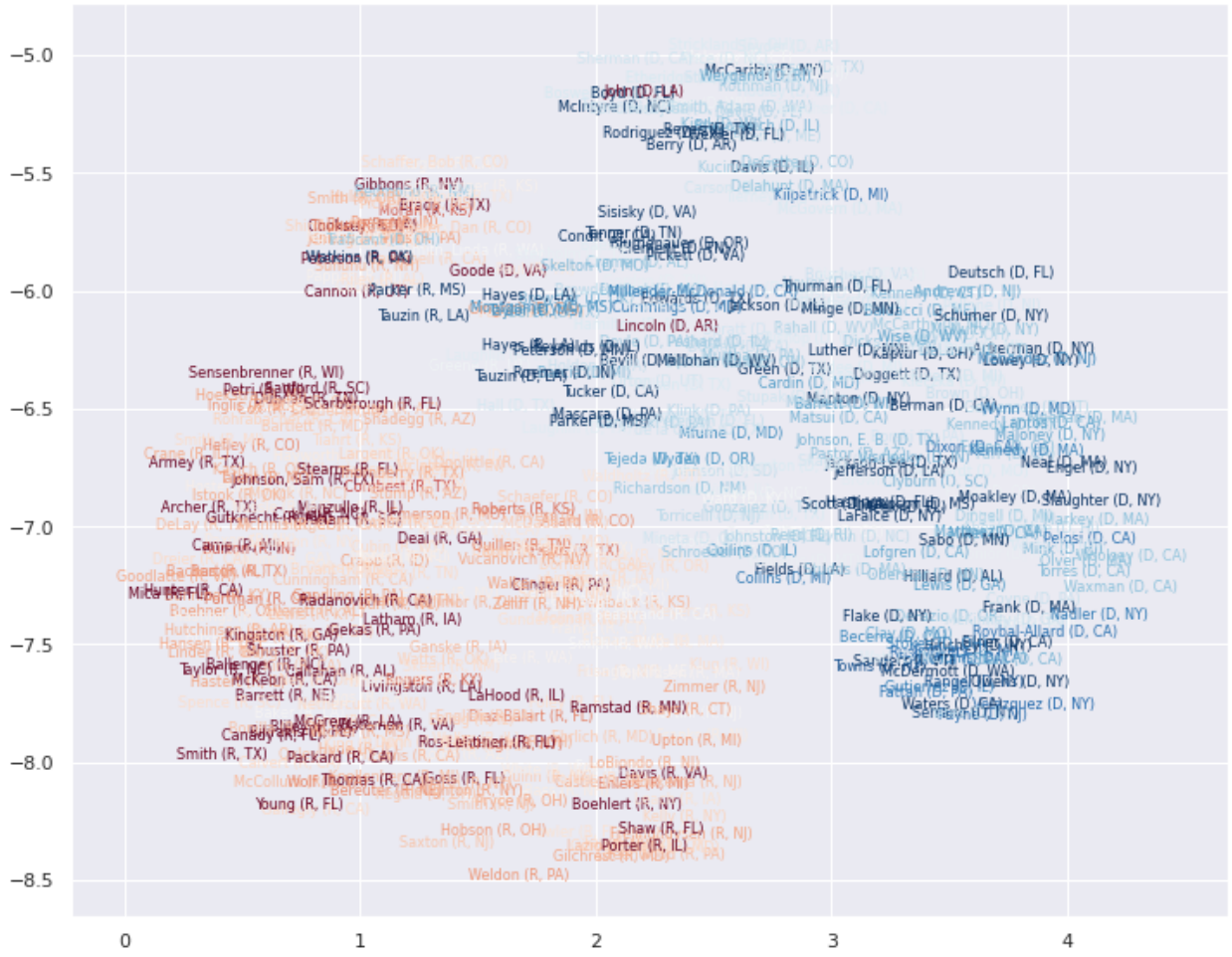
```
Failed to match a name 100 times
```

Now that we have the relative partisan leanings based on district election margins we can color the plot. We can obviously label the plot with the representatives names. The last remaining catch (when using matplotlib for the plotting) is the get the plot bounds (since we will be placing text markers directly into the plot, and thus not autogenerating bounds). This is a simple enough matter of computing some bounds as an adjustment a little outside the data limits.

```
def axis_bounds(embedding):
    left = embedding.T[0].min()
    right = embedding.T[0].max()
    bottom = embedding.T[1].min()
    top = embedding.T[1].max()
    width = right - left
    height = top - bottom
    adj_h = width * 0.1
    adj_v = height * 0.05
    return [left - adj_h, right + adj_h, bottom - adj_v, top + adj_v]
```

Now for the plot. Let's pick a random time slice (you are welcome to try others) and draw the representatives names in their embedded locations for that slice, coloured by their relative election victory margin.

```
fig, ax = plt.subplots(figsize=(12,12))
e = 5
ax.axis(axis_bounds(embeddings[e]))
ax.set_aspect('equal')
for i in range(embeddings[e].shape[0]):
    ax.text(embeddings[e][i, 0],
            embeddings[e][i, 1],
            metadata[e].index.values[i],
            color=plt.cm.RdBu(np.float32(metadata[e]["partisan_lean"].values[i])),
            fontsize=8,
            horizontalalignment='center',
            verticalalignment='center',
           )
```



This gives a good idea of the layout in a single time slices, and by plotting different time slices we can get some idea of how things have evolved. We can go further, however, by plotting a representative as curve through time as their relative political position in congress changes. For that we will need a 3D plot – we need both the UMAP x and y coordinates, as well as a third coordinate giving the year. I found this easiest to do in plotly, so let's import that. To make nice smooth curves through time we will also import the `scipy.interpolate` module which will let is interpolate a smooth curve from the discrete positions that a representatives appears in over time.

```
import plotly.graph_objects as go
import scipy.interpolate
```

Wrangling the data into shape for this is the next step; first let's get everything in a single dataframe that we can extract relevant data from on an as-needed basis.

```
df = pd.DataFrame(np.vstack(embeddings), columns=('x', 'y'))
df['z'] = np.concatenate([[year] * len(embeddings[i]) for i, year in␣
↪enumerate(range(1990, 2021))])
df['representative_id'] = np.concatenate([v.index for v in votes])
df['partisan_lean'] = np.concatenate([m["partisan_lean"].values for m in metadata])
```

Next we'll need that interpolation of the curve for a given representative. We'll write a function to handle that as there is a little bit of case-based logic that makes it non-trivial. We are going to get handed year data and want to interpolate the UMAP x and y coordinates for a single representative.

The first major catch is that many representatives don't have a single contiguous block of years for which they were

in congress: they were elected for several years, missed re-election, and then came back to congress several years later (possibly in another district). Each such block of contiguous years needs to be a separate path, and we shouldn't connect them. We therefore need some logic to find the contiguous blocks and generate smooth paths for each of them.

Another catch is that Ssome representatives have only been in office for a year or two (special elections and so forth) and we can't do a cubic spline interpolation for that; we can devolve to linear interpolation or quadratic splines for those cases, so simply add the point itself for the odd single year cases.

With those issues in hand we can then simply use the scipy `interp1d` function to generate smooth curves through the points.

```
INTERP_KIND = {2:"linear", 3:"quadratic", 4:"cubic"}

def interpolate_paths(z, x, y, c, rep_id):
    consecutive_year_blocks = np.where(np.diff(z) != 1)[0] + 1
    z_blocks = np.split(z, consecutive_year_blocks)
    x_blocks = np.split(x, consecutive_year_blocks)
    y_blocks = np.split(y, consecutive_year_blocks)
    c_blocks = np.split(c, consecutive_year_blocks)

    paths = []

    for block_idx, zs in enumerate(z_blocks):

        text = f"{rep_id} -- partisan_lean: {np.mean(c_blocks[block_idx]):.2f}"

        if len(zs) > 1:
            kind = INTERP_KIND.get(len(zs), "cubic")
        else:
            paths.append(
                (zs, x_blocks[block_idx], y_blocks[block_idx], c_blocks[block_idx],␣
↪text)
            )
            continue

        z = np.linspace(np.min(zs), np.max(zs), 100)
        x = scipy.interpolate.interp1d(zs, x_blocks[block_idx], kind=kind)(z)
        y = scipy.interpolate.interp1d(zs, y_blocks[block_idx], kind=kind)(z)
        c = scipy.interpolate.interp1d(zs, c_blocks[block_idx], kind="linear")(z)

        paths.append((z, x, y, c, text))

    return paths
```

And now we can use plotly to draw the resulting curves. For plotly we use the `Scatter3D` method, which supports a "lines" mode that can draw curves in 3D space. We can colour the curves by the partisan lean score we derived from the election data – in fact the colour can vary through the trace as the election margins vary. Since this is a plotly plot it is interactive, so you can rotate it around and view it from all angles.

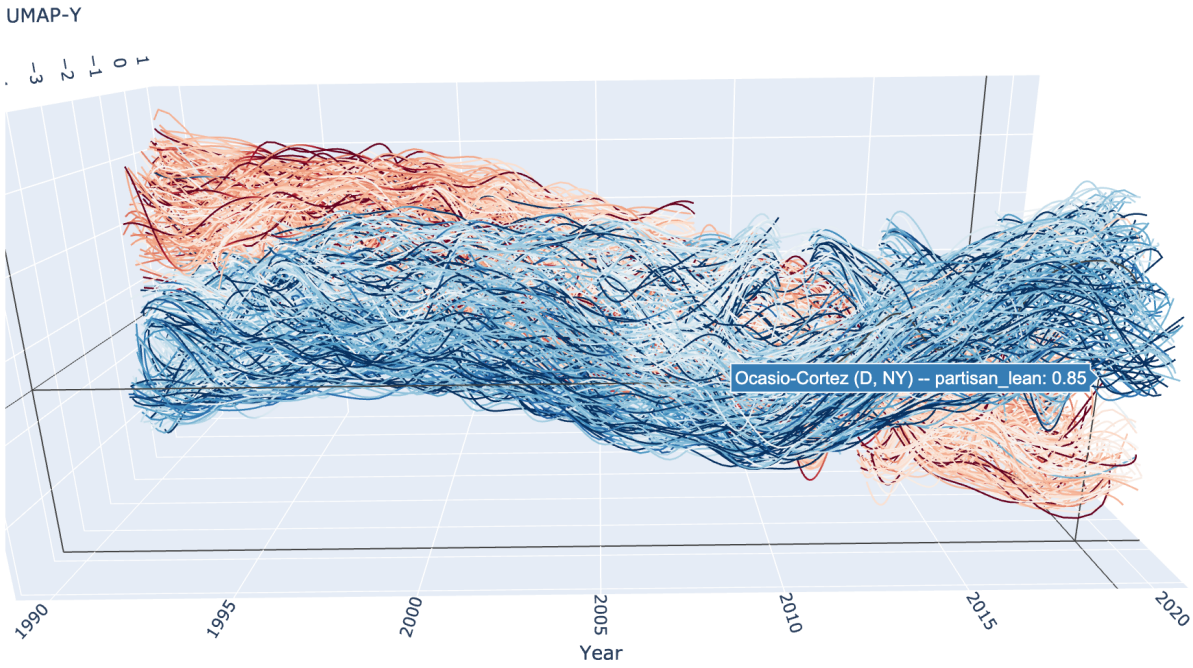
Unfortunately the interactive plotly plot does not embed into the documentation well, so we present here a static image. If you run this yourself, however, you will get the interactive version.

```
traces = []
for rep in df.representative_id.unique():
    z = df.z[df.representative_id == rep].values
    x = df.x[df.representative_id == rep].values
    y = df.y[df.representative_id == rep].values
    c = df.partisan_lean[df.representative_id == rep]
```

(continues on next page)

(continued from previous page)

```
    for z, x, y, c, text in interpolate_paths(z, x, y, c, rep):
        trace = go.Scatter3d(
            x=x, y=z, z=y,
            mode="lines",
            hovertext=text,
            hoverinfo="text",
            line=dict(
                color=c,
                cmin=0.0,
                cmid=0.5,
                cmax=1.0,
                cauto=False,
                colorscale="RdBu",
                colorbar=dict(),
                width=2.5,
            ),
            opacity=1.0,
        )
        traces.append(trace)

fig = go.Figure(data=traces)
fig.update_layout(
    width=800,
    height=600,
    scene=dict(
        aspectratio = dict( x=0.5, y=1.25, z=0.5 ),
        yaxis_title="Year",
        xaxis_title="UMAP-X",
        zaxis_title="UMAP-Y",
    ),
    scene_camera=dict(eye=dict( x=0.5, y=0.8, z=0.75 )),
    autosize=False,
    showlegend=False,
)
fig_widget = go.FigureWidget(fig)
fig_widget
```



This concludes our exploration for now.

CHAPTER 18

Release Notes

Some notes on new features in various releases

18.1 What's new in 0.5

- ParametricUMAP learns embeddings with neural networks
- AlignedUMAP can align multiple embeddings using relations between datasets
- DensMAP can preserve local density information in embeddings
- UMAP now depends on PyNNDescent, but has faster more parallel performance as a result
- UMAP now supports an `update` method to add new data and retrain.
- Various performance improvements and bug fixes
- Additional plotting support, including text searching in interactive plots
- disconnection_distance allows disconnecting points beyond a certain distance from our manifold. (Thanks to John Healy)

18.2 What's new in 0.4

- Inverse transform method. Generate points in the original space corresponding to points in embedded space. (Thanks to Joseph Courtney)
- Different embedding spaces. Support for embedding to a variety of different spaces other than Euclidean. (Thanks to Joseph Courtney)
- New metrics, including Hellinger distance for sparse count data.
- New discrete/label metrics, including hierarchical categories, counts, ordinal data, and string edit distance.
- Support for parallelism in neighbor search and layout optimization. (Thanks to Tom White)

- Support for alternative methods to handling duplicated data samples. (Thanks to John Healy)
- New plotting methods for fast and easy plots.
- Initial support for dataframe embedding – still experimental, but worth trying.
- Support for transform methods with sparse data.
- Multithreading support when no random seed is set.

18.3 What's new in 0.3

- Supervised and semi-supervised dimension reduction. Support for using labels or partial labels for dimension reduction.
- Transform method. Support for adding new unseen points to an existing embedding.
- Performance improvements.

18.4 What's new in 0.2

- A new layout algorithm that handles large datasets (more) correctly.
- Performance improvements.

CHAPTER 19

Frequently Asked Questions

Compiled here are a set of frequently asked questions, along with answers. If you don't find your question listed here then please feel free to add an issue on github. More questions are always welcome, and the authors will do their best to answer. If you feel you have a common question that isn't answered here then please suggest that the question (and answer) be added to the FAQ when you file the issue.

19.1 Should I normalise my features?

The default answer is yes, but, of course, the real answer is "it depends". If your features have meaningful relationships with one another (say, latitude and longitude values) then normalising per feature is not a good idea. For features that are essentially independent it does make sense to get all the features on (relatively) the same scale. The best way to do this is to use pre-processing tools from scikit-learn. All the advice given there applies as sensible preprocessing for UMAP, and since UMAP is scikit-learn compatible you can put all of this together into a scikit-learn pipeline.

19.2 Can I cluster the results of UMAP?

This is hard to answer well, but essentially the answer is "yes, with care". To start with it matters what clustering algorithm you are going to use. Since UMAP does not necessarily produce clean spherical clusters something like K-Means is a poor choice. I would recommend HDBSCAN or similar. The catch here is that UMAP, with its uniform density assumption, does not preserve density well. What UMAP will do, however, is contract connected components of the manifold together. Providing you have enough data for UMAP to distinguish that information then you can get *useful* clustering results out since algorithms like HDBSCAN will easily pick out the components after applying UMAP.

UMAP does offer significant improvements over algorithms like t-SNE for clustering. First, by preserving more global structure and creating meaningful separation between connected components of the manifold on which the data lies, UMAP offers more meaningful clusters. Second, because it supports arbitrary embedding dimensions, UMAP allows embedding to larger dimensional spaces that make it more amenable to clustering.

19.3 The clusters are all squashed together and I can't see internal structure

One of UMAPs goals is to have distance between clusters of points be meaningful. This means that clusters can end up spread out with a fair amount of space between them. As a result the clusters themselves can end up more visually packed together than in, say, t-SNE. This is intended. A catch, however, is that many plots (for example matplotlib's scatter plot with default parameters) tend to show the clusters only as indistinct blobs with no internal structure. The solution for this is really a matter of tuning the plot more than anything else.

If you are using matplotlib consider using the `s` parameter that specifies the glyph size in scatter plots. Depending on how much data you have reducing this to anything from 5 to 0.001 can have a notable effect. The `size` parameter in bokeh is similarly useful (but does not need to be quite so small).

More generally the real solution, particular with large datasets, is to use datashader for plotting. Datashader is a plotting library that handles aggregation of large scale data in scatter plots in a way that can better show the underlying detail that can otherwise be lost. We highly recommend investing the time to learn datashader for UMAP plot particularly for larger datasets.

19.4 I ran out of memory. Help!

For some datasets the default options for approximate nearest neighbor search can result in excessive memory use. If your dataset is not especially large but you have found that UMAP runs out of memory when operating on it consider using the `low_memory=True` option, which will switch to a slower but less memory intensive approach to computing the approximate nearest neighbors. This may alleviate your issues.

19.5 UMAP is eating all my cores. Help!

If run without a random seed UMAP will use numba's parallel implementation to do multithreaded work and use many cores. By default this will make use of as many cores as are available. If you are on a shared machine or otherwise don't wish to use *all* the cores at once you can restrict the number of threads that numba uses by making use of the environment variable `NUMBA_NUM_THREADS`; see the numba documentation for more details.

19.6 Is there GPU or multicore-CPU support?

There is basic multicore support as of version 0.4. In the future it is possible that GPU support may be added.

There is a UMAP implementation for GPU available in the NVIDIA RAPIDS cuML library, so if you need GPU support that is currently the best place to go.

19.7 Can I add a custom loss function?

To allow for fast performance the SGD phase of UMAP has been hand-coded for the specific needs of UMAP. This makes custom loss functions a little difficult to handle. Now that Numba (as of version 0.38) supports passing functions it is possible that future versions of UMAP may support such functionality. In the meantime you should definitely look into smallvis, a library for t-SNE, LargeVis, UMAP, and related algorithms. Smallvis only works for small datasets, but provides much greater flexibility and control.

19.8 Is there support for the R language?

Yes! A number of people have worked hard to make UMAP available to R users.

If you want to use the reference implementation under the hood but want a nice R interface then we recommend umap, which wraps the python code with reticulate. Another reticulate interface is umapr, but it may not be under active development.

If you want a pure R version then we recommend uwot at this time. umap also provides a pure R implementation in addition to its reticulate wrapper.

Both umap and uwot are available on CRAN.

19.9 Is there a C/C++ implementation?

Not that we are aware of. For now Numba has done a very admirable job of providing high performance and the developers of UMAP have not felt the need to move to lower level languages. At some point a multithreaded C++ implementation may be made available, but there are no time-frames for when that would happen.

19.10 I can't get UMAP to run properly!

There are, inevitably, a number of issues and corner cases that can cause issues for UMAP. Some know issues that can cause problems are:

- UMAP doesn't currently support 32-bit Windows. This is due to issues with Numba of that platform and will not likely be resolved soon. Sorry :-(
- If you have pip installed the package `umap` at any time (instead of `umap-learn`) this can cause serious issues. You will want to purge/remove everything umap related in your `site-packages` directory and re-install `umap-learn`.
- Having any files called `umap.py` in the current directory you will have issues as that will be loaded instead of the `umap` module.

It is worth checking the issues page on github for potential solutions. If all else fails please add an issue on github.

19.11 What is the difference between PCA / UMAP / VAEs?

This is an example of an embedding for a popular Fashion MNIST dataset.

Note that FMNIST is mostly a toy dataset (MNIST on steroids). On such a simplistic case UMAP shows distillation results (i.e. if we use its embedding in a downstream task like classification) comparable to VAEs, which are more computationally expensive.

By definition:

- PCA is linear transformation, you can apply it to mostly any kind of data in an unsupervised fashion. Also it works really fast. For most real world tasks its embeddings are mostly too simplistic / useless.
- VAE is a kind of encoder-decoder neural network, trained with KLD loss and BCE (or MSE) loss to enforce the resulting embedding to be continuous. VAE is an extension of auto-encoder networks, which by design should produce embeddings that are not only relevant to actually encoding the data, but are also smooth.

From a more practical standpoint:

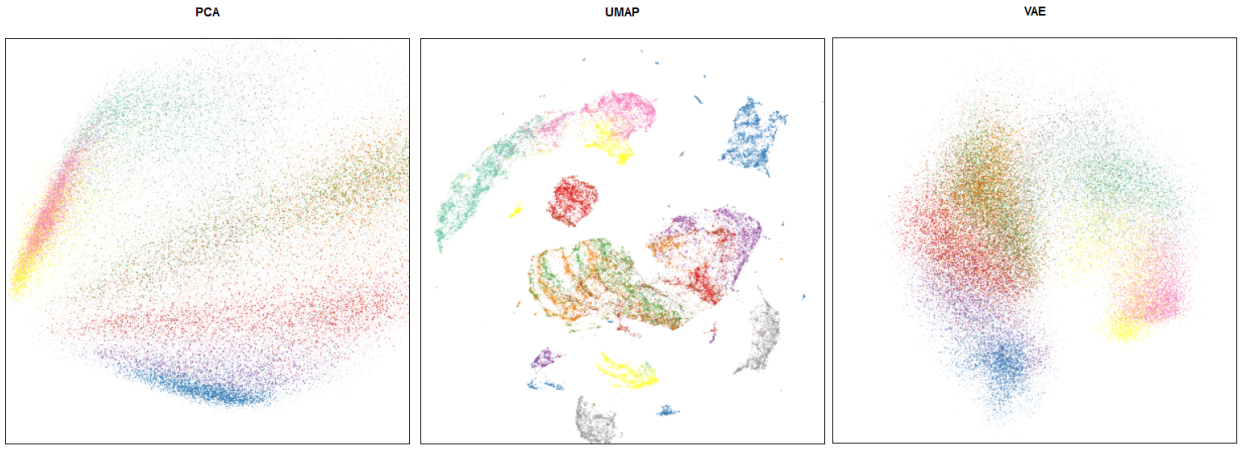


Fig. 1: Comparison of PCA / UMAP / VAE embeddings

- PCA mostly works for any reasonable dataset on a modern machine. (up to tens or hundreds of millions of rows);
- VAEs have been shown to work only for toy datasets and to our knowledge there was no real life useful application to a real world sized dataset (i.e. ImageNet);
- Applying UMAP to real world tasks usually provides a good starting point for downstream tasks (data visualization, clustering, classification) and works reasonably fast;
- Consider a typical pipeline: high-dimensional embedding (300+) => PCA to reduce to 50 dimensions => UMAP to reduce to 10-20 dimensions => HDBSCAN for clustering / some plain algorithm for classification;

Which tool should I use?

- PCA for very large or high dimensional datasets (or maybe consider finding a domain specific matrix factorization technique, e.g. topic modelling for texts);
- UMAP for smaller datasets;
- VAEs are mostly experimental;

Where can I learn more?

- While PCA is ubiquitous, you may look at this example comparing PCA / UMAP / VAEs;

19.12 How UMAP can go wrong

One way UMAP can go wrong is the introduction of data points that are maximally far apart from all other points in your data set. In other words, a points nearest neighbour is maximally far from it. A common example of this could be a point which shares no features in common with any other point under a Jaccard distance or a point whose nearest neighbour is np.inf from it under a continuous distance function. In both these cases UMAPs assumption of all points lying on a connected manifold can lead us astray. From this points perspective all other points are equally valid nearest neighbours so its k-nearest neighbour query will return a random selection of neighbours all at this maximal distance. Next we will normalize this distance by applying our UMAP kernel which says that a point should be maximally similar to it's nearest neighbour. Since all k-nearest neighbours are identically far apart they will all be considered maximally similar by our point in question. When we try to embed our data into a low dimensional space our optimization will attempt to pull all these randomly selected points together. Add a sufficiently large number of these points and our entire space gets pulled together destroying any of the structure we had hoped to identify.

To circumvent this problem we've added a disconnection_distance parameter to UMAP which will cut any edge with a distance greater than the value passed in. This parameter defaults to `None`. When set to `None` the disconnection_distance will be set to the maximal value for any of our supported bounded metrics and otherwise set to np.inf. Removing these edges from the UMAP graph will disconnect our manifold and cause these points to start where they are initialized and get pushed away from all other points via the our optimization.

If a user has a good understanding of their distance metric they can set this value by hand to prevent data in particularly sparse regions of their space from becoming connected to their manifold.

If vertices in your graph are disconnected a warning message will be thrown. At that point a user can make use of the umap.utils.disconnected_vertices() function to identify the disconnected points. This can be used either for filtering and retraining a new UMAP model or simple to bed used as a filter for visualization purposes as seen below.

```
umap_model = umap.UMAP().fit(data)
disconnected_points = umap.utils.disconnected_vertices(umap_model)
umap.plot.points(umap_model, subset_points=~disconnected_points)
```

19.13 Successful use-cases

UMAP can be / has been successfully applied to the following domains:

- Single cell data visualization in biology;
- Mapping malware based on behavioural data;
- Pre-processing phrase vectors for clustering;
- Pre-processing image embeddings (Inception) for clustering;

and many more – if you have a successful use-case please submit a pull request adding it to this list!

CHAPTER 20

How UMAP Works

UMAP is an algorithm for dimension reduction based on manifold learning techniques and ideas from topological data analysis. It provides a very general framework for approaching manifold learning and dimension reduction, but can also provide specific concrete realizations. This article will discuss how the algorithm works in practice. There exist deeper mathematical underpinnings, but for the sake of readability by a general audience these will merely be referenced and linked. If you are looking for the mathematical description please see the UMAP paper.

To begin making sense of UMAP we will need a little bit of mathematical background from algebraic topology and topological data analysis. This will provide a basic algorithm that works well in theory, but unfortunately not so well in practice. The next step will be to make use of some basic Riemannian geometry to bring real world data a little closer to the underlying assumptions of the topological data analysis algorithm. Unfortunately this will introduce new complications, which will be resolved through a combination of deep math (details of which will be elided) and fuzzy logic. We can then put the pieces back together again, and combine them with a new approach to finding a low dimensional representation more fitting to the new data structures at hand. Putting this all together we arrive at the basic UMAP algorithm.

20.1 Topological Data Analysis and Simplicial Complexes

Simplicial complexes are a means to construct topological spaces out of simple combinatorial components. This allows one to reduce the complexities of dealing with the continuous geometry of topological spaces to the task of relatively simple combinatorics and counting. This method of taming geometry and topology will be fundamental to our approach to topological data analysis in general, and dimension reduction in particular.

The first step is to provide some simple combinatorial building blocks called *simplices*. Geometrically a simplex is a very simple way to build an k-dimensional object. A k dimensional simplex is called a k-simplex, and it is formed by taking the convex hull of $k+1$ independent points. Thus a 0-simplex is a point, a 1-simplex is a line segment (between two zero simplices), a 2-simplex is a triangle (with three 1-simplices as "faces"), and a 3-simplex is a tetrahedron (with four 2-simplices as "faces"). Such a simple construction allows for easy generalization to arbitrary dimensions.

This has a very simple combinatorial underlying structure, and ultimately one can regard a k-simplex as an arbitrary set of $k+1$ objects with faces (and faces of faces etc.) given by appropriately sized subsets – one can always provide a "geometric realization" of this abstract set description by constructing the corresponding geometric simplex.

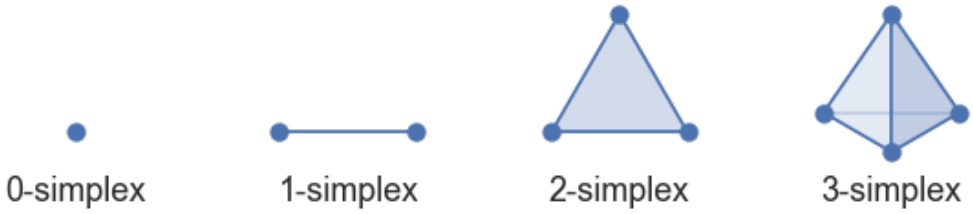


Fig. 1: Low dimensional simplices

Simplices can provide building blocks, but to construct interesting topological spaces we need to be able to glue together such building blocks. This can be done by constructing a *simplicial complex*. Ostensibly a simplicial complex is a set of simplices glued together along faces. More explicitly a simplicial complex $\mathcal{K}$ is a set of simplices such that any face of any simplex in $\mathcal{K}$ is also in $\mathcal{K}$ (ensuring all faces exist), and the intersection of any two simplices in $\mathcal{K}$ is a face of both simplices. A large class of topological spaces can be constructed in this way – just gluing together simplices of various dimensions along their faces. A little further abstraction will get to *simplicial sets* which are purely combinatorial, have a nice category theoretic presentation, and can generate a much broader class of topological spaces, but that will take us too far afield for this article. The intuition of simplicial complexes will be enough to illustrate the relevant ideas and motivation.

How does one apply these theoretical tools from topology to finite sets of data points? To start we'll look at how one might construct a simplicial complex from a topological space. The tool we will consider is the construction of a Čech complex given an open cover of a topological space. That's a lot of verbiage if you haven't done much topology, but we can break it down fairly easily for our use case. An open cover is essentially just a family of sets whose union is the whole space, and a Čech complex is a combinatorial way to covert that into a simplicial complex. It works fairly simply: let each set in the cover be a 0-simplex; create a 1-simplex between two such sets if they have a non-empty intersection; create a 2-simplex between three such sets if the triple intersection of all three is non-empty; and so on. Now, that doesn't sound very advanced – just looking at intersections of sets. The key is that the background topological theory actually provides guarantees about how well this simple process can produce something that represents the topological space itself in a meaningful way (the Nerve theorem is the relevant result for those interested). Obviously the quality of the cover is important, and finer covers provide more accuracy, but the reality is that despite its simplicity the process captures much of the topology.

Next we need to understand how to apply that process to a finite set of data samples. If we assume that the data samples are drawn from some underlying topological space then to learn about the topology of that space we need to generate an open cover of it. If our data actually lie in a metric space (i.e. we can measure distance between points) then one way to approximate an open cover is to simply create balls of some fixed radius about each data point. Since we only have finite samples, and not the topological space itself, we cannot be sure it is truly an open cover, but it might be as good an approximation as we could reasonably expect. This approach also has the advantage that the Čech complex associated to the cover will have a 0-simplex for each data point.

To demonstrate the process let's consider a test dataset like this

If we fix a radius we can then picture the open sets of our cover as circles (since we are in a nice visualizable two dimensional case). The result is something like this

We can then depict the the simplicial complex of 0-, 1-, and 2-simplices as points, lines, and triangles

It is harder to easily depict the higher dimensional simplices, but you can imagine how they would fit in. There are two things to note here: first, the simplicial complex does a reasonable job of starting to capture the fundamental topology of the dataset; second, most of the work is really done by the 0- and 1-simplices, which are easier to deal with computationally (it is just a graph, in the nodes and edges sense). The second observation motivates the Vietoris-Rips

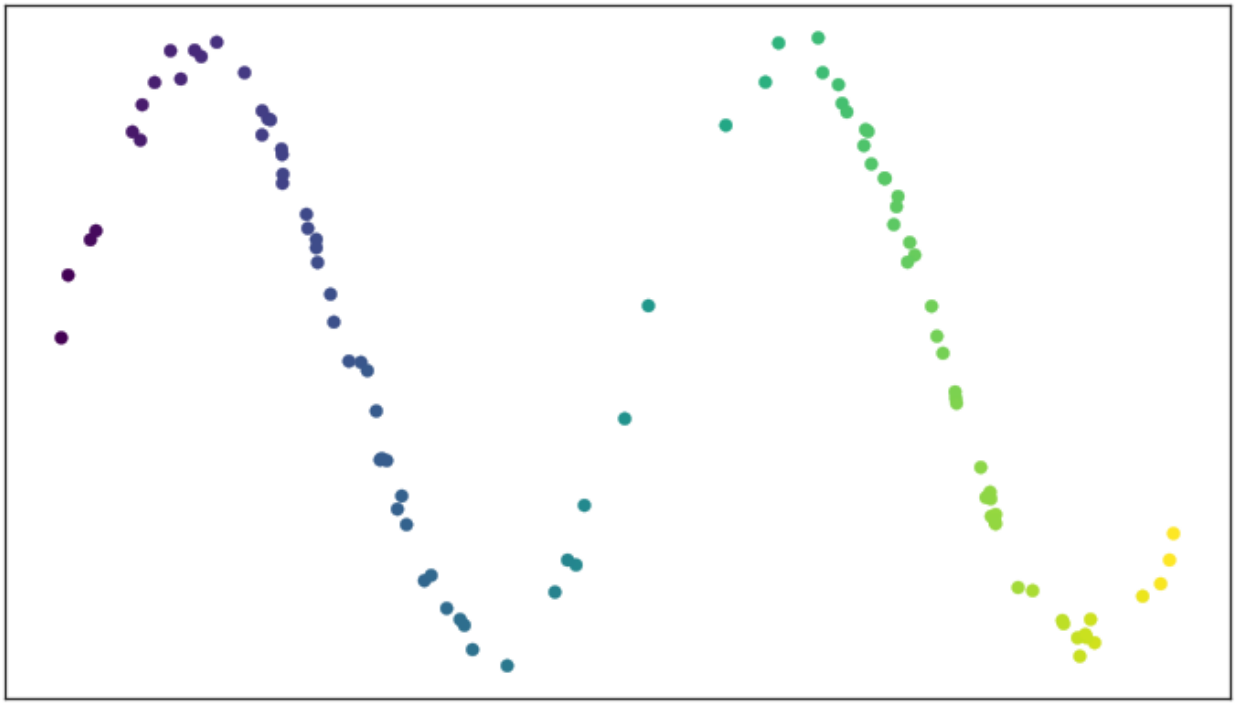


Fig. 2: Test data set of a noisy sine wave

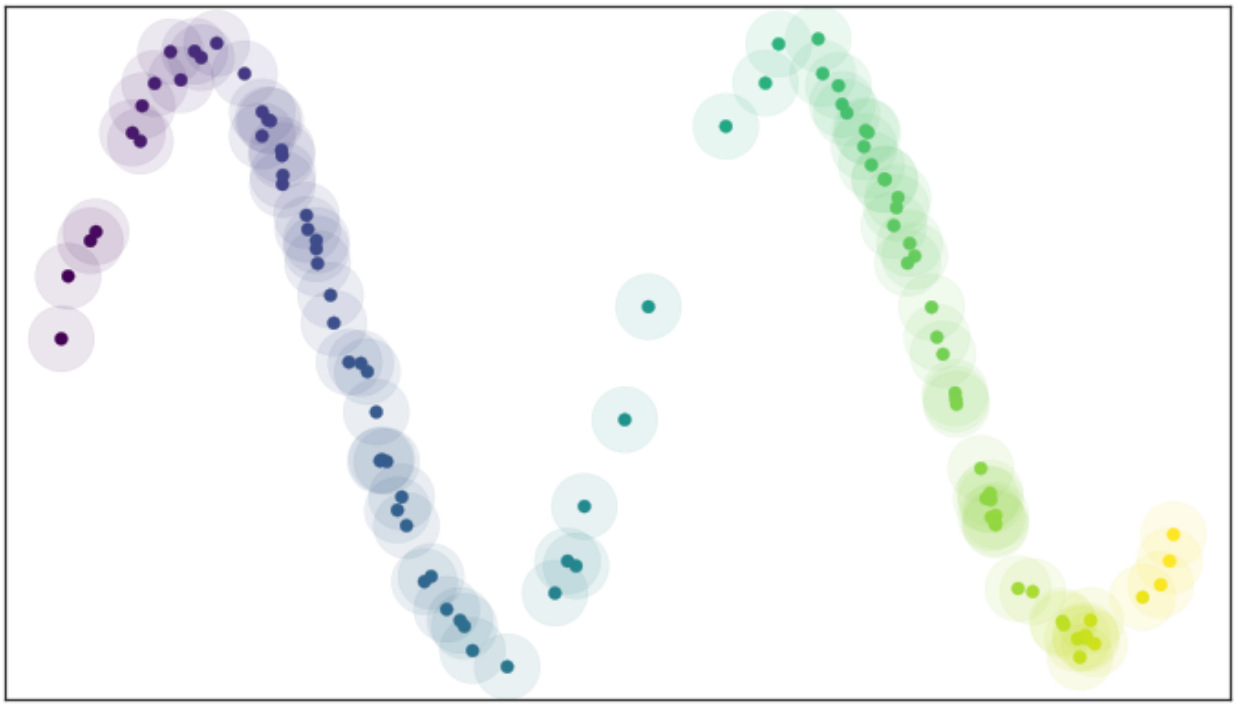


Fig. 3: A basic open cover of the test data

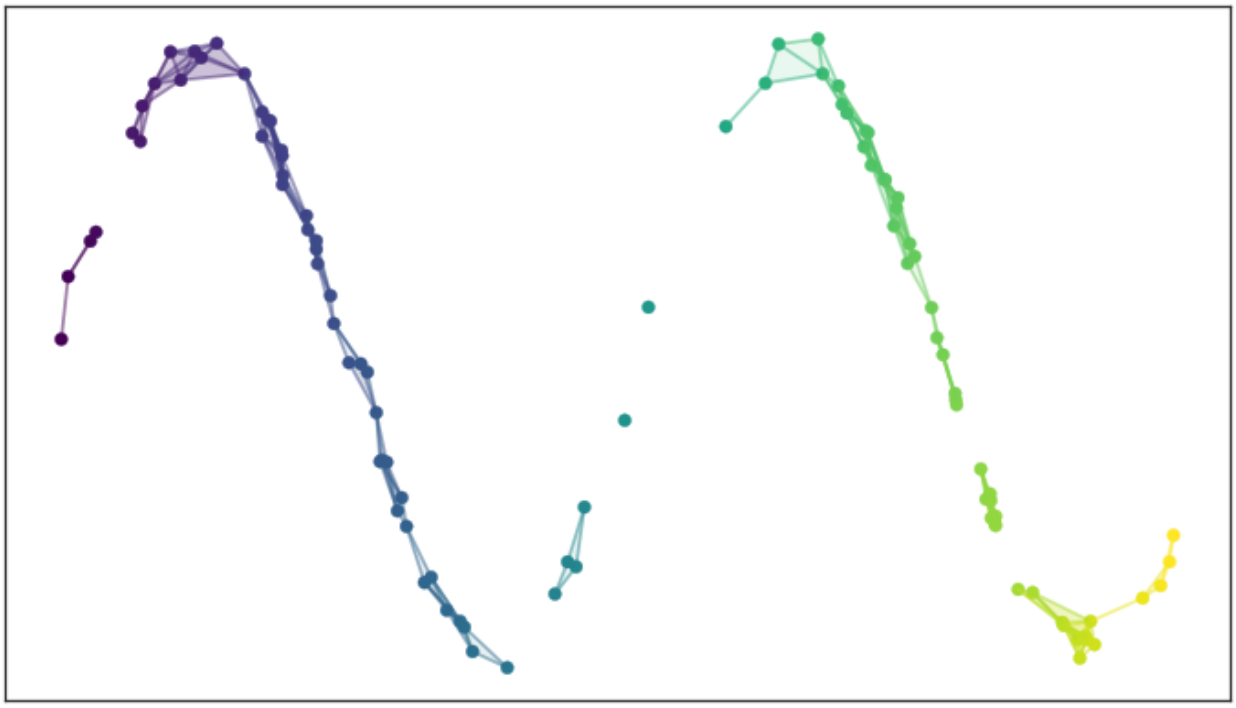


Fig. 4: A simplicial complex built from the test data

complex, which is similar to the Čech complex but is entirely determined by the 0- and 1-simplices. Vietoris-Rips complexes are much easier to work with computationally, especially for large datasets, and are one of the major tools of topological data analysis.

If we take this approach to get a topological representation then we can build a dimension reduction algorithm by finding a low dimensional representation of the data that has a similar topological representation. If we only care about the 0- and 1-simplices then the topological representation is just a graph, and finding a low dimensional representation can be described as a **'graph layout problem <>'__**. If one wants to use, for example, spectral methods for graph layout then we arrive at algorithms like **'Laplacian eigenmaps <>'__** and **'Diffusion maps <>'__**. Force directed layouts are also an option, and provide algorithms closer to **'MDS <>'__** or **'Sammon mapping <>'__** in flavour.

I would not blame those who have read this far to wonder why we took such an abstract roundabout road to simply building a neighborhood-graph on the data and then laying out that graph. There are a couple of reasons. The first reason is that the topological approach, while abstract, provides sound theoretical justification for what we are doing. While building a neighborhood-graph and laying it out in lower dimensional space makes heuristic sense and is computationally tractable, it doesn't provide the same underlying motivation of capturing the underlying topological structure of the data faithfully – for that we need to appeal to the powerful topological machinery I've hinted lies in the background. The second reason is that it is this more abstract topological approach that will allow us to generalize the approach and get around some of the difficulties of the sorts of algorithms described above. While ultimately we will end up with a process that is fairly simple computationally, understanding *why* various manipulations matter is important to truly understanding the algorithm (as opposed to merely computing with it).

20.2 Adapting to Real World Data

The approach described above provides a nice theory for why a neighborhood graph based approach should capture manifold structure when doing dimension reduction. The problem tends to come when one tries to put the theory into practice. The first obvious difficulty (and we can see it even our example above) is that choosing the right radius for

the balls that make up the open cover is hard. If you choose something too small the resulting simplicial complex splits into many connected components. If you choose something too large the simplicial complex turns into just a few very high dimensional simplices (and their faces etc.) and fails to capture the manifold structure anymore. How should one solve this?

The dilemma is in part due to the theorem (called the Nerve theorem) that provides our justification that this process captures the topology. Specifically, the theorem says that the simplicial complex will be (homtopically) equivalent to the union of the cover. In our case, working with finite data, the cover, for certain radii, doesn't cover the whole of the manifold that we imagine underlies the data – it is that lack of coverage that results in the disconnected components. Similarly, where the points are too bunched up, our cover does cover "too much" and we end up with higher dimensional simplices than we might ideally like. If the data were *uniformly distributed* across the manifold then selecting a suitable radius would be easy – the average distance between points would work well. Moreover with a uniform distribution we would be guaranteed that our cover would actually cover the whole manifold with no "gaps" and no unnecessarily disconnected components. Similarly, we would not suffer from those unfortunate bunching effects resulting in unnecessarily high dimensional simplices.

If we consider data that is uniformly distributed along the same manifold it is not hard to pick a good radius (a little above half the average distance between points) and the resulting open cover looks pretty good:

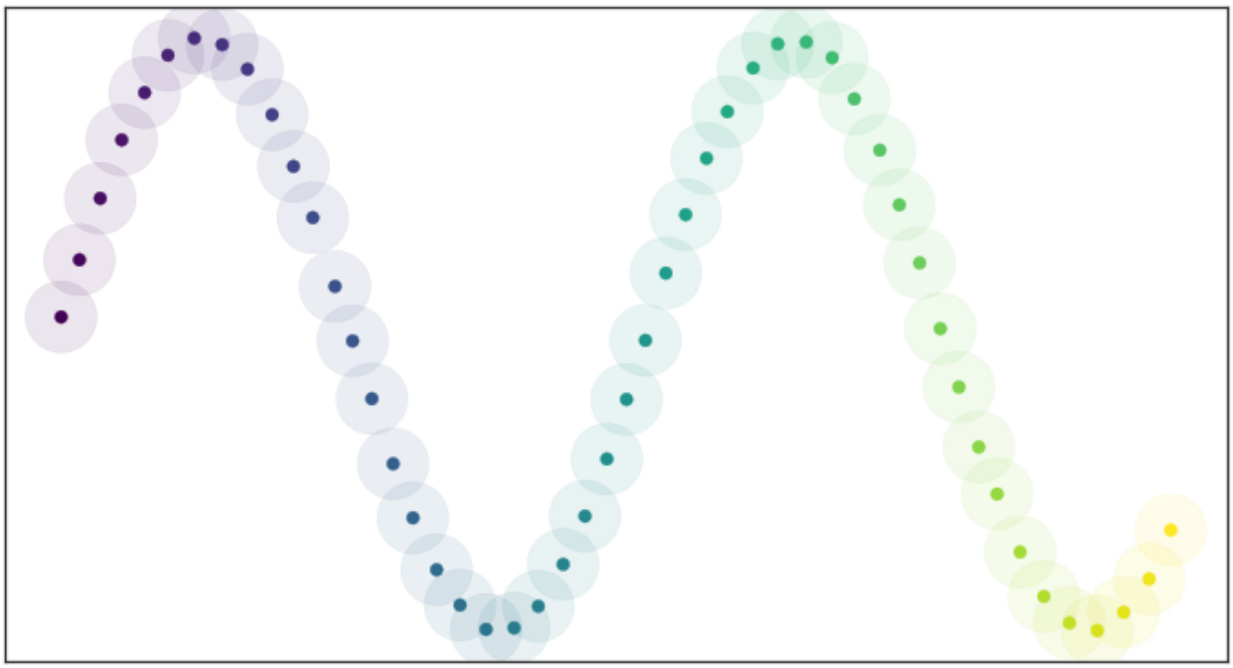


Fig. 5: Open balls over uniformly_distributed_data

Because the data is evenly spread we actually cover the underlying manifold and don't end up with clumping. In other words, all this theory works well assuming that the data is uniformly distributed over the manifold.

Unsurprisingly this uniform distribution assumption crops up elsewhere in manifold learning. The proofs that Laplacian eigenmaps work well require the assumption that the data is uniformly distributed on the manifold. Clearly if we had a uniform distribution of points on the manifold this would all work a lot better – but we don't! Real world data simply isn't that nicely behaved. How can we resolve this? By turning the problem on its head: assume that the data is uniformly distributed on the manifold, and ask what that tells us about the manifold itself. If the data *looks* like it isn't uniformly distributed that must simply be because the notion of distance is varying across the manifold – space itself is warping: stretching or shrinking according to where the data appear sparser or denser.

By assuming that the data is uniformly distributed we can actually compute (an approximation of) a local notion of distance for each point by making use of a little standard Riemannian geometry. In practical terms, once you push the

math through, this turns out to mean that a unit ball about a point stretches to the k-th nearest neighbor of the point, where k is the sample size we are using to approximate the local sense of distance. Each point is given its own unique distance function, and we can simply select balls of radius one with respect to that local distance function!

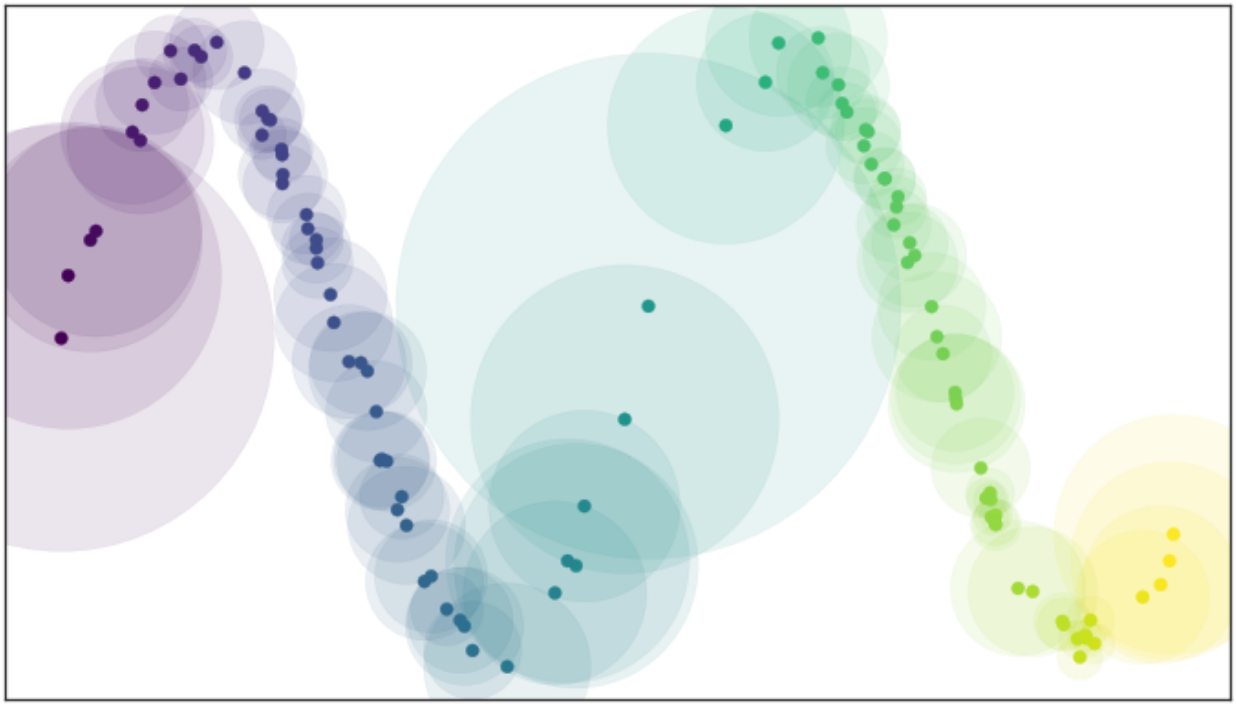


Fig. 6: Open balls of radius one with a locally varying metric

This theoretically derived result matches well with many traditional graph based algorithms: a standard approach for such algorithms is to use a k-neighbor graph instead of using balls of some fixed radius to define connectivity. What this means is that each point in the dataset is given an edge to each of its k nearest neighbors – the effective result of our locally varying metric with balls of radius one. Now, however, we can explain why this works in terms of simplicial complexes and the Nerve theorem.

Of course we have traded choosing the radius of the balls for choosing a value for k. However it is often easier to pick a resolution scale in terms of number of neighbors than it is to correctly choose a distance. This is because choosing a distance is very dataset dependent: one needs to look at the distribution of distances in the dataset to even begin to select a good value. In contrast, while a k value is still dataset dependent to some degree, there are reasonable default choices, such as the 10 nearest neighbors, that should work acceptably for most datasets.

At the same time the topological interpretation of all of this gives us a more meaningful interpretation of k. The choice of k determines how locally we wish to estimate the Riemannian metric. A small choice of k means we want a very local interpretation which will more accurately capture fine detail structure and variation of the Riemannian metric. Choosing a large k means our estimates will be based on larger regions, and thus, while missing some of the fine detail structure, they will be more broadly accurate across the manifold as a whole, having more data to make the estimate with.

We also get a further benefit from this Riemannian metric based approach: we actually have a local metric space associated to each point, and can meaningfully measure distance, and thus we could weight the edges of the graph we might generate by how far apart (in terms of the local metric) the points on the edges are. In slightly more mathematical terms we can think of this as working in a fuzzy topology where being in an open set in a cover is no longer a binary yes or no, but instead a fuzzy value between zero and one. Obviously the certainty that points are in a ball of a given radius will decay as we move away from the center of the ball. We could visualize such a fuzzy cover as looking something like this

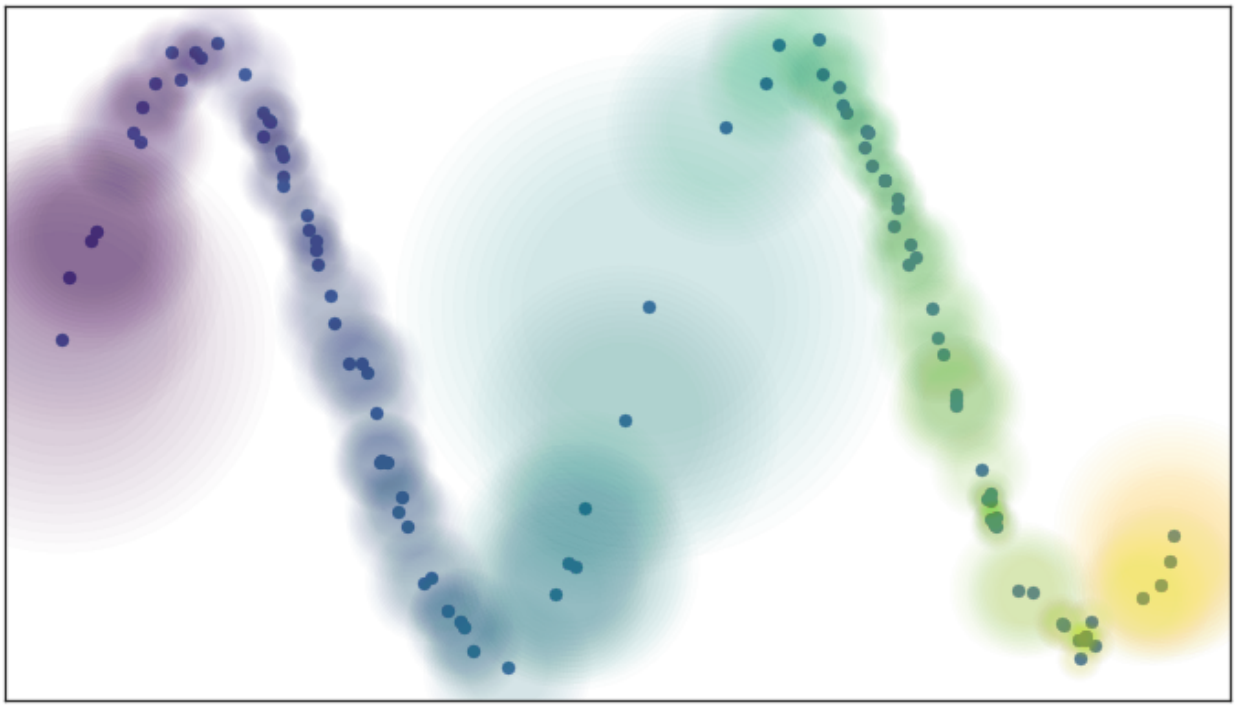


Fig. 7: Fuzzy open balls of radius one with a locally varying metric

None of that is very concrete or formal – it is merely an intuitive picture of what we would like to have happen. It turns out that we can actually formalize all of this by stealing the singular set and geometric realization functors from algebraic topology and then adapting them to apply to metric spaces and fuzzy simplicial sets. The mathematics involved in this is outside the scope of this exposition, but for those interested you can look at the original work on this by David Spivak and our paper. It will have to suffice to say that there is some mathematical machinery that lets us realize this intuition in a well defined way.

This resolves a number of issues, but a new problem presents itself when we apply this sort of process to real data, especially in higher dimensions: a lot of points become essentially totally isolated. One would imagine that this shouldn't happen if the manifold the data was sampled from isn't pathological. So what property are we expecting that manifold to have that we are somehow missing with the current approach? What we need to add is the idea of local connectivity.

Note that this is not a requirement that the manifold as a whole be connected – it can be made up of many connected components. Instead it is a requirement that at any point on the manifold there is some sufficiently small neighborhood of the point that *is* connected (this "in a sufficiently small neighborhood" is what the "local" part means). For the practical problem we are working with, where we only have a finite approximation of the manifold, this means that no point should be *completely* isolated – it should connect to at least one other point. In terms of fuzzy open sets what this amounts to is that we should have complete confidence that the open set extends as far as the closest neighbor of each point. We can implement this by simply having the fuzzy confidence decay in terms of distance *beyond* the first nearest neighbor. We can visualize the result in terms of our example dataset again.

Again this can be formalized in terms of the aforementioned mathematical machinery from algebraic topology. From a practical standpoint this plays an important role for high dimensional data – in high dimensions distances tend to be larger, but also more similar to one another (see the curse of dimensionality). This means that the distance to the first nearest neighbor can be quite large, but the distance to the tenth nearest neighbor can often be only slightly larger (in relative terms). The local connectivity constraint ensures that we focus on the difference in distances among nearest neighbors rather than the absolute distance (which shows little differentiation among neighbors).

Just when we think we are almost there, having worked around some of the issues of real world data, we run aground

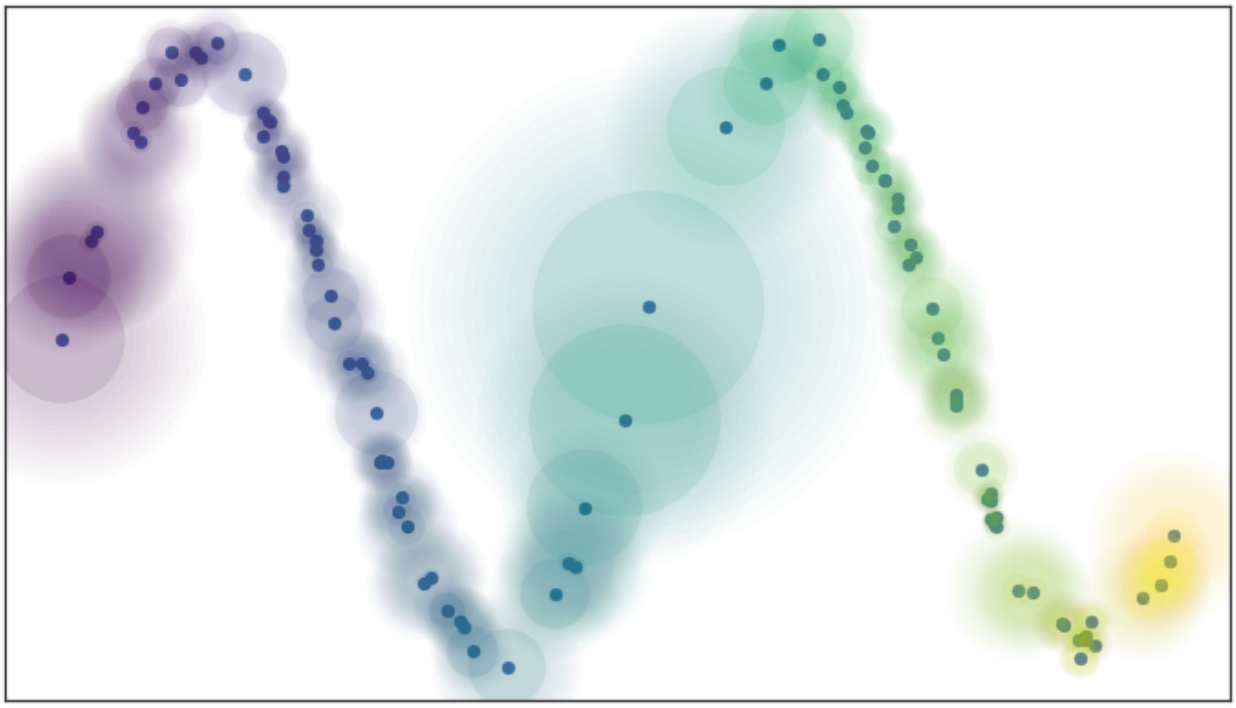


Fig. 8: Local connectivity and fuzzy open sets

on a new obstruction: our local metrics are not compatible! Each point has its own local metric associated to it, and from point *a*'s perspective the distance from point *a* to point *b* might be 1.5, but from the perspective of point *b* the distance from point *b* to point *a* might only be 0.6. Which point is right? How do we decide? Going back to our graph based intuition we can think of this as having directed edges with varying weights something like this.

Between any two points we might have up to two edges and the weights on those edges disagree with one another. There are a number of options for what to do given two disagreeing weights – we could take the maximum, the minimum, the arithmetic mean, the geometric mean, or something else entirely. What we would really like is some principled way to make the decision. It is at this point that the mathematical machinery we built comes into play. Mathematically we actually have a family of fuzzy simplicial sets, and the obvious choice is to take their union – a well defined operation. There are a a few ways to define fuzzy unions, depending on the nature of the logic involved, but here we have relatively clear probabilistic semantics that make the choice straightforward. In graph terms what we get is the following: if we want to merge together two disagreeing edges with weight *a* and *b* then we should have a single edge with combined weight $a + b - a \cdot b$. The way to think of this is that the weights are effectively the probabilities that an edge (1-simplex) exists. The combined weight is then the probability that at least one of the edges exists.

If we apply this process to union together all the fuzzy simplicial sets we end up with a single fuzzy simplicial complex, which we can again think of as a weighted graph. In computational terms we are simply applying the edge weight combination formula across the whole graph (with non-edges having a weight of 0). In the end we have something that looks like this.

So in some sense in the end we have simply constructed a weighted graph (although we could make use of higher dimensional simplices if we wished, just at significant extra computational cost). What the mathematical theory lurking in the background did for us is determine *why* we should construct *this* graph. It also helped make the decisions about exactly *how* to compute things, and gives a concrete interpretation of *what* this graph means. So while in the end we just constructed a graph, the math answered the important questions to get us here, and can help us determine what to do next.

So given that we now have a fuzzy topological representation of the data (which the math says will capture the topology

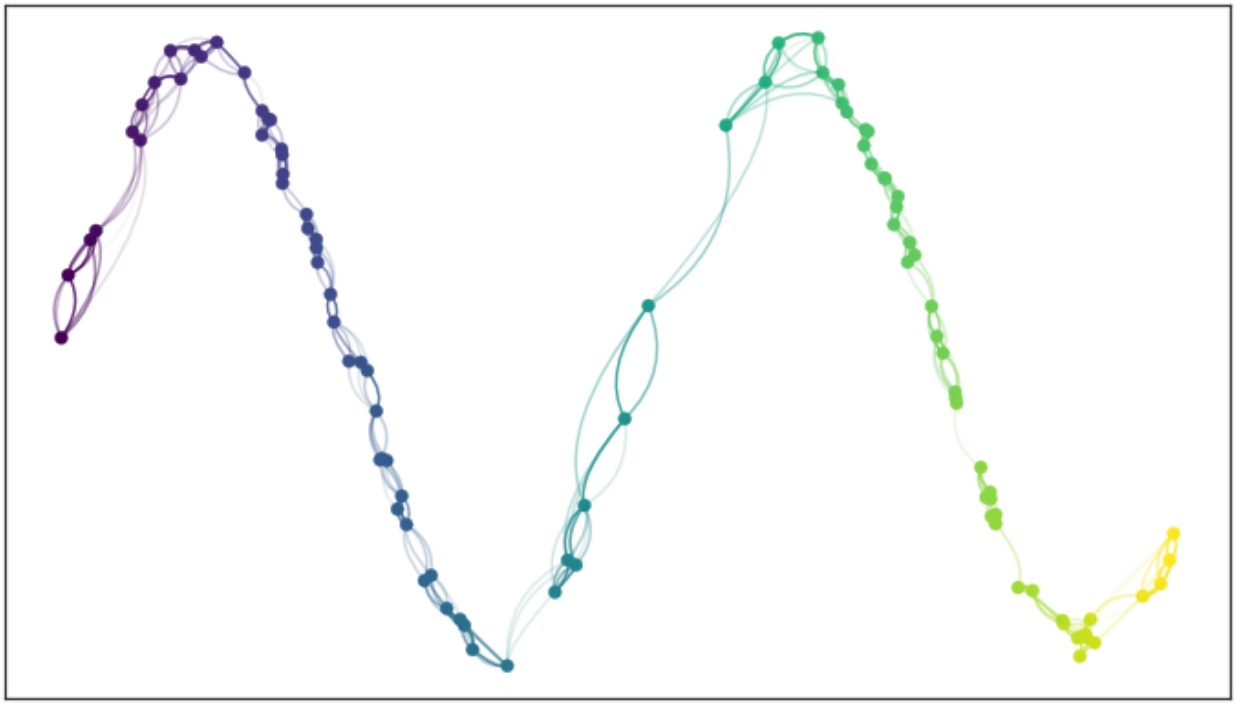


Fig. 9: Edges with incompatible weights

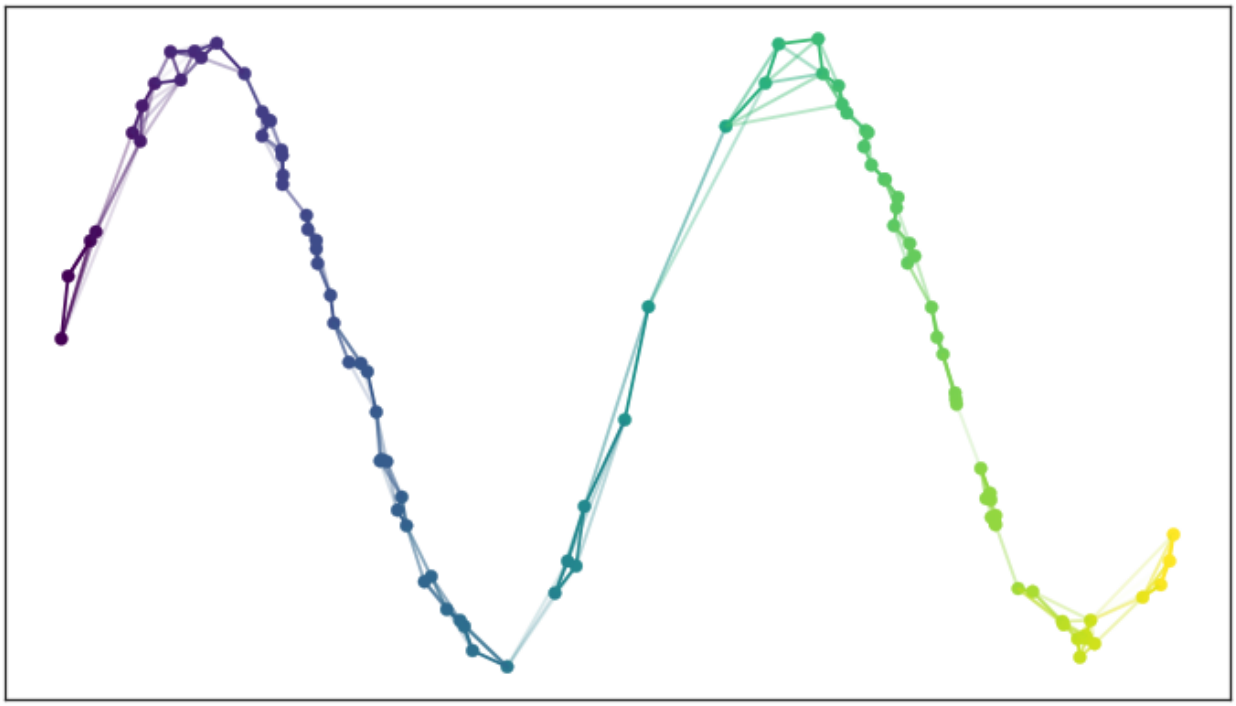


Fig. 10: Graph with combined edge weights

of the manifold underlying the data), how do we go about converting that into a low dimensional representation?

20.3 Finding a Low Dimensional Representation

Ideally we want the low dimensional representation to have as similar a fuzzy topological structure as possible. The first question is how do we determine the fuzzy topological structure of a low dimensional representation, and the second question is how do we find a good one.

The first question is largely already answered – we should presumably follow the same procedure we just used to find the fuzzy topological structure of our data. There is a quirk, however: this time around the data won't be lying on some manifold, we'll have a low dimensional representation that is lying on a very particular manifold. That manifold is, of course, just the low dimensional euclidean space we are trying to embed into. This means that all the effort we went to previously to make vary the notion of distance across the manifold is going to be misplaced when working with the low dimensional representation. We explicitly *want* the distance on the manifold to be standard euclidean distance with respect to the global coordinate system, not a varying metric. That saves some trouble. The other quirk is that we made use of the distance to the nearest neighbor, again something we computed given the data. This is also a property we would like to be globally true across the manifold as we optimize toward a good low dimensional representation, so we will have to accept it as a hyper-parameter `min_dist` to the algorithm.

The second question, 'how do we find a good low dimensional representation', hinges on our ability to measure how "close" a match we have found in terms of fuzzy topological structures. Given such a measure we can turn this into an optimization problem of finding the low dimensional representation with the closest fuzzy topological structure. Obviously if our measure of closeness turns out to have various properties the nature of the optimization techniques we can apply will differ.

Going back to when we were merging together the conflicting weights associated to simplices, we interpreted the weights as the probability of the simplex existing. Thus, since both topological structures we are comparing share the same 0-simplices, we can imagine that we are comparing the two vectors of probabilities indexed by the 1-simplices. Given that these are Bernoulli variables (ultimately the simplex either exists or it doesn't, and the probability is the parameter of a Bernoulli distribution), the right choice here is the cross entropy.

Explicitly, if the set of all possible 1-simplices is E, and we have weight functions such that $w_h(e)$ is the weight of the 1-simplex e in the high dimensional case and $w_l(e)$ is the weight of e in the low dimensional case, then the cross entropy will be

$$\sum_{e \in E} w_h(e) \log\left(\frac{w_h(e)}{w_l(e)}\right) + (1 - w_h(e)) \log\left(\frac{1 - w_h(e)}{1 - w_l(e)}\right)$$

This might look complicated, but if we go back to thinking in terms of a graph we can view minimizing the cross entropy as a kind of force directed graph layout algorithm.

The first term, $w_h(e) \log\left(\frac{w_h(e)}{w_l(e)}\right)$, provides an attractive force between the points e spans whenever there is a large weight associated to the high dimensional case. This is because this term will be minimized when $w_l(e)$ is as large as possible, which will occur when the distance between the points is as small as possible.

In contrast the second term, $(1 - w_h(e)) \log\left(\frac{1-w_h(e)}{1-w_l(e)}\right)$, provides a repulsive force between the ends of e whenever $w_h(e)$ is small. This is because the term will be minimized by making $w_l(e)$ as small as possible.

On balance this process of pull and push, mediated by the weights on edges of the topological representation of the high dimensional data, will let the low dimensional representation settle into a state that relatively accurately represents the overall topology of the source data.

20.4 The UMAP Algorithm

Putting all these pieces together we can construct the UMAP algorithm. The first phase consists of constructing a fuzzy topological representation, essentially as described above. The second phase is simply optimizing the low dimensional representation to have as close a fuzzy topological representation as possible as measured by cross entropy.

When constructing the initial fuzzy topological representation we can take a few shortcuts. In practice, since fuzzy set membership strengths decay away to be vanishingly small, we only need to compute them for the nearest neighbors of each point. Ultimately that means we need a way to quickly compute (approximate) nearest neighbors efficiently, even in high dimensional spaces. We can do this by taking advantage of the Nearest-Neighbor-Descent algorithm of Dong et al. The remaining computations are now only dealing with local neighbors of each point and are thus very efficient.

In optimizing the low dimensional embedding we can again take some shortcuts. We can use stochastic gradient descent for the optimization process. To make the gradient descent problem easier it is beneficial if the final objective function is differentiable. We can arrange for that by using a smooth approximation of the actual membership strength function for the low dimensional representation, selecting from a suitably versatile family. In practice UMAP uses the family of curves of the form $\frac{1}{1+ax^{2b}}$. Equally we don't want to have to deal with all possible edges, so we can use the negative sampling trick (as used by word2vec and LargeVis), to simply sample negative examples as needed. Finally since the Laplacian of the topological representation is an approximation of the Laplace-Beltrami operator of the manifold we can use spectral embedding techniques to initialize the low dimensional representation into a good state.

Putting all these pieces together we arrive at an algorithm that is fast and scalable, yet still built out of sound mathematical theory. Hopefully this introduction has helped provide some intuition for that underlying theory, and for how the UMAP algorithm works in practice.

CHAPTER 21

Performance Comparison of Dimension Reduction Implementations

Different dimension reduction techniques can have quite different computational complexity. Beyond the algorithm itself there is also the question of how exactly it is implemented. These two factors can have a significant role in how long it actually takes to run a given dimension reduction. Furthermore the nature of the data you are trying to reduce can also matter – mostly the involves the dimensionality of the original data. Here we will take a brief look at the performance characterstics of a number of dimension reduction implementations.

To start let's get the basic tools we'll need loaded up – numpy and pandas obviously, but also tools to get and resample the data, and the time module so we can perform some basic benchmarking.

```
import numpy as np
import pandas as pd
from sklearn.datasets import fetch_openml
from sklearn.utils import resample
import time
```

Next we'll need the actual dimension reduction implementations. For the purposes of this explanation we'll mostly stick with scikit-learn, but for the sake of comparison we'll also include the MulticoreTSNE implementation of t-SNE, and openTSNE both of which have historically had significantly better performance than scikit-learn t-SNE (more recent versions of scikit-learn have improved t-SNE performance).

```
from sklearn.manifold import TSNE, LocallyLinearEmbedding, Isomap, MDS,␣
↪SpectralEmbedding
from sklearn.decomposition import PCA
from MulticoreTSNE import MulticoreTSNE
from openTSNE import TSNE as OpenTSNE
from umap import UMAP
```

Next we'll need out plotting tools, and, of course, some data to work with. For this performance comparison we'll default to the now standard benchmark of manifold learning: the MNIST digits dataset. We can use scikit-learn's `fetch_openml` to grab it for us.

```
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

```
sns.set(context='notebook',
        rc={'figure.figsize':(12,10)},
        palette=sns.color_palette('tab10', 10))
```

```
mnist = fetch_openml('mnist_784', version=1, return_X_y=True)
```

```
mnist_data = mnist[0]
mnist_labels = mnist[1].astype(int)
```

Now it is time to start looking at performance. To start with let's look at how performance scales with increasing dataset size.

21.1 Performance scaling by dataset size

As the size of a dataset increases the runtime of a given dimension reduction algorithm will increase at varying rates. If you ever want to run your algorithm on larger datasets you will care not just about the comparative runtime on a single small dataset, but how the performance scales out as you move to larger datasets. We can similate this by subsampling from MNIST digits (via scikit-learn's convenient `resample` utility) and looking at the runtime for varying sized subsamples. Since there is some randomness involved here (both in the subsample selection, and in some of the algorithms which have stochastic aspects) we will want to run a few examples for each dataset size. We can easily package all of this up in a simple function that will return a convenient pandas dataframe of dataset sizes and runtimes given an algorithm.

```
def data_size_scaling(algorithm, data, sizes=[100, 200, 400, 800, 1600], n_runs=5):
    result = []
    for size in sizes:
        for run in range(n_runs):
            subsample = resample(data, n_samples=size)
            start_time = time.time()
            algorithm.fit(subsample)
            elapsed_time = time.time() - start_time
            del subsample
            result.append((size, elapsed_time))
    return pd.DataFrame(result, columns=('dataset size', 'runtime (s)'))
```

Now we just want to run this for each of the various dimension reduction implementations so we can look at the results. Since we don't know how long these runs might take we'll start off with a very small set of samples, scaling up to only 1600 samples.

```
all_algorithms = [
    PCA(),
    UMAP(),
    MulticoreTSNE(),
    OpenTSNE(),
    TSNE(),
    LocallyLinearEmbedding(),
    SpectralEmbedding(),
    Isomap(),
    MDS(),
]
performance_data = {}
for algorithm in all_algorithms:
    if 'openTSNE' in str(algorithm.__class__):
```

(continues on next page)

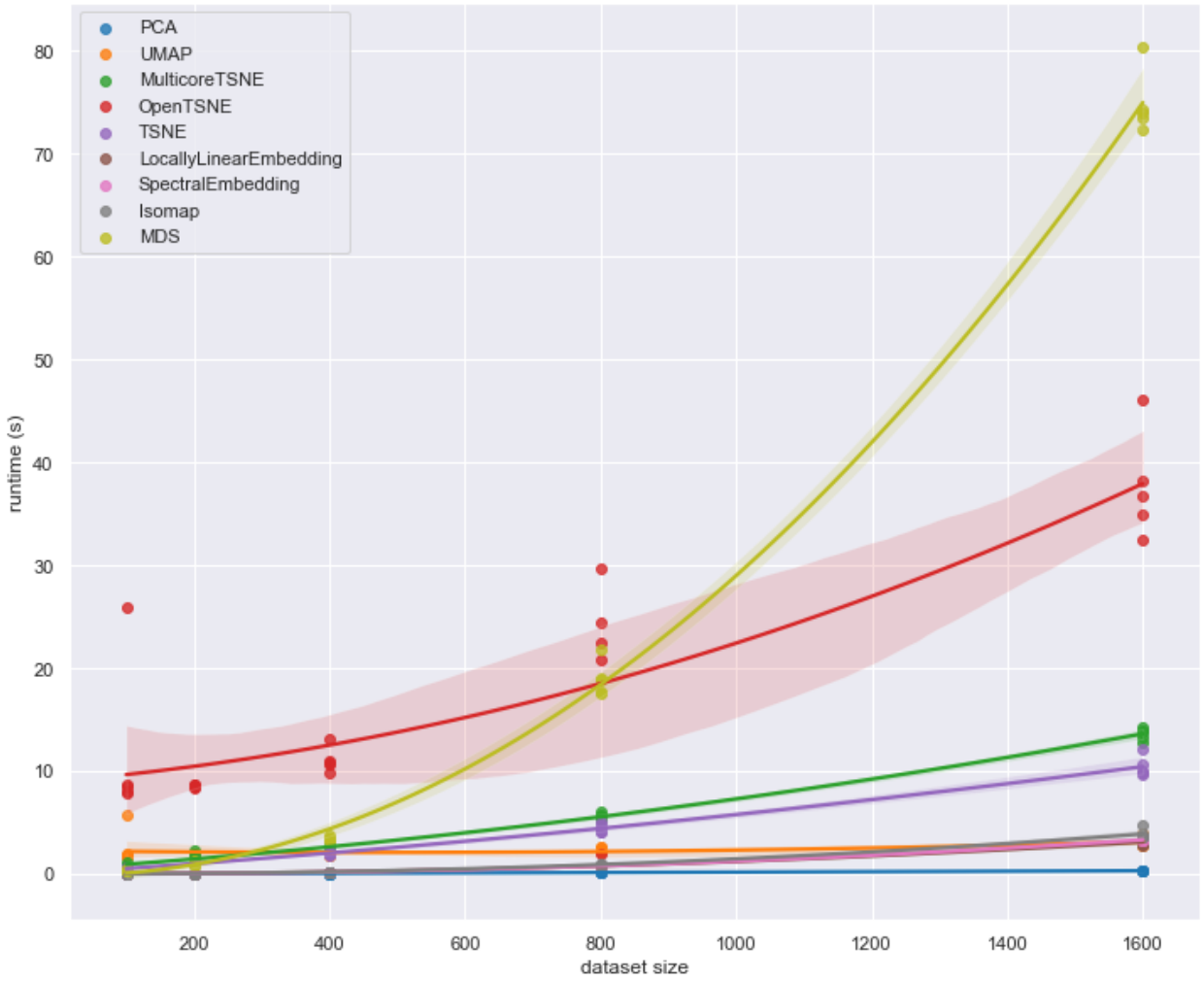
(continued from previous page)

```
        alg_name = "OpenTSNE"
    elif 'MulticoreTSNE' in str(algorithm.__class__):
        alg_name = "MulticoreTSNE"
    else:
        alg_name = str(algorithm).split('(')[0]

    performance_data[alg_name] = data_size_scaling(algorithm, mnist_data, n_runs=5)

    print(f"[{time.asctime(time.localtime())}] Completed {alg_name}")
```

```
[Sat Feb 22 09:50:24 2020] Completed PCA
[Sat Feb 22 09:51:23 2020] Completed UMAP
[Sat Feb 22 09:53:24 2020] Completed MulticoreTSNE
[Sat Feb 22 10:00:50 2020] Completed OpenTSNE
[Sat Feb 22 10:02:22 2020] Completed TSNE
[Sat Feb 22 10:02:44 2020] Completed LocallyLinearEmbedding
[Sat Feb 22 10:03:06 2020] Completed SpectralEmbedding
[Sat Feb 22 10:03:31 2020] Completed Isomap
[Sat Feb 22 10:11:45 2020] Completed MDS
```

Now let's plot the results so we can see what is going on. We'll use seaborn's regression plot to interpolate the effective scaling. For some algorithms this can be a little noisy, especially in this relatively small dataset regime, but it will give us a good idea of what is going on.

```
for alg_name, perf_data in performance_data.items():
    sns.regplot('dataset size', 'runtime (s)', perf_data, order=2, label=alg_name)
plt.legend()
```



We can see straight away that there are some outliers here. It is notable that openTSNE does poorly on small datasets. It does not have the scaling properties of MDS however; for larger dataset sizes MDS is going to quickly become completely unmanageable which openTSNE has fairly flat scaling. At the same time MulticoreTSNE demonstrates that t-SNE can run fairly efficiently. It is hard to tell much about the other implementations other than the fact that PCA is far and away the fastest option. To see more we'll have to look at runtimes on larger dataset sizes. Both MDS, Isomap and SpectralEmbedding will actually take too long to run so let's restrict ourselves to the fastest performing implementations and see what happens as we extend out to larger dataset sizes.

```
fast_algorithms = [
    PCA(),
    UMAP(),
    MulticoreTSNE(),
    OpenTSNE(),
    TSNE(),
    LocallyLinearEmbedding(),
]
fast_performance_data = {}
for algorithm in fast_algorithms:
    if 'openTSNE' in str(algorithm.__class__):
        alg_name = "OpenTSNE"
    elif 'MulticoreTSNE' in str(algorithm.__class__):
        alg_name = "MulticoreTSNE"
```

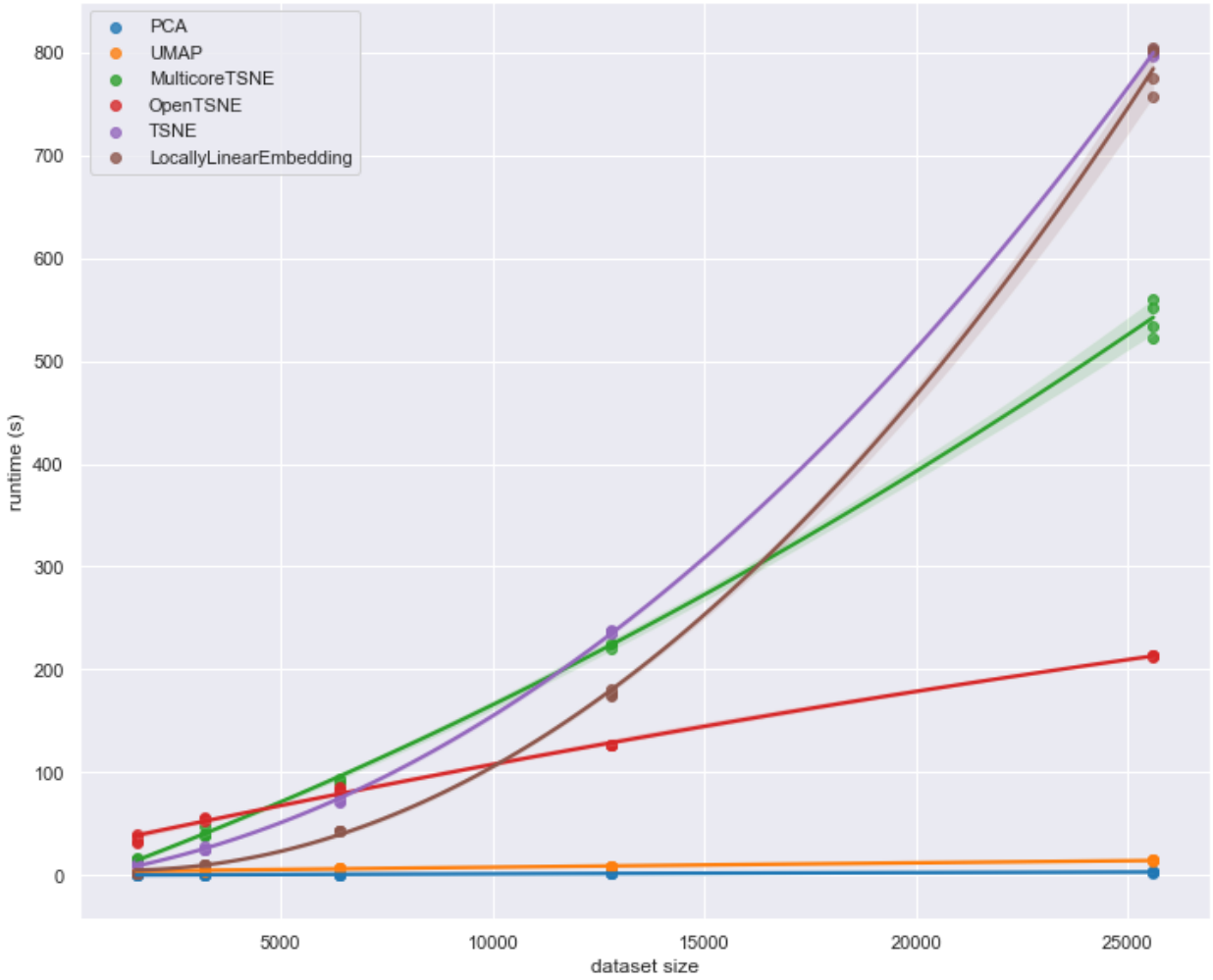
(continues on next page)

(continued from previous page)

```
    else:
        alg_name = str(algorithm).split('(')[0]

    fast_performance_data[alg_name] = data_size_scaling(algorithm, mnist_data,
                                                        sizes=[1600, 3200, 6400, 12800,␣
↪25600], n_runs=4)

    print(f"[{time.asctime(time.localtime())}] Completed {alg_name}")
```

```
[Sat Feb 22 10:12:15 2020] Completed PCA
[Sat Feb 22 10:14:51 2020] Completed UMAP
[Sat Feb 22 11:16:05 2020] Completed MulticoreTSNE
[Sat Feb 22 11:50:17 2020] Completed OpenTSNE
[Sat Feb 22 13:06:38 2020] Completed TSNE
[Sat Feb 22 14:14:36 2020] Completed LocallyLinearEmbedding
```

```
for alg_name, perf_data in fast_performance_data.items():
    sns.regplot('dataset size', 'runtime (s)', perf_data, order=2, label=alg_name)
plt.legend()
```



At this point we begin to see some significant differentiation among the different implementations. In the earlier plot

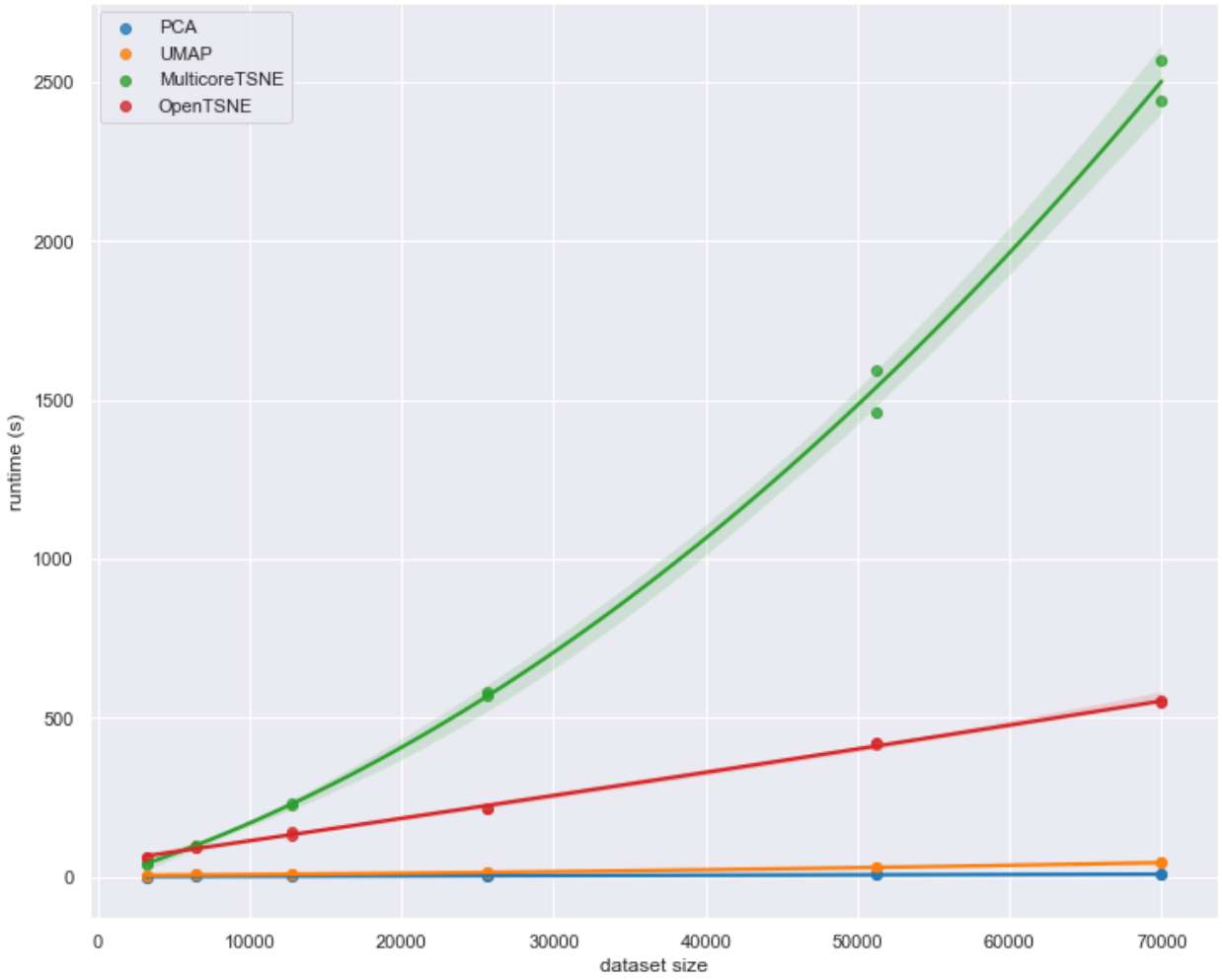
OpenTSNE looked to be performing relatively poorly, but now the scaling effects kick in, and we see that is is faster than most. Similarly MulticoreTSNE looked to be slower than some of the other algorithms in th earlier plot, but as we scale out to larger datasets we see that its relative scaling performance is superior to the scikit-learn implementations of TSNE and locally linear embedding.

It is probably worth extending out further – up to the full MNIST digits dataset. To manage to do that in any reasonable amount of time we'll have to restrict out attention to an even smaller subset of implementations. We will pare things down to just OpenTSNE, MulticoreTSNE, PCA and UMAP.

```
very_fast_algorithms = [
    PCA(),
    UMAP(),
    MulticoreTSNE(),
    OpenTSNE(),
]
vfast_performance_data = {}
for algorithm in very_fast_algorithms:
    if 'openTSNE' in str(algorithm.__class__):
        alg_name = "OpenTSNE"
    elif 'MulticoreTSNE' in str(algorithm.__class__):
        alg_name = "MulticoreTSNE"
    else:
        alg_name = str(algorithm).split('(')[0]

    vfast_performance_data[alg_name] = data_size_scaling(algorithm, mnist_data,
                                                         sizes=[3200, 6400, 12800, 25600,␣
↪51200, 70000], n_runs=2)

    print(f"[{time.asctime(time.localtime())}] Completed {alg_name}")
```

```
[Sat Feb 22 14:15:22 2020] Completed PCA
[Sat Feb 22 14:18:59 2020] Completed UMAP
[Sat Feb 22 17:04:58 2020] Completed MulticoreTSNE
[Sat Feb 22 17:54:14 2020] Completed OpenTSNE
```

```
for alg_name, perf_data in vfast_performance_data.items():
    sns.regplot('dataset size', 'runtime (s)', perf_data, order=2, label=alg_name)
plt.legend()
```



Here we see UMAP's advantages over t-SNE really coming to the forefront. While UMAP is clearly slower than PCA, its scaling performance is dramatically better than MulticoreTSNE, and, despite the impressive scaling performance of openTSNE, UMAP continues to outperform it. Based on the slopes of the lines, for even larger datasets the difference between UMAP and t-SNE is only going to grow.

This concludes our look at scaling by dataset size. The short summary is that PCA is far and away the fastest option, but you are potentially giving up a lot for that speed. UMAP, while not competitive with PCA, is clearly the next best option in terms of performance among the implementations explored here. Given the quality of results that UMAP can provide we feel it is clearly a good option for dimension reduction.

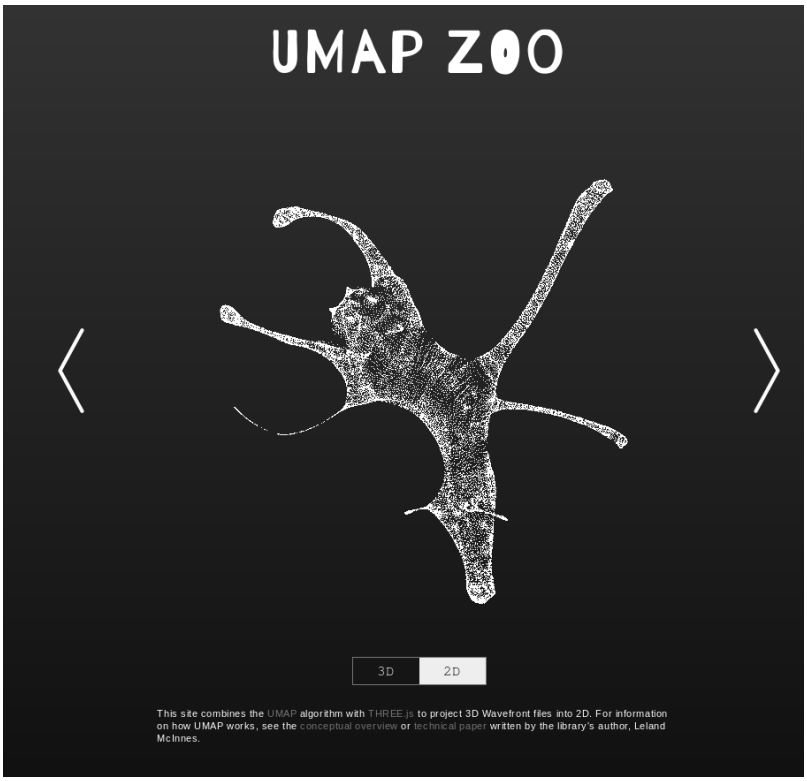
CHAPTER 22

Interactive Visualizations

UMAP has found use in a number of interesting interactive visualization projects, analyzing everything from images from photo archives, to word embedding, animal point clouds, and even sound. Sometimes it has also been used in interesting interactive tools that simply help a user to get an intuition for what the algorithm is doing (by applying it to intuitive 3D data). Below are some amazing projects that make use of UMAP.

22.1 UMAP Zoo

An exploration of how UMAP behaves when dimension reducing point clouds of animals. It is interactive, letting you switch between 2D and 3D representations and has a wide selection of different animals. Attempting to guess the animal from the 2D UMAP representation is a fun game. In practice this tool can go a long way to helping to build at least some intuitions for what UMAP tends to do with data.

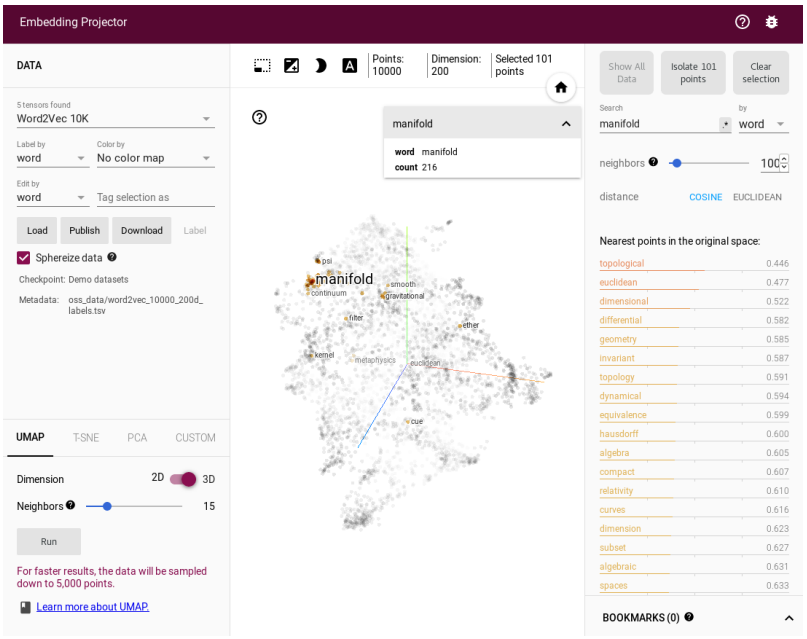


UMAP Zoo

Thanks to Douglas Duhaime.

22.2 Tensorflow Embedding Projector

If you just want to explore UMAP embeddings of datasets then the Embedding Projector from Tensorflow is a great way to do that. As well as having a good interactive 3D view it also has facilities for inspecting and searching labels and tags on the data. By default it loads up word2vec vectors, but you can upload any data you wish. You can then select the UMAP option among the tabs for embeddings choices (alongside PCA and t-SNE).

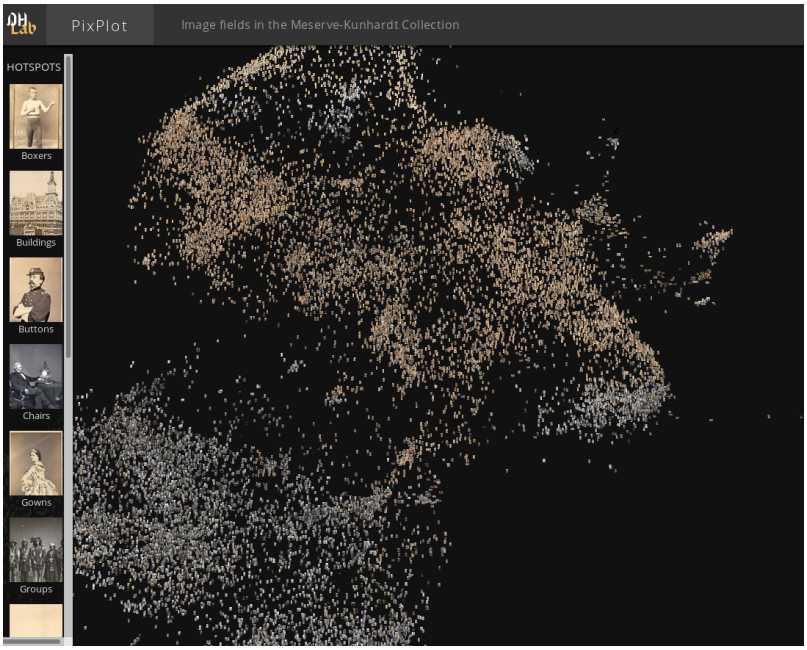


Embedding Projector

Thanks to Andy Coenen and the Embedding Projector team.

22.3 PixPlot

PixPlot provides an overview of large photo-collections. In the demonstration app from Yale's Digital Humanities lab it provides a window on the Meserve-Kunhardt Collection of historical photographs. The approach uses convolutional neural nets to reduce the images to 2048 dimensions, and then uses UMAP to present them in a 2-dimensional map which the user can interactive pan and zoom around in. This process results in similar photos ending up in similar regions of the map allowing for easy perusal of large photo collections. The PixPlot project is also available on github in case you wish to train it on your own photo collection.



PixPlot

Thanks to Douglas Duhaime and the Digital Humanities lab at Yale.

22.4 UMAP Explorer

A great demonstration of building a web based app for interactively exploring a UMAP embedding. In this case it provides an exploration of UMAP run on the MNIST digits dataset. Each point in the embedding is rendered as the digit image, and coloured according to the digit class. Mousing over the images will make them larger and provide a view of the digit in the upper left. You can also pan and zoom around the emebdding to get a better understanding of how UMAP has mapped the different styles of handwritten digits down to 2 dimensions.

UMAP Explorer

Thanks for Grant Custer.

22.5 Audio Explorer

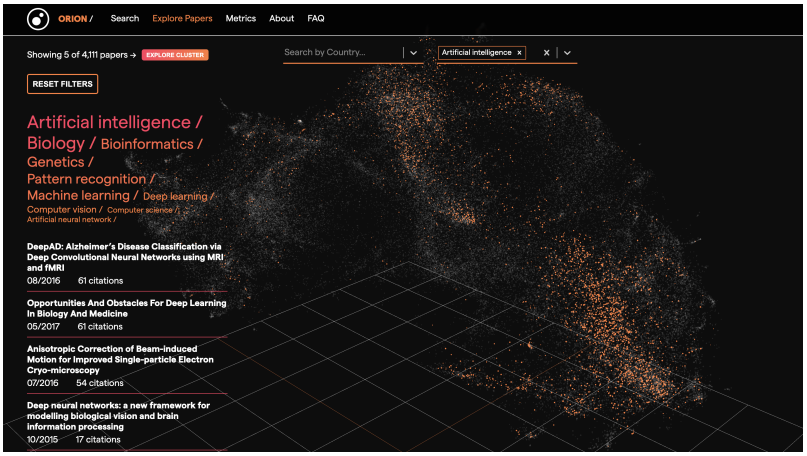
The Audio Explorer uses UMAP to embed sound samples into a 2 dimensional space for easy exploration. The goal here is to take a large library of sounds samples and put similar sounds in similar regions of the map, allowing a user to quickly mouse over and listen to various variations of a given sample to quickly find exactly the right sound sample to use. Audio explorer uses MFCCs and/or WaveNet to provide an initial useful vector representation of the sound samples, before applying UMAP to generate the 2D embedding.

Audio Explorer

Thanks to Leon Fedden.

22.6 Orion Search

Orion is an open source research measurement and knowledge discovery tool that enables you to monitor progress in science, visually explore the scientific landscape and search for relevant publications. Orion encodes bioRxiv paper abstracts to dense vectors with Sentence Transformers and projects them to an interactive 3D visualisation with UMAP. You can filter the UMAP embeddings by topic and country. You can also select a subset of the UMAP embeddings and retrieve those papers and their metadata.

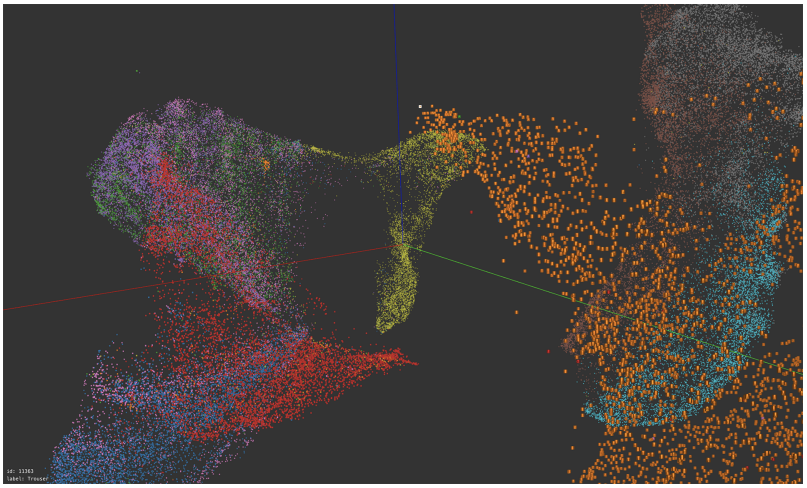


Orion Search

Thanks to Kostas Stathoulopoulos, Zac Ioannidis and Lilia Villafuerte.

22.7 Exploring Fashion MNIST

A web based interactive exploration of a 3D UMAP embedding ran on the Fashion MNIST dataset. Users can freely navigate the 3D space, jumping to a specific image by clicking an image or entering an image id. Like Grant Custer's UMAP Explorer, each point is rendered as the actual image and colored according to the label. It is also similar to the Tensorflow Embedding Projector, but designed more specifically for Fashion MNIST, thus more efficient and capable of showing all the 70k images.



Exploring Fashion MNIST

Thanks to stwind.

CHAPTER 23

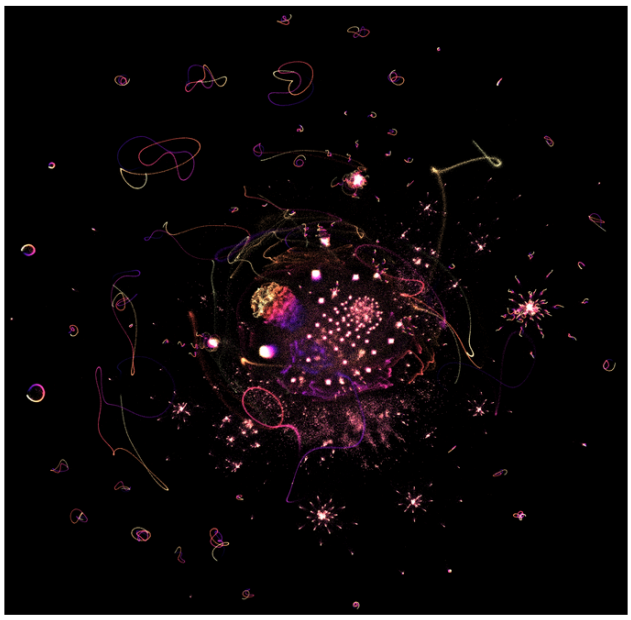
Exploratory Analysis of Interesting Datasets

UMAP is a useful tool for general exploratory analysis of data – it can provide a unique lens through which to view data that can highlight structures and properties hiding in data that are not as apparent when analysed with other techniques. Below is a selection of uses cases of UMAP being used for interesting explorations of intriguing datasets – everything from pure math and outputs of neural networks, to philosophy articles, and scientific texts.

23.1 Prime factorizations of numbers

What would happen if we applied UMAP to the integers? First we would need a way to express an integer in a high dimensional space. That can be done by looking at the prime factorization of each number. Next you have to take enough numbers to actually generate an interesting visualization. John Williamson set about doing exactly this, and the results are fascinating. While they may not actually tell us anything new about number theory they do highlight interesting structures in prime factorizations, and demonstrate how UMAP can aid in interesting explorations of datasets that we might think we know well. It's worth visiting the linked article below as Dr. Williamson provides a rich and detailed exploration of UMAP as applied to prime factorizations of integers.

1 What do numbers look like?



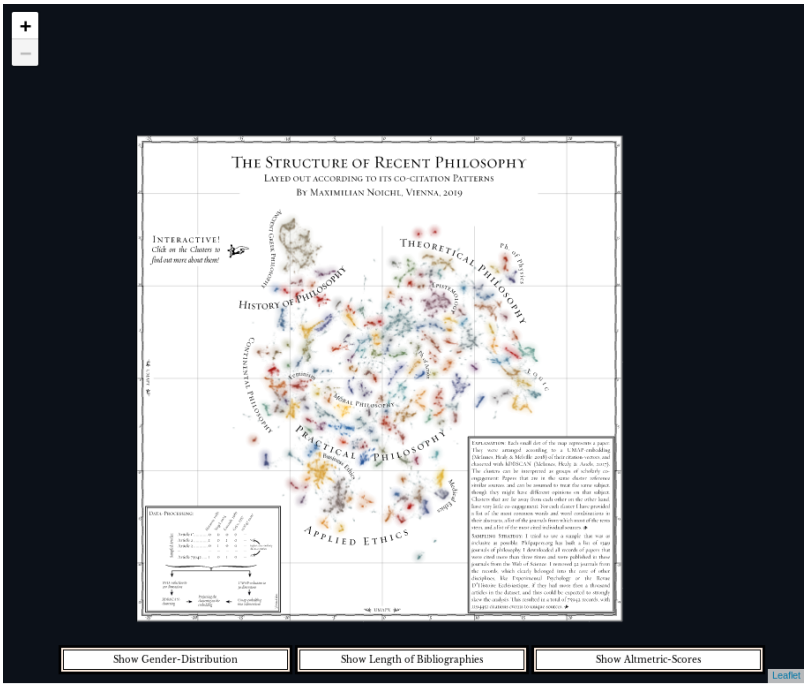
One million integers embedded into 2D space with UMAP

UMAP on prime factorizations

Thanks to John Williamson.

23.2 Structure of Recent Philosophy

Philosophy is an incredibly diverse subject, ranging from social and moral philosophy to logic and philosophy of math; from analysis of ancient Greek philosophy to modern business ethics. If we could get an overview of all the philosophy papers published in the last century what might it look like? Maximilian Noichl provides just such an exploration, looking at a large sampling of philosophy papers and comparing them according to their citations. The results are intriguing, and can be explored interactively in the viewer Maximilian built for it.

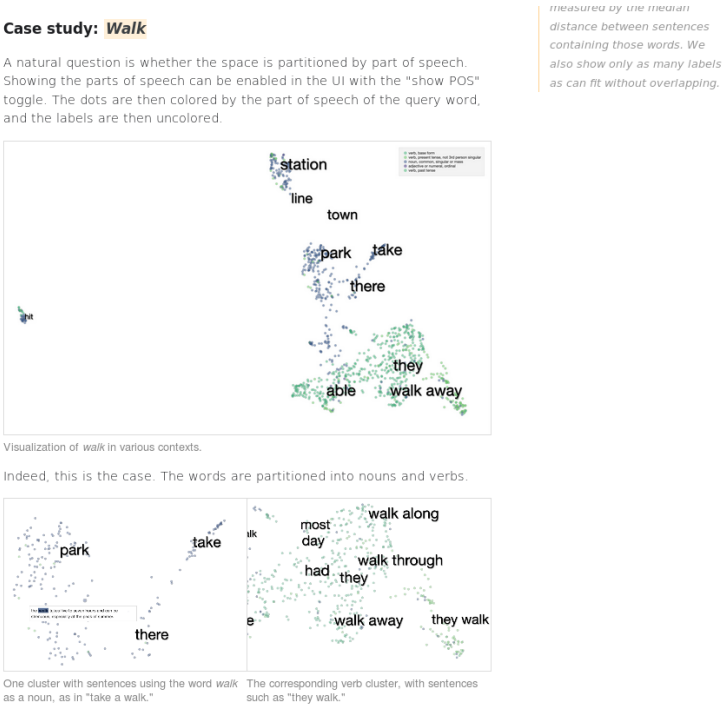


Structure of Recent Philosophy

Thanks to Maximilian Noichl.

23.3 Language, Context, and Geometry in Neural Networks

Among recent developments in natural language processing is the BERT neural network based technique for analysis of language. Among many things that BERT can do one is context sensitive embeddings of words – providing numeric vector representations of words that are sensitive to the context of how the word is used. Exactly what goes on inside the neural network to do this is a little mysterious (since the network is very complex with many many parameters). A tram of researchers from Google set out to explore the word embedding space generated by BERT, and among the tools used was UMAP. The linked blog post provides a detailed and inspiring analysis of what BERT's word embeddings look like, and how the different layers of BERT represent different aspects of language.

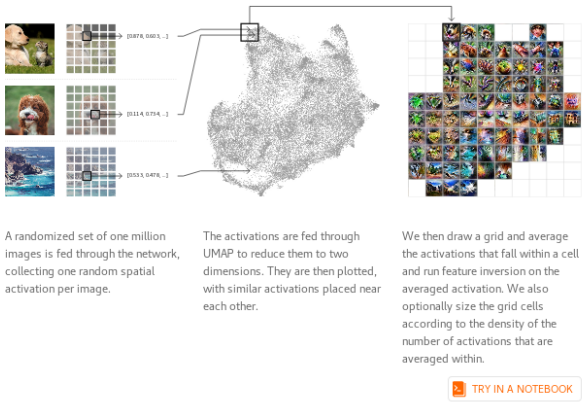


Language, Context, and Geometry in Neural Networks

Thanks to Andy Coenen, Emily Reif, Ann Yuan, Been Kim, Adam Pearce, Fernanda Viégas, and Martin Wattenberg.

23.4 Activation Atlas

Understanding the image processing capabilities (and deficits!) of modern convolutional neural networks is a challenge. Certainly these models are capable of amazing feats in, for example, image classification. They can also be brittle in unexpected ways, with carefully designed images able to induce otherwise baffling mis-classifications. To better understand this researchers from Google and OpenAI built the activation atlas – analysing the space of activations of a neural network. Here UMAP provides a means to compress the activation landscape down to 2 dimensions for visualization. The result was an impressive interactive paper in the Distill journal, providing rich visualizations and new insights into the working of convolutional neural networks.

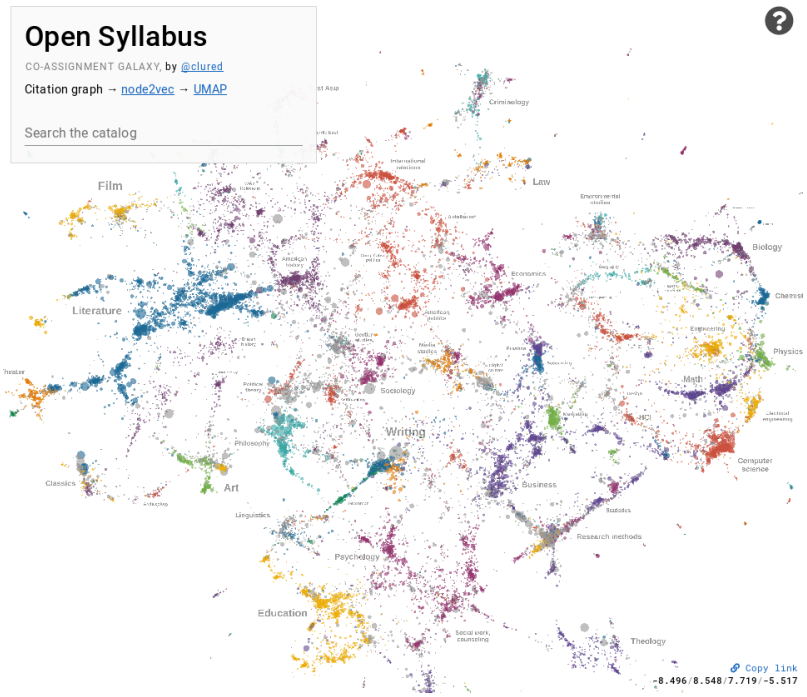


The Activation Atlas

Thanks to Shan Carter, Zan Armstrong, Ludwig Schubert, Ian Johnson, and Chris Olah.

23.5 Open Syllabus Galaxy

Suppose you wanted to explore the space of commonly assigned texts from Open Syllabus? That gives you over 150,000 texts to consider. Since the texts are open you can actually analyse the text content involved. With some NLP and neural network wizardry David McClure build a network of such texts and then used node2vec and UMAP to generate a map of them. The result is a galaxy of textbooks showing inter-relationships between subjects, similar and related texts, and generally just a an interesting ladscape of science to be explored. As with some of the other projects here David made a great interactive viewer allowing for rich exploration of the results.



Open Syllabus Galaxy

Thanks to David McClure.

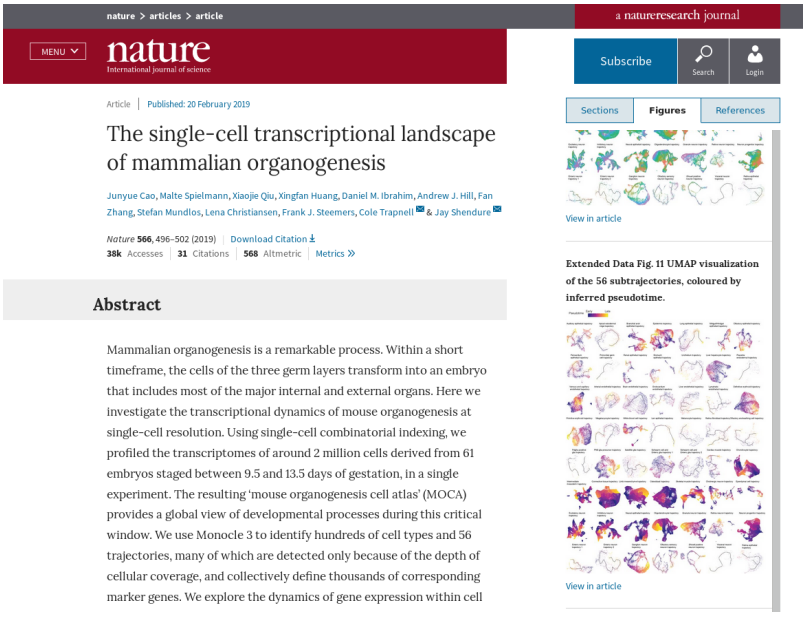
CHAPTER 24

Scientific Papers

UMAP has been used in a wide variety of scientific publications from a diverse range of fields. Here we will highlight a small selection of papers that demonstrate both the depth of analysis, and breadth of subjects, UMAP can be used for. These range from biology, to machine learning, and even social science.

24.1 The single-cell transcriptional landscape of mammalian organogenesis

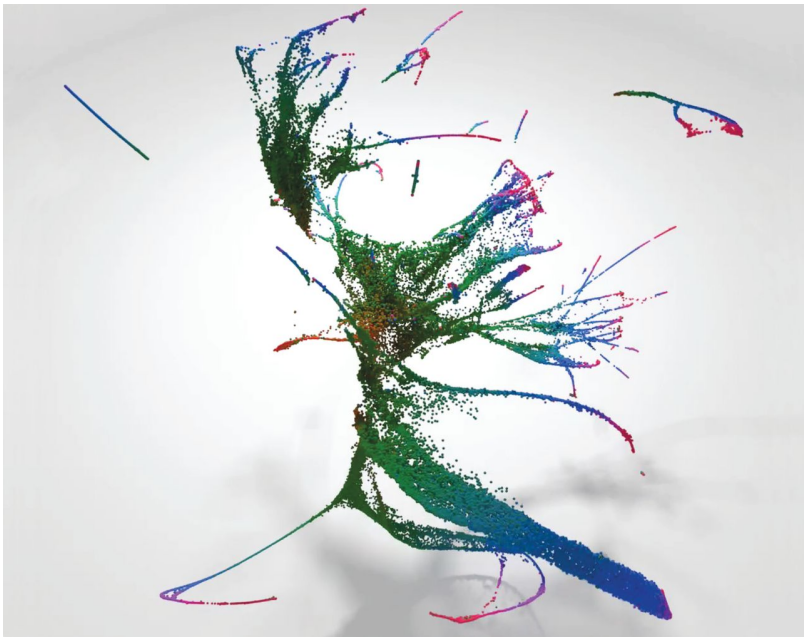
A detailed look at the development of mouse embryos from a single-cell view. UMAP is used as a core piece of The Monocle3 software suite for identifying cell types and trajectories. This was a major paper in Nature, demonstrating the power of UMAP for large scale scientific endeavours.



Link to the paper

24.2 A lineage-resolved molecular atlas of C. elegans embryogenesis at single-cell resolution

Still in the realm of single cell biology this paper looks at the developmental landscape of the round-word C. elegans. UMAP is used for detailed analysis of the developmental trajectories of cells, looking at global scales, and then digging down to look at individual organs. The result is an impressive array of UMAP visualisations that tease out ever finer structures in cellular development.



Link to the paper

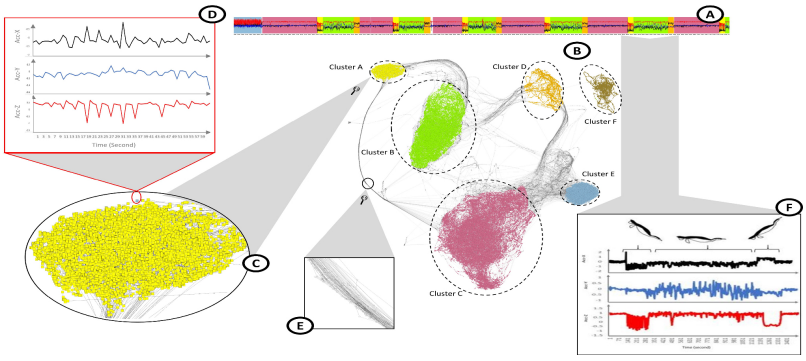
24.3 Exploring Neural Networks with Activation Atlases

Understanding the image processing capabilities (and deficits!) of modern convolutional neural networks is a challenge. This interactive paper from Distill seeks to provide a way to "peek inside the black box" by looking at the activations throughout the network. By mapping this high dimensional data down to 2D with UMAP the authors can construct an "atlas" of how different images are perceived by the network.

Link to the paper

24.4 TimeCluster: dimension reduction applied to temporal data for visual analytics

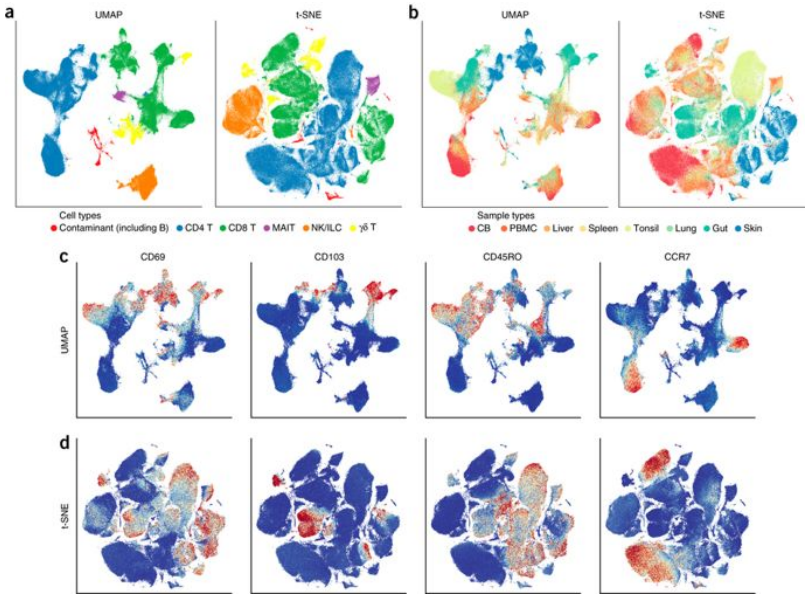
An interesting approach to time-series analysis, targeted toward cases where the time series has repeating patterns – though no necessarily of a consistently periodic nature. The approach involves dimension reduction and clustering of sliding window blocks of the time-series. The result is a map where repeating behaviour is exposed as loop structures. This can be useful for both clustering similar blocks within a time-series, or finding outliers.



Link to the paper

24.5 Dimensionality reduction for visualizing single-cell data using UMAP

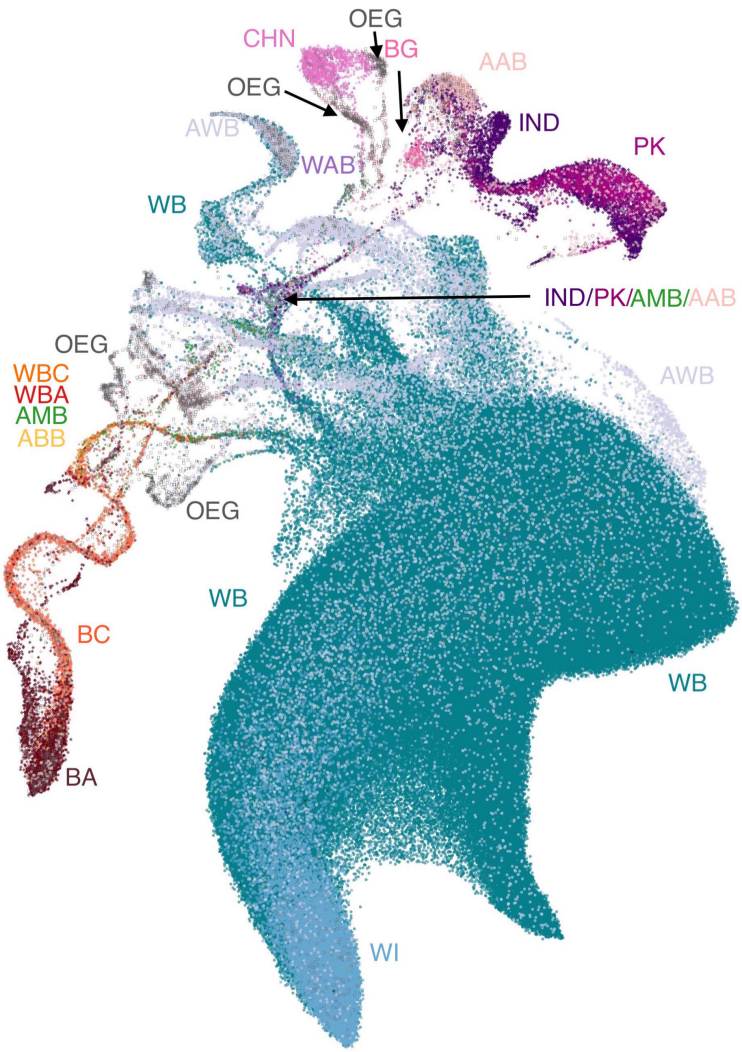
An early paper on applying UMAP to single-cell biology data. It looks at both, gene-expression data and flow-cytometry data, and compares UMAP to t-SNE both in terms of performance and quality of results. This is a good introduction to using UMAP for single-cell biology data.



Link to the paper

24.6 Revealing multi-scale population structure in large cohorts

A paper looking at population genetics which uses UMAP as a means to visualise population structures. This produced some intriguing visualizations, and was one of the first of several papers taking this visualization approach. It also includes some novel visualizations using UMAP projections to 3D as RGB color specifications for data points, allowing the UMAP structure to be visualized in geographic maps based on where the samples were drawn from.

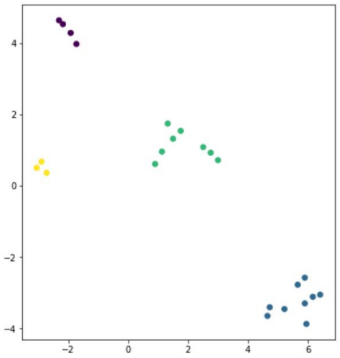


Link to the paper

24.7 Understanding Vulnerability of Children in Surrey

An example of the use of UMAP in sociological studies – in this case looking at children in Surrey, British Columbia. Here UMAP is used as a tool to aid in general data analysis, and proves effective for the tasks to which it was put.

Validating Clustering results with UMAP



UMAP Clustering (Right) shows four distinct clusters on all-waves.

Hopkins Statistic (Below) to reject the null hypothesis that these clusters reasonably random.

	t-SNE A-clusters					
Cluster	0	1	**2**	3	4	**5**
H	0.4563	0.5478	**0.5706**	0.4166	0.6080	**0.4311**

Table 2: Hopkin's statistic over the t-SNE all-wave clusters.

	UMAP UA-clusters			
Cluster	**0**	1	2	**3**
H	**0.5706**	0.5023	0.5308	**0.4311**

Table 3: Hopkin's statistic over the UMAP all-wave clusters.

Link to the paper

CHAPTER 25

UMAP API Guide

UMAP has only a single class `UMAP`.

25.1 UMAP

class umap.umap_.**UMAP**(*n_neighbors=15*, *n_components=2*, *metric='euclidean'*, *metric_kwds=None*, *output_metric='euclidean'*, *output_metric_kwds=None*, *n_epochs=None*, *learning_rate=1.0*, *init='spectral'*, *min_dist=0.1*, *spread=1.0*, *low_memory=True*, *n_jobs=-1*, *set_op_mix_ratio=1.0*, *local_connectivity=1.0*, *repulsion_strength=1.0*, *negative_sample_rate=5*, *transform_queue_size=4.0*, *a=None*, *b=None*, *random_state=None*, *angular_rp_forest=False*, *target_n_neighbors=-1*, *target_metric='categorical'*, *target_metric_kwds=None*, *target_weight=0.5*, *transform_seed=42*, *transform_mode='embedding'*, *force_approximation_algorithm=False*, *verbose=False*, *unique=False*, *densmap=False*, *dens_lambda=2.0*, *dens_frac=0.3*, *dens_var_shift=0.1*, *output_dens=False*, *disconnection_distance=None*)

Uniform Manifold Approximation and Projection

Finds a low dimensional embedding of the data that approximates an underlying manifold.

n_neighbors: float (optional, default 15) The size of local neighborhood (in terms of number of neighboring sample points) used for manifold approximation. Larger values result in more global views of the manifold, while smaller values result in more local data being preserved. In general values should be in the range 2 to 100.

n_components: int (optional, default 2) The dimension of the space to embed into. This defaults to 2 to provide easy visualization, but can reasonably be set to any integer value in the range 2 to 100.

metric: string or function (optional, default 'euclidean') The metric to use to compute distances in high dimensional space. If a string is passed it must match a valid predefined metric. If a general metric is required a function that takes two 1d arrays and returns a float can be provided. For performance purposes it is required that this be a numba jit'd function. Valid string metrics include:

- euclidean
- manhattan
- chebyshev
- minkowski
- canberra
- braycurtis
- mahalanobis
- wminkowski
- seuclidean
- cosine
- correlation
- haversine
- hamming
- jaccard
- dice
- russelrao
- kulsinski
- ll_dirichlet
- hellinger
- rogerstanimoto
- sokalmichener
- sokalsneath
- yule

Metrics that take arguments (such as minkowski, mahalanobis etc.) can have arguments passed via the metric_kwds dictionary. At this time care must be taken and dictionary elements must be ordered appropriately; this will hopefully be fixed in the future.

n_epochs: int (optional, default None) The number of training epochs to be used in optimizing the low dimensional embedding. Larger values result in more accurate embeddings. If None is specified a value will be selected based on the size of the input dataset (200 for large datasets, 500 for small).

learning_rate: float (optional, default 1.0) The initial learning rate for the embedding optimization.

init: string (optional, default 'spectral')

How to initialize the low dimensional embedding. Options are:

- 'spectral': use a spectral embedding of the fuzzy 1-skeleton
- 'random': assign initial embedding positions at random.
- A numpy array of initial embedding positions.

min_dist: float (optional, default 0.1) The effective minimum distance between embedded points. Smaller values will result in a more clustered/clumped embedding where nearby points on the manifold are drawn

closer together, while larger values will result on a more even dispersal of points. The value should be set relative to the `spread` value, which determines the scale at which embedded points will be spread out.

spread: float (optional, default 1.0) The effective scale of embedded points. In combination with `min_dist` this determines how clustered/clumped the embedded points are.

low_memory: bool (optional, default False) For some datasets the nearest neighbor computation can consume a lot of memory. If you find that UMAP is failing due to memory constraints consider setting this option to True. This approach is more computationally expensive, but avoids excessive memory use.

set_op_mix_ratio: float (optional, default 1.0) Interpolate between (fuzzy) union and intersection as the set operation used to combine local fuzzy simplicial sets to obtain a global fuzzy simplicial sets. Both fuzzy set operations use the product t-norm. The value of this parameter should be between 0.0 and 1.0; a value of 1.0 will use a pure fuzzy union, while 0.0 will use a pure fuzzy intersection.

local_connectivity: int (optional, default 1) The local connectivity required – i.e. the number of nearest neighbors that should be assumed to be connected at a local level. The higher this value the more connected the manifold becomes locally. In practice this should be not more than the local intrinsic dimension of the manifold.

repulsion_strength: float (optional, default 1.0) Weighting applied to negative samples in low dimensional embedding optimization. Values higher than one will result in greater weight being given to negative samples.

negative_sample_rate: int (optional, default 5) The number of negative samples to select per positive sample in the optimization process. Increasing this value will result in greater repulsive force being applied, greater optimization cost, but slightly more accuracy.

transform_queue_size: float (optional, default 4.0) For transform operations (embedding new points using a trained **model_** this will control how aggressively to search for nearest neighbors. Larger values will result in slower performance but more accurate nearest neighbor evaluation.

a: float (optional, default None) More specific parameters controlling the embedding. If None these values are set automatically as determined by `min_dist` and `spread`.

b: float (optional, default None) More specific parameters controlling the embedding. If None these values are set automatically as determined by `min_dist` and `spread`.

random_state: int, RandomState instance or None, optional (default: None) If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

metric_kwds: dict (optional, default None) Arguments to pass on to the metric, such as the `p` value for Minkowski distance. If None then no arguments are passed on.

angular_rp_forest: bool (optional, default False) Whether to use an angular random projection forest to initialise the approximate nearest neighbor search. This can be faster, but is mostly on useful for metric that use an angular style distance such as cosine, correlation etc. In the case of those metrics angular forests will be chosen automatically.

target_n_neighbors: int (optional, default -1) The number of nearest neighbors to use to construct the target simplcial set. If set to -1 use the `n_neighbors` value.

target_metric: string or callable (optional, default 'categorical') The metric used to measure distance for a target array is using supervised dimension reduction. By default this is 'categorical' which will measure distance in terms of whether categories match or are different. Furthermore, if semi-supervised is required target values of -1 will be trated as unlabelled under the 'categorical' metric. If the target array takes continuous values (e.g. for a regression problem) then metric of 'l1' or 'l2' is probably more appropriate.

target_metric_kwds: dict (optional, default None) Keyword argument to pass to the target metric when performing supervised dimension reduction. If None then no arguments are passed on.

target_weight: float (optional, default 0.5) weighting factor between data topology and target topology. A value of 0.0 weights entirely on data, a value of 1.0 weights entirely on target. The default of 0.5 balances the weighting equally between data and target.

transform_seed: int (optional, default 42) Random seed used for the stochastic aspects of the transform operation. This ensures consistency in transform operations.

verbose: bool (optional, default False) Controls verbosity of logging.

unique: bool (optional, default False) Controls if the rows of your data should be uniqued before being embedded. If you have more duplicates than you have n_neighbour you can have the identical data points lying in different regions of your space. It also violates the definition of a metric. For to map from internal structures back to your data use the variable _unique_inverse_.

densmap: bool (optional, default False) Specifies whether the density-augmented objective of densMAP should be used for optimization. Turning on this option generates an embedding where the local densities are encouraged to be correlated with those in the original space. Parameters below with the prefix 'dens' further control the behavior of this extension.

dens_lambda: float (optional, default 2.0) Controls the regularization weight of the density correlation term in densMAP. Higher values prioritize density preservation over the UMAP objective, and vice versa for values closer to zero. Setting this parameter to zero is equivalent to running the original UMAP algorithm.

dens_frac: float (optional, default 0.3) Controls the fraction of epochs (between 0 and 1) where the density-augmented objective is used in densMAP. The first (1 - dens_frac) fraction of epochs optimize the original UMAP objective before introducing the density correlation term.

dens_var_shift: float (optional, default 0.1) A small constant added to the variance of local radii in the embedding when calculating the density correlation objective to prevent numerical instability from dividing by a small number

output_dens: float (optional, default False) Determines whether the local radii of the final embedding (an inverse measure of local density) are computed and returned in addition to the embedding. If set to True, local radii of the original data are also included in the output for comparison; the output is a tuple (embedding, original local radii, embedding local radii). This option can also be used when densmap=False to calculate the densities for UMAP embeddings.

disconnection_distance: float (optional, default np.inf or maximal value for bounded distances) Disconnect any vertices of distance greater than or equal to disconnection_distance when approximating the manifold via our k-nn graph. This is particularly useful in the case that you have a bounded metric. The UMAP assumption that we have a connected manifold can be problematic when you have points that are maximally different from all the rest of your data. The connected manifold assumption will make such points have perfect similarity to a random set of other points. Too many such points will artificially connect your space.

fit (*X*, *y=None*)
Fit X into an embedded space.

Optionally use y for supervised dimension reduction.

X [array, shape (n_samples, n_features) or (n_samples, n_samples)] If the metric is 'precomputed' X must be a square distance matrix. Otherwise it contains a sample per row. If the method is 'exact', X may be a sparse matrix of type 'csr', 'csc' or 'coo'.

y [array, shape (n_samples)] A target array for supervised dimension reduction. How this is handled is determined by parameters UMAP was instantiated with. The relevant attributes are `target_metric` and `target_metric_kwds`.

fit_transform (*X*, *y=None*)
Fit X into an embedded space and return that transformed output.

X [array, shape (n_samples, n_features) or (n_samples, n_samples)] If the metric is 'precomputed' X must be a square distance matrix. Otherwise it contains a sample per row.

y [array, shape (n_samples)] A target array for supervised dimension reduction. How this is handled is determined by parameters UMAP was instantiated with. The relevant attributes are `target_metric` and `target_metric_kwds`.

X_new [array, shape (n_samples, n_components)] Embedding of the training data in low-dimensional space.

or a tuple (X_new, r_orig, r_emb) if `output_dens` flag is set, which additionally includes:

r_orig: array, shape (n_samples) Local radii of data points in the original data space (log-transformed).

r_emb: array, shape (n_samples) Local radii of data points in the embedding (log-transformed).

inverse_transform(*X*)

Transform X in the existing embedded space back into the input data space and return that transformed output.

X [array, shape (n_samples, n_components)] New points to be inverse transformed.

X_new [array, shape (n_samples, n_features)] Generated data points new data in data space.

transform(*X*)

Transform X into the existing embedded space and return that transformed output.

X [array, shape (n_samples, n_features)] New data to be transformed.

X_new [array, shape (n_samples, n_components)] Embedding of the new data in low-dimensional space.

A number of internal functions can also be accessed separately for more fine tuned work.

25.2 Useful Functions

class umap.umap_.**UMAP**(*n_neighbors=15*, *n_components=2*, *metric='euclidean'*, *metric_kwds=None*, *output_metric='euclidean'*, *output_metric_kwds=None*, *n_epochs=None*, *learning_rate=1.0*, *init='spectral'*, *min_dist=0.1*, *spread=1.0*, *low_memory=True*, *n_jobs=-1*, *set_op_mix_ratio=1.0*, *local_connectivity=1.0*, *repulsion_strength=1.0*, *negative_sample_rate=5*, *transform_queue_size=4.0*, *a=None*, *b=None*, *random_state=None*, *angular_rp_forest=False*, *target_n_neighbors=-1*, *target_metric='categorical'*, *target_metric_kwds=None*, *target_weight=0.5*, *transform_seed=42*, *transform_mode='embedding'*, *force_approximation_algorithm=False*, *verbose=False*, *unique=False*, *densmap=False*, *dens_lambda=2.0*, *dens_frac=0.3*, *dens_var_shift=0.1*, *output_dens=False*, *disconnection_distance=None*)

Uniform Manifold Approximation and Projection

Finds a low dimensional embedding of the data that approximates an underlying manifold.

n_neighbors: float (optional, default 15) The size of local neighborhood (in terms of number of neighboring sample points) used for manifold approximation. Larger values result in more global views of the manifold, while smaller values result in more local data being preserved. In general values should be in the range 2 to 100.

n_components: int (optional, default 2) The dimension of the space to embed into. This defaults to 2 to provide easy visualization, but can reasonably be set to any integer value in the range 2 to 100.

metric: string or function (optional, default 'euclidean') The metric to use to compute distances in high dimensional space. If a string is passed it must match a valid predefined metric. If a general metric is required a function that takes two 1d arrays and returns a float can be provided. For performance purposes it is required that this be a numba jit'd function. Valid string metrics include:

- euclidean
- manhattan
- chebyshev
- minkowski
- canberra
- braycurtis
- mahalanobis
- wminkowski
- seuclidean
- cosine
- correlation
- haversine
- hamming
- jaccard
- dice
- russelrao
- kulsinski
- ll_dirichlet
- hellinger
- rogerstanimoto
- sokalmichener
- sokalsneath
- yule

Metrics that take arguments (such as minkowski, mahalanobis etc.) can have arguments passed via the metric_kwds dictionary. At this time care must be taken and dictionary elements must be ordered appropriately; this will hopefully be fixed in the future.

n_epochs: int (optional, default None) The number of training epochs to be used in optimizing the low dimensional embedding. Larger values result in more accurate embeddings. If None is specified a value will be selected based on the size of the input dataset (200 for large datasets, 500 for small).

learning_rate: float (optional, default 1.0) The initial learning rate for the embedding optimization.

init: string (optional, default 'spectral')

How to initialize the low dimensional embedding. Options are:

- 'spectral': use a spectral embedding of the fuzzy 1-skeleton

- 'random': assign initial embedding positions at random.
- A numpy array of initial embedding positions.

min_dist: float (optional, default 0.1) The effective minimum distance between embedded points. Smaller values will result in a more clustered/clumped embedding where nearby points on the manifold are drawn closer together, while larger values will result on a more even dispersal of points. The value should be set relative to the `spread` value, which determines the scale at which embedded points will be spread out.

spread: float (optional, default 1.0) The effective scale of embedded points. In combination with `min_dist` this determines how clustered/clumped the embedded points are.

low_memory: bool (optional, default False) For some datasets the nearest neighbor computation can consume a lot of memory. If you find that UMAP is failing due to memory constraints consider setting this option to True. This approach is more computationally expensive, but avoids excessive memory use.

set_op_mix_ratio: float (optional, default 1.0) Interpolate between (fuzzy) union and intersection as the set operation used to combine local fuzzy simplicial sets to obtain a global fuzzy simplicial sets. Both fuzzy set operations use the product t-norm. The value of this parameter should be between 0.0 and 1.0; a value of 1.0 will use a pure fuzzy union, while 0.0 will use a pure fuzzy intersection.

local_connectivity: int (optional, default 1) The local connectivity required – i.e. the number of nearest neighbors that should be assumed to be connected at a local level. The higher this value the more connected the manifold becomes locally. In practice this should be not more than the local intrinsic dimension of the manifold.

repulsion_strength: float (optional, default 1.0) Weighting applied to negative samples in low dimensional embedding optimization. Values higher than one will result in greater weight being given to negative samples.

negative_sample_rate: int (optional, default 5) The number of negative samples to select per positive sample in the optimization process. Increasing this value will result in greater repulsive force being applied, greater optimization cost, but slightly more accuracy.

transform_queue_size: float (optional, default 4.0) For transform operations (embedding new points using a trained **model_** this will control how aggressively to search for nearest neighbors. Larger values will result in slower performance but more accurate nearest neighbor evaluation.

a: float (optional, default None) More specific parameters controlling the embedding. If None these values are set automatically as determined by `min_dist` and `spread`.

b: float (optional, default None) More specific parameters controlling the embedding. If None these values are set automatically as determined by `min_dist` and `spread`.

random_state: int, RandomState instance or None, optional (default: None) If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

metric_kwds: dict (optional, default None) Arguments to pass on to the metric, such as the `p` value for Minkowski distance. If None then no arguments are passed on.

angular_rp_forest: bool (optional, default False) Whether to use an angular random projection forest to initialise the approximate nearest neighbor search. This can be faster, but is mostly on useful for metric that use an angular style distance such as cosine, correlation etc. In the case of those metrics angular forests will be chosen automatically.

target_n_neighbors: int (optional, default -1) The number of nearest neighbors to use to construct the target simplcial set. If set to -1 use the `n_neighbors` value.

target_metric: string or callable (optional, default 'categorical') The metric used to measure distance for a target array is using supervised dimension reduction. By default this is 'categorical' which will measure distance in terms of whether categories match or are different. Furthermore, if semi-supervised is required

target values of -1 will be trated as unlabelled under the 'categorical' metric. If the target array takes continuous values (e.g. for a regression problem) then metric of 'l1' or 'l2' is probably more appropriate.

target_metric_kwds: dict (optional, default None) Keyword argument to pass to the target metric when performing supervised dimension reduction. If None then no arguments are passed on.

target_weight: float (optional, default 0.5) weighting factor between data topology and target topology. A value of 0.0 weights entirely on data, a value of 1.0 weights entirely on target. The default of 0.5 balances the weighting equally between data and target.

transform_seed: int (optional, default 42) Random seed used for the stochastic aspects of the transform operation. This ensures consistency in transform operations.

verbose: bool (optional, default False) Controls verbosity of logging.

unique: bool (optional, default False) Controls if the rows of your data should be uniqued before being embedded. If you have more duplicates than you have n_neighbour you can have the identical data points lying in different regions of your space. It also violates the definition of a metric. For to map from internal structures back to your data use the variable _unique_inverse_.

densmap: bool (optional, default False) Specifies whether the density-augmented objective of densMAP should be used for optimization. Turning on this option generates an embedding where the local densities are encouraged to be correlated with those in the original space. Parameters below with the prefix 'dens' further control the behavior of this extension.

dens_lambda: float (optional, default 2.0) Controls the regularization weight of the density correlation term in densMAP. Higher values prioritize density preservation over the UMAP objective, and vice versa for values closer to zero. Setting this parameter to zero is equivalent to running the original UMAP algorithm.

dens_frac: float (optional, default 0.3) Controls the fraction of epochs (between 0 and 1) where the density-augmented objective is used in densMAP. The first (1 - dens_frac) fraction of epochs optimize the original UMAP objective before introducing the density correlation term.

dens_var_shift: float (optional, default 0.1) A small constant added to the variance of local radii in the embedding when calculating the density correlation objective to prevent numerical instability from dividing by a small number

output_dens: float (optional, default False) Determines whether the local radii of the final embedding (an inverse measure of local density) are computed and returned in addition to the embedding. If set to True, local radii of the original data are also included in the output for comparison; the output is a tuple (embedding, original local radii, embedding local radii). This option can also be used when densmap=False to calculate the densities for UMAP embeddings.

disconnection_distance: float (optional, default np.inf or maximal value for bounded distances) Disconnect any vertices of distance greater than or equal to disconnection_distance when approximating the manifold via our k-nn graph. This is particularly useful in the case that you have a bounded metric. The UMAP assumption that we have a connected manifold can be problematic when you have points that are maximally different from all the rest of your data. The connected manifold assumption will make such points have perfect similarity to a random set of other points. Too many such points will artificially connect your space.

fit (*X*, *y=None*)
Fit X into an embedded space.

Optionally use y for supervised dimension reduction.

X [array, shape (n_samples, n_features) or (n_samples, n_samples)] If the metric is 'precomputed' X must be a square distance matrix. Otherwise it contains a sample per row. If the method is 'exact', X may be a sparse matrix of type 'csr', 'csc' or 'coo'.

y [array, shape (n_samples)] A target array for supervised dimension reduction. How this is handled is determined by parameters UMAP was instantiated with. The relevant attributes are `target_metric` and `target_metric_kwds`.

fit_transform(*X*, *y=None*)

Fit X into an embedded space and return that transformed output.

X [array, shape (n_samples, n_features) or (n_samples, n_samples)] If the metric is 'precomputed' X must be a square distance matrix. Otherwise it contains a sample per row.

y [array, shape (n_samples)] A target array for supervised dimension reduction. How this is handled is determined by parameters UMAP was instantiated with. The relevant attributes are `target_metric` and `target_metric_kwds`.

X_new [array, shape (n_samples, n_components)] Embedding of the training data in low-dimensional space.

or a tuple (X_new, r_orig, r_emb) if `output_dens` flag is set, which additionally includes:

r_orig: array, shape (n_samples) Local radii of data points in the original data space (log-transformed).

r_emb: array, shape (n_samples) Local radii of data points in the embedding (log-transformed).

inverse_transform(*X*)

Transform X in the existing embedded space back into the input data space and return that transformed output.

X [array, shape (n_samples, n_components)] New points to be inverse transformed.

X_new [array, shape (n_samples, n_features)] Generated data points new data in data space.

transform(*X*)

Transform X into the existing embedded space and return that transformed output.

X [array, shape (n_samples, n_features)] New data to be transformed.

X_new [array, shape (n_samples, n_components)] Embedding of the new data in low-dimensional space.

umap.umap_.**compute_membership_strengths**

Construct the membership strength data for the 1-skeleton of each local fuzzy simplicial set – this is formed as a sparse matrix where each row is a local fuzzy simplicial set, with a membership strength for the 1-simplex to each other data point.

knn_indices: array of shape (n_samples, n_neighbors) The indices on the `n_neighbors` closest points in the dataset.

knn_dists: array of shape (n_samples, n_neighbors) The distances to the `n_neighbors` closest points in the dataset.

sigmas: array of shape(n_samples) The normalization factor derived from the metric tensor approximation.

rhos: array of shape(n_samples) The local connectivity adjustment.

return_dists: bool (optional, default False) Whether to return the pairwise distance associated with each edge

bipartite: bool (optional, default False) Does the nearest neighbour set represent a bipartite graph? That is are the nearest neighbour indices from the same point set as the row indices?

rows: array of shape (n_samples * n_neighbors) Row data for the resulting sparse matrix (coo format)

cols: array of shape (n_samples * n_neighbors) Column data for the resulting sparse matrix (coo format)

vals: array of shape (n_samples * n_neighbors) Entries for the resulting sparse matrix (coo format)

dists: array of shape (n_samples * n_neighbors) Distance associated with each entry in the resulting sparse matrix

umap.umap_.**discrete_metric_simplicial_set_intersection**(*simplicial_set*, *discrete_space*, *unknown_dist=1.0*, *far_dist=5.0*, *metric=None*, *metric_kws={}*, *metric_scale=1.0*)

Combine a fuzzy simplicial set with another fuzzy simplicial set generated from discrete metric data using discrete distances. The target data is assumed to be categorical label data (a vector of labels), and this will update the fuzzy simplicial set to respect that label data.

TODO: optional category cardinality based weighting of distance

simplicial_set: sparse matrix The input fuzzy simplicial set.

discrete_space: array of shape (n_samples) The categorical labels to use in the intersection.

unknown_dist: float (optional, default 1.0) The distance an unknown label (-1) is assumed to be from any point.

far_dist: float (optional, default 5.0) The distance between unmatched labels.

metric: str (optional, default None) If not None, then use this metric to determine the distance between values.

metric_scale: float (optional, default 1.0) If using a custom metric scale the distance values by this value – this controls the weighting of the intersection. Larger values weight more toward target.

simplicial_set: sparse matrix The resulting intersected fuzzy simplicial set.

umap.umap_.**fast_intersection**

Under the assumption of categorical distance for the intersecting simplicial set perform a fast intersection.

rows: array An array of the row of each non-zero in the sparse matrix representation.

cols: array An array of the column of each non-zero in the sparse matrix representation.

values: array An array of the value of each non-zero in the sparse matrix representation.

target: array of shape (n_samples) The categorical labels to use in the intersection.

unknown_dist: float (optional, default 1.0) The distance an unknown label (-1) is assumed to be from any point.

far_dist float (optional, default 5.0) The distance between unmatched labels.

None

umap.umap_.**fast_metric_intersection**

Under the assumption of categorical distance for the intersecting simplicial set perform a fast intersection.

rows: array An array of the row of each non-zero in the sparse matrix representation.

cols: array An array of the column of each non-zero in the sparse matrix representation.

values: array of shape An array of the values of each non-zero in the sparse matrix representation.

discrete_space: array of shape (n_samples, n_features) The vectors of categorical labels to use in the intersection.

metric: numba function The function used to calculate distance over the target array.

scale: float A scaling to apply to the metric.

None

umap.umap_.**find_ab_params**(*spread*, *min_dist*)

Fit a, b params for the differentiable curve used in lower dimensional fuzzy simplicial complex construction. We want the smooth curve (from a pre-defined family with simple gradient) that best matches an offset exponential decay.

umap.umap_.**fuzzy_simplicial_set**(*X*, *n_neighbors*, *random_state*, *metric*, *metric_kwds={}*, *knn_indices=None*, *knn_dists=None*, *angular=False*, *set_op_mix_ratio=1.0*, *local_connectivity=1.0*, *apply_set_operations=True*, *verbose=False*, *return_dists=None*)

Given a set of data X, a neighborhood size, and a measure of distance compute the fuzzy simplicial set (here represented as a fuzzy graph in the form of a sparse matrix) associated to the data. This is done by locally approximating geodesic distance at each point, creating a fuzzy simplicial set for each such point, and then combining all the local fuzzy simplicial sets into a global one via a fuzzy union.

X: array of shape (n_samples, n_features) The data to be modelled as a fuzzy simplicial set.

n_neighbors: int The number of neighbors to use to approximate geodesic distance. Larger numbers induce more global estimates of the manifold that can miss finer detail, while smaller values will focus on fine manifold structure to the detriment of the larger picture.

random_state: numpy RandomState or equivalent A state capable being used as a numpy random state.

metric: string or function (optional, default 'euclidean') The metric to use to compute distances in high dimensional space. If a string is passed it must match a valid predefined metric. If a general metric is required a function that takes two 1d arrays and returns a float can be provided. For performance purposes it is required that this be a numba jit'd function. Valid string metrics include:

- euclidean (or l2)
- manhattan (or l1)
- cityblock
- braycurtis
- canberra
- chebyshev
- correlation
- cosine
- dice
- hamming
- jaccard
- kulsinski
- ll_dirichlet
- mahalanobis
- matching
- minkowski
- rogerstanimoto
- russellrao

- seuclidean
- sokalmichener
- sokalsneath
- sqeuclidean
- yule
- wminkowski

Metrics that take arguments (such as minkowski, mahalanobis etc.) can have arguments passed via the metric_kwds dictionary. At this time care must be taken and dictionary elements must be ordered appropriately; this will hopefully be fixed in the future.

metric_kwds: dict (optional, default {}) Arguments to pass on to the metric, such as the `p` value for Minkowski distance.

knn_indices: array of shape (n_samples, n_neighbors) (optional) If the k-nearest neighbors of each point has already been calculated you can pass them in here to save computation time. This should be an array with the indices of the k-nearest neighbors as a row for each data point.

knn_dists: array of shape (n_samples, n_neighbors) (optional) If the k-nearest neighbors of each point has already been calculated you can pass them in here to save computation time. This should be an array with the distances of the k-nearest neighbors as a row for each data point.

angular: bool (optional, default False) Whether to use angular/cosine distance for the random projection forest for seeding NN-descent to determine approximate nearest neighbors.

set_op_mix_ratio: float (optional, default 1.0) Interpolate between (fuzzy) union and intersection as the set operation used to combine local fuzzy simplicial sets to obtain a global fuzzy simplicial sets. Both fuzzy set operations use the product t-norm. The value of this parameter should be between 0.0 and 1.0; a value of 1.0 will use a pure fuzzy union, while 0.0 will use a pure fuzzy intersection.

local_connectivity: int (optional, default 1) The local connectivity required – i.e. the number of nearest neighbors that should be assumed to be connected at a local level. The higher this value the more connected the manifold becomes locally. In practice this should be not more than the local intrinsic dimension of the manifold.

verbose: bool (optional, default False) Whether to report information on the current progress of the algorithm.

return_dists: bool or None (optional, default None) Whether to return the pairwise distance associated with each edge.

fuzzy_simplicial_set: coo_matrix A fuzzy simplicial set represented as a sparse matrix. The (i, j) entry of the matrix represents the membership strength of the 1-simplex between the ith and jth sample points.

`umap.umap_.`**`init_graph_transform`**`(`*graph*, *embedding*`)`

Given a bipartite graph representing the 1-simplices and strengths between the new points and the original data set along with an embedding of the original points

initialize the positions of new points relative to the strengths (of their neighbors in the source data).

If a point is in our original data set it embeds at the original points coordinates. If a point has no neighbours in our original dataset it embeds as the np.nan vector. Otherwise a point is the weighted average of it's neighbours embedding locations.

graph: csr_matrix (n_new_samples, n_samples) A matrix indicating the the 1-simplices and their associated strengths. These strengths should be values between zero and one and not normalized. One indicating that the new point was identical to one of our original points.

embedding: array of shape (n_samples, dim) The original embedding of the source data.

new_embedding: array of shape (n_new_samples, dim) An initial embedding of the new sample points.

umap.umap_.**init_transform**
Given indices and weights and an original embeddings initialize the positions of new points relative to the indices and weights (of their neighbors in the source data).

indices: array of shape (n_new_samples, n_neighbors) The indices of the neighbors of each new sample

weights: array of shape (n_new_samples, n_neighbors) The membership strengths of associated 1-simplices for each of the new samples.

embedding: array of shape (n_samples, dim) The original embedding of the source data.

new_embedding: array of shape (n_new_samples, dim) An initial embedding of the new sample points.

umap.umap_.**make_epochs_per_sample**(*weights*, *n_epochs*)
Given a set of weights and number of epochs generate the number of epochs per sample for each weight.

weights: array of shape (n_1_simplices) The weights ofhow much we wish to sample each 1-simplex.

n_epochs: int The total number of epochs we want to train for.

An array of number of epochs per sample, one for each 1-simplex.

umap.umap_.**nearest_neighbors**(*X*, *n_neighbors*, *metric*, *metric_kwds*, *angular*, *random_state*, *low_memory=True*, *use_pynndescent=True*, *n_jobs=-1*, *verbose=False*)
Compute the n_neighbors nearest points for each data point in X under metric. This may be exact, but more likely is approximated via nearest neighbor descent.

X: array of shape (n_samples, n_features) The input data to compute the k-neighbor graph of.

n_neighbors: int The number of nearest neighbors to compute for each sample in X.

metric: string or callable The metric to use for the computation.

metric_kwds: dict Any arguments to pass to the metric computation function.

angular: bool Whether to use angular rp trees in NN approximation.

random_state: np.random state The random state to use for approximate NN computations.

low_memory: bool (optional, default False) Whether to pursue lower memory NNdescent.

verbose: bool (optional, default False) Whether to print status data during the computation.

knn_indices: array of shape (n_samples, n_neighbors) The indices on the n_neighbors closest points in the dataset.

knn_dists: array of shape (n_samples, n_neighbors) The distances to the n_neighbors closest points in the dataset.

rp_forest: list of trees The random projection forest used for searching (if used, None otherwise)

umap.umap_.**raise_disconnected_warning**(*edges_removed*, *vertices_disconnected*, *disconnection_distance*, *total_rows*, *threshold=0.1*, *verbose=False*)
A simple wrapper function to avoid large amounts of code repetition.

umap.umap_.**reset_local_connectivity**(*simplicial_set*, *reset_local_metric=False*)

Reset the local connectivity requirement – each data sample should have complete confidence in at least one 1-simplex in the simplicial set. We can enforce this by locally rescaling confidences, and then remerging the different local simplicial sets together.

simplicial_set: sparse matrix The simplicial set for which to recalculate with respect to local connectivity.

simplicial_set: sparse_matrix The recalculated simplicial set, now with the local connectivity assumption restored.

umap.umap_.**simplicial_set_embedding**(*data*, *graph*, *n_components*, *initial_alpha*, *a*, *b*, *gamma*, *negative_sample_rate*, *n_epochs*, *init*, *random_state*, *metric*, *metric_kwds*, *densmap*, *densmap_kwds*, *output_dens*, *output_metric=CPUDispatcher(<function euclidean_grad>)*, *output_metric_kwds={}*, *euclidean_output=True*, *parallel=False*, *verbose=False*)

Perform a fuzzy simplicial set embedding, using a specified initialisation method and then minimizing the fuzzy set cross entropy between the 1-skeletons of the high and low dimensional fuzzy simplicial sets.

data: array of shape (n_samples, n_features) The source data to be embedded by UMAP.

graph: sparse matrix The 1-skeleton of the high dimensional fuzzy simplicial set as represented by a graph for which we require a sparse matrix for the (weighted) adjacency matrix.

n_components: int The dimensionality of the euclidean space into which to embed the data.

initial_alpha: float Initial learning rate for the SGD.

a: float Parameter of differentiable approximation of right adjoint functor

b: float Parameter of differentiable approximation of right adjoint functor

gamma: float Weight to apply to negative samples.

negative_sample_rate: int (optional, default 5) The number of negative samples to select per positive sample in the optimization process. Increasing this value will result in greater repulsive force being applied, greater optimization cost, but slightly more accuracy.

n_epochs: int (optional, default 0) The number of training epochs to be used in optimizing the low dimensional embedding. Larger values result in more accurate embeddings. If 0 is specified a value will be selected based on the size of the input dataset (200 for large datasets, 500 for small).

init: string

How to initialize the low dimensional embedding. Options are:

- 'spectral': use a spectral embedding of the fuzzy 1-skeleton
- 'random': assign initial embedding positions at random.
- A numpy array of initial embedding positions.

random_state: numpy RandomState or equivalent A state capable being used as a numpy random state.

metric: string or callable The metric used to measure distance in high dimensional space; used if multiple connected components need to be layed out.

metric_kwds: dict Key word arguments to be passed to the metric function; used if multiple connected components need to be layed out.

densmap: bool Whether to use the density-augmented objective function to optimize the embedding according to the densMAP algorithm.

densmap_kwds: dict Key word arguments to be used by the densMAP optimization.

output_dens: bool Whether to output local radii in the original data and the embedding.

output_metric: function Function returning the distance between two points in embedding space and the gradient of the distance wrt the first argument.

output_metric_kwds: dict Key word arguments to be passed to the output_metric function.

euclidean_output: bool Whether to use the faster code specialised for euclidean output metrics

parallel: bool (optional, default False) Whether to run the computation using numba parallel. Running in parallel is non-deterministic, and is not used if a random seed has been set, to ensure reproducibility.

verbose: bool (optional, default False) Whether to report information on the current progress of the algorithm.

embedding: array of shape (n_samples, n_components) The optimized of `graph` into an `n_components` dimensional euclidean space.

aux_data: dict Auxiliary output returned with the embedding. When densMAP extension is turned on, this dictionary includes local radii in the original data (`rad_orig`) and in the embedding (`rad_emb`).

umap.umap_.**smooth_knn_dist**

Compute a continuous version of the distance to the kth nearest neighbor. That is, this is similar to knn-distance but allows continuous k values rather than requiring an integral k. In essence we are simply computing the distance such that the cardinality of fuzzy set we generate is k.

distances: array of shape (n_samples, n_neighbors) Distances to nearest neighbors for each samples. Each row should be a sorted list of distances to a given samples nearest neighbors.

k: float The number of nearest neighbors to approximate for.

n_iter: int (optional, default 64) We need to binary search for the correct distance value. This is the max number of iterations to use in such a search.

local_connectivity: int (optional, default 1) The local connectivity required – i.e. the number of nearest neighbors that should be assumed to be connected at a local level. The higher this value the more connected the manifold becomes locally. In practice this should be not more than the local intrinsic dimension of the manifold.

bandwidth: float (optional, default 1) The target bandwidth of the kernel, larger values will produce larger return values.

knn_dist: array of shape (n_samples,) The distance to kth nearest neighbor, as suitably approximated.

nn_dist: array of shape (n_samples,) The distance to the 1st nearest neighbor for each point.

CHAPTER 26

Indices and tables

- genindex
- modindex
- search

Python Module Index

u

Index