umap Documentation

Release 0.3

Leland McInnes

Jul 12, 2018

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 littl easier.

```
import numpy as np
from sklearn.datasets import load_iris, load_digits
from sklearn.model_selection import train_test_split
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 Iris data

The next step is to get some data to work with. To ease us into things we'll start with the iris 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. We can load the iris dataset from sklearn.

```
iris = load_iris()
print(iris.DESCR)
```

```
Iris Plants Database
====================

Notes
-----
Data Set Characteristics:
    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, predictive attributes and the class
    :Attribute Information:
        - sepal length in cm
        - sepal width in cm
        - petal length in cm
        - petal width in cm
        - class:
                - Iris-Setosa
                - Iris-Versicolour
                - Iris-Virginica
    :Summary Statistics:

    ============== ==== ==== ======= ===== ====================
                    Min  Max   Mean    SD   Class Correlation
    ============== ==== ==== ======= ===== ====================
    sepal length:   4.3  7.9   5.84   0.83    0.7826
    sepal width:    2.0  4.4   3.05   0.43   -0.4194
    petal length:   1.0  6.9   3.76   1.76    0.9490  (high!)
    petal width:    0.1  2.5   1.20  0.76     0.9565  (high!)
    ============== ==== ==== ======= ===== ====================

    :Missing Attribute Values: None
    :Class Distribution: 33.3% for each of 3 classes.
    :Creator: R.A. Fisher
    :Donor: Michael Marshall (MARSHALL%PLU@io.arc.nasa.gov)
    :Date: July, 1988

This is a copy of UCI ML iris datasets.
http://archive.ics.uci.edu/ml/datasets/Iris

The famous Iris database, first used by Sir R.A Fisher

This is perhaps the best known database to be found in the
pattern recognition literature.  Fisher's paper is a classic in the field and
is referenced frequently to this day.  (See Duda & Hart, for example.)  The
data set contains 3 classes of 50 instances each, where each class refers to a
type of iris plant.  One class is linearly separable from the other 2; the
latter are NOT linearly separable from each other.

References
----------
   - Fisher,R.A. "The use of multiple measurements in taxonomic problems"
     Annual Eugenics, 7, Part II, 179-188 (1936); also in "Contributions to
     Mathematical Statistics" (John Wiley, NY, 1950).
   - Duda,R.O., & Hart,P.E. (1973) Pattern Classification and Scene Analysis.
     (Q327.D83) John Wiley & Sons.  ISBN 0-471-22361-1.  See page 218.
   - Dasarathy, B.V. (1980) "Nosing Around the Neighborhood: A New System
     Structure and Classification Rule for Recognition in Partially Exposed
     Environments".  IEEE Transactions on Pattern Analysis and Machine
     Intelligence, Vol. PAMI-2, No. 1, 67-71.
```

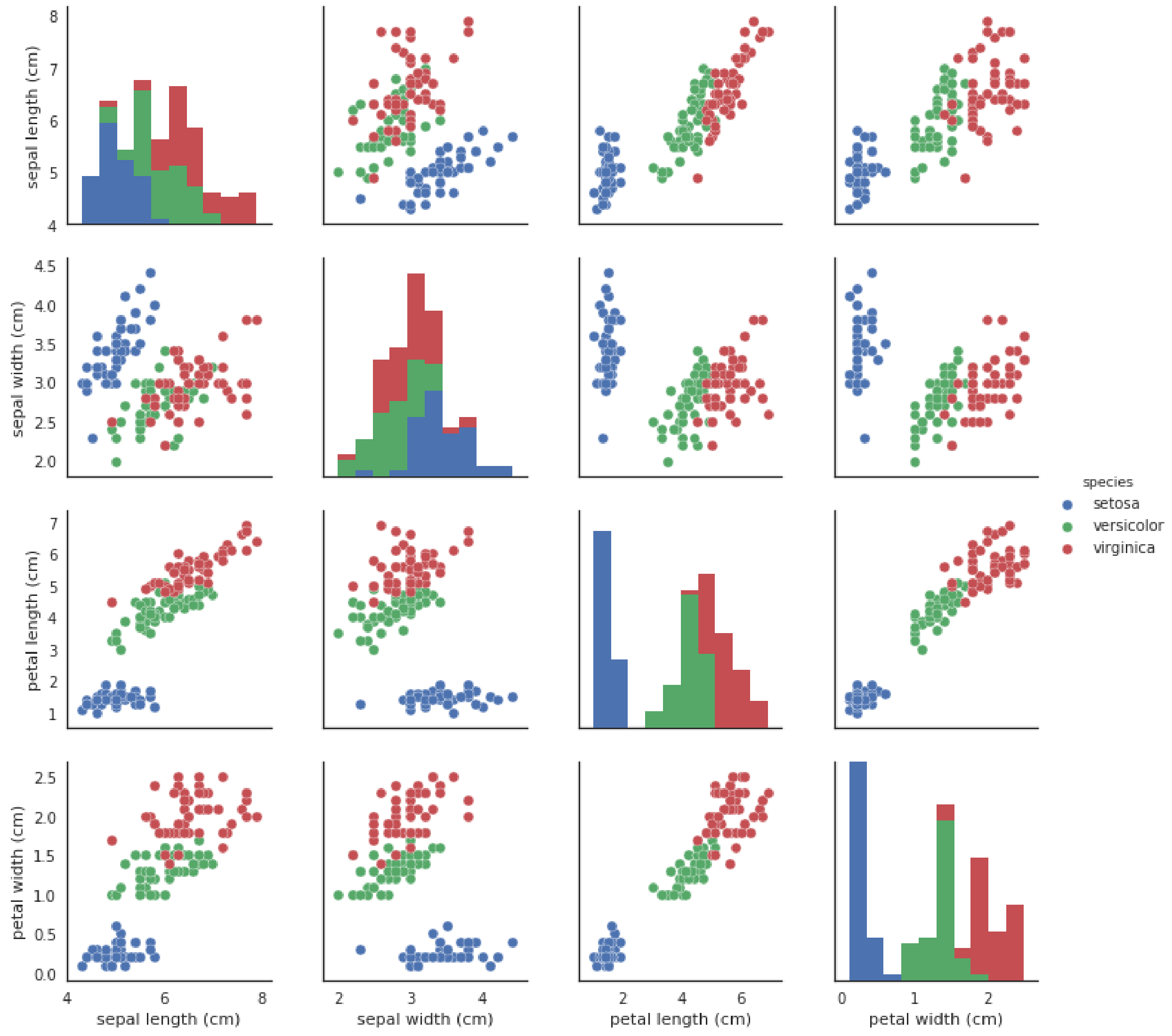
(continues on next page)

(continued from previous page)

```
   - Gates, G.W. (1972) "The Reduced Nearest Neighbor Rule".  IEEE Transactions
     on Information Theory, May 1972, 431-433.
   - See also: 1988 MLC Proceedings, 54-64.  Cheeseman et al"s AUTOCLASS II
     conceptual clustering system finds 3 classes in the data.
   - Many, many more ...
```

The description tells us a fair amount about the dataset – it consists of measurements of petals and sepals of iris flowers. There are 3 species of flower represented, each with 50 sets of measurements. 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 (once we get the data into a pandas dataframe).

```
iris_df = pd.DataFrame(iris.data, columns=iris.feature_names)
iris_df['species'] = pd.Series(iris.target).map(dict(zip(range(3),iris.target_names)))
sns.pairplot(iris_df, hue='species');
```



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 inuitions 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()
```

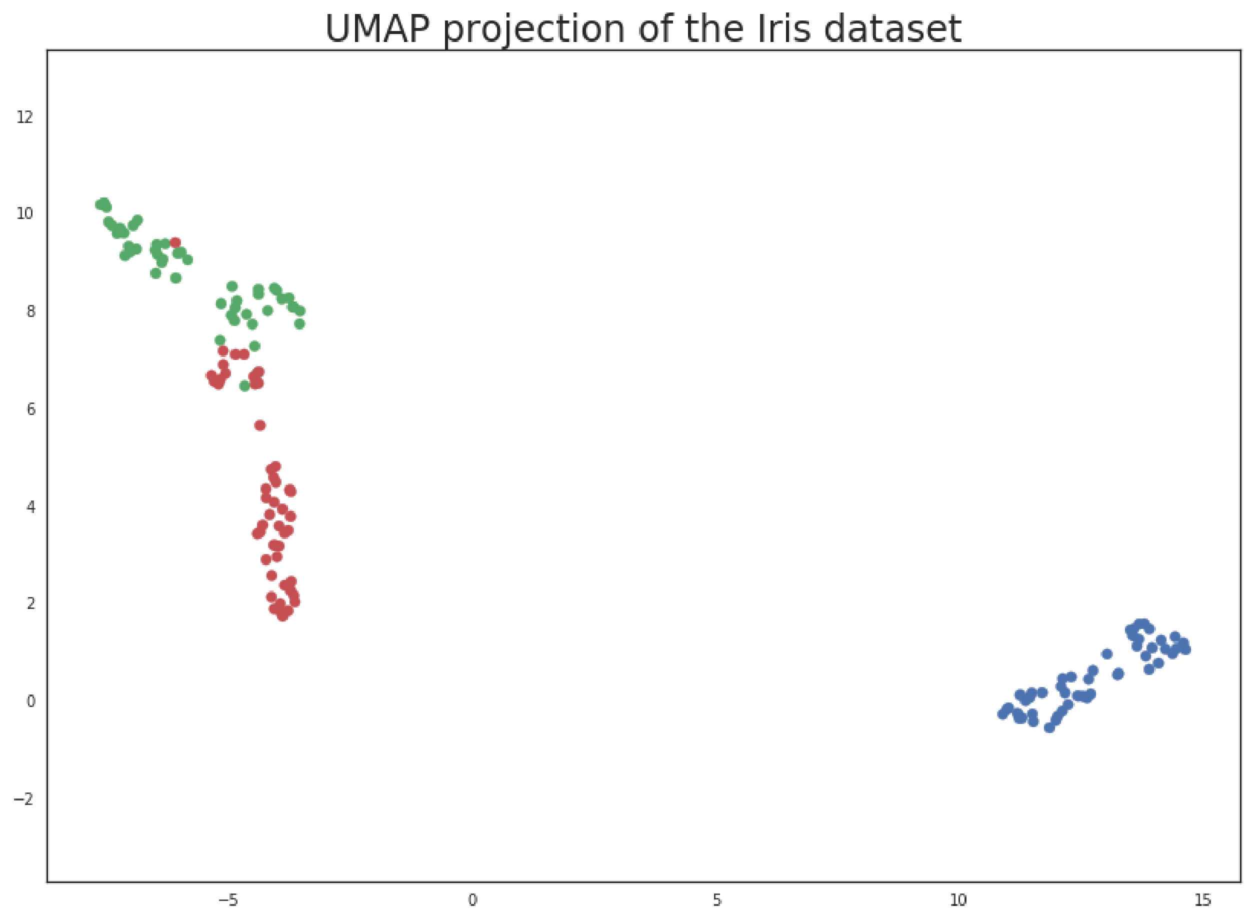
Now we need to train our reducer, letting it learn about the manifold. For this UMAP follows the sklearn API and has a method `fit` which we pass the data we want the model to learn from. Since, at the end of the day, we are going to want to reduced representation of the data we will use, instead, the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

```
embedding = reducer.fit_transform(iris.data)
embedding.shape
```

```
(150, 2)
```

The result is an array with 150 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 flower. 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 iris.
↪target])
plt.gca().set_aspect('equal', 'datalim')
plt.title('UMAP projection of the Iris 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 differnt 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 Iris 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)
```

```
Optical Recognition of Handwritten Digits Data Set
===================================================

Notes
-----
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
```

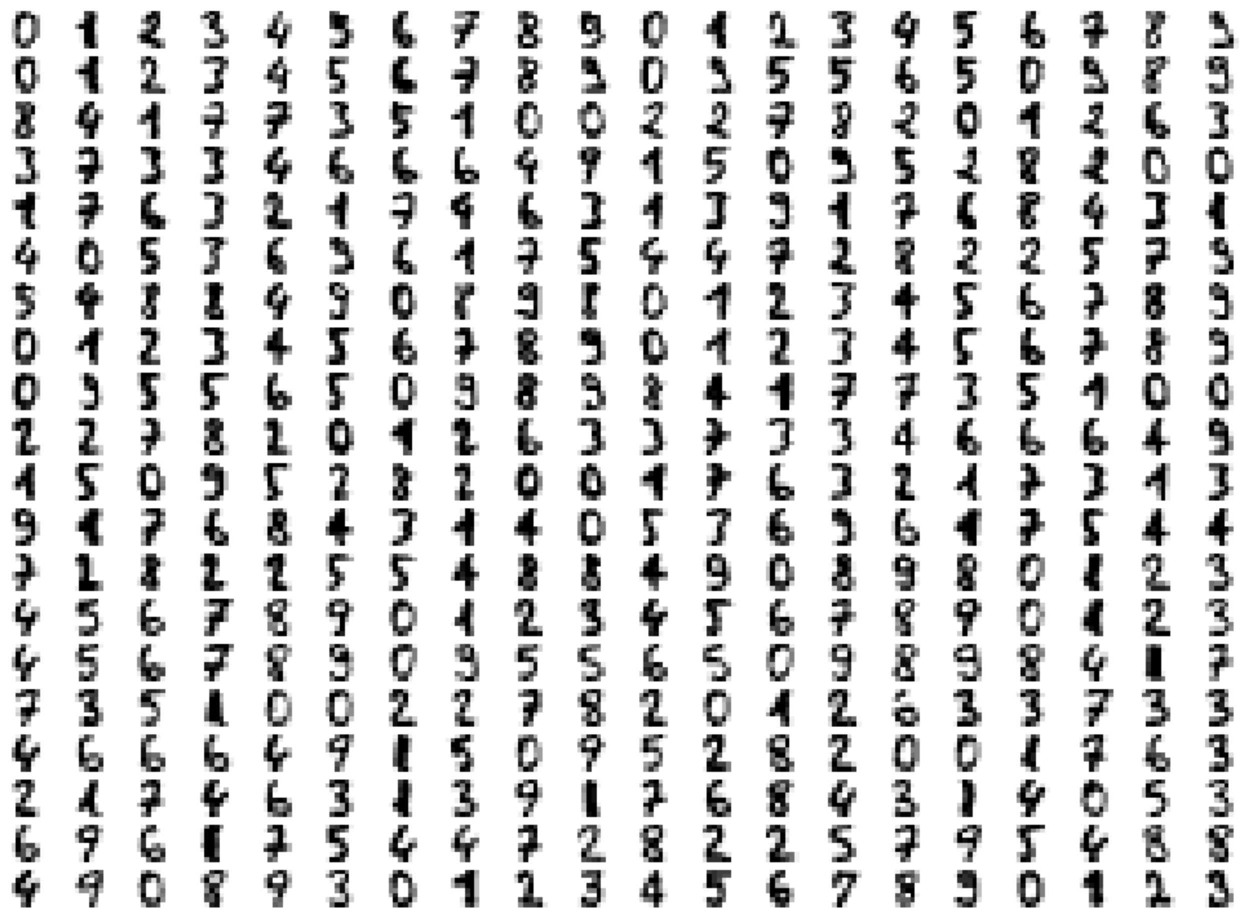
(continues on next page)

(continued from previous page)

```
    :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
http://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.

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.

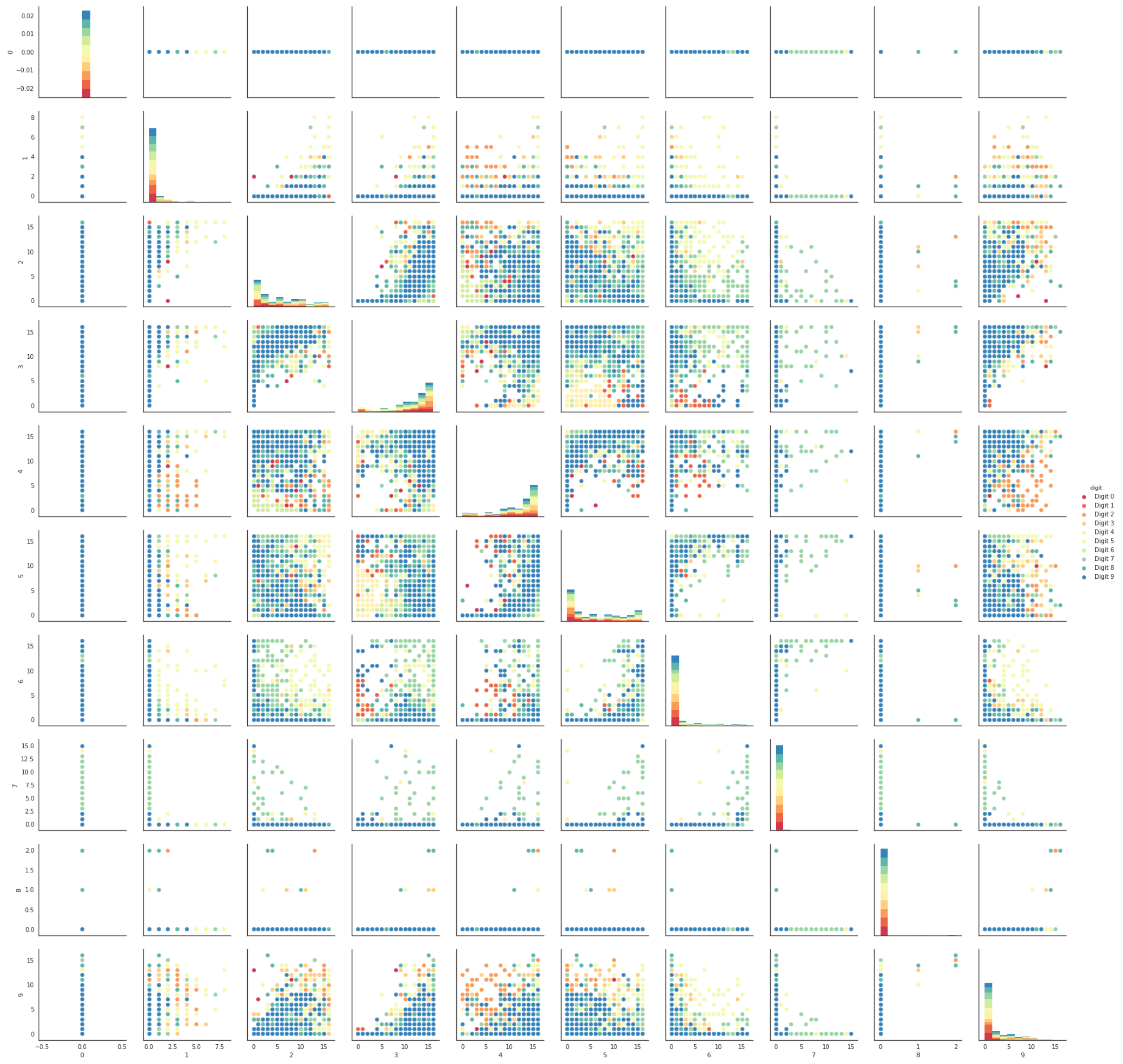
```
fig, ax_array = plt.subplots(20, 20)
axes = ax_array.flatten()
for i, ax in enumerate(axes):
    ax.imshow(digits.images[i], cmap='gray_r')
plt.setp(axes, xticks=[], yticks=[], frame_on=False)
plt.tight_layout(h_pad=0.5, w_pad=0.01)
```



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[:,:10])
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 Iris.

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

```
UMAP(a=1.576943460405378, alpha=1.0, angular_rp_forest=False,
   b=0.8950608781227859, bandwidth=1.0, gamma=1.0, init='spectral',
   local_connectivity=1.0, metric='euclidean', metric_kwds={},
   min_dist=0.1, n_components=2, n_epochs=None, n_neighbors=15,
   negative_sample_rate=5, random_state=42, set_op_mix_ratio=1.0,
   spread=1.0, target_metric='categorical', target_metric_kwds={},
   transform_queue_size=4.0, transform_seed=42, verbose=False)
```

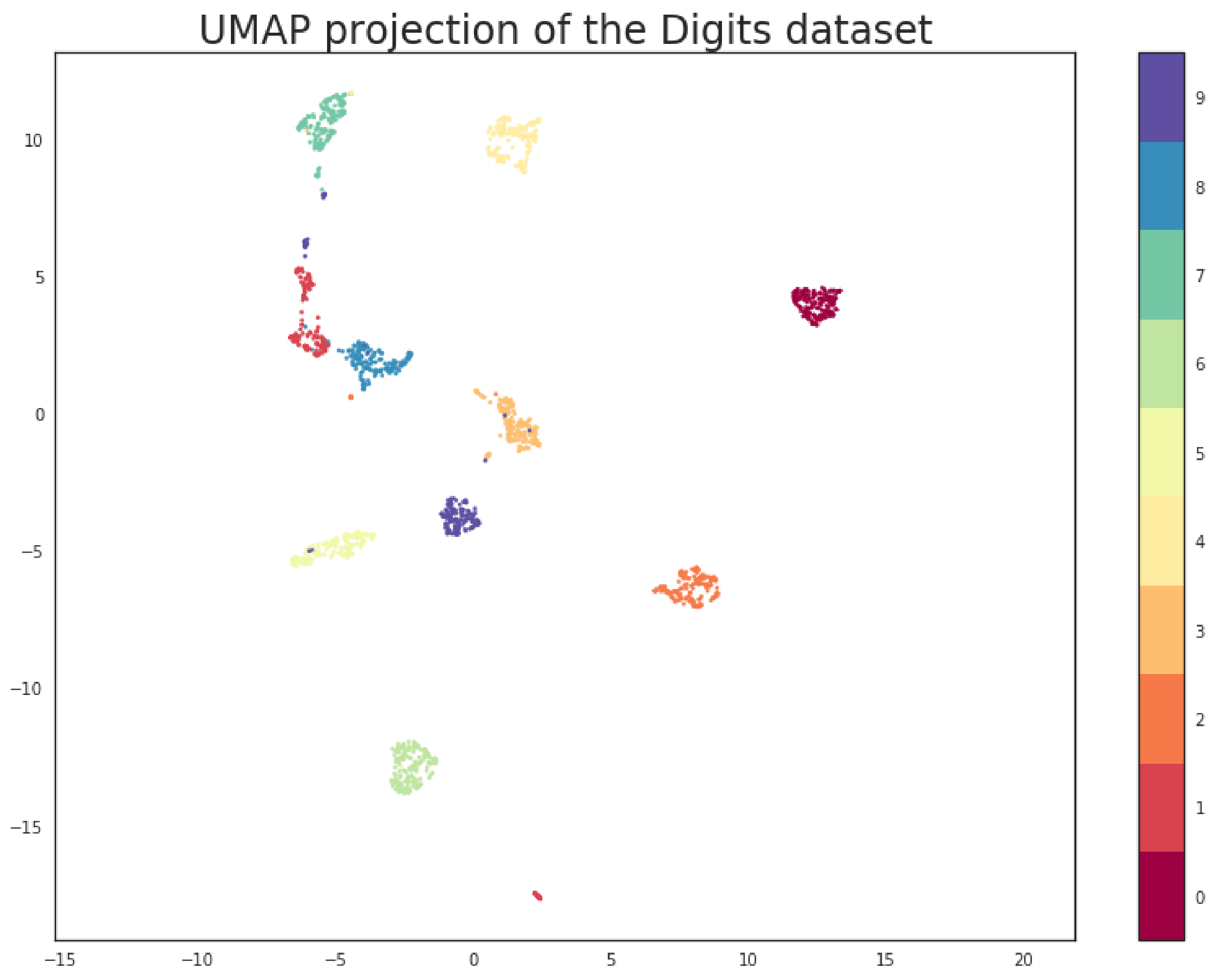
Now, instead of returning an embedding we simply get back the reducer object, now having trained on the dataset we passed it. To access the resulting transform we can either look at the `embedding_` attribute of the reducer object, or call transform on the original data.

```
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 Iris example we can now plot the resulting embedding, coloring the data points by the class that theyr 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 'data:image/png;base64,' + 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>
"""))

plot_figure.circle(
    'x',
```

(continues on next page)

(continued from previous page)

```
    '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.

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 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)
```

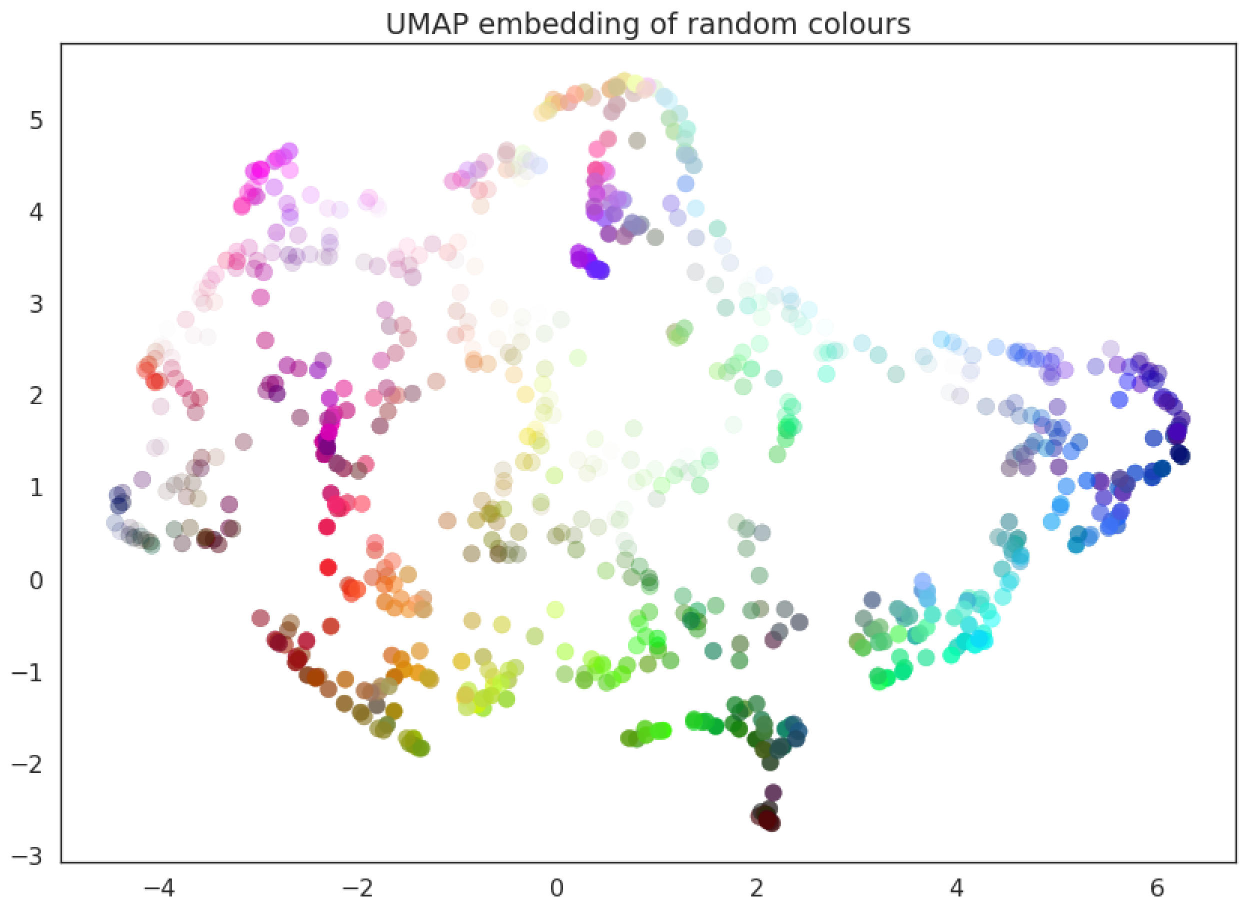
Now we need to find a low dimensional representation of the data. As in the Basic Usage documentation, we can do this by using the *`fit_transform()`* method on a *`UMAP`* object.

```
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 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.

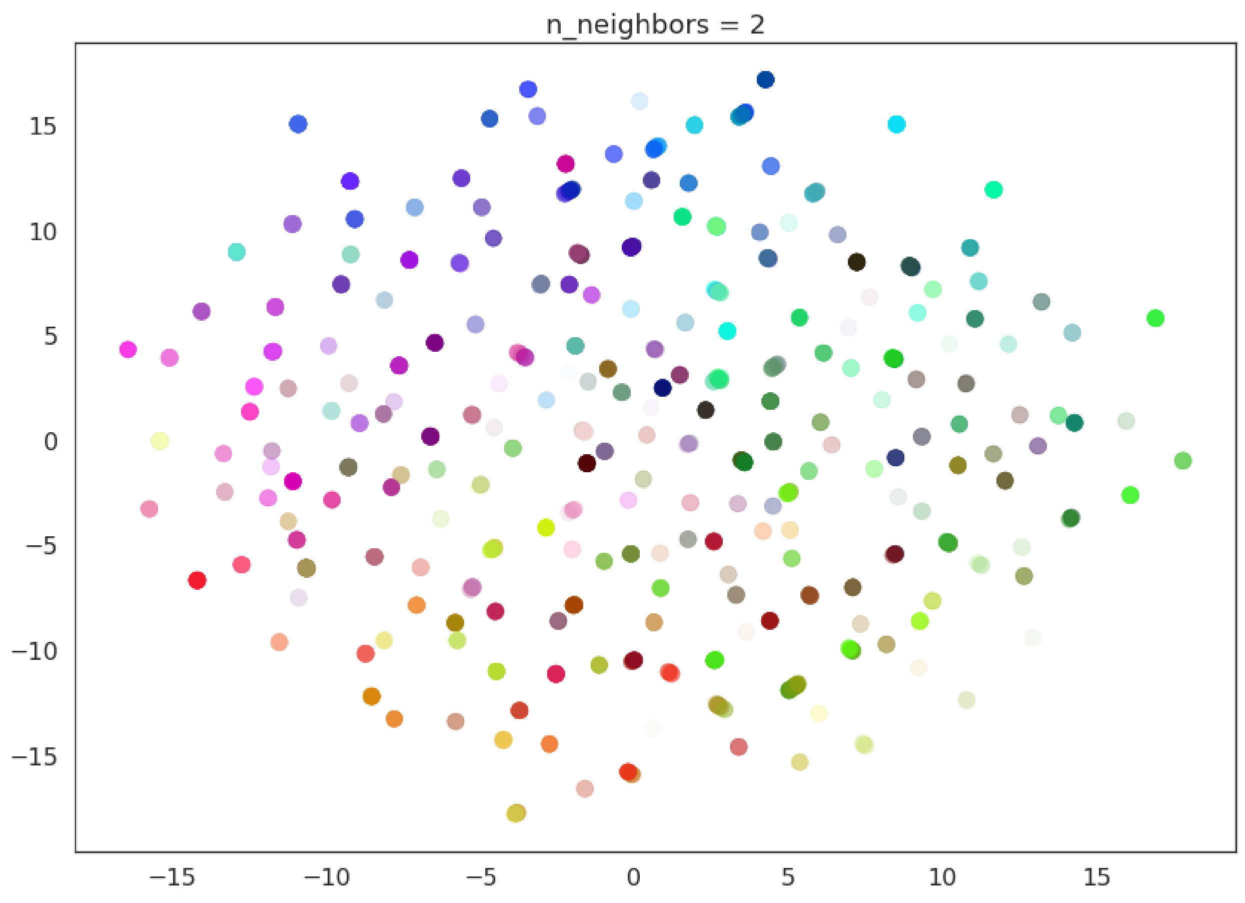
```
def draw_umap(n_neighbors=15, min_dist=0.1, n_components=2, metric='euclidean', title=
↪''):
    fit = umap.UMAP(
        n_neighbors=n_neighbors,
        min_dist=min_dist,
        n_components=n_components,
        metric=metric
    )
    u = fit.fit_transform(data);
    fig = plt.figure()
    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)
```

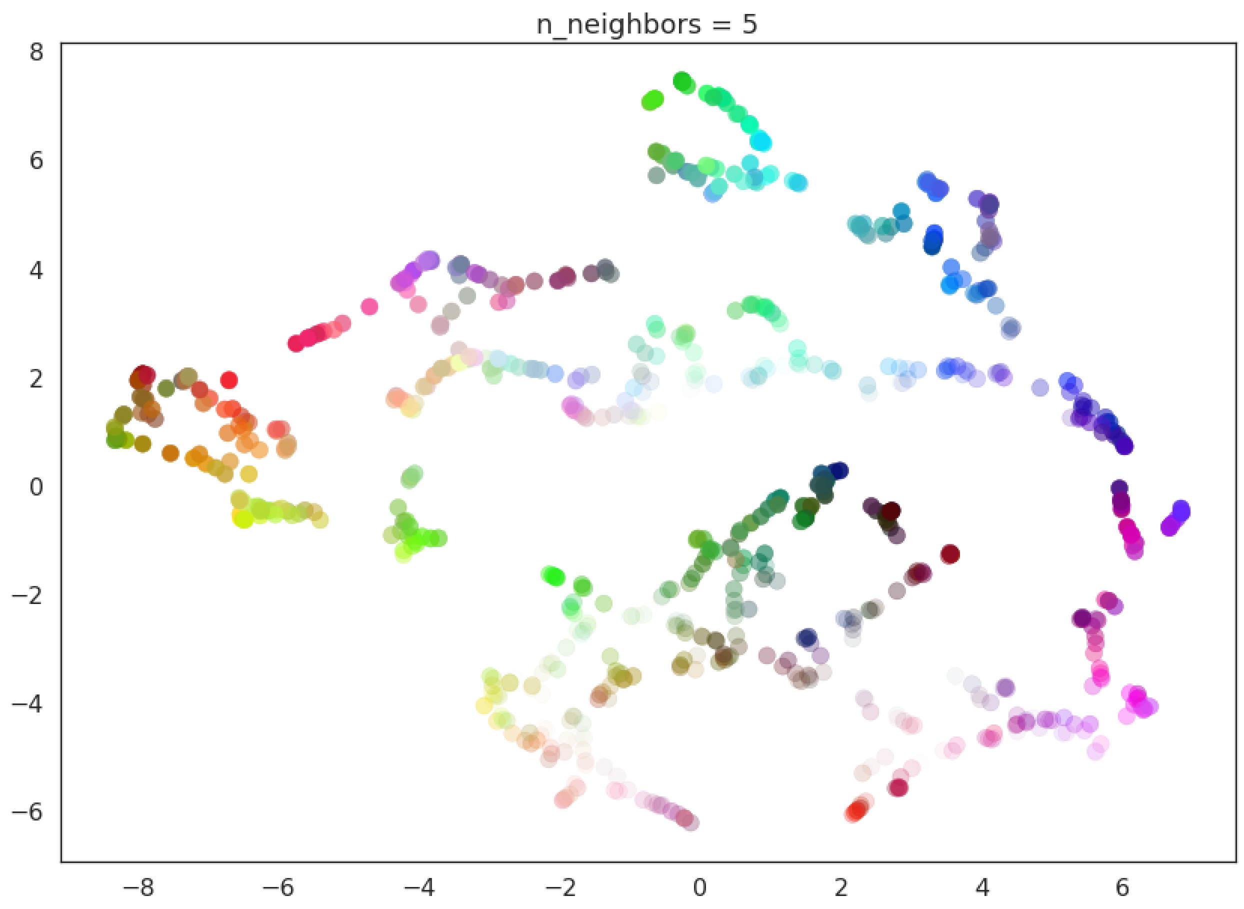
2.1 n_neighbors

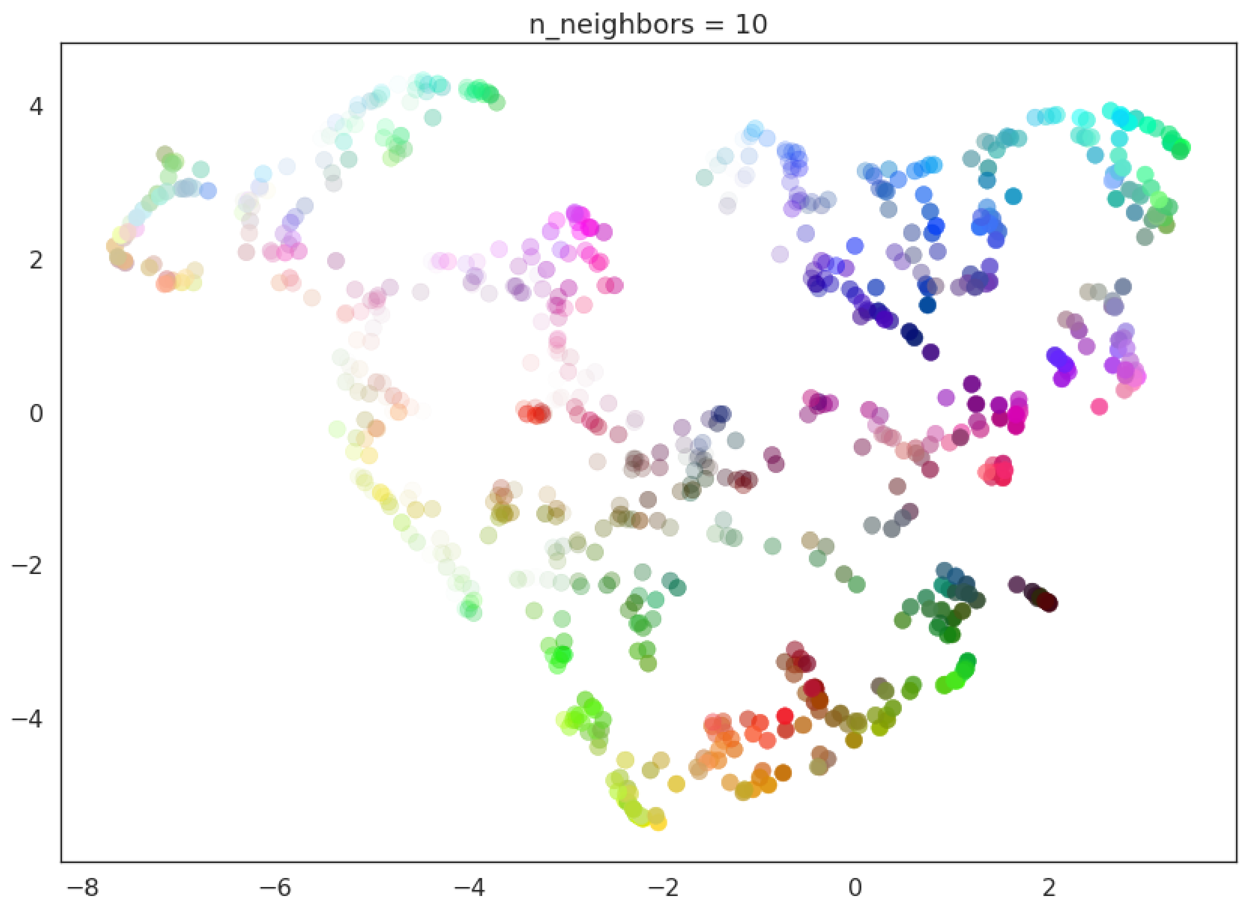
This parameter controls how UMAP balances local versus global structure in the data. It does this by constraining the size of the local neighborhood UMAP will look at when attempting to learn the manifold structure of the data. This means that low values of n_neighbors will force UMAP to concentrate 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, loosing fine detail structure for the sake of getting the broader of the data.

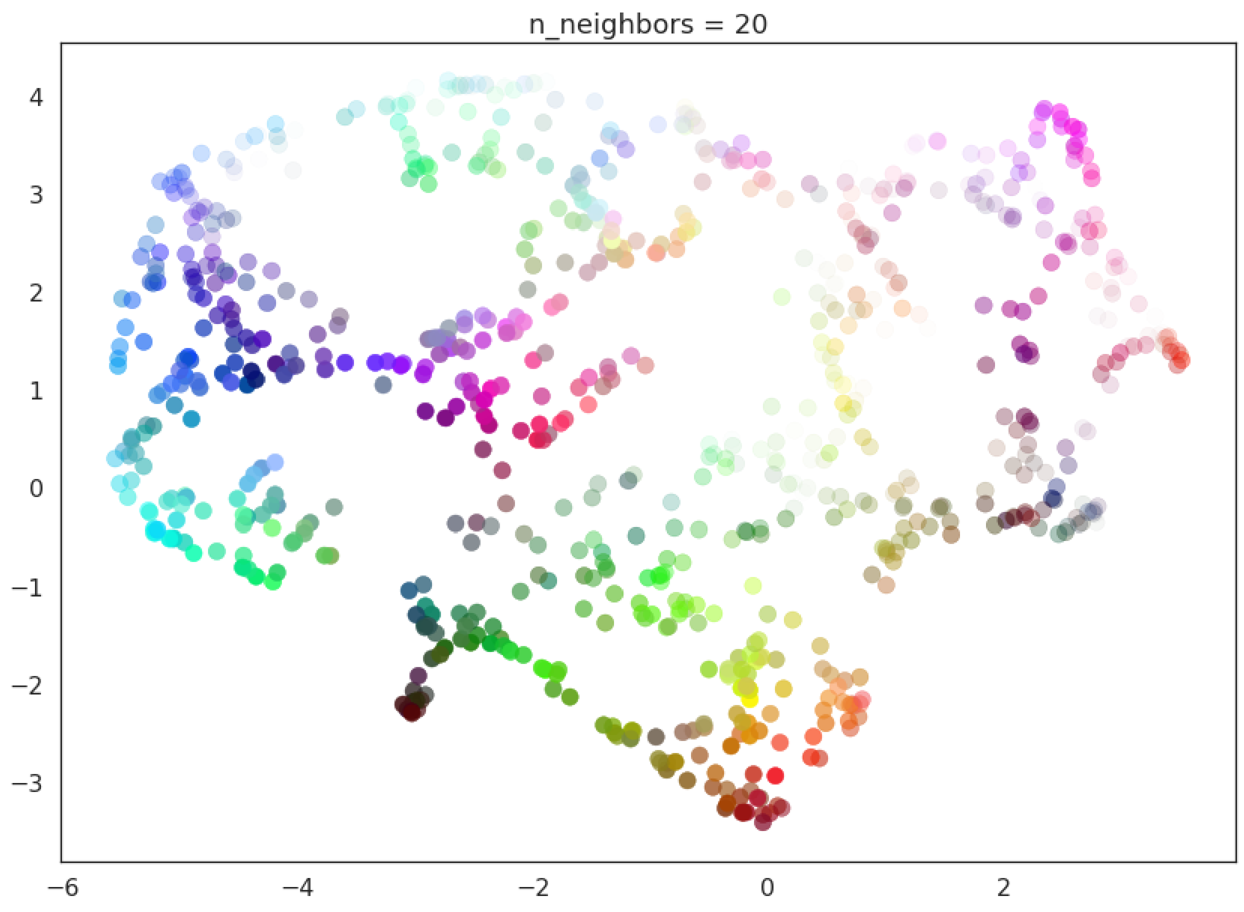
We can see that in practice by fitting our dataset with UMAP using a range of n_neighbors values. The default value of n_neighbors for UMAP (as used above) is 15, but we will look at values ranging from 2 (a very local view of the manifold) up to 200 (a quarter of the data).

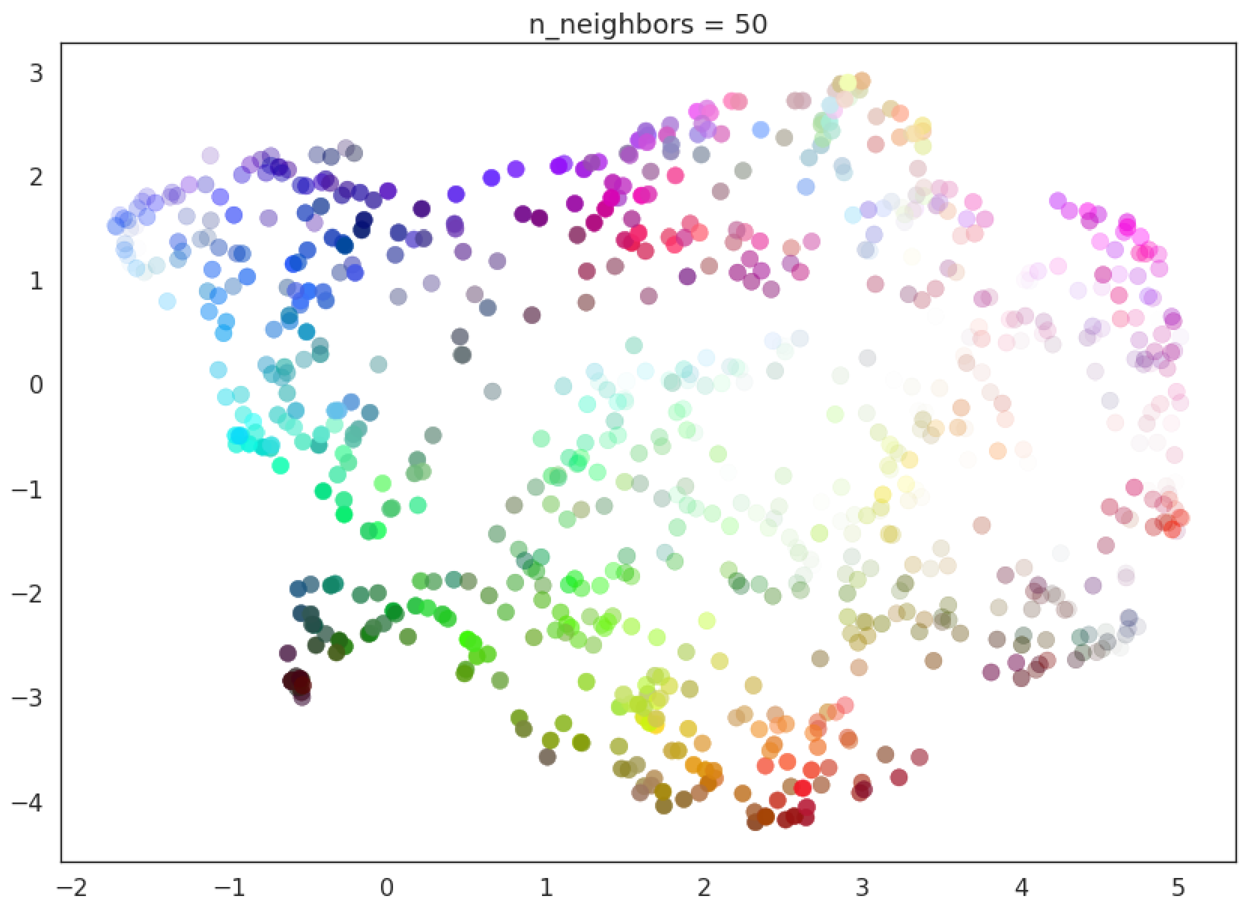
```
for n in (2, 5, 10, 20, 50, 100, 200):
    draw_umap(n_neighbors=n, title='n_neighbors = {}'.format(n))
```

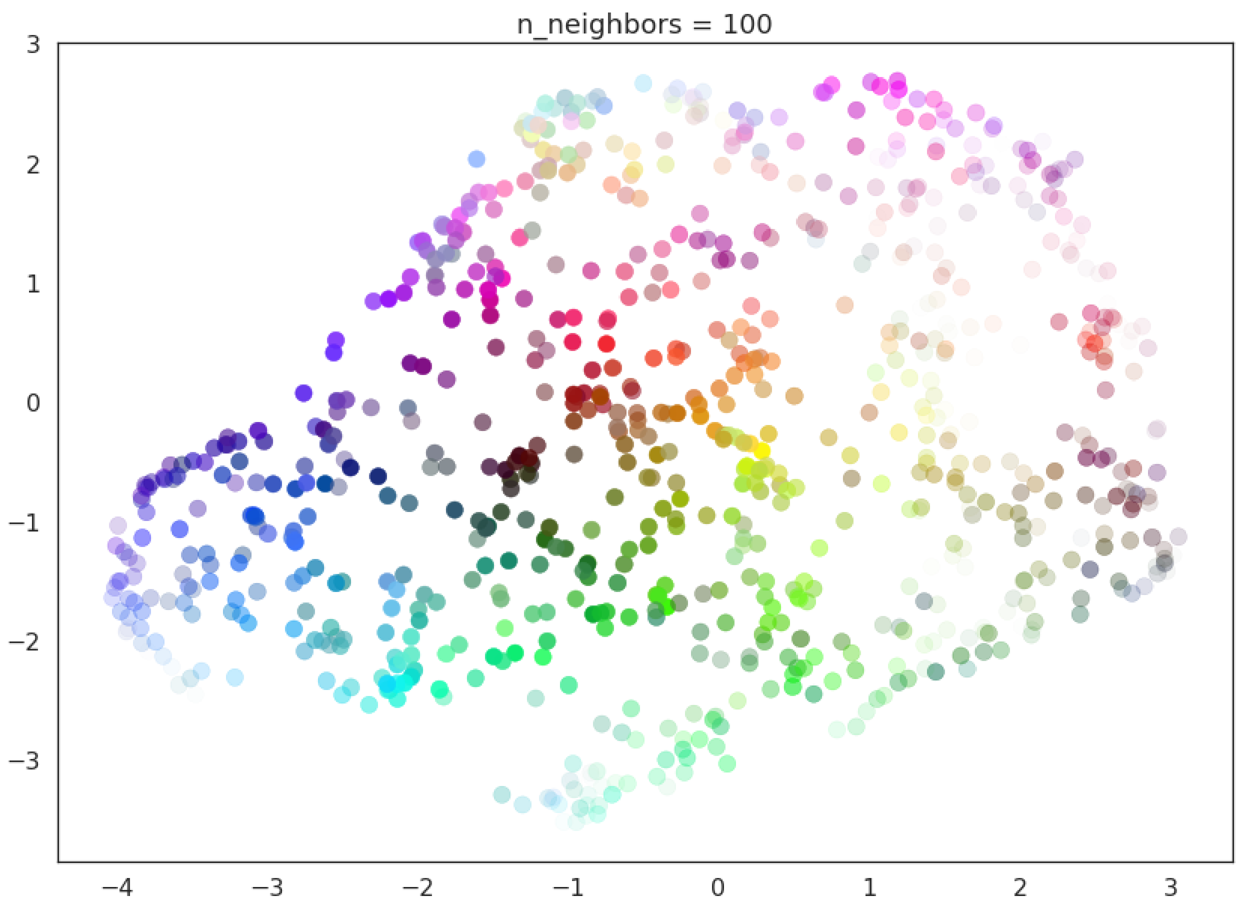

n_neighbors = 2
15
10
5
0
−5
−10
−15
−15
−10
−5
0
5
10
15

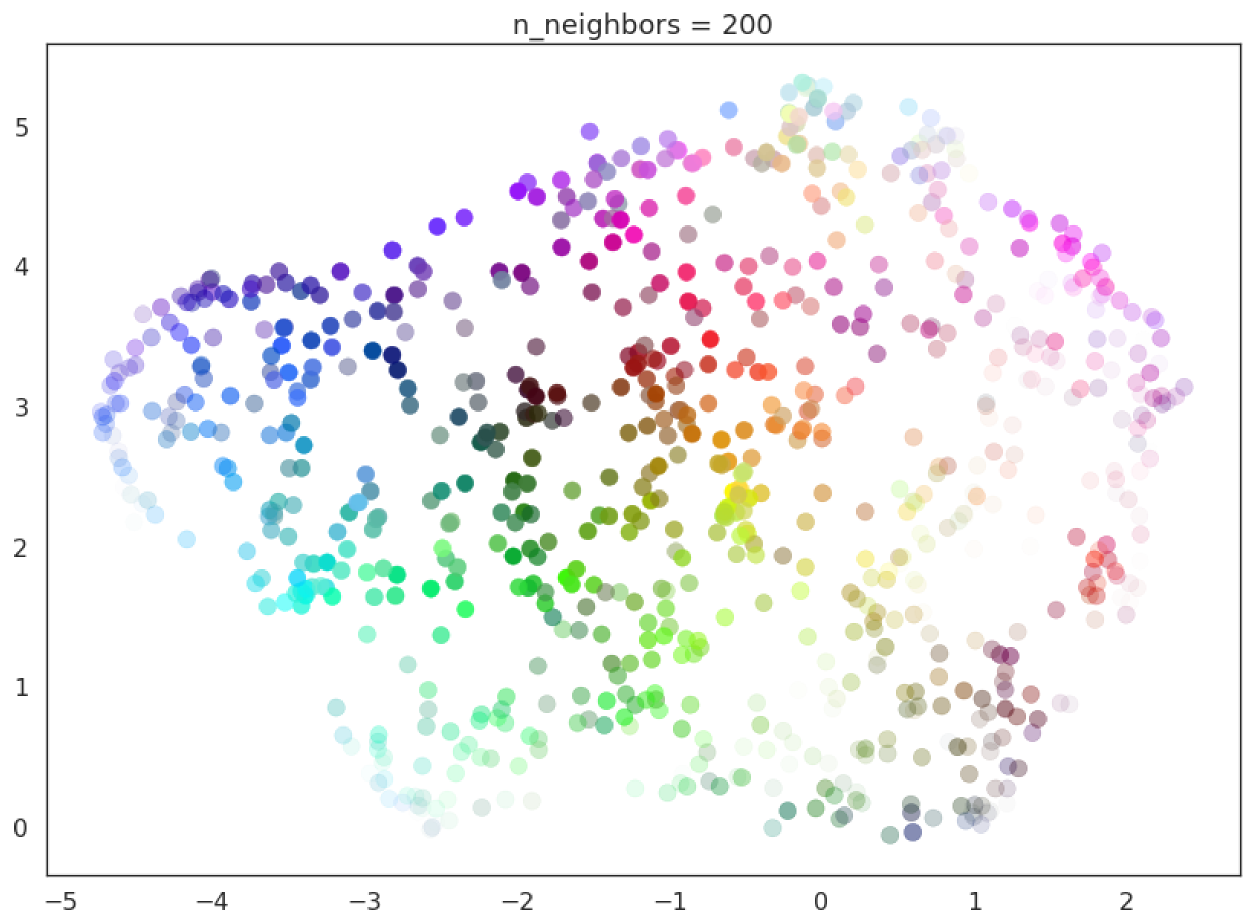

n_neighbors = 5
8
6
4
2
0
−2
−4
−6
−8
−6
−4
−2
0
2
4
6


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 sturcture (individual colors are no longer necessarily immediately near their closest color match).

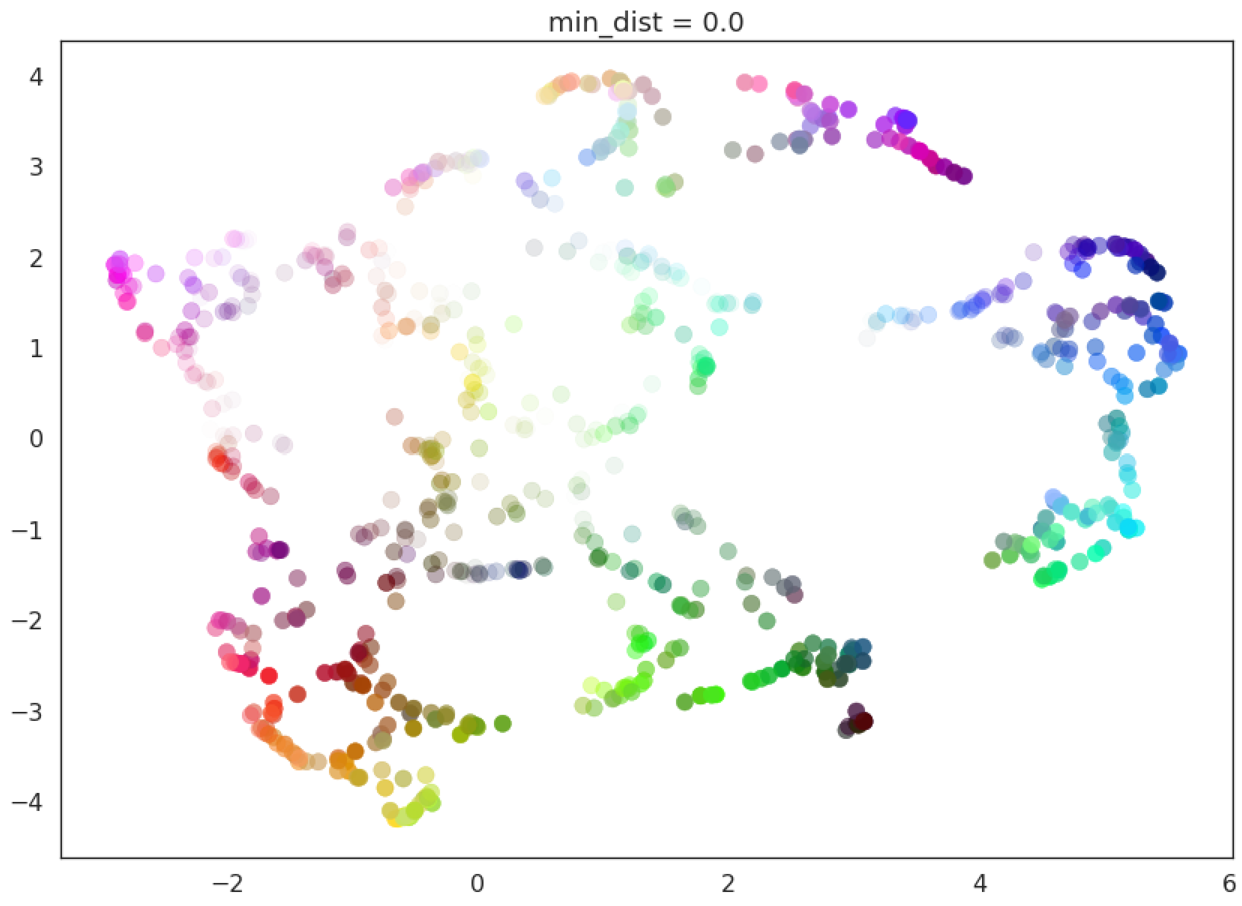
This effect well exemplifies the local/global tradeoff provided by `n_neighbors`.

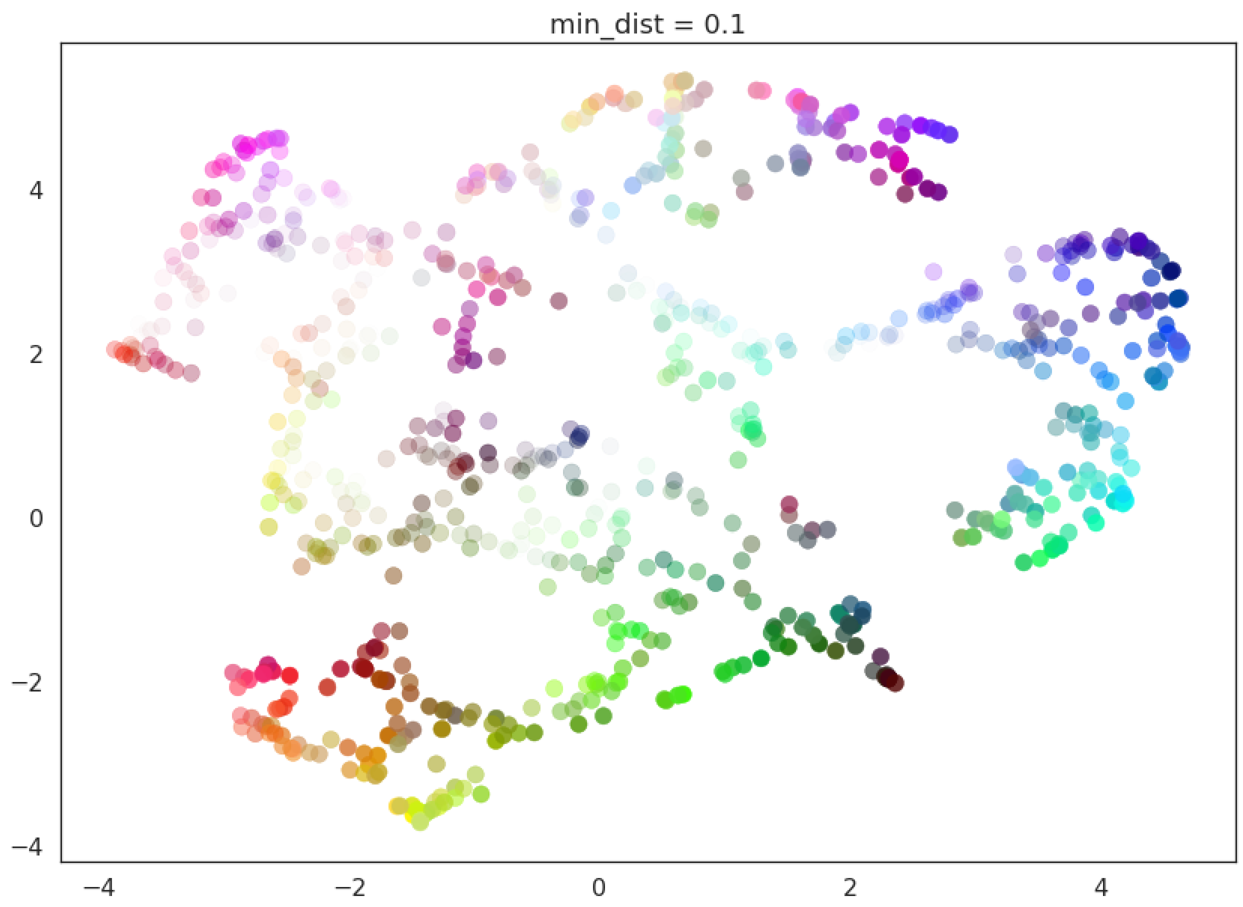
2.2 `min_dist`

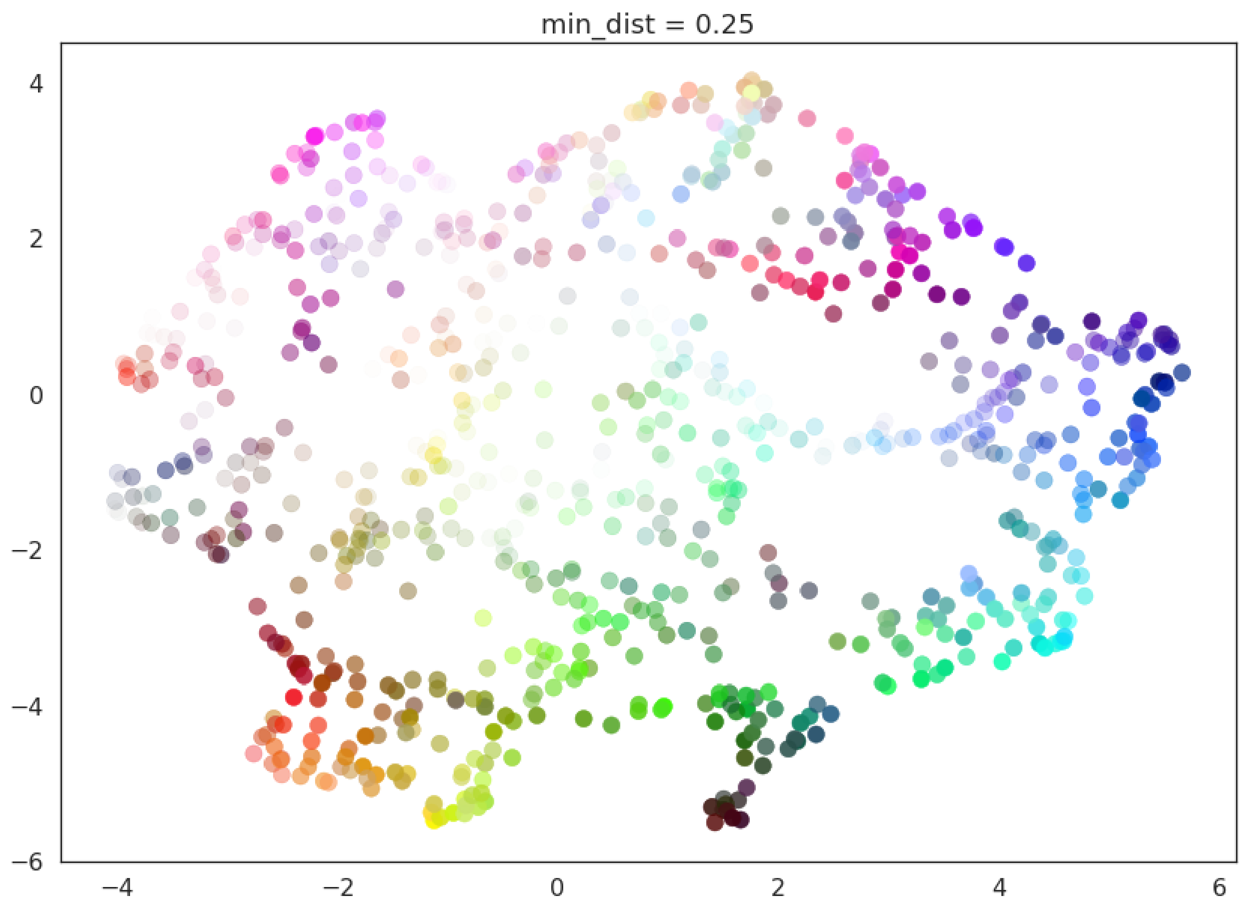
The `min_dist` parameter controls how tightly UMAP is allowed to pack points together. It, quite literally, provides the minimum distance apart that points are allowed to be in the low dimensional representation. This means that low values of `min_dist` will result in clumpier embeddings. This can be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

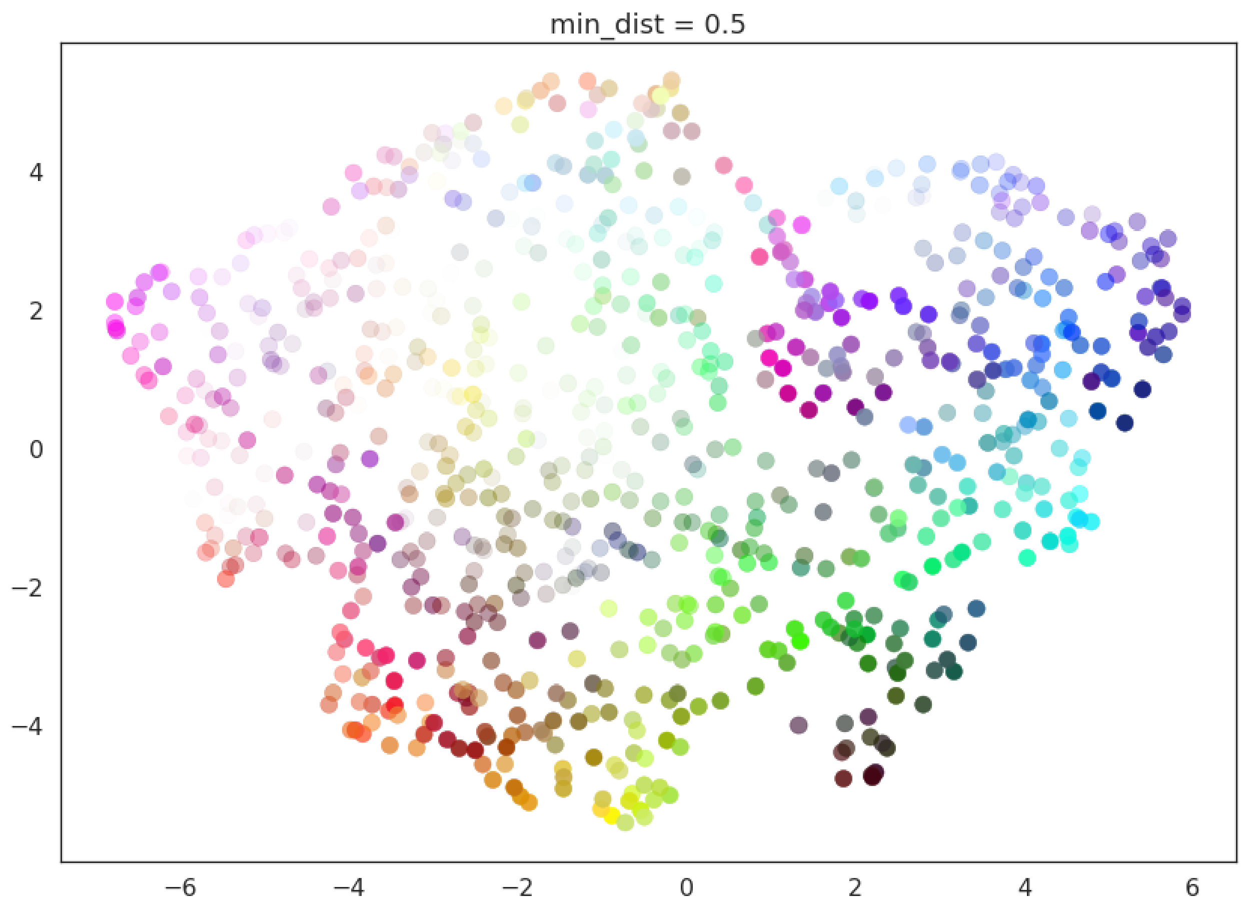
The default value for `min_dist` (as used above) is 0.1. We will look at a range of values from 0.0 through to 0.99.

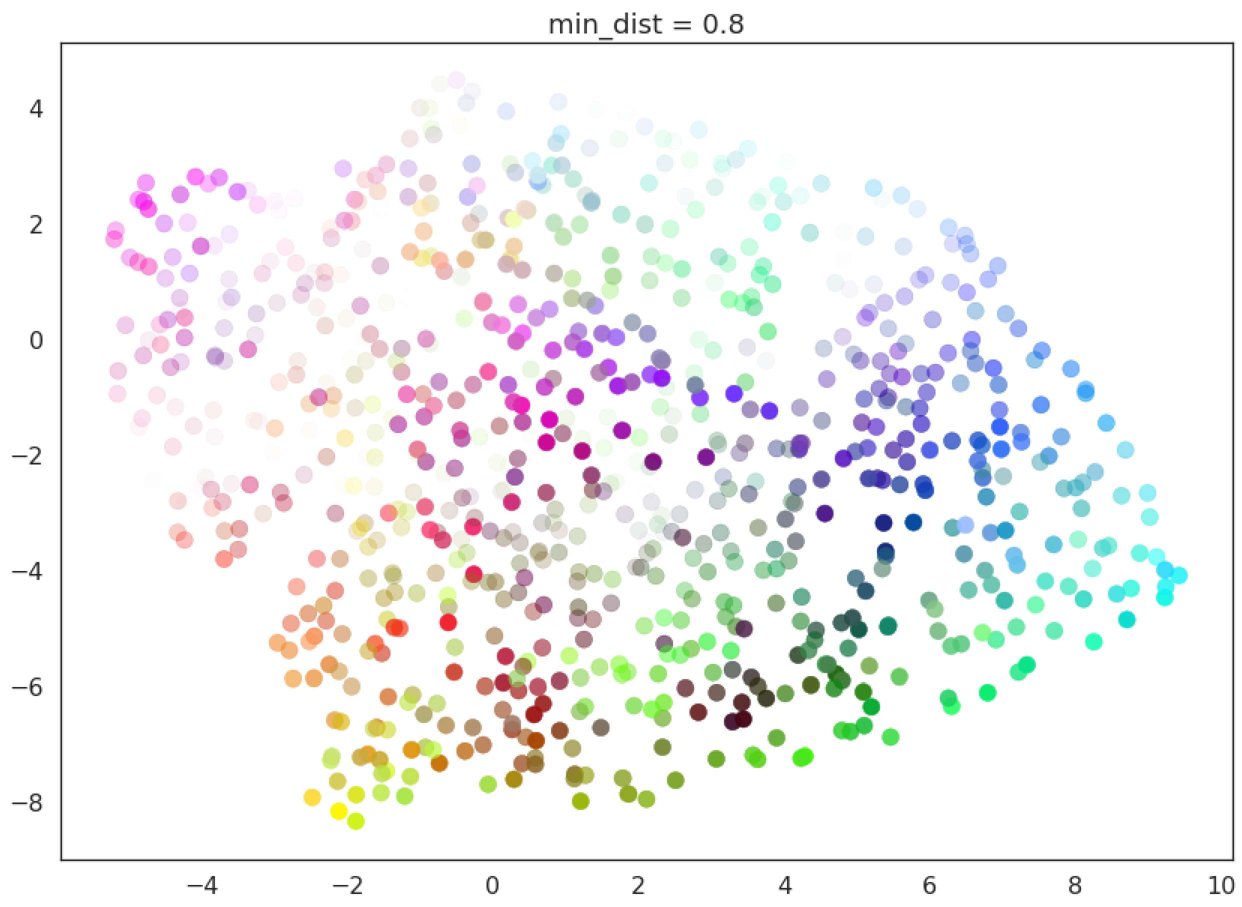
```
for d in (0.0, 0.1, 0.25, 0.5, 0.8, 0.99):
    draw_umap(min_dist=d, title='min_dist = {}'.format(d))
```

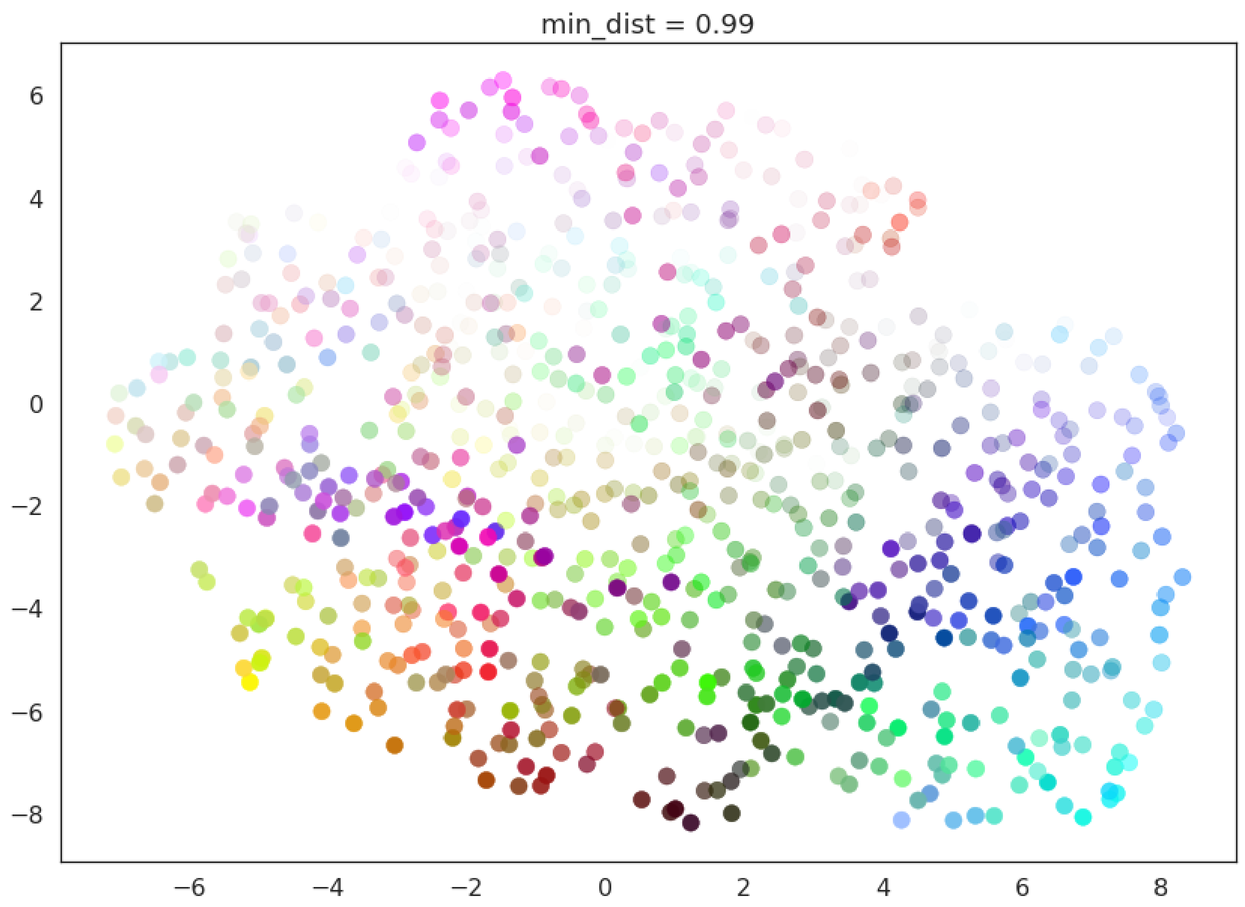



min_dist = 0.1
4
2
0
−2
−4
−4
−2
0
2
4


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.

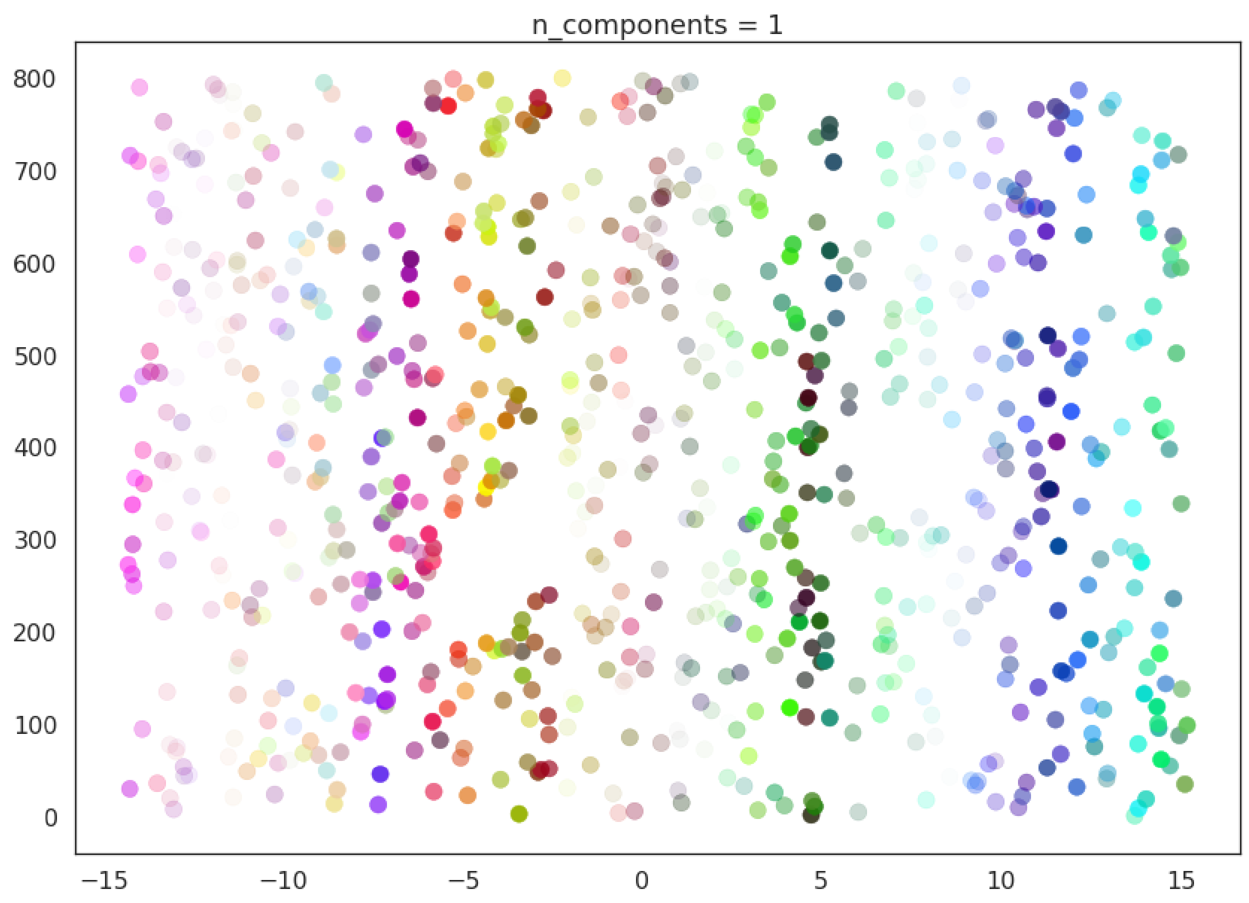
2.3 `n_components`

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 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.

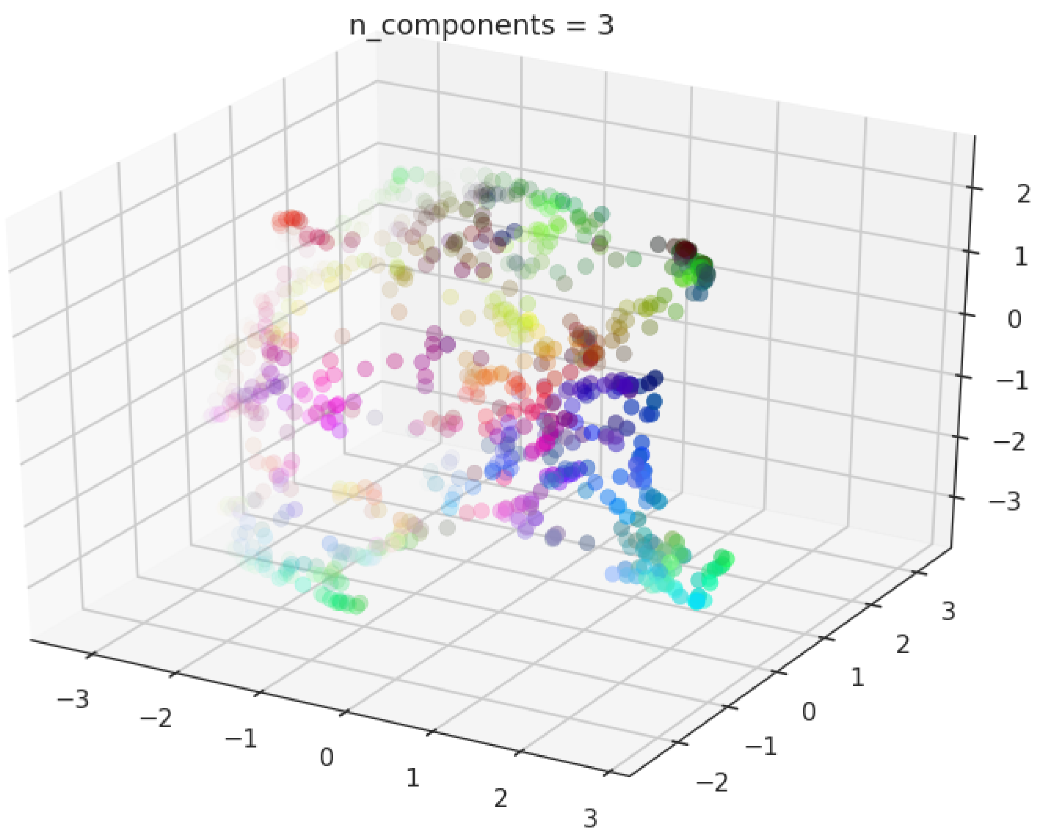
```
draw_umap(n_components=1, title='n_components = 1')
```



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` at 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
- russelrao
- 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)

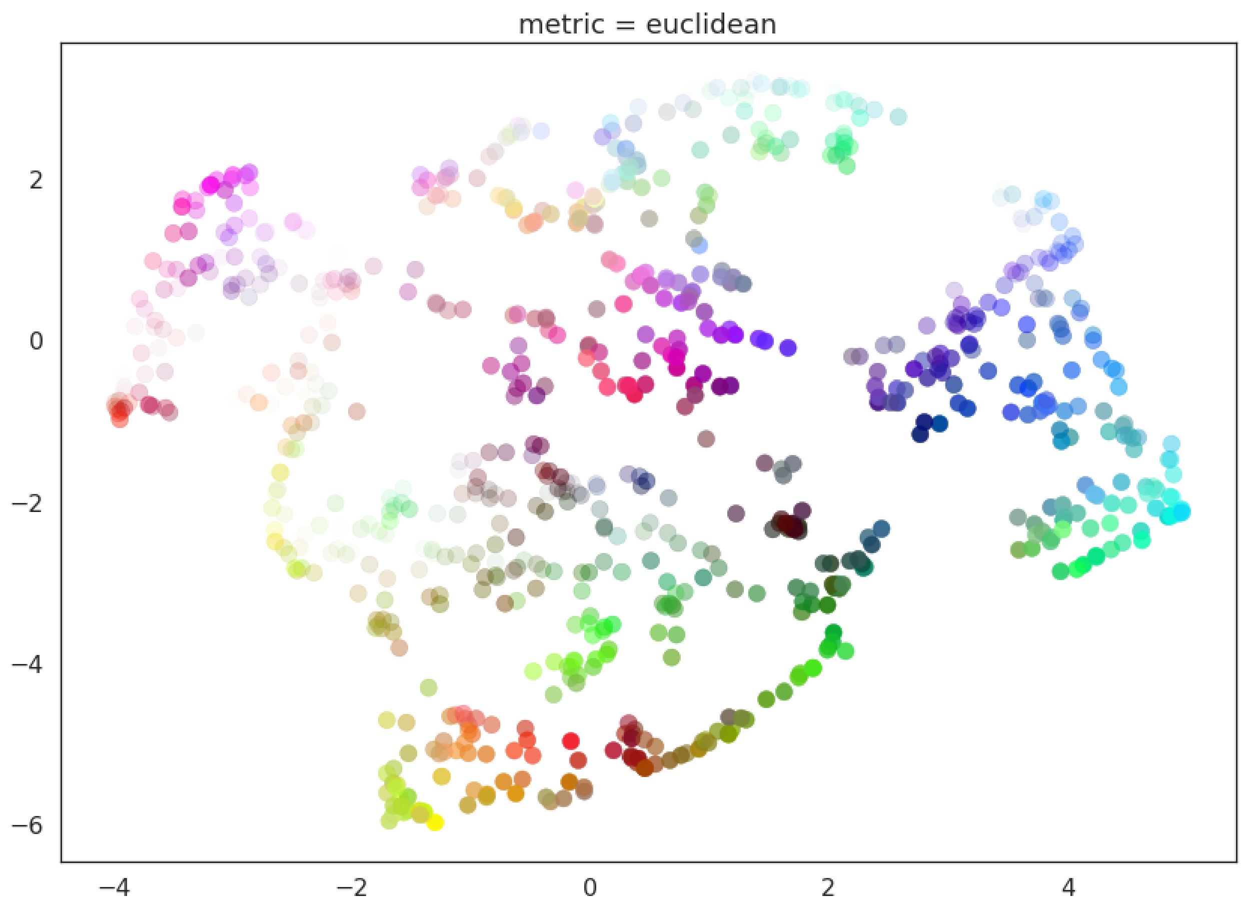
```
        return ((r - g) / delta) + 4

@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))
```

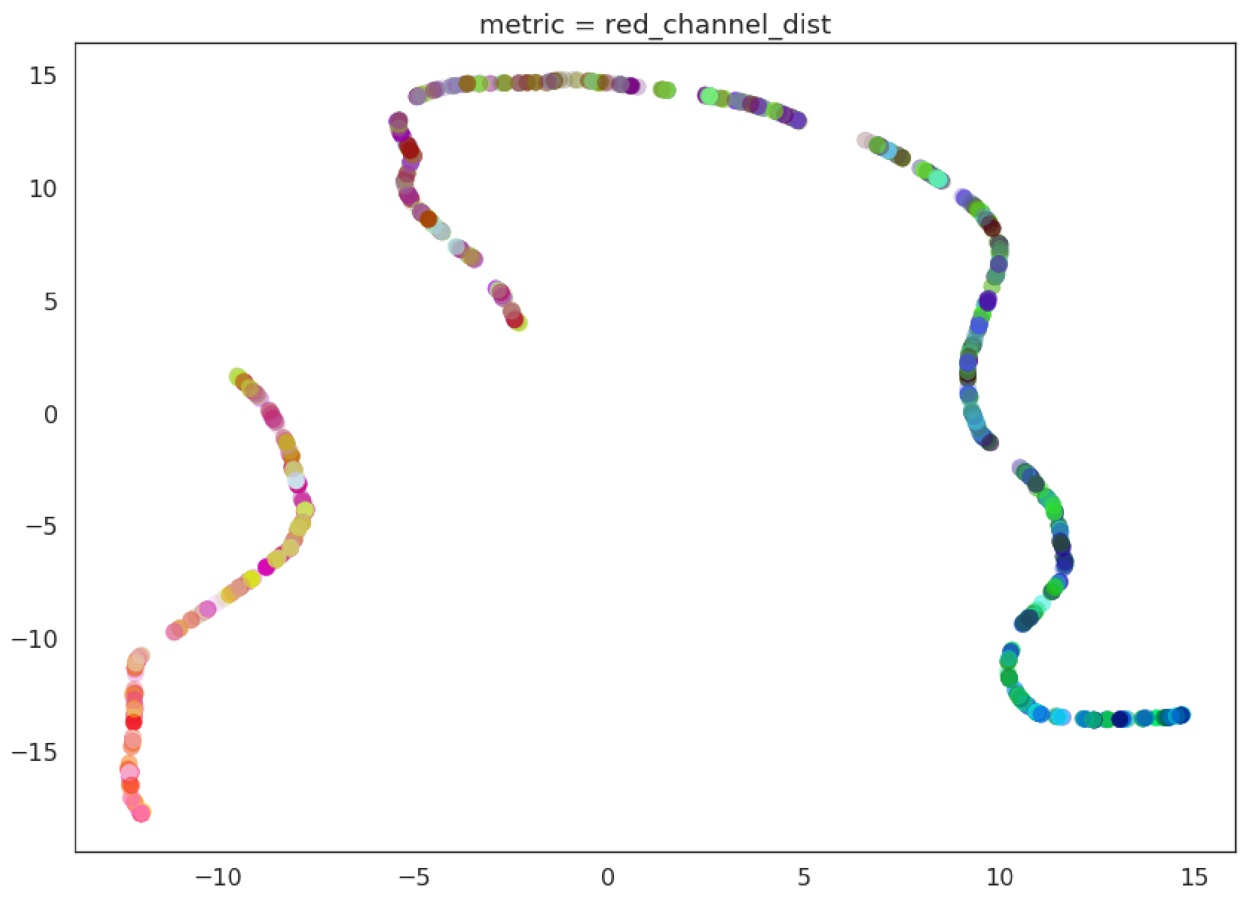
With that in hand we can define three extra distances. The first simply measures the difference in hue, the second measures the euclidean distance in a combined saturation and lightness space, while the third measures distance in the full HSL space.

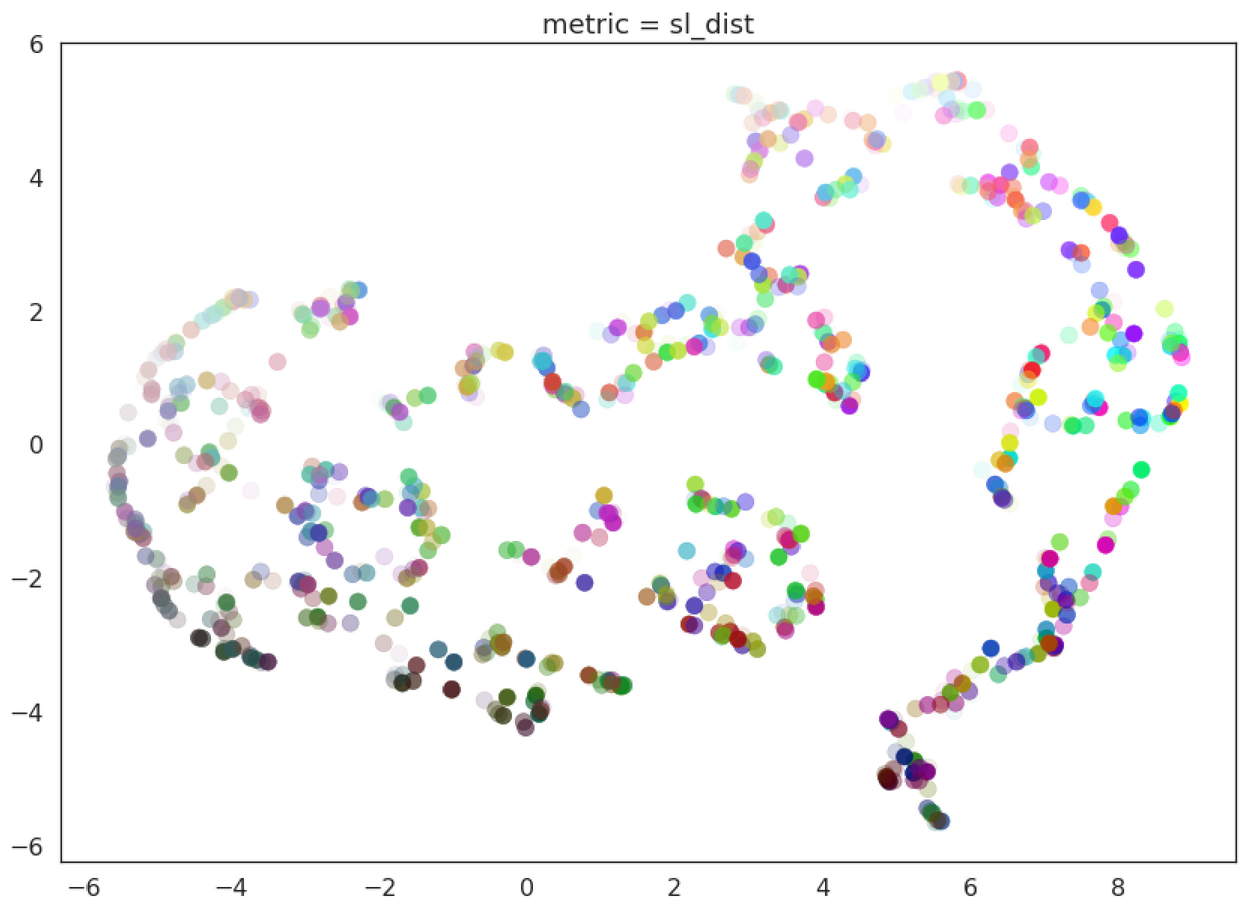
```
@numba.njit()
def hue_dist(a, b):
    diff = (hue(a[0], a[1], a[2]) - hue(b[0], b[1], b[2])) % 6
    if diff < 0:
        return diff + 6
    else:
        return diff

@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)
```

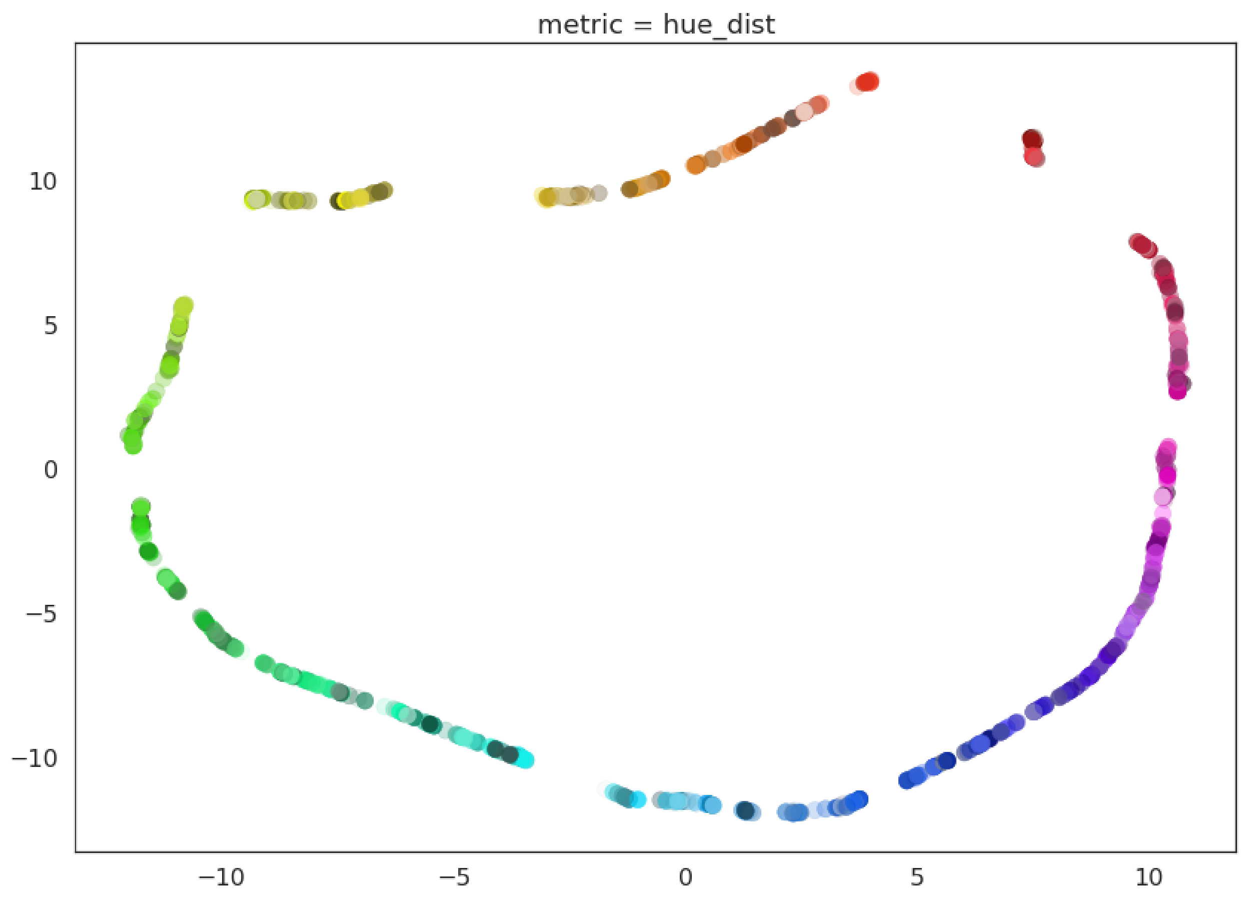
With such custom metrics in hand we can get UMAP to embed the data using those metrics to measure distance between our input data points. Note that `numba` provides significant flexibility in what we can do in defining distance functions. Despite this we retain the high performance we expect from UMAP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

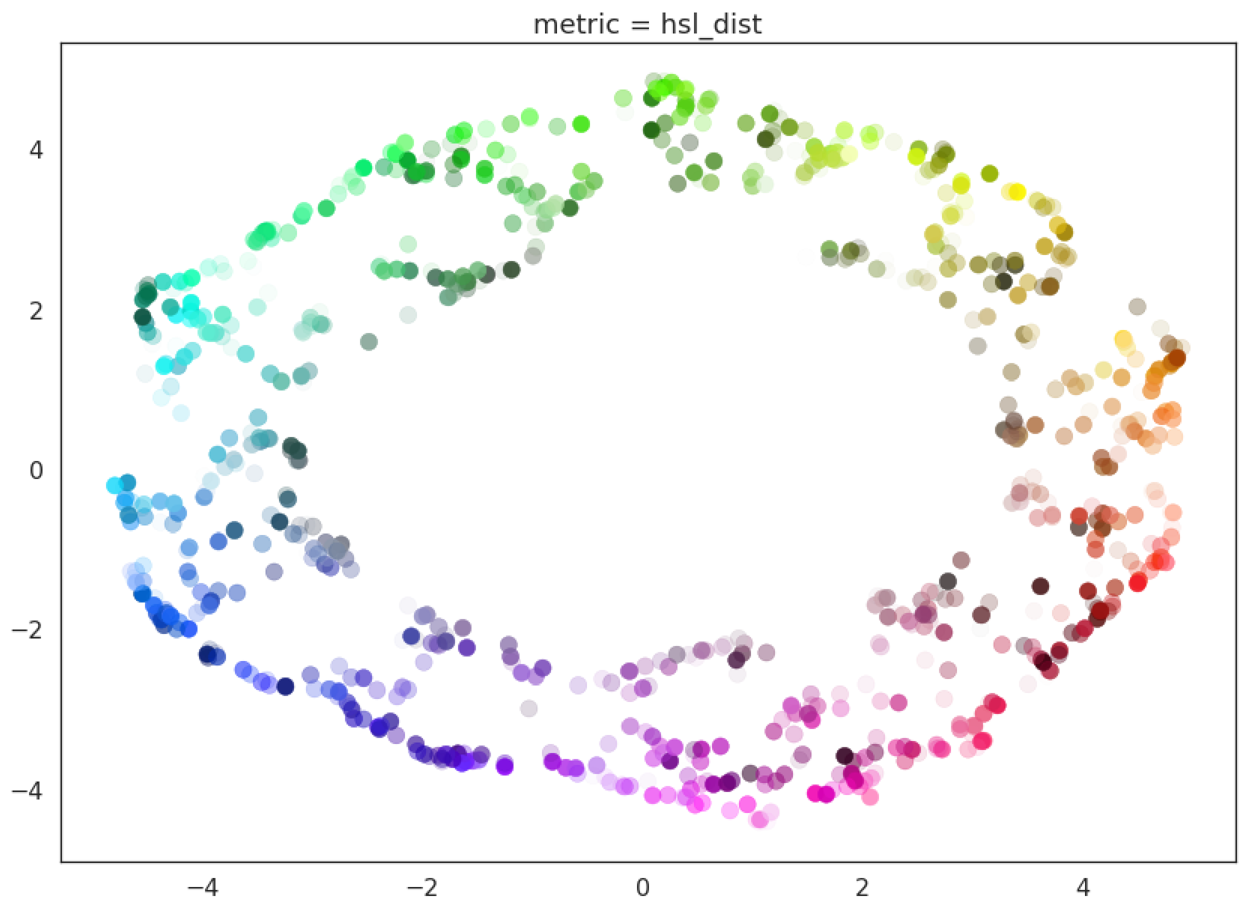
```
for m in ("euclidean", red_channel_dist, sl_dist, hue_dist, hsl_dist):
    name = m if type(m) is str else m.__name__
    draw_umap(n_components=2, metric=m, title='metric = {}'.format(name))
```


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 see 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

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 lets 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 tets 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 mode) 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.

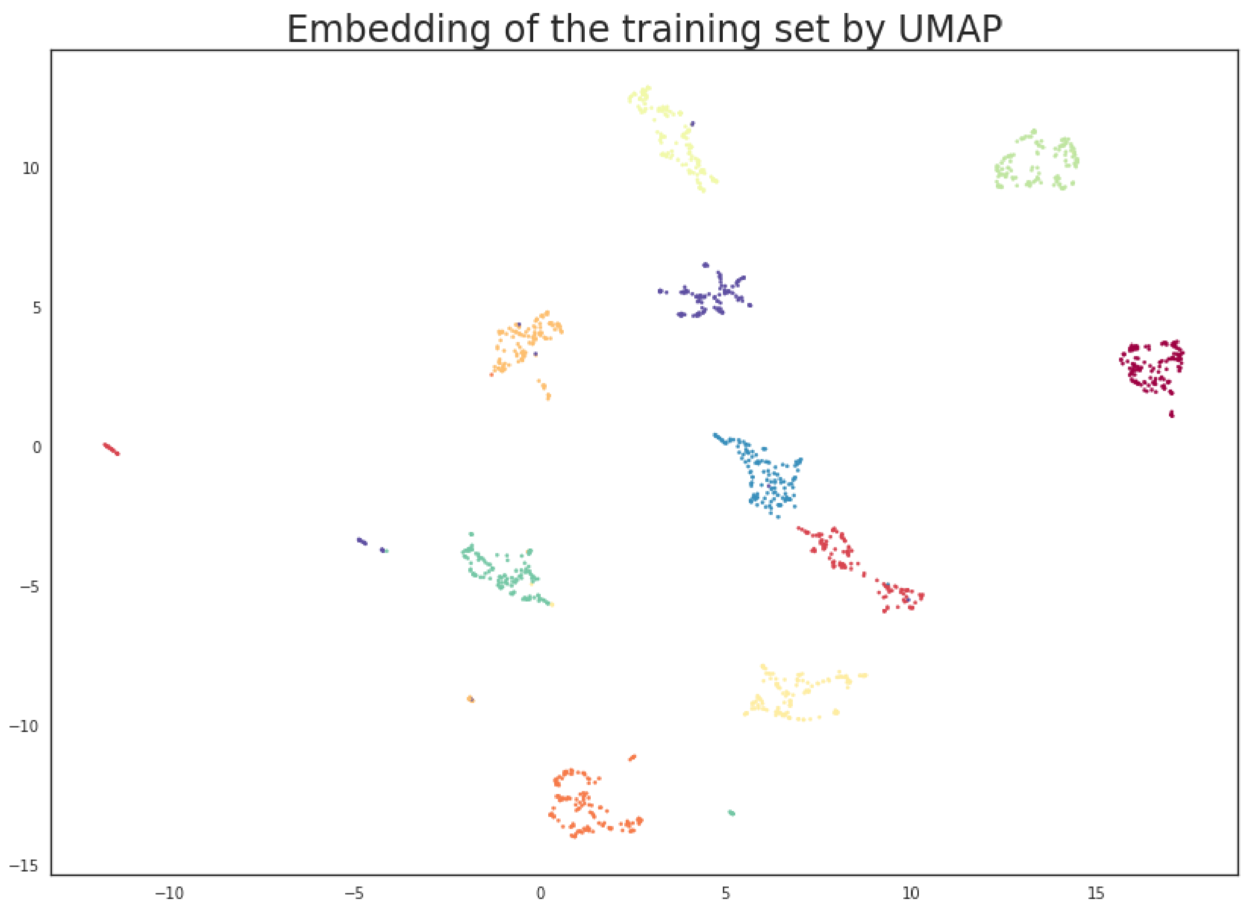
```
import umap
```

To make use of UMAP as a data transformer we first need to fir 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.

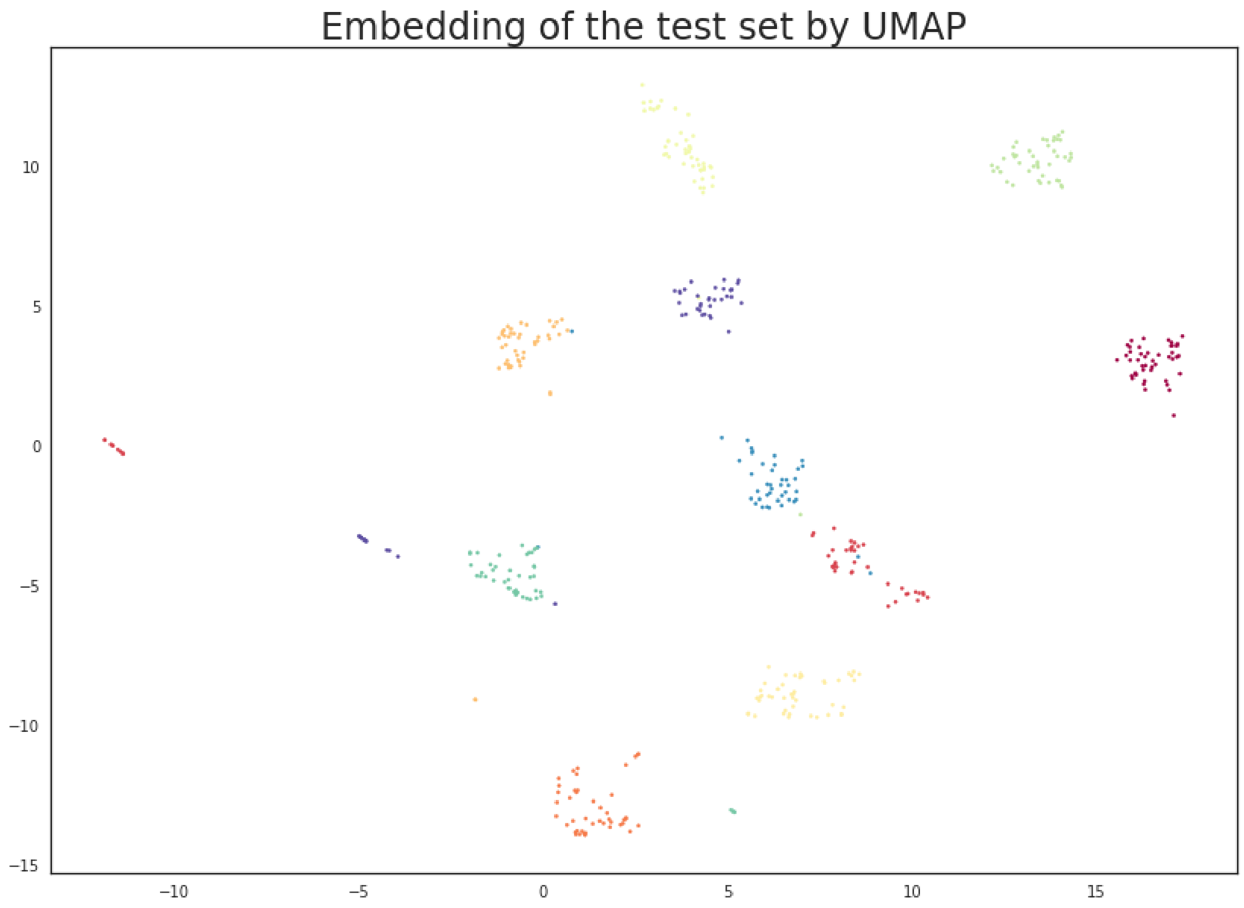
```
%time test_embedding = trans.transform(X_test)
```

```
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 of 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 had 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 may 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 4

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 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 desgined 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 do dimension reduction on this dataset.

```
import umap
```

4.1 UMAP on Fashion MNIST

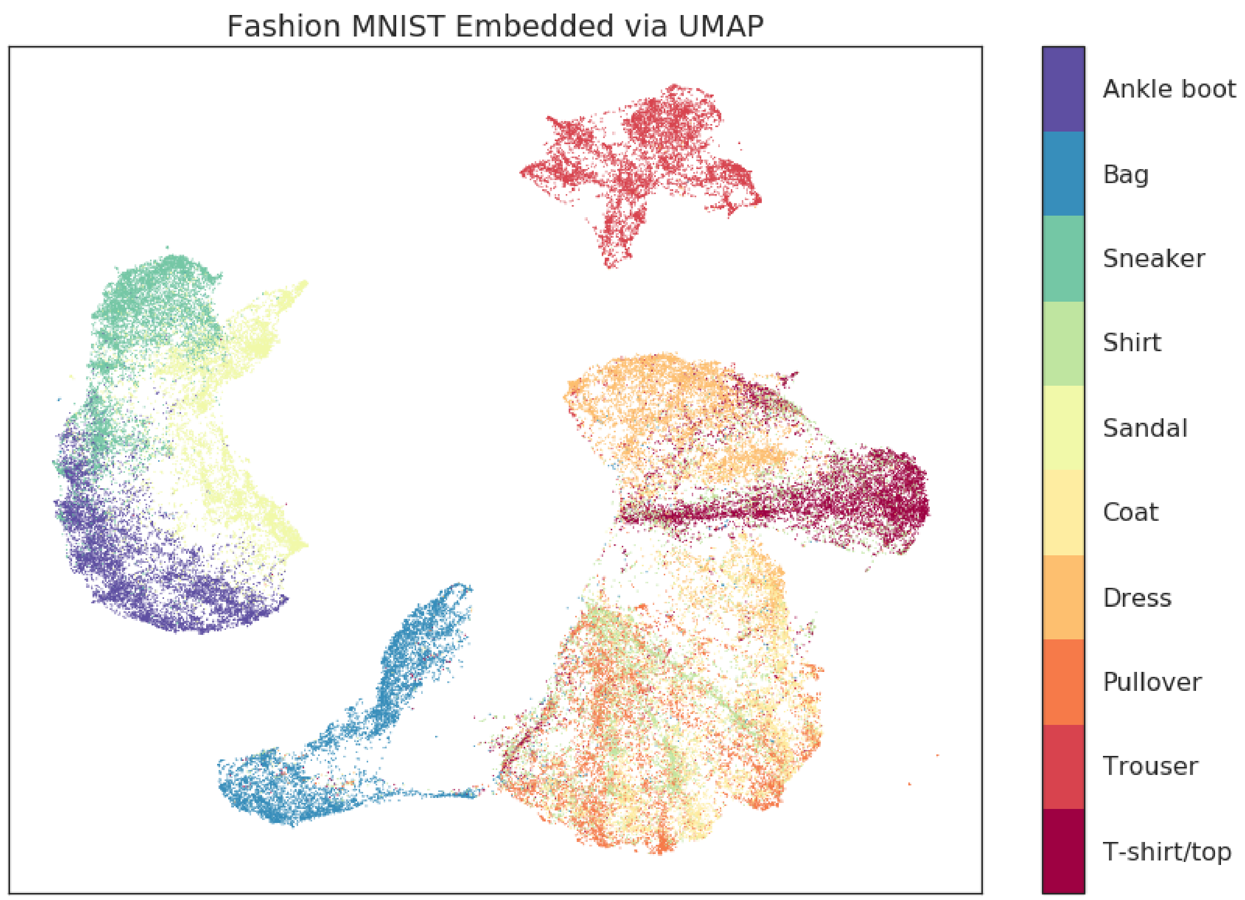
First we'll just do standard unsupervised dimension reduction using UMAP so we have a baseline of what the results look like for later comparison. This is simply a matter of instantiating a *`UMAP`* object (in this case setting the `n_neighbors` parameter to be 5 – we are interested mostly in very local 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 matplotlibs 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 unsruprisingly 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!

4.2 Using Labels to Separate Classes (Supervised UMAP)

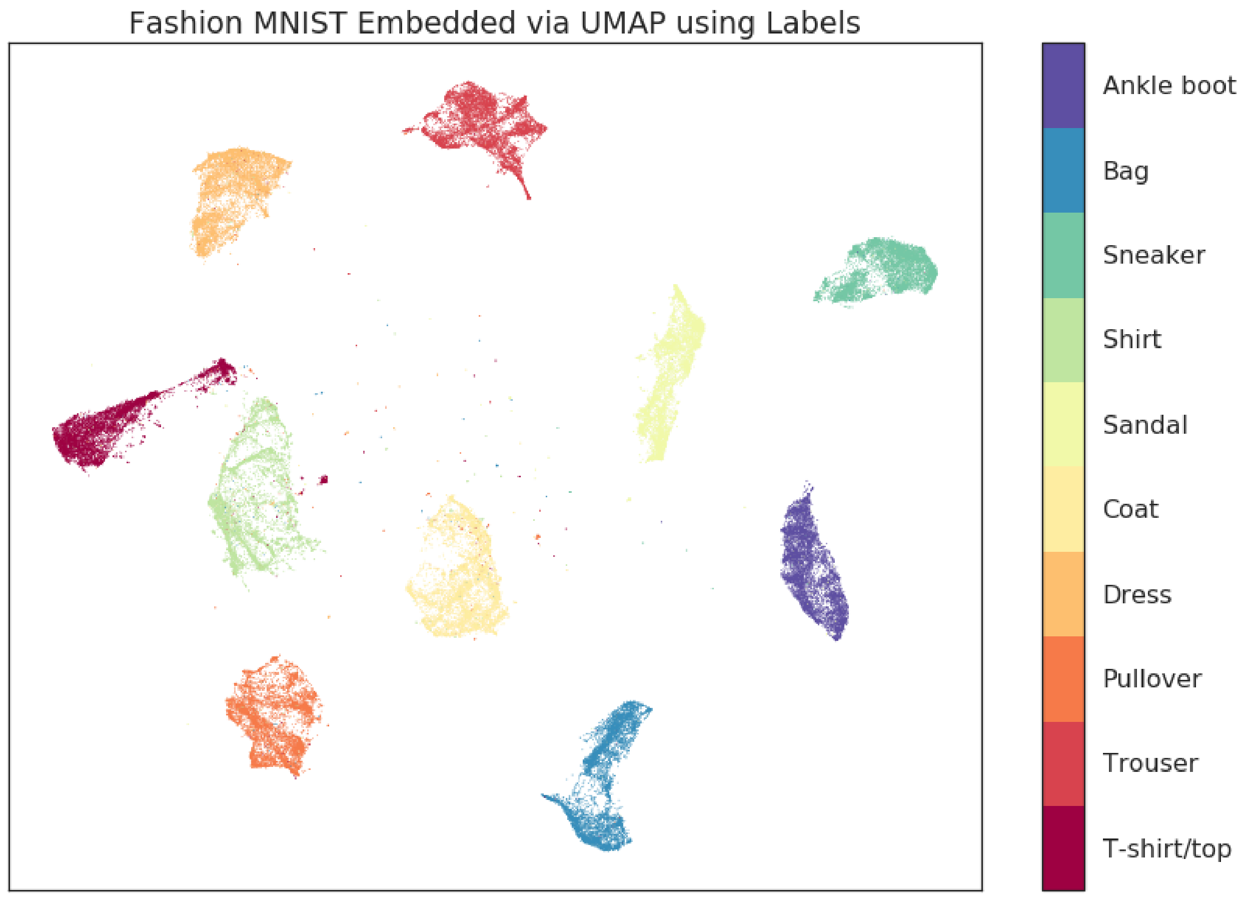
How do we go about coercing UMAP to make use of target labels? If you are familiar with the sklearn API you'll know that the *fit()* method takes a target parameter `y` that specifies supervised target information (for example when training a supervised classification model). We can simply pass the *UMAP* 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 seprated 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 seprate them while still having a meaningful embedding of individual points then supervised UMAP can provide exactly what you need.

4.3 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 to. To set up the example we'll mask some of the target information – we'll do this by using the sklearn standard of having unlabelled point be given the 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
```

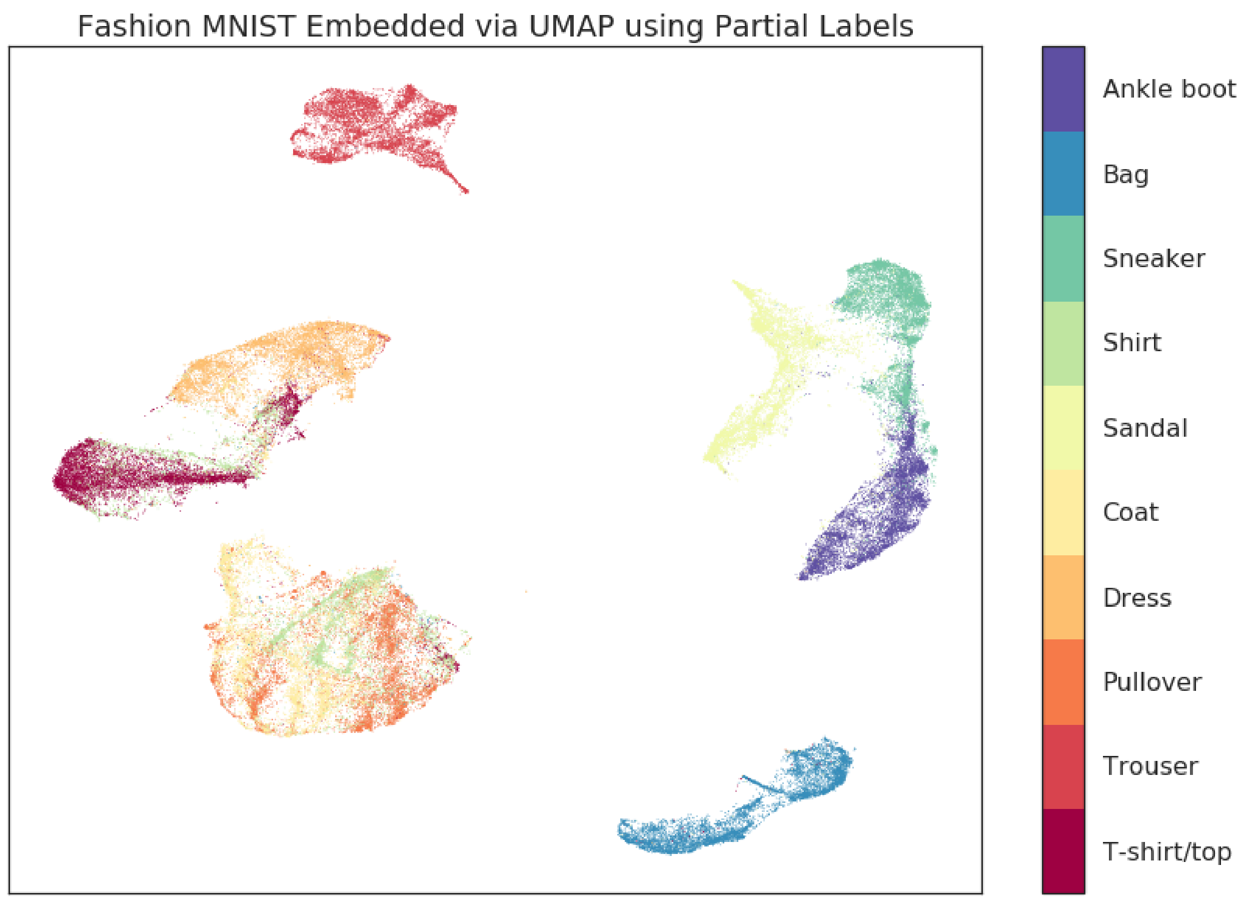
Now that we have randomly masked some of the labels we can try to perform supervised learning again. Everything works as before, but UMAP will interpret the -1 label as beingan unlabelled point and learn accordingly.

```
%%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.

4.4 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
```

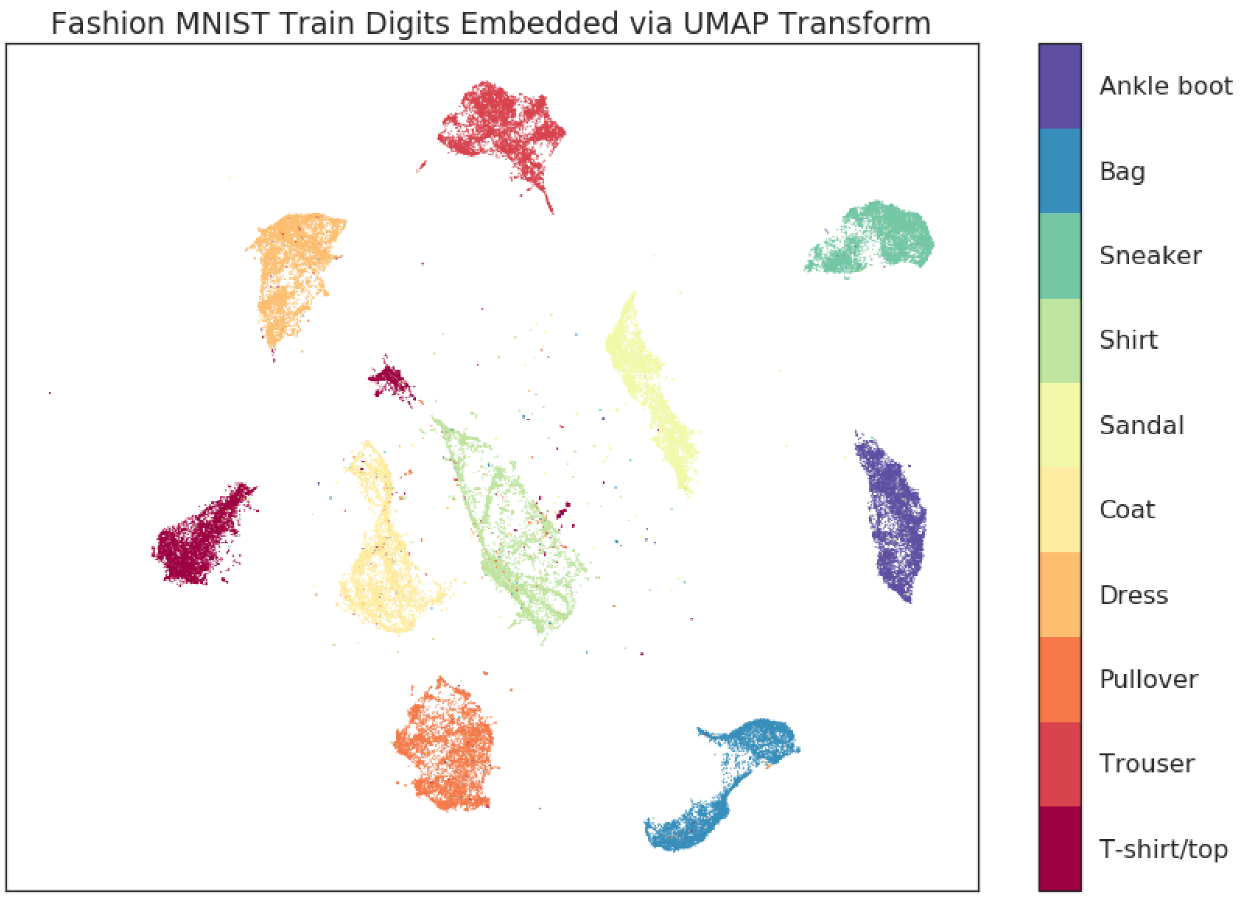
Next we can use the `transform()` method on that model to transform the test set into the learned space. This time we won't pass the label information and let the model attempt to place the data correctly.

```
%%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
```

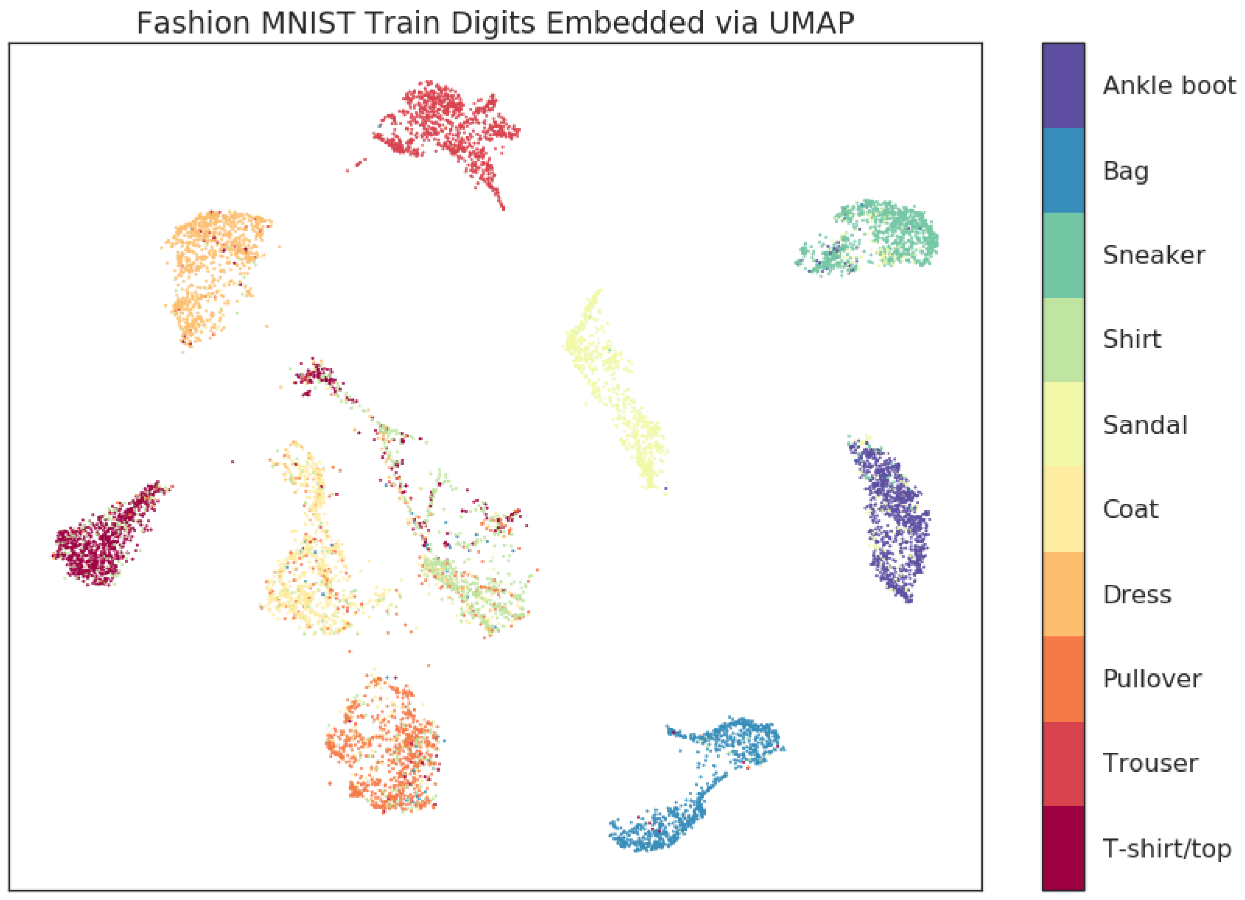
UMAP transforms are not as fast as some approaches, but as you can see this was still fairly efficient. The important question is how well we managed to embed the test data into the existing learned space. To start let's visualise the embedding of the training data so we can get a sense of where things *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 *:meth:'~umap.umap_.UMAP.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 in some t-shirts that ended up in mixed with the shirts, and some pullovers which are confused with the coats. Given the difficulty of the problemn this is a good result, particularly when compared with current state-of-the-art approaches such as siamese and triplet networks.

CHAPTER 5

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 tears in clusters that are not actually present, 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_mldata` 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.

```python
from sklearn.datasets import fetch_mldata
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
```

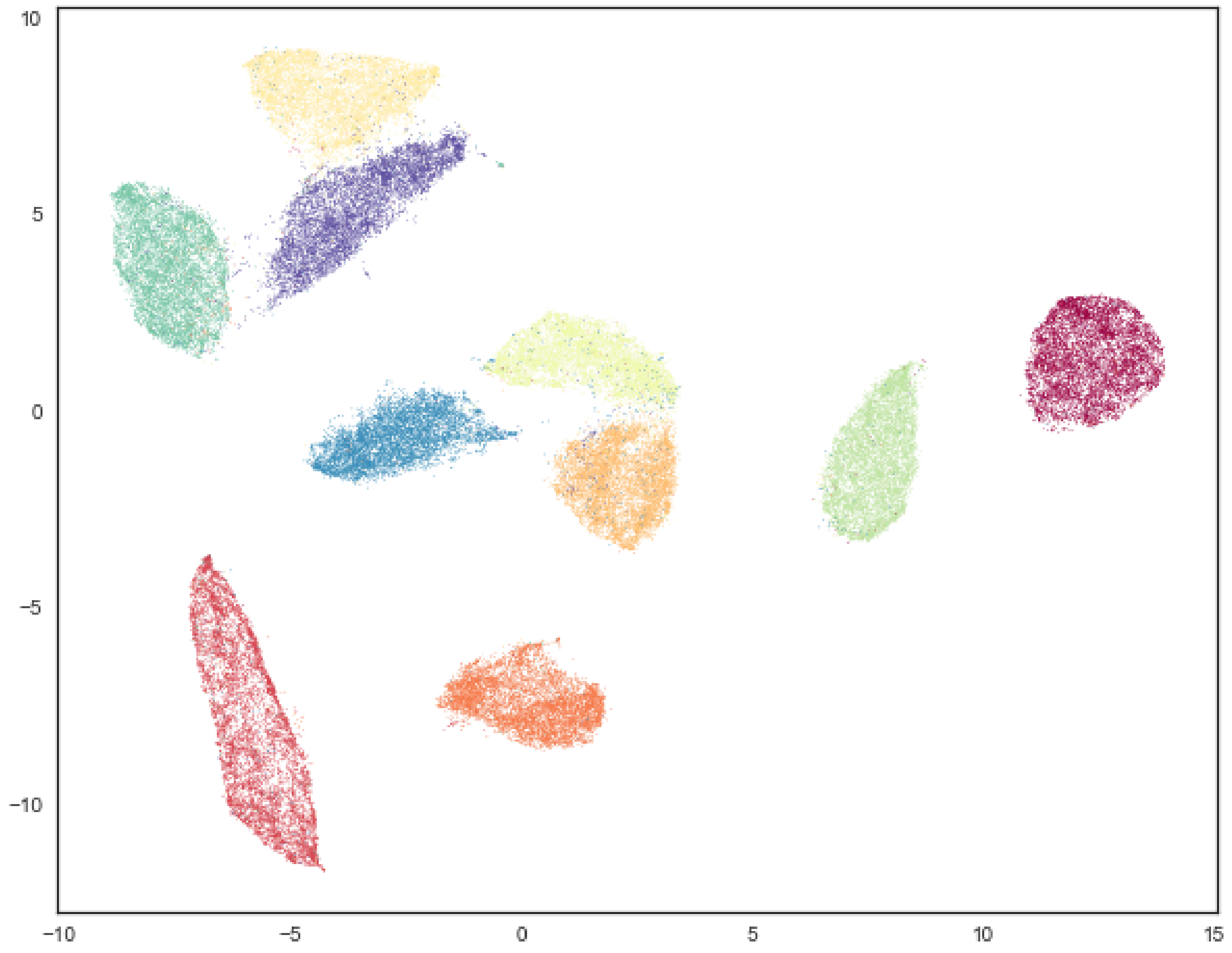
Now let's set up the plotting and grab the data we'll be using – in this case the MNIST handwritten digits dataset. MNIST consists of 28x28 pixel grayscale images of handwritten digits (0 through 9). These can be unraveled such that each digit is described by a 784 dimensional vector (the gray scale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

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 a 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, s=0.1,
↪ cmap='Spectral');
```



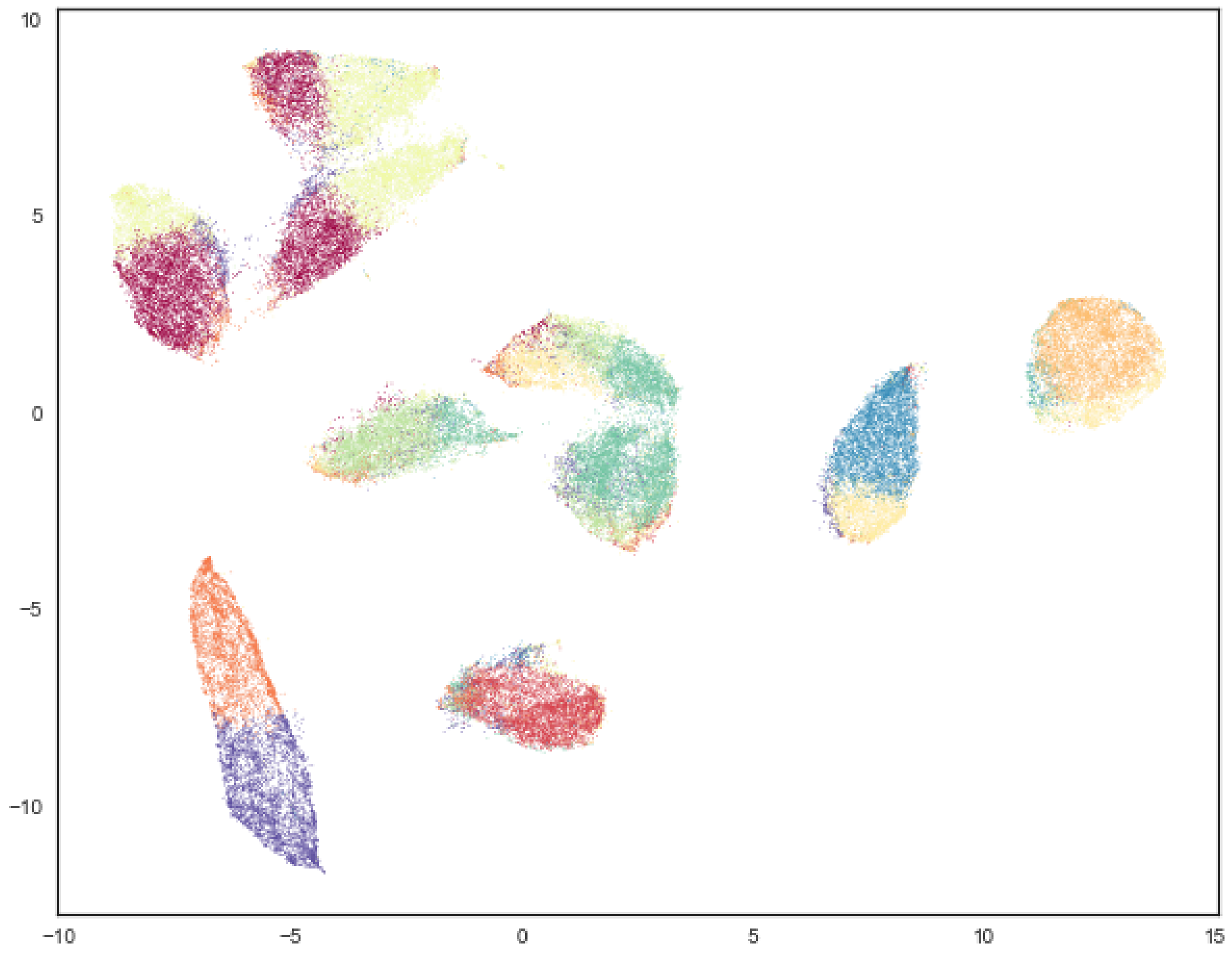
5.1 Traditional clustering

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 – the two clusters are the far right 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)
)
```

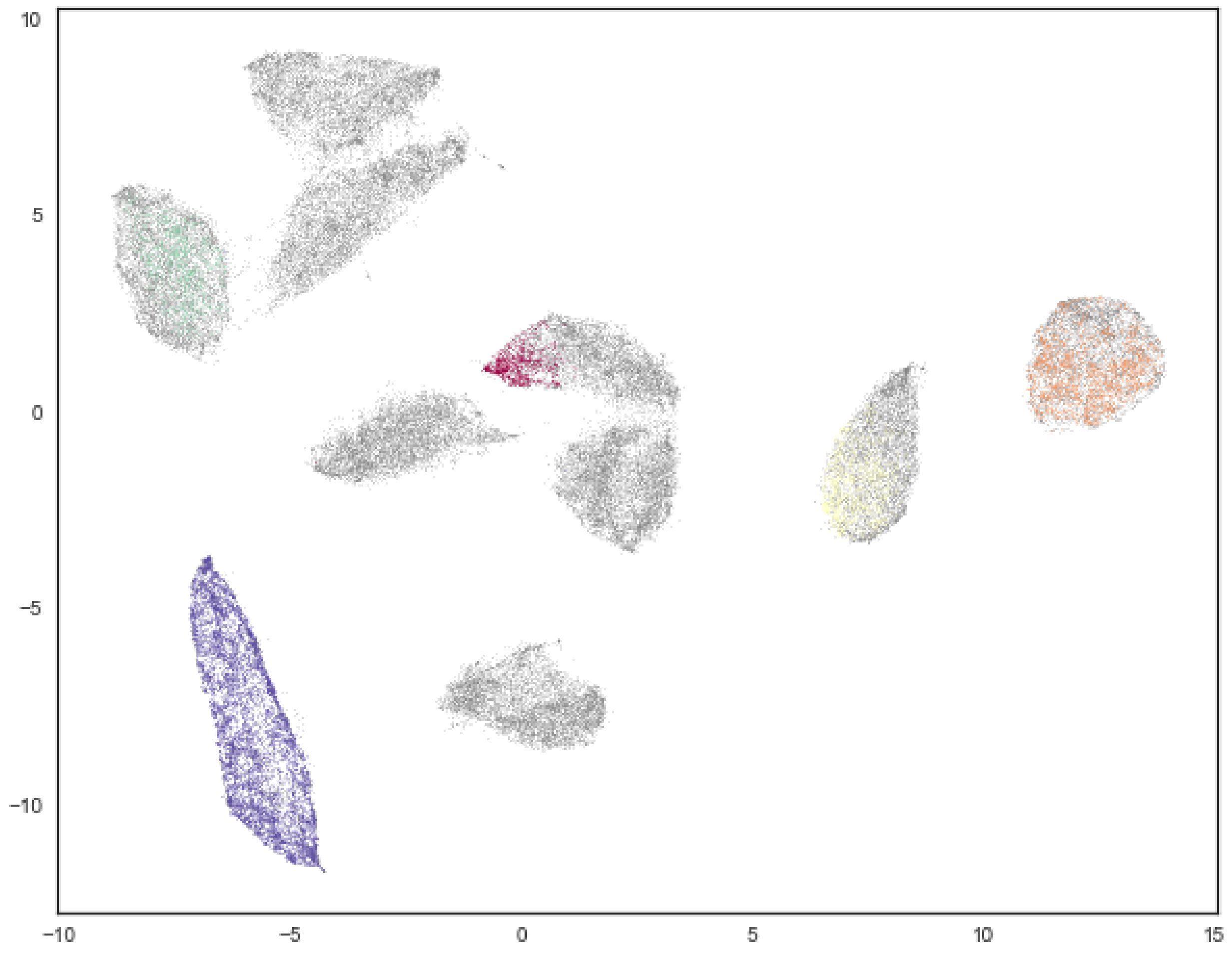
```
(0.366752951359725552, 0.49614118437750965)
```

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 tehcniques. 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 the as "noise". To visualize this aspect we will colorpoints 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 the 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.

5.2 UMAP enhanced clustering

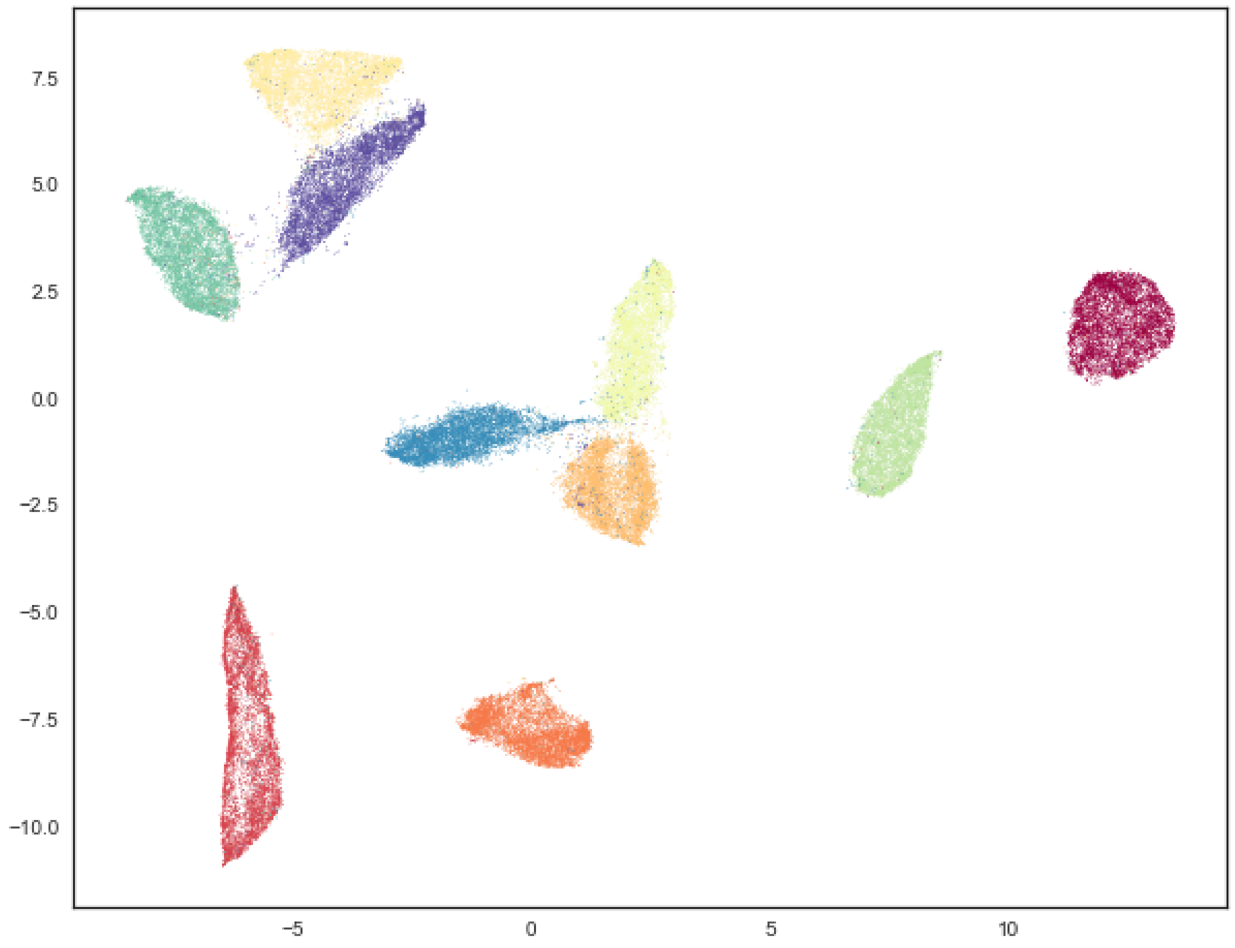
Our goal is to make use of UMAP to perform non-linear manifold aware dimension reduction so we can get the dataset down to a number of dimensions small enough for a density based clustering algorithm to make progress. One advantage of UMAP for this is that it doesn't require you to reduce to only two 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,
```

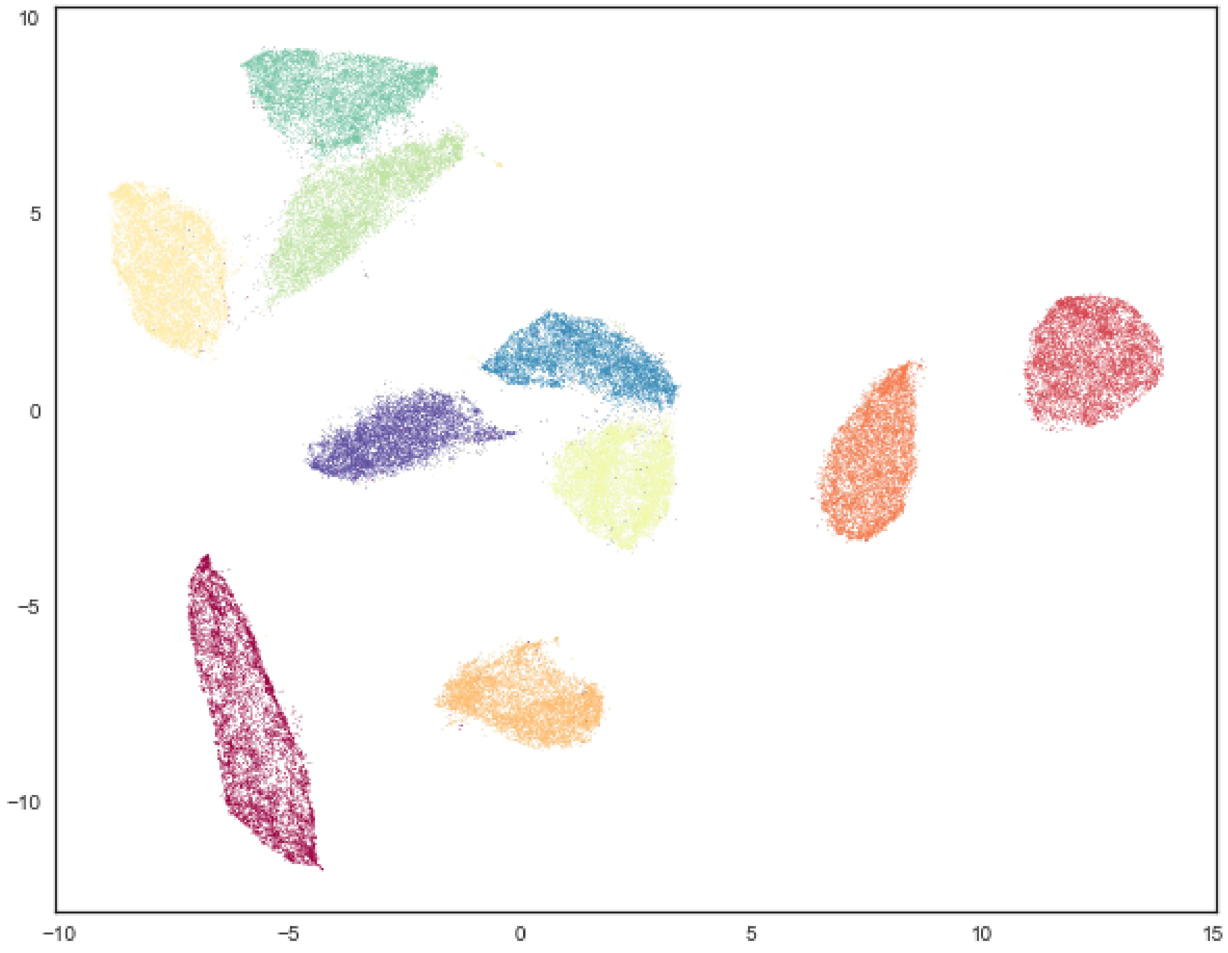
(continues on next page)

(continued from previous page)

```
    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 have 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 score 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 thing care must be taken, but clearly UMAP can provide significantly better clustering results when used judiciously.

CHAPTER 6

Gallery of Examples of UMAP usage

A small gallery collection examples of UMAP usage. Do you have an interesting UMAP plot that uses publicly available data? Submit a pull request to have it added as an example!

Note: Click *here* to download the full example code

6.1 UMAP on the MNIST Digits dataset

A simple example demonstrating how to use UMAP on a larger dataset such as MNIST. We first pull the MNIST dataset and then use UMAP to reduce it to only 2-dimensions for easy visualisation.

Note that UMAP manages to both group the individual digit classes, but also to retain the overall global structure among the different digit classes – keeping 1 far from 0, and grouping triplets of 3,5,8 and 4,7,9 which can blend into one another in some cases.

```
import umap
from sklearn.datasets import fetch_mldata
import matplotlib.pyplot as plt
import seaborn as sns

sns.set(context="paper", style="white")

mnist = fetch_mldata("MNIST original")

reducer = umap.UMAP(random_state=42)
embedding = reducer.fit_transform(mnist.data)

fig, ax = plt.subplots(figsize=(12, 10))
plt.scatter(
    embedding[:, 0], embedding[:, 1], c=mnist.target, cmap="Spectral", s=0.1
)
```

(continues on next page)

(continued from previous page)

```
plt.setp(ax, xticks=[], yticks=[])
plt.title("MNIST data embedded into two dimensions by UMAP", fontsize=18)

plt.show()
```

Total running time of the script: (0 minutes 0.000 seconds)

Note: Click *here* to download the full example code

6.2 UMAP on the Fashion MNIST Digits dataset using Datashader

This is a simple example of using UMAP on the Fashion-MNIST dataset. The goal of this example is largely to demonstrate the use of datashader as an effective tool for visualising UMAP results. In particular datashader allows visualisation of very large datasets where overplotting can be a serious problem. It supports coloring by categorical variables (as shown in this example), or by continuous variables, or by density (as is common in datashader examples).

```
import umap
import numpy as np
import pandas as pd
import requests
import os
import datashader as ds
import datashader.utils as utils
import datashader.transfer_functions as tf
import matplotlib.pyplot as plt
import seaborn as sns

sns.set(context="paper", style="white")

if not os.path.isfile('fashion-mnist.csv'):
    csv_data = requests.get(
        'https://www.openml.org/data/get_csv/18238735/phpnBqZGZ'
    )
    with open('fashion-mnist.csv', 'w') as f:
        f.write(csv_data.text)
source_df = pd.read_csv('fashion-mnist.csv')

data = source_df.iloc[:, :784].values.astype(np.float32)
target = source_df['class'].values

pal = [
 '#9e0142',
 '#d8434e',
 '#f67a49',
 '#fdbf6f',
 '#feeda1',
 '#f1f9a9',
 '#bfe5a0',
 '#74c7a5',
 '#378ebb',
 '#5e4fa2'
]
```

(continues on next page)

(continued from previous page)

```
color_key = {str(d):c for d,c in enumerate(pal)}

reducer = umap.UMAP(random_state=42)
embedding = reducer.fit_transform(data)

df = pd.DataFrame(embedding, columns=('x', 'y'))
df['class'] = pd.Series([str(x) for x in target], dtype="category")

cvs = ds.Canvas(plot_width=400, plot_height=400)
agg = cvs.points(df, 'x', 'y', ds.count_cat('class'))
img = tf.shade(agg, color_key=color_key, how='eq_hist')

utils.export_image(img, filename='fashion-mnist', background='black')

image = plt.imread('fashion-mnist.png')
fig, ax = plt.subplots(figsize=(6, 6))
plt.imshow(image)
plt.setp(ax, xticks=[], yticks=[])
plt.title("Fashion MNIST data embedded\n"
          "into two dimensions by UMAP\n"
          "visualised with Datashader",
          fontsize=12)

plt.show()
```

Total running time of the script: (0 minutes 0.000 seconds)

Note: Click *here* to download the full example code

6.3 Comparison of Dimension Reduction Techniques

A comparison of several different dimension reduction techniques on a variety of toy datasets. The datasets are all toy datasets, but should provide a representative range of the strengths and weaknesses of the different algorithms.

The time to perform the dimension reduction with each algorithm and each dataset is recorded in the lower right of each plot.

Things to note about the datasets:

- **Blobs: A set of five gaussian blobs in 10 dimensional** space. This should be a prototypical example of something that should clearly separate even in a reduced dimension space.
- **Iris: a classic small dataset with one distinct class** and two classes that are not clearly separated.
- **Digits: handwritten digits – ideally different digit** classes should form distinct groups. Due to the nature of handwriting digits may have several forms (crossed or uncrossed sevens, capped or straight line oes, etc.)
- **Wine: wine characteristics ideally used for a toy** regression. Ultimately the data is essentially one dimensional in nature.
- **Swiss Roll: data is essentially a rectangle, but** has been "rolled up" like a swiss roll in three dimensional space. Ideally a dimension reduction technique should be able to "unroll" it. The data has been coloured according to one dimension of the rectangle, so should form a rectangle of smooth color variation.

- **Sphere: the two dimensional surface of a three** dimensional sphere. This cannot be represented accurately in two dimensions without tearing. The sphere has been coloured with hue around the equator and black to white from the south to north pole.

```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import time

from sklearn import datasets, decomposition, manifold, preprocessing
from colorsys import hsv_to_rgb

import umap

sns.set(context="paper", style="white")

blobs, blob_labels = datasets.make_blobs(
    n_samples=500, n_features=10, centers=5, random_state=42
)
iris = datasets.load_iris()
digits = datasets.load_digits(n_class=10)
wine = datasets.load_wine()
swissroll, swissroll_labels = datasets.make_swiss_roll(
    n_samples=1000, noise=0.1, random_state=42
)
sphere = np.random.normal(size=(600, 3))
sphere = preprocessing.normalize(sphere)
sphere_hsv = np.array(
    [
        (
            (np.arctan2(c[1], c[0]) + np.pi) / (2 * np.pi),
            np.abs(c[2]),
            min((c[2] + 1.1), 1.0),
        )
        for c in sphere
    ]
)
sphere_colors = np.array([hsv_to_rgb(*c) for c in sphere_hsv])

reducers = [
    (manifold.TSNE, {"perplexity": 50}),
    # (manifold.LocallyLinearEmbedding, {'n_neighbors':10, 'method':'hessian'}),
    (manifold.Isomap, {"n_neighbors": 30}),
    (manifold.MDS, {}),
    (decomposition.PCA, {}),
    (umap.UMAP, {"n_neighbors": 30, "min_dist": 0.3}),
]

test_data = [
    (blobs, blob_labels),
    (iris.data, iris.target),
    (digits.data, digits.target),
    (wine.data, wine.target),
    (swissroll, swissroll_labels),
    (sphere, sphere_colors),
]
dataset_names = ["Blobs", "Iris", "Digits", "Wine", "Swiss Roll", "Sphere"]
```

(continues on next page)

(continued from previous page)

```
n_rows = len(test_data)
n_cols = len(reducers)
ax_index = 1
ax_list = []

# plt.figure(figsize=(9 * 2 + 3, 12.5))
plt.figure(figsize=(10, 8))
plt.subplots_adjust(
    left=.02, right=.98, bottom=.001, top=.96, wspace=.05, hspace=.01
)
for data, labels in test_data:
    for reducer, args in reducers:
        start_time = time.time()
        embedding = reducer(n_components=2, **args).fit_transform(data)
        elapsed_time = time.time() - start_time
        ax = plt.subplot(n_rows, n_cols, ax_index)
        if isinstance(labels[0], tuple):
            ax.scatter(*embedding.T, s=10, c=labels, alpha=0.5)
        else:
            ax.scatter(
                *embedding.T, s=10, c=labels, cmap="Spectral", alpha=0.5
            )
        ax.text(
            0.99,
            0.01,
            "{:.2f} s".format(elapsed_time),
            transform=ax.transAxes,
            size=14,
            horizontalalignment="right",
        )
        ax_list.append(ax)
        ax_index += 1
plt.setp(ax_list, xticks=[], yticks=[])

for i in np.arange(n_rows) * n_cols:
    ax_list[i].set_ylabel(dataset_names[i // n_cols], size=16)
for i in range(n_cols):
    ax_list[i].set_xlabel(repr(reducers[i][0]()).split("(")[0], size=16)
    ax_list[i].xaxis.set_label_position("top")

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.000 seconds)

CHAPTER 7

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.

7.1 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 vales) 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

7.2 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.

7.3 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 llibrary 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.

7.4 Is there GPU or multicore-CPU support?

Not at this time. The bottlenecks in the code are the (approximate) nearest neighbor search and the optimization of the low dimensional representation. The first of these (ANN) is performed by a random projection forest and nearest-neighbor-descent. Both of those are, at the least, parellelisable in principle, and could be converted to support multicore (at the cost of single core performance). The optimization is performed via a (slightly custom) stochastic gradient descent. SGD is both parallelisable and amenable to GPUs. This means that in principle UMAP could support multicore and use GPUs for optimization. In practice this would involve GPU expertise and would potentially hurt single core performance, and so has been deferred for now. If you have expertise in GPU programming with Numba and would be interested in adding GPU support we would welcome your contributions.

7.5 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 posisble 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.

7.6 Is the 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 the we recommend umapr which wraps the python code with reticulate.

If you want a pure R version then there are a couple of options: UMAP and UWOT. Both are good reimplementations of UMAP in R. If performance is a major factor we recommend UWOT at this time.

7.7 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.

7.8 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.

CHAPTER 8

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.

8.1 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 to 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.

Fig. 1: Low dimensional simplices

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.

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 to 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 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 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 lies 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 is 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

Fig. 2: Test data set of a noisy sine wave

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

Fig. 3: A basic open cover of the test data

We can then depict the the simplicial complex of 0-, 1-, and 2-simplices as points, lines, and triangles

Fig. 4: A simplicial complex built from the test data

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 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 make 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).

8.2 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 with 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 covers “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:

_distributed_data _distributed_data

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 appears 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 or 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 *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 in 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? 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.

Fig. 8: Local connectivity and fuzzy open sets

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 on a new obstruction: out 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.

Fig. 9: Edges with incompatible weights

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 the merge together two disagreeing edges with weight *a* and *b* then we should have a single edge with combined weight $ a + b - a:raw-latex:‘times ‘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.

Fig. 10: Graph with combined edge weights

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 of the manifold underlying the data), how do we go about converting that into a low dimensional representation?

8.3 Finding a Low Dimensional Representation

Ideally we want the low dimensional representation to have as similar as possible 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 the 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 structure 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 terms 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 that relatively accurately represents the overall topology of the source data.

8.4 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 as possible a fuzzy topological representation 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 is 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 for $\frac{1}{1+ax^{2b}}$. 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 9

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.

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, which has significantly better performance than the current scikit-learn t-SNE.

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_mldata` to grab it for us.

Now it is time to start looking at performance. To start with let's look at how performance scales with increasing dataset size.

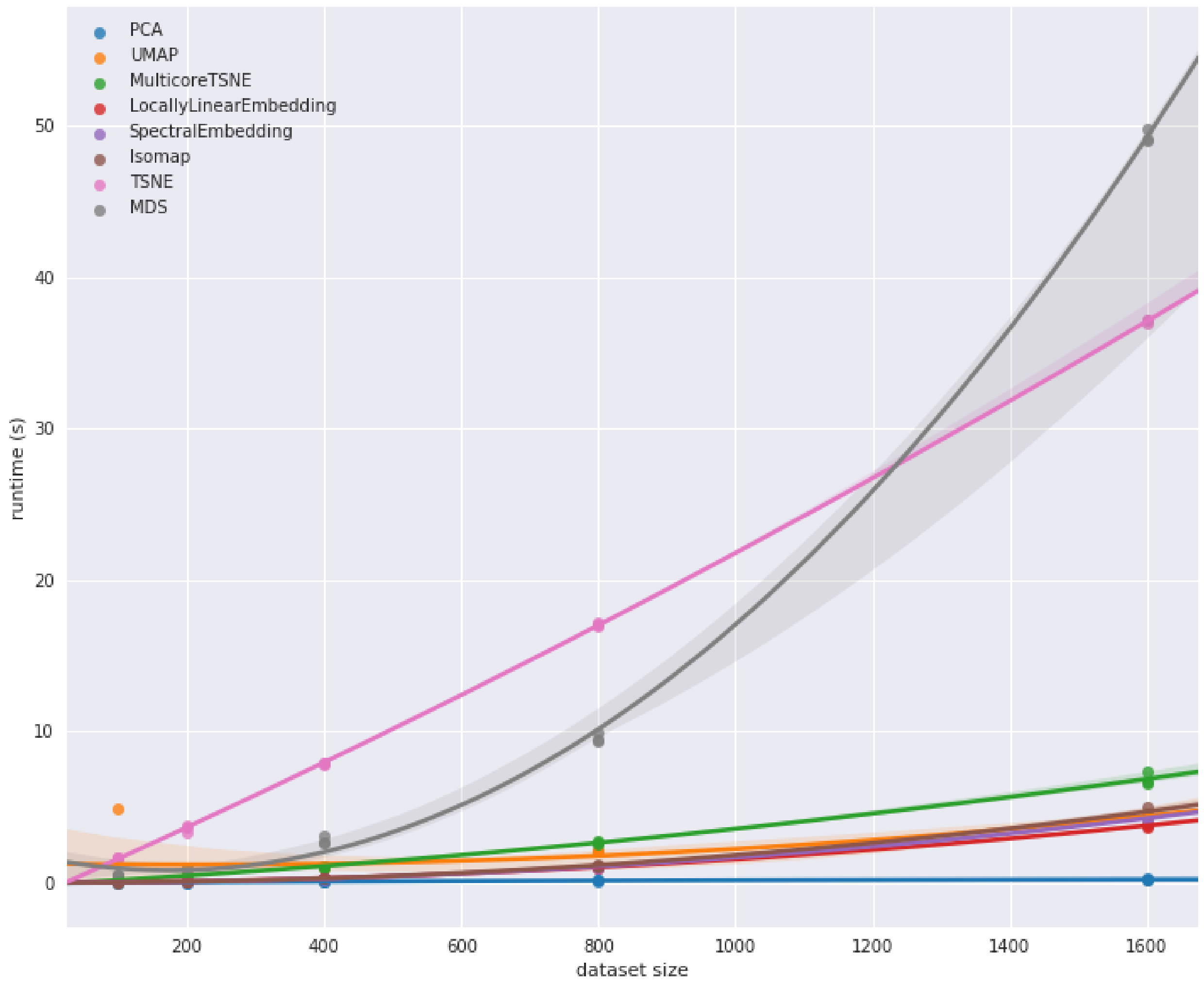
9.1 Performance scaling by dataset size

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.

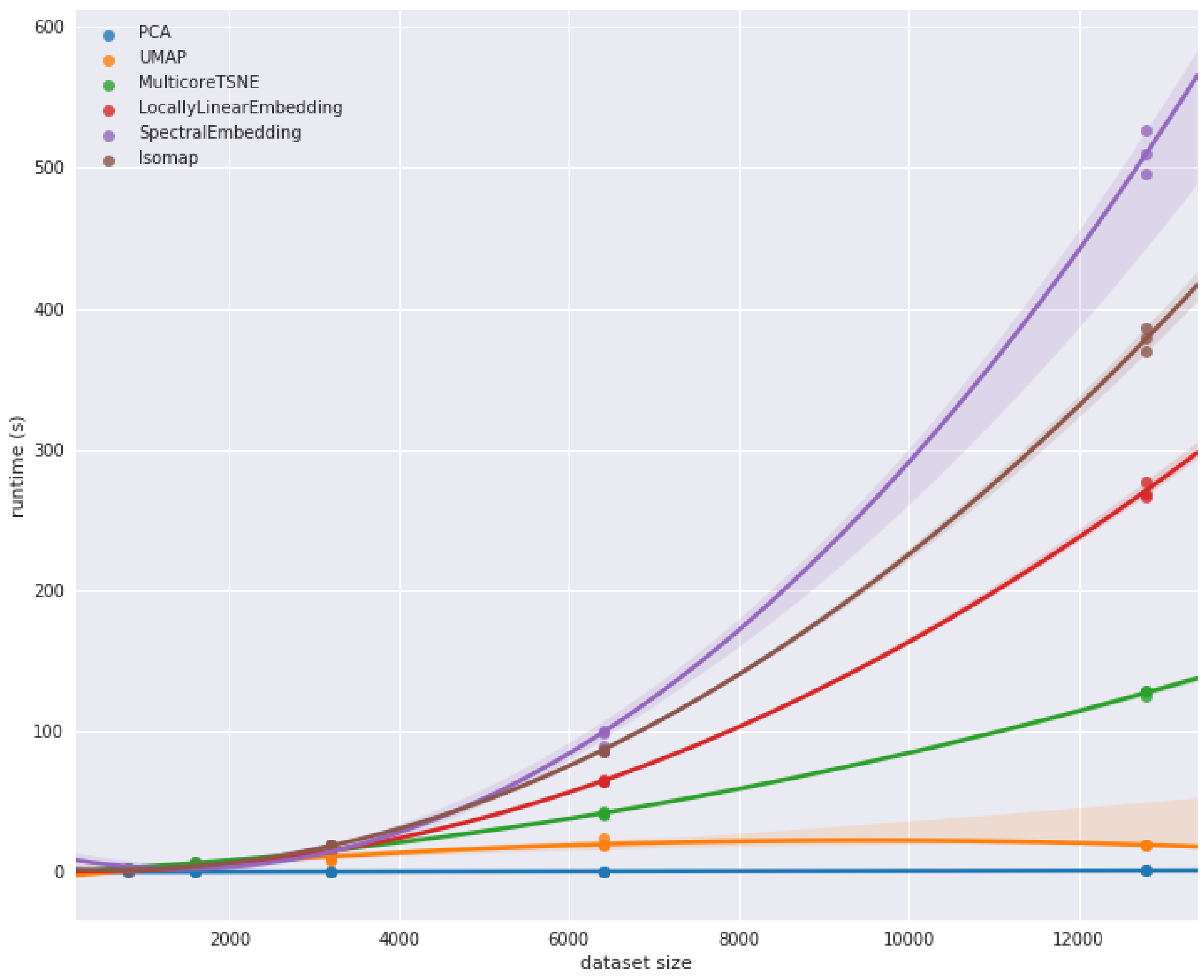
Now we just want to run this for each of the various dimension reduction implementations so we can look at the results. Since we don't know how long these runs might take we'll start off with a very small set of samples, scaling up to only

1600 samples.

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.

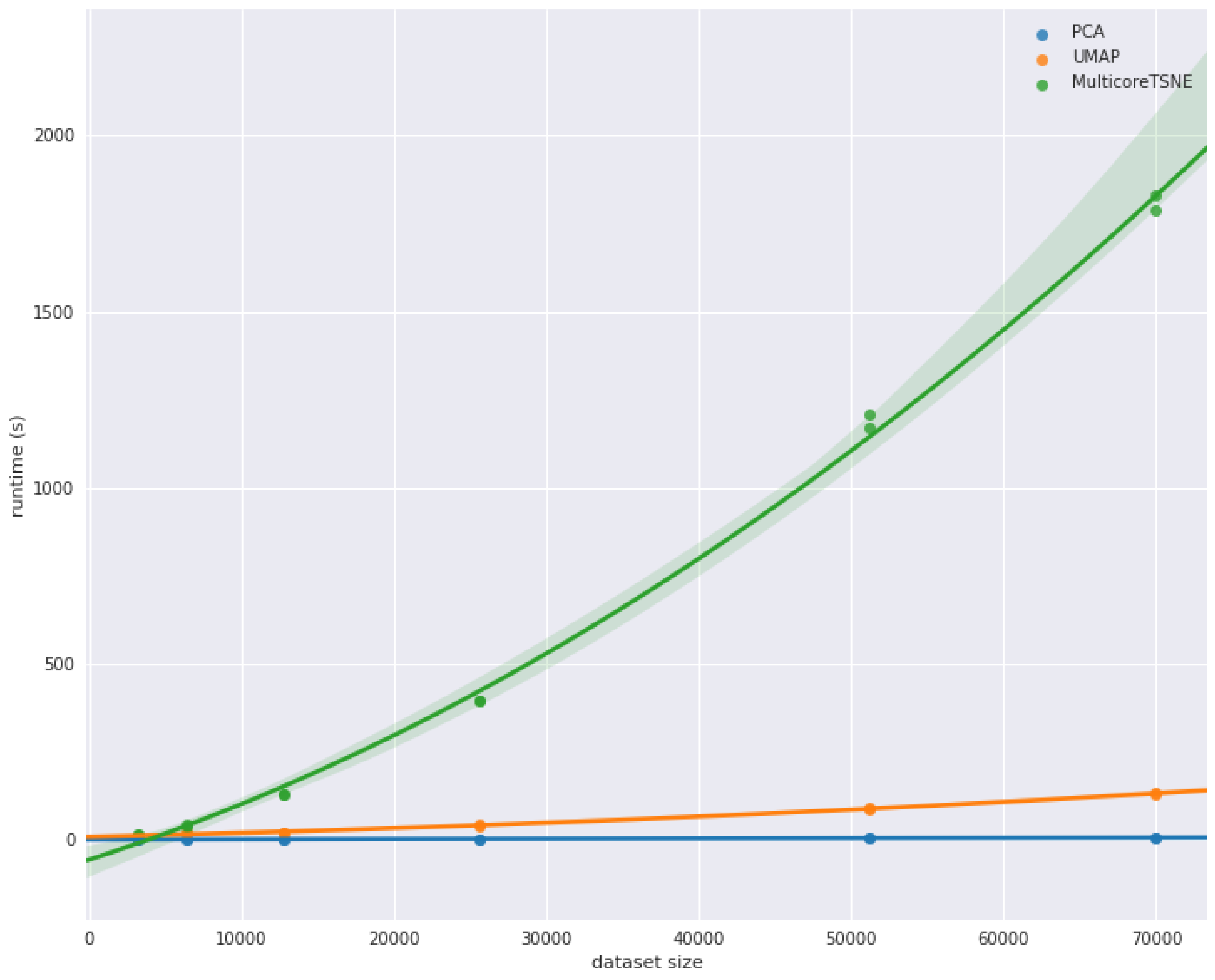


We can see straight away that there are some outliers here. The scikit-learn t-SNE is clearly much slower than most of the other algorithms. It does not have the scaling properties of MDS however; for larger dataset sizes MDS is going to quickly become completely unmanageable. At the same time MulticoreTSNE 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 and scikit-learn's t-SNE are going to 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.



At this point we begin to see some significant differentiation among the different implementations. In the earlier plot MulticoreTSNE looked to be slower than some of the other algorithms, but as we scale out to larger datasets we see that its relative scaling performance is far superior to the scikit-learn implementations of Isomap, spectral embedding, 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 MulticoreTSNE, PCA and UMAP.



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 for even larger datasets the difference 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 10

UMAP API Guide

UMAP has only a single class `UMAP`.

10.1 UMAP

class `umap.umap_.`**`UMAP`**(*n_neighbors=15*, *n_components=2*, *metric='euclidean'*, *n_epochs=None*, *learning_rate=1.0*, *init='spectral'*, *min_dist=0.1*, *spread=1.0*, *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*, *metric_kwds=None*, *angular_rp_forest=False*, *target_n_neighbors=-1*, *target_metric='categorical'*, *target_metric_kwds=None*, *target_weight=0.5*, *transform_seed=42*, *verbose=False*)

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
- 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.

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.

`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.

`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.

10.2 Useful Functions

class `umap.umap_.`**`UMAP`**(*n_neighbors=15*, *n_components=2*, *metric='euclidean'*, *n_epochs=None*, *learning_rate=1.0*, *init='spectral'*, *min_dist=0.1*, *spread=1.0*, *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*, *metric_kwds=None*, *angular_rp_forest=False*, *target_n_neighbors=-1*, *target_metric='categorical'*, *target_metric_kwds=None*, *target_weight=0.5*, *transform_seed=42*, *verbose=False*)
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
- 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.

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.

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.

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_.`**categorical_simplicial_set_intersection**

Combine a fuzzy simplicial set with another fuzzy simplicial set generated from categorical data using categorical 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.

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.

simplicial_set: sparse matrix The resulting intersected fuzzy simplicial set.

`umap.umap_.`**clip**

Standard clamping of a value into a fixed range (in this case -4.0 to 4.0)

val: float The value to be clamped.

The clamped value, now fixed to be in the range -4.0 to 4.0.

`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.

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)

`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.

simplicial_set: sparse matrix The resulting intersected fuzzy simplicial set.

`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`**

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

- manhattan
- chebyshev
- minkowski
- canberra
- braycurtis
- mahalanobis
- wminkowski
- seuclidean
- cosine
- correlation
- haversine
- hamming
- jaccard
- dice
- russelrao
- kulsinski
- 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.

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.

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_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`**

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*, *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.

verbose: bool 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.

`umap.umap_.`**`optimize_layout`**

Improve an embedding using stochastic gradient descent to minimize the fuzzy set cross entropy between the 1-skeletons of the high dimensional and low dimensional fuzzy simplicial sets. In practice this is done by sampling edges based on their membership strength (with the (1-p) terms coming from negative sampling similar to word2vec).

head_embedding: array of shape (n_samples, n_components) The initial embedding to be improved by SGD.

tail_embedding: array of shape (source_samples, n_components) The reference embedding of embedded points. If not embedding new previously unseen points with respect to an existing embedding this is simply the head_embedding (again); otherwise it provides the existing embedding to embed with respect to.

head: array of shape (n_1_simplices) The indices of the heads of 1-simplices with non-zero membership.

tail: array of shape (n_1_simplices) The indices of the tails of 1-simplices with non-zero membership.

n_epochs: int The number of training epochs to use in optimization.

n_vertices: int The number of vertices (0-simplices) in the dataset.

epochs_per_samples: array of shape (n_1_simplices) A float value of the number of epochs per 1-simplex. 1-simplices with weaker membership strength will have more epochs between being sampled.

a: float Parameter of differentiable approximation of right adjoint functor

b: float Parameter of differentiable approximation of right adjoint functor

rng_state: array of int64, shape (3,) The internal state of the rng

gamma: float (optional, default 1.0) Weight to apply to negative samples.

initial_alpha: float (optional, default 1.0) Initial learning rate for the SGD.

negative_sample_rate: int (optional, default 5) Number of negative samples to use per positive sample.

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 embedding.

`umap.umap_.`**`rdist`**

Reduced Euclidean distance.

x: array of shape (embedding_dim,) y: array of shape (embedding_dim,)

The squared euclidean distance between x and y

`umap.umap_.`**`reset_local_connectivity`**

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*, *verbose*)

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 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.

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.

`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 esscence 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 11

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index