n2d Documentation

Release 0.3.1

David Josephs

Mar 26, 2020

Contents:

CHAPTER 1

About N2D

N2D is a python library implementation of the "deep" clustering method described in this brilliant paper, and by all metrics represents the absolute state of the art in time series/sequence and image clustering. The source code for the software is available here.

In this section we will talk about the motivations for N2D, what it is, and the goals for this package.

1.1 What is N2D?

N2D is short for "Not too deep" clustering. A "not too deep" clustering algorithm works as follows:

1. The data goes into an autoencoder (or other representation learning neural network), which is trained, learning a powerful, concise representation (embedding) of the data.
2. The autoencoded embedding then goes into a manifold learner, in this case primarily UMAP (while t-sne and ISOMAP are also usable), which finds a *local manifold* within the data
3. The local manifold is then sent into a clustering algorithm, which clusters the data

1.1.1 What does it do?

The idea of N2D is as follows: by first learning an embedding of the data, and then learning the manifold of the autoencoded data, we transform the data into a form that is readily clusterable, again demonstrated in the paper. N2D is competitive with the most state of the art deep clustering techniques out there, with the benefit of being simple, relatively fast, and intuitive, and represents an excellent path for future research.

1.2 Purpose of the library

The purpose of this library is to provide A) an easy library for regular use and B) an extensible framework for future research.

1.3 Citation

Please cite the original authors of the algorithm if you use N2D in your research.

```
@article{2019arXiv190805968M,
title = {N2D:(Not Too) Deep Clustering via Clustering the Local Manifold of an␣
↪Autoencoded Embedding},
author = {{McConville}, Ryan and {Santos-Rodriguez}, Raul and {Piechocki}, Robert J␣
↪and {Craddock}, Ian},
journal = {arXiv preprint arXiv:1908.05968},
year = "2019",
}
```

CHAPTER 2

Getting started

Here we will talk about getting started with N2D so you can get clustering!!

2.1 Installation

N2D is on Pypi and readily installable

```
pip install n2d
```

Please note that if you want GPU support, **You will also need to install tensorflow with GPU support**

2.2 Loading Data

N2D comes with **5** built in datasets: 3 image datasets and two time series datasets, described below:

- MNIST - **Description**: Standard handwritten image dataset. 10 classes
- MNIST-Test - **Description**: Test set of MNIST. 10 classes
- MNIST-Fashion - **Description**: Pictures of articles of clothing, similar to MNIST but much more difficult. 10 classes
- Human Activity Recognition (HAR) - **Description**: Time series of accelerometer data, used to determine whether the recorded human is sitting, walking, going upstairs/downstairs etc. 6 classes
- Pendigits - **Description**: Pressure sensor data of humans writing. Used to determine what number the human is writing. 10 classes

To actually load the data, we import the datasets from n2d, shown below along with the data import functions and their outputs

```
from n2d import datasets as data

# imports mnist
data.load_mnist() # x, y

# imports mnist_test
data.load_mnist_test() # x, y

# imports fashion
data.load_fashion() # x, y, y_names

# imports HAR
data.load_har() # x, y, y_names

# imports pendigits
data.load_pendigits # x, y
```

In this example, we are going to use HAR.

```
x, y, y_names = data.load_har()
```

2.3 Building the model

To build an N2D model, we are going to need 2 pieces: an autoencoder, and a manifold clustering algorithm. Both are provided with the library thankfully! First, we will load up any libraries we want to use in this example:

```
import n2d
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
plt.style.use(['seaborn-white', 'seaborn-paper'])
sns.set_context("paper", font_scale = 1.3)
matplotlib.use('agg')
np.random.seed(0)

from n2d import datasets as data
# load in the data
x, y, y_names = data.load_har()
```

The first step of any not too deep clustering procedure is the autoencoded embedding. Therefore, we will initialize that first. We do this with the AutoEncoder class:

2.3.1 The AutoEncoder Class

So lets go ahead and initialize the autoencoder. This again uses the N2D AutoEncoder class:

```
n_clusters = 6
latent_dim = n_clusters

ae = n2d.AutoEncoder(x.shape[-1], latent_dim)
```

In the simplest possible example, this is it! The Autoencoder class **requires** the input dimensions of the data, and the number of dimensions we would like to reduce that to (latent dimensions, embedding dimensions). We can also modify the internal architecture of the AutoEncoder with the **architecture** argument. By default, the shape of the

encoder is *[input_dim, 500, 500, 2000, latent_dim]* and the shape of the **decoder** is *[latent_dim, 2000, 500, 500, input_dim]*, or the reverse of the encoder. The autoencoder consists of these two ends stacked together, giving a network with dimensions: *[input_dim, 500, 500, 2000, latent_dim, 2000, 500, 500, input_dim]*. The shape of the network in between the input and latent dimensions can be replaced with a list, for example if we wanted the first three layers of the encoder to be 2000 neurons, and the next 4000 we would say (expecting the decoder to be the reverse of this):

```
ae_huge = n2d.Autoencoder(x.shaep[-1], latent_dim, architecture = [2000, 2000, 2000,␣
↪4000])
```

We can also change the activation function of our hidden layers by specifying **act**. Below is a table of all the parameters for AutoEncoder:

Table 1: n2d.AutoEncoder Arguments

Argument	Default	Description
input_dim	no default	The data's dimensions, typically data.shape[-1]
latent_dim	10	Number of dimensions you wish to represent the data in with the autoencoder
architecture	[500, 500, 2000]	The layout of the hidden layers in the network, presented in list form
act	'relu'	The activation function for the hidden layers of the network
x_lambda	lambda x: x	Function used to transform the inputs to the network, but hold the outputs constant

It is important to note that while we set the latent dimensions to be the same as the number of clusters, this is not a hard and fast rule. Use your head and some sense when choosing dimensions!

The next step in Not Too Deep clustering is to learn the manifold in the embedding and cluster that. In the original paper describing N2D, UMAP and Gaussian mixing performed the best, and therefore are implemented in the library. To do this, we use the UmapGMM class (replacing the autoencoder/manifold learner/clustering algorithm will be discussed in the next chapter).

2.3.2 Clustering the Embedded Manifold: UmapGMM

Lets talk a bit more about how we learn the manifold and cluster it!! This is done primarily with the UmapGMM object

```
manifoldGMM = n2d.UmapGMM(n_clusters)
```

This initializes the hybrid manifold learner/clustering arguments. In general, UmapGMM performs best, but in a later section we will talk about replacing it with other clustering/manifold learning techniques. The arguments for UmapGMM are shown below:

Table 2: UmapGMM Arguments

Argument	Default	Description
n_clusters	no default	The number of clusters
umap_dim	2	Number of dimensions of the manifold.
umap_neighbors	10	Number of nearest neighbors to consider for UMAP. Defaults to 10, to recreate cutting edge results shown in the paper, however often 20 is a better value
umap_min_distance	float(0)	Minimum distance between points within the manifold. Smaller numbers get tighter, better clusters while larger numbers are better for visualization
umap_metric	'euclidean'	The distance metric to use for UMAP.
random_state	0	The random seed

For our use case, there are two main tunables: **umap_dim**, and **umap_neighbors**. **umap_dim** is the number of dimensions you wish to project the autoencoded embedding in. In general, values between **2** and **the number of clusters** are acceptable. It is best to start at 2 (the default value) and then go up from there. All of the breakthrough results in the paper were done with umap_dim =2. **umap_neighbors** is the number of nearest neighbors UMAP will use when constructing its KNN graph. In the case of N2D, this should be a small value, as we want to learn the **local manifold**. The default value for umap_neighbors is **10**, as it will allow you to reproduce the results in the paper, however umap_neighbors = **20** sometimes performs slightly better, *especially if the autoencoder loss is high*. Since umapGMM takes just a few seconds to run, it is worth it to tune these two values in general.

Finally, we are ready to get clustering!

2.3.3 Initializing N2D

Next, we initialize the **n2d** object. We feed it first an autoencoder, and second a manifold clusterer:

```
harcluster = n2d.n2d(ae, manifoldGMM)
```

and that's it! Now we can fit and predict!

2.3.4 Learning an Embedding

Next, we need to train the autoencoder to learn the embedding. This step is pretty easy. As this is our first run of the autoencoder, the only thing we need to input is the name we would like the weights to be stored under, as well as create a weights directory.

```
harcluster.fit(x, weight_id = "weights/har-1000-ae_weights.h5")
```

This will train the autoencoder, and store the weights in **weights/[WEIGHT_ID]-[NUM_EPOCHS]-ae_weights.h5**. The arguments to the preTrainEncoder method are shown in the table below:

Table 3: fit Arguments

Argument	Default	Description
batch_size	256	The batch size
epochs	1000	number of epochs
loss	"mse"	The loss function. Anything that tf.keras accepts will do.
optimizer	"adam"	The optimizier
weights	None	The name of the weight file. If None, the model will be trained
verbose	0	The verbosity of the training
weight_id	None	if None, the encoder weights will not be saved. If string, it will save the weights to that file path
patience	None	int or None. If None, nothing special happens, if int, the tolerance for early stopping

Please note the patience parameter! It can save lots of time. Also please note, if you do not tell N2D where to save the model weights, it will not save them!!

On our next round of the autoencoder, while we fiddle with clustering algorithms, visualizations, or whatever, we can use the preTrainEncoder method to load in our weights as follows.

```
harcluster.fit(x, weights = "weights/har-1000-ae_weights.h5")
```

Finally, we can actually cluster the data! To do this, we pass the clustering mechanism into the N2D predict method.

```
preds = harcluster.predict(x)
```

This will save the prediction internally and externally (for visualization convenience). The prediction is internally stored in

```
harcluster.preds
```

for your convenience if you want to access the predictions for plotting/further analysis

2.3.5 fit_predict

We can wrap these two commands into one using the fit_predict method, which takes the same arguments as fit:

```
harcluster.fit_predict(x, weight_id = "weights/har-1000-ae_weights.h5")
```

2.3.6 predict_proba

If your clusterer has the method "predict_proba", you can also do that:

```
probs = harcluster.predict_proba(x)
```

2.3.7 Assessing and Visualization

To assess the quality of the clusters, you can A) use some custom assessment method on the predictions or B) if you have labels run

```
harcluster.assess(y)
# (0.81212, 0.71669, 0.64013)
```

This prints out the cluster accuracy, NMI, and ARI metrics for our clusters. These values are top of the line for all clustering models on HAR.

To visualize, we again have a built in method as well as tools for creating your own visualizations:

Built in:

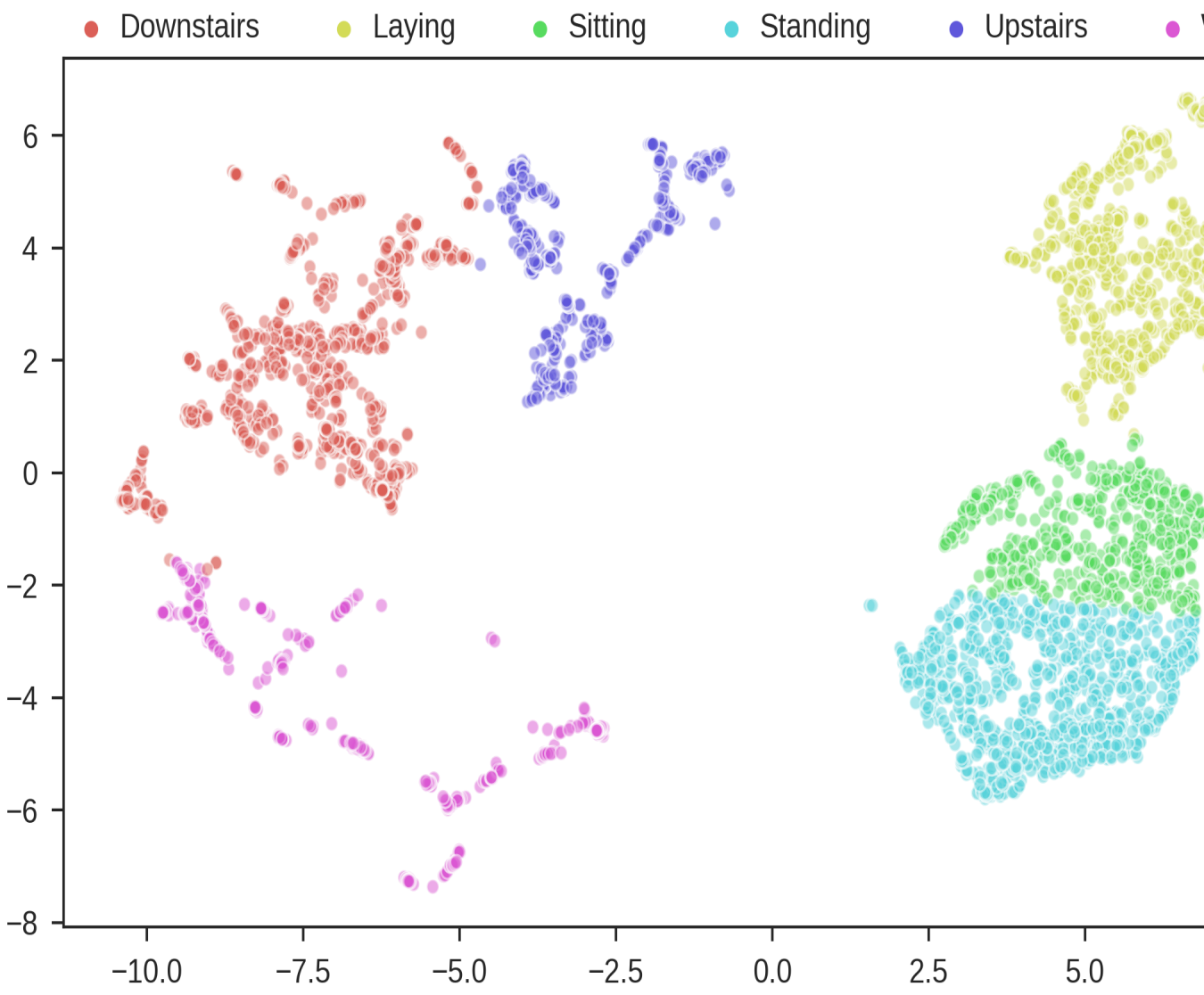
```
harcluster.visualize(y, y_names, n_clusters = n_clusters)
plt.show()
```

Custom :

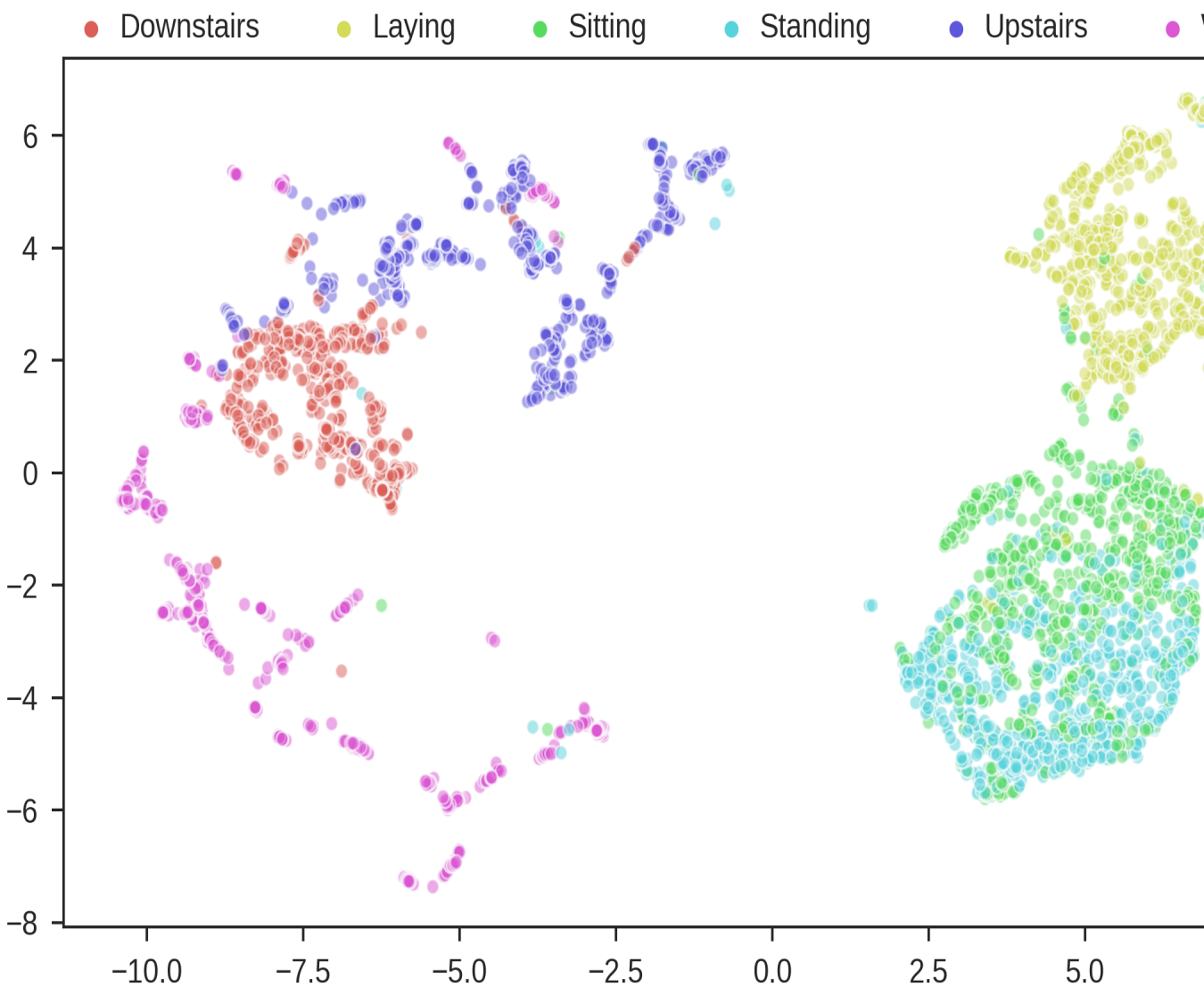
We need a few things for a visualization: The embedding and the the predictions. The embedding is stored in

```
harcluster.hle
```

You typically want to plot the embedding as x and the clusters as y! Lets also check out what our clusters look like!



These are the predicted clusters, now lets look at the real groupings!



Looks like we did a pretty good job!! One very interesting thing to note, is even though it got some things wrong, where it got them wrong is still useful. The stationary activities are all near each other, while the active activities are all together. N2D, with no features and labels, not only found useful clusters, but ones that provide real world intuition! This is a very powerful result.

2.4 Predicting on new data

Once the everything has been fitted, we can easily make fast predictions on new data:

```
x_test, y_test = some test set
new_preds = harcluster.predict(x_test)
```

This will use the autoencoder to map the data into the proper number of dimensions, and then transform it to the manifold learned during fitting, and finally cluster it using the trained clustering mechanism.

2.5 Saving and Loading

N2D models can be saved for deployment with the **save_n2d** and the **load_n2d** functions. Currently, this is managed by saving the **encoder** to an h5 file, and pickling the **manifold clusterer**. This is an open option area for development, ideally the whole model will be serialized in an h5 file. If you wish to contribute, please see the issue. To save an n2d model, follow the following procedure:

```
n2d.save_n2d(harcluster, encoder_id='models/har.h5', manifold_id='models/hargmm.sav')
```

to load, we follow a similar mechanism:

```
hcluster = n2d.load_n2d('models/har.h5', 'models/hargmm.sav')
```

Please note that **for rapid development and experimentation** you should use the **weight saving** in the *.fit* method, as that is its intended use. You can train the network and then fiddle around with the rest of the model. This means that **save_n2d** and **load_n2d** should **only be used for deploying the model**.

CHAPTER 3

Advanced Usage

As mentioned earlier, N2D is an entirely extensible framework for not too deep clustering. In this section we will discuss modifying the clustering/manifold learning methods, and modifying the autoencoder. The independence of each step of N2D means we can change the autoencoder into a convolutional autoencoder or some other more complex LSTM based autoencoder, depending on the application, or change clustering method. We will discuss changing both parts of the algorithm below.

3.1 Changing the Manifold Clustering Step:

To extend N2D to include your favorite autoencoder or clustering algorithm, you can use either of the two **generator** classes. To replace the manifold clustering step, we use the **manifold_cluster_generator** class. This class takes in 4 arguments:

1. The class of the manifold learner, for example, umap.UMAP
2. A dict of arguments to initialize the manifold learner with
3. The class of the clusterer
4. A dict of arguments for the clusterer

Objects created by **generators** can be passed directly into N2D, without needing any boilerplate code. Lets go ahead and look at an example. Let us assume that we want to use density based clustering with UMAP and our standard autoencoder based dimensionality reduction. First, we import our libraries:

```
import n2d
import numpy as np
import n2d.datasets as data
import hdbscan
import umap

x, y = data.load_mnist()
```

First, we make our autoencoder, for now using the AutoEncoder class:

```
ae = n2d.AutoEncoder(input_dim = x.shape[-1], latent_dim = 20) # chosen arbitrarily
```

Next, lets define the arguments we wish to initialize hdbscan and umap with. Please note these values are chosen either arbitrarily or for visualization:

```
# hdbscan arguments
hdbscan_args = {"min_samples":10,"min_cluster_size":500, 'prediction_data':True}

# umap arguments
umap_args = {"metric":"euclidean", "n_components":2, "n_neighbors":30,"min_dist":0}
```

Next, lets go ahead and generate something we can use to cluster our embedding!!:

```
db_clust = n2d.manifold_cluster_generator(umap.UMAP, umap_args, hdbscan.HDBSCAN,␣
↪hdbscan_args)
```

Now we pass those into **n2d.n2d** and we are good to go!:

```
n2d_db = n2d.n2d(ae, db_clust)
```

We can fit as usual:

```
n2d_db.fit(x, epochs = 10) # for times sake, this is just an example
```

Because this is dbscan, after fitting we can say we are done! The fitted n2d object can do anything the parent clustering class can do (it also shares its limitations). This means that we can just go ahead and grab the predictions which hdbscan already so kindly made for us:

```
# the probabilities
print(n2d_db.clusterer.probabilities_)
# the labels
print(n2d_db.clusterer.labels_)
```

The clustering algorithm is stored in .clusterer, while the manifold learner is stored in .manifolder::
print(n2d_db.clusterer) print(n2d_db.manifolder)

Note that while our fitted n2d object has all the attributes of the clustering mechanism, it also has all of the limitations. That means, in the case of hdbscan, we can do **fit_predict**, however there is no **predict** method.:

```
# works
n2d_db.fit_predict(x, epochs = 10)
# fails
n2d_db.predict(x)
```

However, hdbscan has a neat trick where we can make "approximate predictions". This is allowed! We can write a imple function to get the approximate predictions and make predictions on new data:

```
x_test, y_test = data.load_mnist_test()

# predict on new data with dbscan and not too deep clustering!
def approx_predict(n2d_obj, newdata):
        embedding = n2d_obj.encoder.predict(newdata)
        manifold = n2d_obj.manifolder.transorm(embedding)
        labs, probs = hdbscan.approximate_predict(n2d_obj.clusterer, manifold)
        return labs, probs

labs, probs = approx_predict(n2d_db, x_test)
```

Next, lets look at swapping out the autoencoder!!

3.2 Changing the Autoencoder

To swap out the autoencoder, we can, just as with the clustering step, use a **generator** class. In this case, we will use the **autoencoder_generator** class. This class takes in 2 things: an iterable of model parts, and if needed a lambda function. The lambda function is not necessary, and by default does nothing. However, for some use cases it may be useful to change the inputs to the encoder. We will look at one such case: a denoising autoencoder. **NOTE: this is a simple example to showcase features, there is no real precedent for clustering with a denoising autoencoder**

First, again, we load up our libraries:

```
import n2d
from n2d import datasets as data
from tensorflow.keras.layers import Dense, Input
import seaborn as sns
import umap
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
plt.style.use(['seaborn-white', 'seaborn-paper'])
sns.set_context("paper", font_scale=1.3)

x, y, y_names = data.load_fashion()

n_clusters = 10
```

Next, as usual, we are going to make our autoencoder, however this time without the AutoEncoder class. We are going to want to make a list, tuple, or array that contains pointers to the input layer, the end of the encoder (center layer), and output layer of the encoder. To do that we will use the tf.keras functional API:

```
hidden_dims = [500, 500, 2000]
input_dim = x.shape[-1]
inputs = Input(input_dim)
encoded = inputs
for d in hidden_dims:
    encoded = Dense(d, activation = "relu")(encoded)
encoded = Dense(n_clusters)(encoded)
decoded = encoded
for d in hidden_dims[::-1]:
    decoded = Dense(d, activation = "relu")(decoded)
outputs = Dense(input_dim)(decoded)
```

Lets go ahead and define our first set of inputs for the **autoencoder_generator** class:

```
ae_stages = (inputs, encoded, outputs)
```

Again, the autoencoder_generator class requires an iterable containing the input layer, the encoding, and the decoded output layer of the model. The rest is taken care of internally. As this is a denoising autoencoder, lets also write a function that adds noise to our data:

```
def add_noise(x, noise_factor):
    x_clean = x
    x_noisy = x_clean + noise_factor * np.random.normal(loc = 0.0, scale = 1.0, size␣
↪= x_clean.shape)
```

(continues on next page)

(continued from previous page)

```
    x_noisy = np.clip(x_noisy, 0., 1.)
    return x_noisy
```

Now we can go ahead and generate an autoencoder for N2D:

```
denoising_ae = n2d.autoencoder_generator(ae_stages, x_lambda = lambda x: add_noise(x,␣
↪0.5))
```

Finally, lets initialize UmapGMM and our model, and make a quick prediction:

```
umapgmm = n2d.UmapGMM(n_clusters)
model = n2d.n2d(denousing_ae, umapgmm)
model.fit(x, epochs=10)
model.predict(x)
model.visualize(y, y_names,  n_clusters = n_clusters)
plt.show()
print(model.assess(y))
```

And with that, you are ready to get clustering and testing new and unexplored algorithms! If you are having any troubles, or ideas for features, please make an issue on github!!

CHAPTER 4

Indices and tables

- genindex
- modindex
- search