Unsupervised Learning and Dimensionality Reduction

Blake Wang bwang400@gatech.edu

The Data

Table 1 - Description of the data

	Wine Quality (WQ)	Generated Blobs (GB)
Examples	4898	1000
Features	12	2
Input Feature Type	Float	Float
Target Feature Type	Integer (0 to 10)	Integer representing cluster label (0 to 2)
Source	https://archive.ics.uci.edu/ml/datasets/Wine+Quality	make_blobs(centers=6)

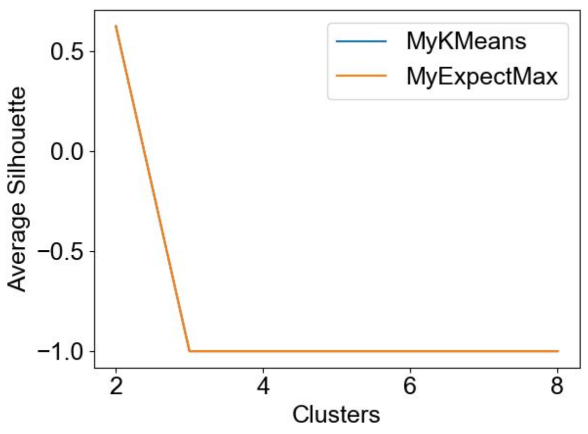
I chose to use Wine Quality (WQ) from my Assignment 1 submission and a new dataset, Generated Blobs (GB), which I generated using scikit-learn's make_blobs dataset generator. I thought this would lead to an interesting juxtaposition between real-world and simulated data. Furthermore, the generated dataset has a few upsides. For one, I can generate 2-dimensional data which can be represented in a plot so that clustering can be easily visualized. Secondly, make_blobs gives us the "true" cluster labels so I can measure clustering accuracy. Note that the clustering algorithms could in fact output an even *better* clustering than the "true" labels, statistically speaking, because the data generation is probabilistic. However, this is still handy for benchmarking my clustering algorithm implementations.

1 Clustering

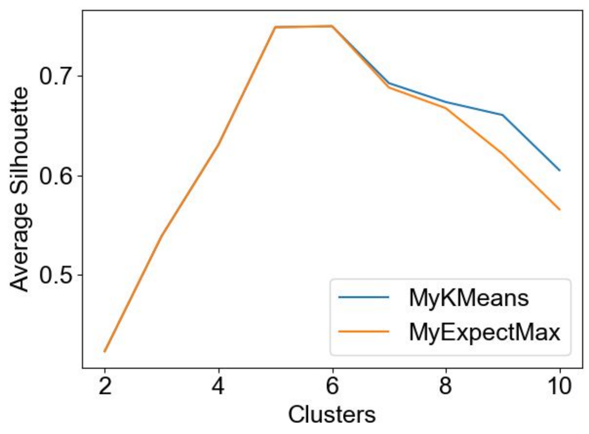
I started by writing my own implementations for MyKMeans and MyExpectMax from scratch in order to better understand the algorithms. Because both datasets have real-valued inputs, I chose to use Euclidean distance in my implementations. I then generated the GB dataset and collected some benchmark metrics using my implementations. I did not like the idea that we have to know the number of clusters ahead of time so I used the average silhouette score to find the best number of clusters. Because I already knew that I generated data with 6 clusters, I was able to verify this method. I looped through numbers of clusters attempting to maximize the average silhouette score, then iterations and wall clock time.

Next, I did the same thing for the WQ dataset. At first, I was getting very poor (negative) average silhouette values from MyExpectMax, but tuning the standard deviation used by the algorithm yielded great improvements. The best value turned out to be very large. This makes sense because the larger the standard deviation, the more MyExpectMax's clusters resemble uniform distributions rather than gaussians. In other words, when standard deviation is very high, MyExpectMax behaves more like MyKMeans.

Figure 1 - Average silhouette over number of clusters

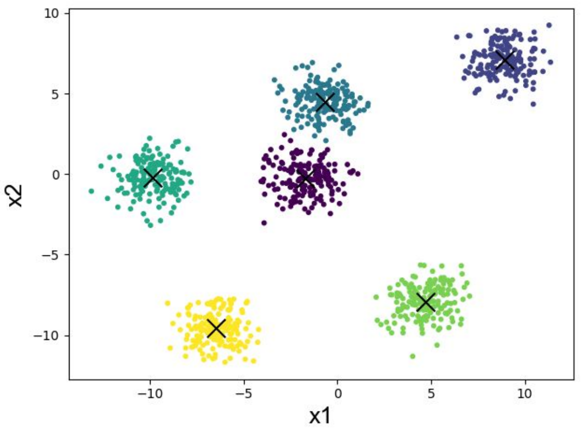


WQ dataset

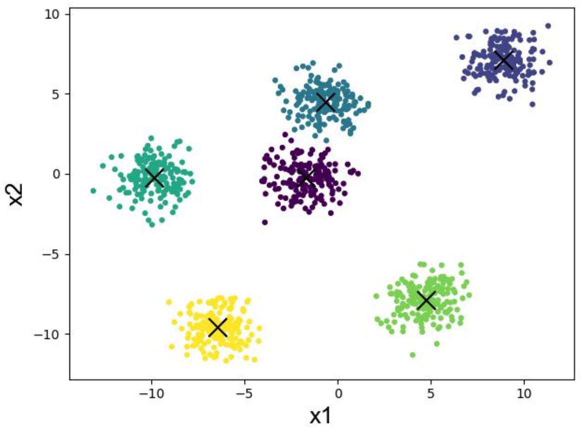


GB dataset

Figure 2 - Clustering output



MyKMeans



MyExpectMax

In the WQ plot of FIgure 1, we see that any more than 2 clusters yields an average silhouette score of -1, the lowest possible value, for both clustering algorithms. This suggests that maybe there is only *good* wine and *bad* wine, and not 11 levels of granularity in wine quality. In the GB dataset plot, both algorithms have a clear peak at 6 clusters which matches the make_blobs labels with extremely low error. This is obviously to be expected because this dataset was “purpose built” for clustering. Figure 2 shows that both algorithms converged on the same set of clusters. The “X” markers represent the centers of each cluster. Note that I “hardened” the soft clusters by assigning each point a color according to its most probable hidden value.

Table 2 - Results from using average silhouette score to find best number of clusters

	Clusters	Score	Error	Iterations	Time
WQ MyKMeans	2	+0.62	17.1%	10	1.12s
WQ MyExpectMax	2	+0.62	17.0%	13	2.62s
GB MyKMeans	6	+0.75	0.2%	6	0.41s
GB MyExpectMax	6	+0.75	0.2%	5	0.67s

Table 2 describes quantitatively that for the WQ dataset both clustering algorithms found their highest average silhouette score with only 2 clusters, though there are actually 7 labels present in the data. MyKMeans was able to converge in a shorter wall clock time than MyExpectMax for both datasets though it executed more iterations. This makes sense given that MyExpectMax’s costlier probabilistic calculations of soft clusters cause it to have greater time per iteration. it should be noted that I again “hardened” the soft clusters in order to compute silhouette and error values for the MyExpectMax algorithm.

2 Dimensionality Reduction

Principal Component Analysis

First, I ran PCA on the GB dataset. I used scikit-learn’s PCA implementation. The results were not very interesting though this was expected given the low dimensionality to begin with. I ran PCA with n_components=0.95 which this implementation interprets as, “give me the principal components that explain 95% of the variance.” When PCA performed the transformation, the feature dimension stayed at 2. It also yielded that the first component explained 67.6% of the variance. So if I were to drop the second principal component, I would be throwing away 32.4% of the information. The fact that PCA was not able to reduce the dimensionality of the GB dataset without unacceptably high information loss is not discouraging or surprising because it was only 2-dimensional to begin with.

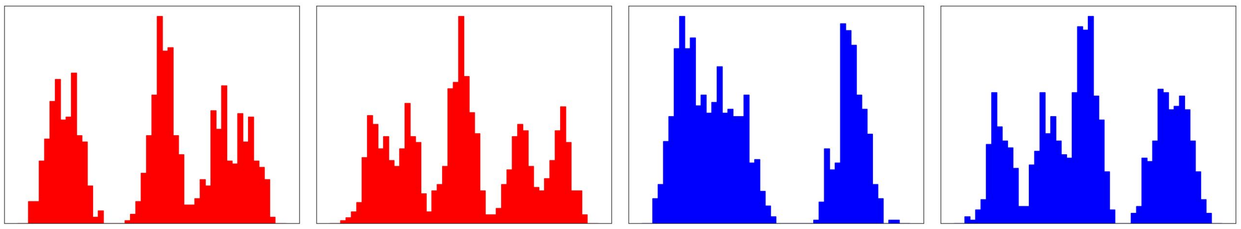
I then ran PCA on the WQ dataset. The effect was far more dramatic! By again running with n_components=0.95, PCA was able to reduce dimensionality from 12 to 1 while still explaining

99.9% of the variance! This will certainly help the neural network training time in the coming sections by effectively lifting the curse of dimensionality.

Independent Component Analysis

ICA proved somewhat trickier because it is less obvious to me how finding independent sources, or *unmixing*, really helps when training a neural network. I first hypothesized that running ICA would transform my data such that the distribution for each independent component was less normal. First, I tried a purely visual approach. I plotted histograms of the WQ and GB datasets both before and after a run of scikit-learn's FastICA implementation. As WQ has 12 features, I'll show only a selection of the WQ histograms.

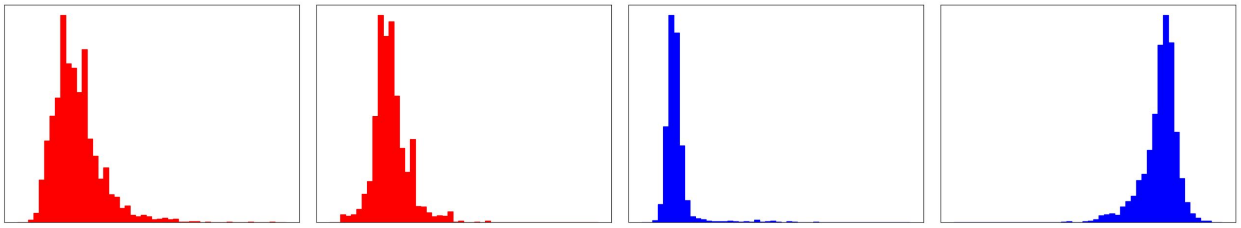
Figure 3 - Distributions before and after FastICA transformation



GB distributions *before* FastICA **GB distributions *after* FastICA**

Qualitatively, there are fewer peaks in the post-FastICA histograms in Figure 3 which suggests that s1 and s2 might better represent the "true" sources. This evidence is not very clear so I chose to investigate further.

Figure 4 - Selection of distributions before and after FastICA transformation



WQ distributions *before* FastICA **WQ distributions *after* FastICA**

Figure 4 shows only a selection of 2 of 12 features (x1 and x2) and 2 independent components (s1 and s2) for the sake of space. I observed that the distributions of the post-FastICA components seem narrower than the original features'. I chose this selection of histograms to help demonstrate that. This again is rather weak evidence of more independent sources.

In order to make some more concrete statements about independence of the components, I tried out 3 different normality tests; the Shapiro-Wilk Test, the D'Agostino's K^2 Test, and the

Anderson-Darling Test. My goal here was to show that the distributions of the FastICA components were *less* normal than the original features'. All tests agreed that both the GB and WQ distributions were highly non-normal in every dimension with p values on the order of 10^{-17} for GB and 10^{-33} for WQ. I interpreted these results to mean I would need to to use a measure of independence that did not rely on gaussian distributions.

The last bit of analysis I did was to compute mutual information between each of the feature vectors before FastICA and between each of the component vectors after FastICA. I used scikit-learn's mutual_info_score module to calculate a matrix of mutual information scores for each pair of vectors. I did not include the mutual information score between each vector and itself (matrix diagonal). I then took mean of all the mutual information values in the matrix to get a mean score for comparing pre and post transformation distributions. I found that FastICA reduced this mean mutual information score from 0.251 for the original distributions to 0.183 for the transformed distributions on the WQ dataset, a 27% reduction. Running FastICA on the GB dataset's distributions yielded a 17% mean mutual information reduction from 1.449 to 1.198. This experiment shows quantitatively that components in the new space are in fact more independent than the original feature vectors. In other words, FastICA did its job!

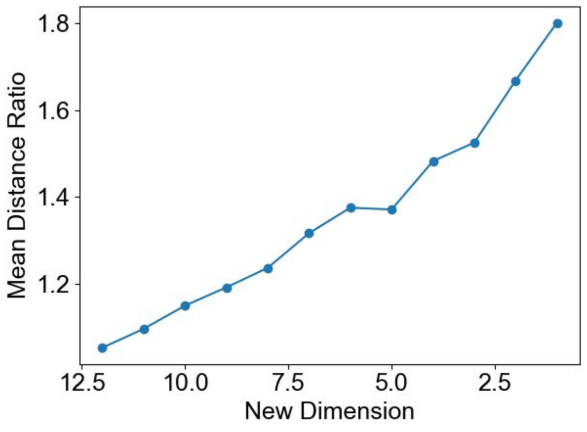
Randomized Projections

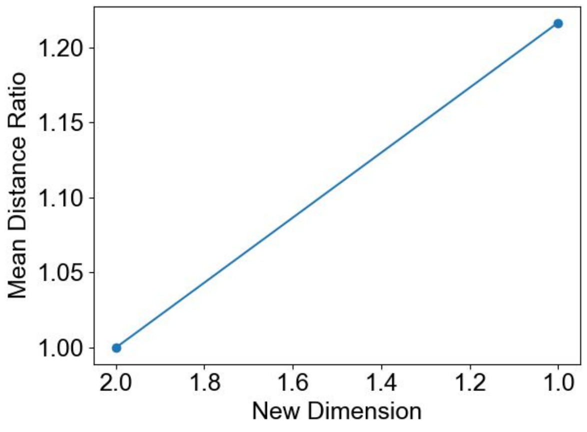
When thinking about randomized projects as a tool for dimensionality reduction, my first question was, "what dimension can I reduce to?" I wanted to keep the dimensionality reduction truly unsupervised by not looking at the labels or introducing a classifier into the feedback loop. One of the key assumptions about random projections is that they are distance preserving. In other words, the distance between a pair of points in one space should be preserved through the transformation to the lower dimensional space. Obviously some amount of information loss is unavoidable when projecting onto a lower dimensional space but I thought that maybe I could use distance preservation as a metric to indicate *how much* information loss really occurred as a result of a random projection.

To this end, I implemented an experiment where I chose a random sample of data point pairs and for each pair, I found the ratio of their distance in the original dimension over their distance in the lower dimension. Just as before, I chose to use Euclidean distance because both my datasets have real-valued (float) inputs. I then simply calculated squared error between the mean of these ratios and 1 (perfect distance preservation) to see how well distance was preserved through the projection.

With my new distance preservation metric in hand, I was now equipped to search over all possible lower dimensions to find out "how low I could go." I chose to use scikit-learn's GaussianRandomProjection module. In the spirit of randomization, I also implemented randomized restarts by feeding GaussianRandomProjection 5 different random seeds and taking the mean distance ratio with the lowest squared error. This smoothed out the distance preservation graphs a bit.

Figure 5 - Distance preservation ratio as dimension of new space decreases





WQ dataset: $R^{12} \rightarrow [R^{12}, R^{1}]$ **GB dataset: $R^{2} \rightarrow [R^{2}, R^{1}]$**

Figure 5 shows that as we project into smaller and smaller dimensional spaces, the mean distance ratio suffers more and more. For both datasets, projecting into a random space with equal dimension preserves distance extremely well. The ratio is very near 1. However let's say that for example we desired to maintain a mean distance ratio of no greater than 1.2 and no less than 0.8, in other words, a 20% difference in mean distance over the sample of data point pairs through the projection. Then we would not be able to reduce the GB dataset dimension at all because the best random $R^2 \rightarrow R^1$ projection yields a mean distance ratio of 1.22. This rather large difference makes sense given that we're halving the dimension. Performing dimensionality reduction on a 2-dimensional dataset is often a rather silly exercise anyways and again highlights the difference between the generated and real-world datasets. Using the same 20% threshold, we could project the WQ dataset like $R^{12} \rightarrow R^9$ because this yields a mean distance ratio of 1.19. This example indicates that a 25% dimensionality reduction only retains 80% of the information as measured by mean distance ratio. This is much worse than say PCA which was able to reduce down to R^1 while still explaining 99.9% of the variance in the WQ dataset.

Truncated Singular Value Decomposition

For the last dimensionality reduction technique, I chose Truncated Singular Value Decomposition (SVD) mainly because it is another technique that does not require us to "peek" at the labels in order to do its work. To me, this is a very desirable characteristic because, in a machine learning context, it does not introduce the often slow learner into the dimensionality reduction process. Though Truncated SVD is often associated with natural language processing in which case it is known as Latent Semantic Analysis (LSA), it can be applied to my datasets effectively as well. It is like PCA in that it uses SVD however it is unlike PCA in that it does not center the data first. I used scikit-learn's TruncatedSVD module. I first ran TruncatedSVD on the WQ dataset and found that the first component explained 99.7% of the variance. This is very similar to my PCA result which makes sense given the similarity between the two approaches. I then ran TruncatedSVD the GB dataset and found that the first component explained 67.6% of the variance. This again is consistent with my PCA results.

Table 3 - Singular values comparison between PCA and Truncated SVD

	WQ											GB
TSVD	1.7e5	5.5e3	790	280	210	53	17	7.5	6.7	5.8	1.8	233.6
PCA	8.8e4	2.6e3	790	280	62	49	8.2	7.5	6.6	5.7	1.2	229.7

Furthermore, as shown in Table 3, the singular values yielded by Truncated SVD are very close to those yielded by PCA. On these two datasets, the two techniques seem to be almost identical.

3 All Together Now

For the following experiments, I set the number of clusters for each dataset to be the number of unique labels. This way, it would be theoretically possible to produce a clustering that perfectly matched the labels. For each dataset, I looped over every combination of dimensionality reduction techniques and clustering algorithms and wrote down the mean silhouette score, the error against the labels, the number of iterations, and the wall clock time. I again used random restarts for the clustering algorithms however I still chose the clustering by best average silhouette score, not lowest error. So while technically I did “peek” at the lebels this time in order to predetermine the number of clusters, labels were not used to find the best cluster labels.

One interesting problem to solve was how to map the cluster labels to data labels to calculate error. For example if my cluster labels were [1, 2, 2] and my data labels were [4, 3, 3], there is actually zero error because the same clusters of examples have common labels To solve this problem, I first mapped both label sets to 2D arrays where each subarray was a binary array representing cluster membership. So [1, 2, 2] mapped to say [[1, 0, 0], [0, 1, 1]] and [4, 3, 3] mapped to say [[0, 1, 1], [1, 0, 0]]. I then compared the first 2D array with every permutation of the second 2D array and returned the lowest error.

Table 4 - MyKMeans clustering after dimensionality reduction

	WQ				GB			
	Score	**Error**	**Iters**	**Time**	**Score**	**Error**	**Iters**	**Time**
Raw	+0.53	23.5%	100	60.26s	+0.75	0.2%	5	0.66s
PCA	+0.55	23.4%	63	23.99s	+0.68	8.8%	5	0.67s
FastICA	+0.08	20.8%	51	17.85s	+0.65	8.0%	6	0.46s

RandProj	+0.54	23.4%	69	21.41s	+0.65	13.9%	6	0.81s
TruncatedSVD	+0.55	23.5%	66	24.31s	+0.68	8.8%	5	0.70s

There is a lot of data here so I'll just point out a few of the more interesting results. First, we can see in Table 4 that none of the dimensionality reduction techniques significantly affected the silhouette score of the MyKMeans clustering of the WQ dataset except for FastICA which lowered the score significantly. Interestingly, FastICA is also the only technique that lowered the corresponding error, iterations, and wall clock time. In other words, a *worse* clustering produced a *better* classifier.

For the GB dataset, the most interesting observation in Table 4 is merely that all the dimensionality reduction techniques increased the error without significantly reducing the wall clock time. This makes sense because the data was generated in clusters and also didn't have much dimension to reduce in the first place so it was purpose-built to be clustered before any transformation. The one technique that did give a performance boost was FastICA which helped MyKMeans run ~30% faster.

Table 5 - MyExpectMax clustering after dimensionality reduction

	WQ				**GB**			
	Score	**Error**	**Iters**	**Time**	**Score**	**Error**	**Iters**	**Time**
Raw	+0.53	23.5%	100	59.32s	+0.75	0.2%	5	0.68s
PCA	+0.55	23.5%	100	59.03s	+0.68	8.8%	5	0.74s
FastICA	-0.08	17.3%	2	1.21s	+0.37	14.5%	2	0.27s
RandProj	+0.54	23.4%	73	42.26s	+0.65	13.9%	6	0.79s
TruncatedSVD	+0.55	23.5%	100	57.7s	+0.68	8.8%	5	0.67s

The MyExpectMax results displayed in Table 5 tell a very similar story to those of Table 4. Again, FastICA drastically reduced the mean silhouette but significantly improved the error, iterations, adn wall clock time. This time the improvements were much more pronounced. Most of the other techniques maxed out at 100 iterations (they may have gone much longer) but MyExpectMax was able to converge after 2 iterations after the data had been transformed by FastICA. For the GB dataset, MyExpectMax again benefited from FastICA with respect to iterations and wall clock time but suffered greater error for similar reasons as for MyKMeans.

4 Neural Network on Dimension-Reduced Data

In this section, I trained a neural network on data that was first transformed by each of the dimensionality reduction techniques. I chose the WQ dataset and scikit-learn's MLPClassifier. I also reproduced the raw (un-transformed) results for this dataset from Assignment 1 for comparison. Because dimensionality reduction in this context should be a neural network performance booster, I wrote down several performance metrics in addition to the test and training set accuracies. Just as in Assignment 1, I performed a grid search for the best hyper-parameters for each newly transformed version of the WQ dataset. The grid search varied over hidden_layer_sizes, learning_rate_init, alpha, and max_iter.

Table 6 - Neural Network training and prediction results on dimensionality-reduced data

	Test Accuracy	**Training Accuracy**	**Training Time**	**Iterations**	**Grid Search Time**
Raw	50.5%	50.0%	2.7s	427	65.5s
PCA	45.5%	44.7%	1.3s	135	51.3s
FastICA	58.9%	58.4%	2.0s	182	185.2s
RamdProj	45.6%	44.8%	0.5s	83	68.1
TruncatedSVD	45.5%	44.7%	3.0s	500	46.2s

Table 6 shows the results for test and training set accuracies as well as the performance metrics. First, note that the benchmark test accuracy on raw data is 50.5% which is not good but not as bad as it appears because the WQ dataset has 11 classes which means random chance is 9.1%. The most surprising result to me is that FastICA actually significantly improved both the training and test set accuracies. I had expected to outperform the other techniques because it didn't *actually* reduce the dimension, but I did not expect ~8% improvement from the raw data. Apparently, the neural network is able to perform much better when components are more statistically independent. It also did so in 26% shorter training time. The rest of the dimensionality reduction techniques' results matched my expectation more closely. Each improved the training time while suffering a little in accuracy. This makes sense because lower dimensional data result in faster training but the transformation always has some information loss. GaussianRandomProjection (RandProj) did particularly well, achieving an 81% reduction in training time. This is because random projection doesn't have to do a lot of the heavy lifting (e.g. SVD) but instead simply projects into a random lower dimensional space.

5 Neural Network on Cluster-Reduced Data

For this last section, I first used my clustering algorithm implementations to transform the 12 features of the WQ data into a single feature, cluster label. I then used these cluster labels to predict the data labels in two ways. First, as a benchmark, I performed a brute force search for the best 1 → 1 mapping of cluster labels to data labels as I described before. Second, I used the cluster labels (inputs) paired with the original data labels (outputs) and fed them to MLPClassifier and performed grid search for the best hyper-parameters as usual. In other words, I used my clustering algorithms as 12 → 1 dimensionality reduction techniques to preprocess the data. For both these methods, I used only data from the training set. While clustering, I “wrote down” the centroids that corresponded to the cluster labels. I then assigned cluster labels to the test set data by checking which of these centroids each test set point was nearest to. For this, I used the same Euclidean distance measure as before. I was then equipped to make test set predictions using the test set cluster labels as inputs in the same two ways (brute force search for 1 → 1 mapping and MLPClassifier).

Table 7 - Results from predicting data labels from cluster labels

	Test Set Accuracy		**Training Set Accuracy**	
	Brute Force	**Neural Network**	**Brute Force**	**Neural Network**
Raw	NA	50.5%	NA	50.0%
MyKMeans	76.6%	45.5%	76.5%	44.7%
MyExpectMax	76.6%	45.5%	76.5%	44.7%

Table 7 shows that for the WQ dataset, simply finding the best 1 → 1 mapping between cluster labels and data labels results in much better predictions than a neural network trained on those same cluster labels. This is quite interesting especially considering I didn’t even let the clustering algorithms “peek” at the data labels. Had I searched for the best clustering by minimizing error instead of maximizing average silhouette score, I suspect the accuracies might be even better! What’s more is that the brute force method wasn’t all that costly. It took m! calculations where m is the number of classes. In a 7-class dataset, this only 5,040 calculations. Remember that I actually did “cheat” a little bit by counting the number of unique labels actually present in the data (7) and fixing the number of clusters to that value. So far in this course, I’ve yet to encounter a scenario where neural networks really shine. Had I chosen to use images or another type of dataset with very high feature count, I may have been better able to highlight their strengths in my experiments. It makes sense that the neural network prediction accuracy suffered a bit when using cluster labels because, just like all the other dimensionality reduction techniques, there is inevitably some loss of information.