CS 4786 - Competition 1

Bhai Jaiveer Singh, Alex Ueki, Divyansh Garg, Ryan Curtis

November 6, 2017

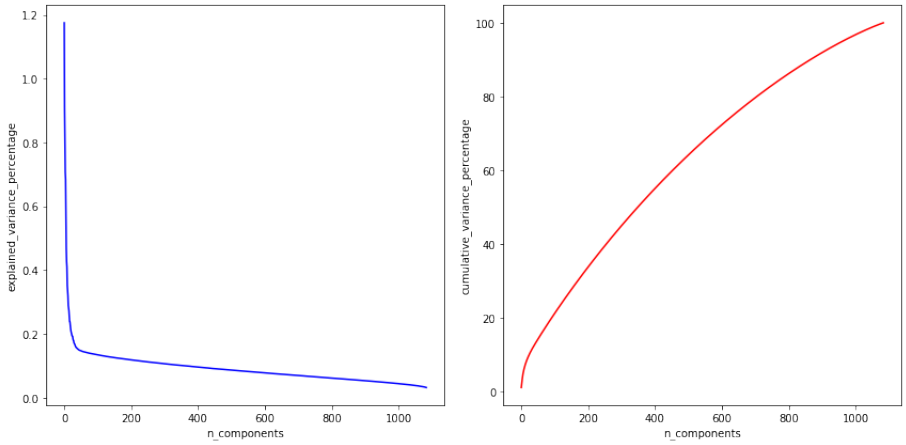
1 Overview of Approach

The model we developed to assign labels mostly revolved around a two-step process. First, it involved extracting and then combining the data from the two inputs: the extracted features and similarity graph. Second, it required clustering the resulting data into 10 clusters to be labeled. Initially, we tried just using PCA and spectral embedding to extract information from each of the inputs respectively, followed by CCA to combine the data and then K-means to perform the clustering. Throughout the competition, we adjusted this model by changing the clustering algorithms, and attempting to reduce noise in the input data. We also completely changed the way we extracted the information, instead choosing to use two versions of spectral embeddings, each of them based on one of the initial data sets, so that we were essentially combining information from two similar views. This led us to ultimately use a model that reduced noise from the similarity graph and feature vectors prior to the CCA and then using a Siamese Neural Network (SNN) to do the clustering.

2 First Attempts

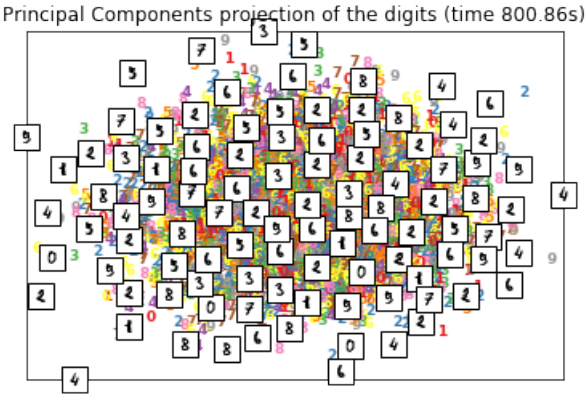
Our preliminary implementation focused on extracting information from the extracted features dataset using PCA, extracting information naively from the similarity graph using Spectral Clustering, and then using Canonical Correlation Analysis (CCA) and clustering to produce labels. We expected this manner of combining information using CCA to work because the similarity graph and extracted features data were taken from different views of the same data points. So using CCA would be useful because CCA looks for the dimensions with the highest correlation in a scale-free manner, and hence it would be able to pick out the commonalities between the two views, and hence hopefully allow us to reduce some noise in the data set.

With this initial approach, the first thing we did was apply PCA to the feature vectors in order to extract information from them. This was done because PCA picks a new dimensionality to project the data on to in a way that maximizes the variance. Since the original data had 1084 features, we thought this would be a good way to remove some noise from the data, reduce the dimensionality, and at the same time figure out which were the most important dimensions that could explain the most variance. Using PCA, we created the following visualizations:



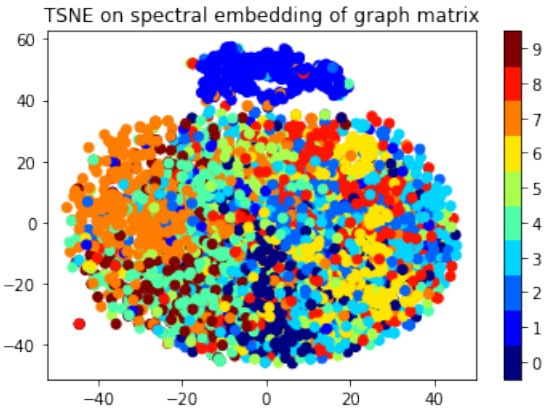
Based on these graphs, we understood that around the first 20 dimensions accounted for a high explained variance percentage, and so if we were to rely on just PCA, k = 20 would be a good choice. This was confirmed by looking at the top eigenvalues, which started at around 12 but after the first 20 or so elements, dropped significantly and then stayed low. However, the cumulative variance percentage visualization showed that PCA alone would not be sufficient to extract information from the data set because the cumulative variance percentage graph had no kink as such, and increased a lot with number of components, meaning that reducing dimensionality might not be as useful as we first thought and that we were actually losing a lot of information.

To evaluate how effective just PCA was, and how effective extracting information from just the extracted features data set was, we then used a simple k-means clustering and then relabeled the clusters according to how far the cluster centers were from each seed of a particular digit. Doing this very basic analysis we learned that most of the seeds of the same digit were not put in the same cluster, and so, there was a lot more noise in the data that could be removed from the naive approach of using just PCA. We also made the following visualization based on just the PCA projections, where the seeds are the highlighted digits:



The visualization clearly indicates that PCA alone did not provide any clear clusters of digits

The next step was to evaluate the effectiveness of extracting information from only the similarity graph (G), using spectral embedding. We first computed the spectral embedding of G and then used the 6000 points in both F and the embedding of G to apply CCA and obtained weights corresponding to the projected space. Using them, we were able to project all 10,000 points in F to the projected space, to produce a new data set (F_{CCA}).
Using S, we then tried to apply a clustering algorithm (initially K-means) to create 10 clusters. Given the 60 label seeds, we assigned labels to the 10 clusters and the last 4000 points were labeled based on the cluster they belong too. This initial attempt resulted in poor clustering results.



This showed that in order to use spectral embedding, we would definitely need to enhance the similarity graph in some way, or come up with our own affinity matrix, as the simple adjacency

matrix was not expressive enough. Similar to using just PCA on features, in this clustering also most of the seeds of the same digit were not put in the same cluster, indicating that the similarity graph had a lot of noise in it, and so, later in the report you will see many variations of the affinity matrix we made to improve it.

Now that we had tested performance on just the features, and just the graph, with an initial attempt at combining the two views using CCA. We expected to get good results by combining the two views, the similarity graph and extracted features, because CCA is predominately used to combine two different sources of data that contain redundant information. Ultimately, this did not give us the results we were looking for. First of all, we later learned that the similarity graph and extracted features are based on separate views of the original data, not necessarily carrying redundant information. Second, the similarity graph as well as the feature vectors appeared to have a lot of noise, and these naive methods of extracting information were losing a lot of the expressiveness of the original data set, so when used with the CCA this was giving inaccurate results.

3 Developing the Final Model

3.1 Extracting information from similarity graph

To extract reasonable information from the similarity graph we tried a variety of approaches. We tried to use a bunch of community detection algorithms, hoping to to use the detected communities as natural clusters resulting from the similarity graph. But this approach was not successful, as we had to deal with a 6000*6000 dimension matrix, and most of the above algorithms either were failed to give proper outputs or had too slow of a runtime.
After this, we tried to look at some more intuitive ways to extract information from the similarity graph. We came up with a nice idea to calculate the exponential of the adjacency matrix (not elemental wise exponentiation but of the whole matrix).
And use it as a similarity matrix for the data-points.

Our logic to this was as follows: Let A be the adjacency matrix.
Suppose B = A^2, then $B[i, j]$ =
Number of paths from i to j such that there is a single in-between node (say k).
From there on, it is not difficult to generalize that for C = A^n, $C[i, j]$ = Number of paths from i to j such that there are exactly n-1 in-between nodes.

Now consider what happens for e^A,
$e^A = \sum_{n=0}^{\infty} \frac{A^n}{n!}$.

This means that e^A[i,j], is the sum of all paths from i to j such that long paths have been penalized and given lower weights than shorter ones.
This naturally expresses something about the connectivity of the graph. i.e. if two nodes are close or have a lot of mutual connections than we want them to have high similarity and vice-versa.
From the above, it can be concluded that e^A, is a similarity matrix for nodes in A.

One of the reasons we went for the above method is that it beautifully captures the similarity information of the graph in terms of e, which allows us to use simple libraries to compute it, without doing something very complicated. Another reason was, for the adjacency matrix, it is simple to compute its exponential as it is diagonalizable.

As if A = VQV^T, then $e^A = Ve^QV^T$ and we only need to exponentiate the diagonal elements of Q.
It was a surprise to us that the method worked well enough that performing spectral clustering on e^A and then doing CCA with the extracted features to obtain predictions was suddenly increased our accuracy to 70% from 50% based on previous approaches.

We went along with retaining 500 of the components of the exponential adjacency after performing spectral clustering.

This choice was experimentally determined by using metrics like homogeneity and completeness score (Part of Scikit's library) and finding the number of components from spectral embedding that gave the highest score.

3.2 Extracting information from feature vectors

To extract information from the feature vectors, PCA was run on the set to get the first 20 principle components, producing K_{20}. Then an adjacency matrix F_{adj} was created for all 10000 feature vectors in K_{20}, following the formula $F_{adj}[i,j] = exp(-(d(F_{20}[i], F_{20}[j])^2))$, where $d(x,y)$ is the linear norm of $x-y$. Finally, we ran spectral embedding on F_{adj} to produce F_{spec} for the first 500 features.

We felt that this method was good because this way we got a spectral embedding of the data based on just the feature vectors, not the similarity graph, and so when we later combined it with the spectral embedding from just the similarity graph, we essentially had two different representations of "distances" between points based on completely different views, so we were able to find correlations between 2 low dimensional representations as desired for a good CCA output.

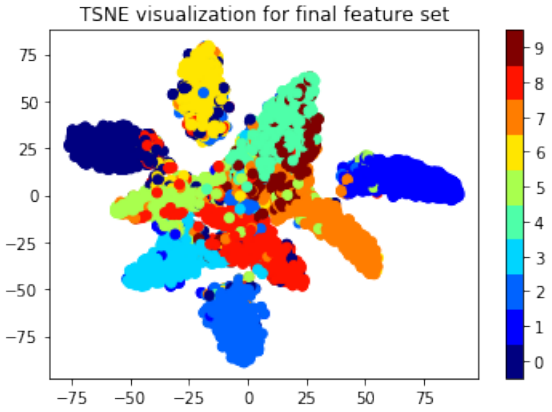
We felt that this was a better method than using just PCA to extract information because it retained a lot more information about the data set, and more importantly, spectral embedding in this manner helped us extract information about how the points were related to each other, based on their distances, not just their projections in each dimension as PCA would have provided. We chose to calculate distance based only on the first 20 dimensions in order to reduce noise, so we did PCA first to reduce the dimensionality before calculating the distances. The exact k=20 was determined by looking at the PCA visualizations which showed a kink in the explained variance graph around k=20, indicating the top 20 dimensions are a lot more important. The exact distance function was chosen because of the nice properties of the exponential function.

We chose first 500 features for spectral embedding because for an expressive enough spectral embedding you need to pick a high enough dimensionality, without picking a dimensionality so high that we lose out on some of the noise clearing benefits. This choice was also experimentally confirmed in a similar manner to the other spectral embedding, by using metrics like homogeneity and completeness score (Part of Scikit's library) and finding the number of components from spectral embedding that gave the highest score.

On a side note, Kernel PCA was explored as a potential alternative to regular PCA. The results were not helpful in expressing the data, and time would have to be spent exploring the hyper-parameters. We opted to stick with regular PCA.

3.3 Combining information from both views

To combine the information from the feature vectors and similarity graph, CCA was used to correlate the data views into a lower dimensional space. Using a cross-validation library PyRCCA, we were about to compare the 500 components of F_{spec} (for the first 6000 elements) and 500 components of EXP_{spec}. Cross-validation projects data points from one set to another and computes what number of components give the best match. The optimal number of components found was 8, so weights were obtained corresponding to the 8-dimensional projected space. Using these weights, all 10,000 points in F_{spec} were projected to create F_{CCA-8}. F_{CCA-8} was then used to cluster the data.



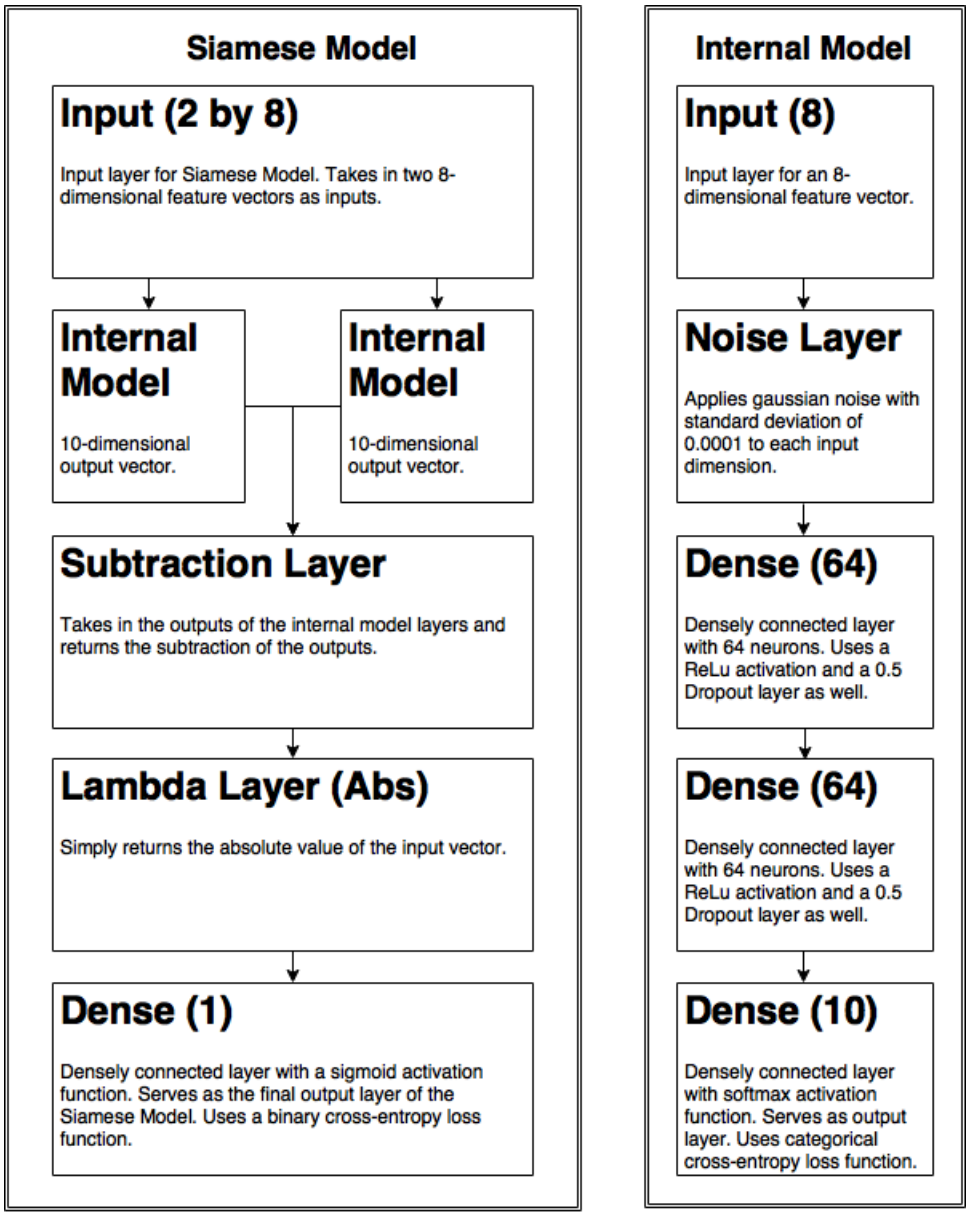
The above graph is a TSNE visualization for the final feature set of 10,000 points obtained using the above methods. It is visible that there is a clear separation between the clusters, which further authenticates on the validity of the above method

3.4 Clustering

Our clustering method used a **Siamese Neural Network** (SNN) to predict similar labels between two representations of points. Specifically, the SNN attempts to create a function $S(x, y)$ such that $S(x, y) \approx 1$ if x and y have the same label. Our SNN uses two identical internal models F that would try to map a representation of a given point x to an array y. y is a 10 element array where each index i is the model's prediction for the x being of label i. The SNN takes in a pair of points x_1, x_2, produces $F(x_1) \to y_1, F(x_2) \to y_2$, and compares the outputs for similarity $S(y_1, y_2) \in [0, 1)$. $S(y_1, y_2)$ was computed by taking $y_{diff} = |y_1 - y_2|$ and passing y_{diff} into a final output layer.

3.4.1 SNN Architecture

Below is a diagram of our SNN structure. The code to implement this model was implemented in models.py under the SiameseModel class.



3.4.2 Training the SNN

To train the SNN, the 60 labeled points in *Seed.csv* were used to produce our training data-set by creating a 60x60 matrix $SEED$ where each index pair $SEED[i, j] = 1$ if i and j have the same label. Each i and j were also mapped to the given feature representation. Then randomly selected inputs feature pairs and output values from $SEED$ were used in training the SNN. The validation set was our whole training set. The training batch size, steps, and epochs were 256, 10000, and 10 respectively, meaning the SNN saw 25,600,000 input-output sets of training data during training. Additionally, a small amount of gaussian noise $std = 0.0001$ was added to all features of each input, to help prevent over-fitting.

3.4.3 Validating the SNN

The SNN needed to pass two property checks after training. The first that data-points are similar to themselves, i.e $S(x, x) = 1$. The second is that similarity is symmetric, $S(x, y) = S(y, x)$.

Due to the structure of the SNN, in particular that identical nature of the internal models, these properties only failed if the implementation of the SNN was incorrect.

3.4.4 Label Predictions

Predictions of labels was done by comparing each of the 10000 data points to each of the 60 labeled seeds. For each given data point, a label was assigned based on maximum similarity, i.e if point x was most similar to seed s, the label of s would also be assigned to x. This was done for all 10000 data points, the last 4000 of which would be submitted.

3.4.5 Strengths

- Naturally labels seeds correctly, no need to try to map labels.
- Training set is based on the square of the labeled dataset, so from the 60 seeds the training data was actually 1770 unique pairs. Note that while these are 3600 possible pairs, 1830 of these pairs are redundant and were only useful to validate properties of the SNN.
- Able to handle non-linear clusters. Even if the seed were distant from each-other in an n-dimensional space, the SNN would still be able to cluster them.

3.4.6 Weaknesses

- Despite the quadratic training set size, training set was still too small to prevent over-fitting.
- Labeling completely dependent on seeds. If a seed was to be an outlier, it would skew training and predictions.
- Non-deterministic results because training data is randomly selected. Results on the same data will be similar, but not exactly the same. Our submission will include the weights used to produce our best result.

4 Failed Approaches, Oversights, and Future Work

4.0.1 Supervised Prediction from Labels

One strategy our team wanted to use was supervised prediction with a neural network given labels with a high degree of confidence. We would use the first 6000 points labeled using our SNN, then train a model M to predict based on some representation of our feature vectors, to finally submit predictions for the last 4000 points. However, we couldn't quickly produce such a model M that could do accurate predictions given labeled data, so we scrapped this approach. The problem was probably with how we reduced our data, but future works should focus on generating such an M that could accurately predict based on a representation of features, and also how to create such a representation of our features. Additional, M would be useful as the internal model of our SNN, because it could better distinguish between features. This was likely the major setback of our project.

4.0.2 Over-reduction of features

Our method involved using a PCA-reduced representation of our feature vectors K to produce an adjacency matrix K_{adj}, which was then itself reduced via spectral embedding. Using two dimensionality reductions on the same data may have lost too much important information about the data. A revised method would probably create a distance matrix on the original features vectors, but using a different distance function that doesn't result in very large distances, which is common in a 1084-dimensional representation, being converted to non-existent values.

4.0.3 Failed clustering methods

Before using the Siamese Neural Network, we tried out a few different clustering methods. The first one we tried out was k-means. With k-means itself we used a variety of approaches. One approach was to make each of the initial clusters centroids the euclidean mean of all of the seeds of a particular digit. However, we found that because there were only 60 seeds, these centroids would shift so much over the k-means implementation that in the end the initial seeds themselves were in completely different clusters.

A different k-means method we tried out was to initialize centroids randomly, allow k-means to run till convergence, and then relabel the clusters according to the correct digits. This relabeling was tried in 2 ways - 1) By checking which cluster had the most number of seeds of the same digit, and assigning that particular digit as the label of the cluster. This method failed because too often, there would be either the same number of seeds in a cluster from different digits, or some clusters left at the end with no seeds in them at all. 2) Relabeling the clusters according to distances of the cluster centers from the euclidean mean of all of the seeds of a particular digit. This method was better than the first method and more robust, and used henceforth to relabel the clusters.

The effectiveness of the clustering methods were evaluated using metrics like homogeneity and completeness score (Part of Scikit's library), and we fine tuned the parameters for clustering by comparing these scores. We also created a method to check the "purity" of a cluster, which compared the clusters to the known labels of the seeds, and determined how good the clustering was independent of the actual labels assigned to the clusters. Note that the SNN naturally has a very high purity score

These methods generally showed that k-means was not a great clustering method for this. The most likely reason is that k-means looks for circular clusters, whereas the data was likely to consist of more ellipsoidal clusters, especially given that PCA revealed that some eigenvalues were much larger than others, indicating that some directions were more squished than others.

So we tried ellipsoid and gaussian mixture model for clustering as well. Gaussian mixture model is effective in finding ellipsoidal clusterings, and the added benefit was that it provided soft clustering, and this actually lead to slightly better results than k-means, as determined by the metrics mentioned above. However, these clusterings were not as good as the SNN, and so we did not use them in the final model.

One of the main reasons we believe these clusterings didn't work as well as the Siamese Neural Network is that there were too few seeds to use them effectively to generate labels for the clusters. We kept falling into the trap of over-fitting to these few seeds, and we even tried to alleviate this by extending our seeds, as described below.

4.0.4 Attempt to extend the seed set

In order to improve how we relabeled our clusters, we attempted to extend the set of seeds by adding data points to it if we were relatively sure of their labels. Our method for doing this was label propagation, which is a form of semi supervised learning that tries to extend a labeled set based on a similarity graph, and then clamps the soft labels it assigns, repeating this until convergence. Although this method seemed promising, even the new labels it provided with a high degree of confidence often did not correspond at all to the labels we had gotten for the same points using our other methods, which had already proven to be at least 70% accurate. The most likely reason for this was that we were propagating the labels based just on the information extracted from the similarity graph, and so we were propagating labels based on only 1 view of the data, hence leading to inaccurate labels. An improvement to this would have been using label propagation on a similarity graph that encodes information from both the views.

4.0.5 Alternative spectral embedding affinity matrices

Before deciding on using the affinity matrices described above for spectral embeddings, we tried a few different ones as well. One of the ones we tried was formed by first creating the euclidean

distance based matrix, following the formula $F_{adj}[i,j] = exp(-(d(F_{20}[i], F_{20}[j])^2))$, and then just finding the dot product of this matrix and the matrix based on the simple adjacency matrix formed from the initial similarity graph consisting of only 1 and 0 values indicating an edge. The reason this matrix did not provide a good spectral embedding was that it prematurely combined information from the two views, and so when we later tried to use CCA on this spectral embedding and any other low dimensional representation based on just 1 of the views, we got poor results, because the two inputs to CCA were not truly from 2 different views.

Another affinity matrix we created was based on a measure of distance determined by how many edges are in the shortest path between two nodes, with the edges being those from the similarity graph. Although this similarity matrix was actually quite expressive and effective, it was ultimately outperformed by the exponential affinity matrix described in the "Extracting information from similarity graph" section, and so was not used in the final model.

4.0.6 Alternative for reduction of features

Our team did not explore any other method to reduce the feature vectors. PCA was the most accessible method, but we found that it would remove lots of the explained variance in the data unless we took large numbers of components. We discounted some methods like random projections, but one interesting method that was not considered was using an auto-encoder. This was not covered in class, but could potentially have found representations of features that would lose minimal amounts of information.

4.0.7 Training the SNN

The training method used on the SNN did not completely separate the validation and training data. As a result, the validation results were inflated and may have over-fit the model.

4.0.8 The Fat Siamese Neural Net

One attempted method at increasing our training set size was to take linear combinations of each seed with the others of the same label to produce a new seed. This was done by taking all combinations, from 2 to 6 elements, of a given label and adding the centroid as a new seed. We dubbed this a Fat SNN. This did not improve accuracy, which is reasonable considering that using linear combinations goes against a previously mentioned strength of the SNN: its ability to handle non-linear clusters.

5 Conclusion

While there was clearly room for improvement, as well as room for trying out other methods, we feel that our method was quite effective overall and derived from a strong understanding of principled approaches of extracting information from each of the data sets, and we feel that we were able to effectively combine the information from each of the data sets to reveal new information about the data points. We were also able to use visualizations and metrics effectively to test and iteratively improve our model to arrive at our final version, and we also developed some new methods, tweaked existing ones, and tried out methods not discussed in class - such as the Siamese Neural Network, and our method for creating the affinity matrix used for spectral embedding.