# Assignment_4

March 20, 2019

## 1 Assignment 4 - Unsupervised Learning and More Supervised Learning

### 1.1 *Anna Berman*

Netid: *aeb100*

### 1.2 1

#### 1.2.1 [35 points] Clustering

Clustering can be used to determine structure, assign group membership, and representing data through compression. Here you'll dive deeply into clustering exploring the impact of a number of classifiers on

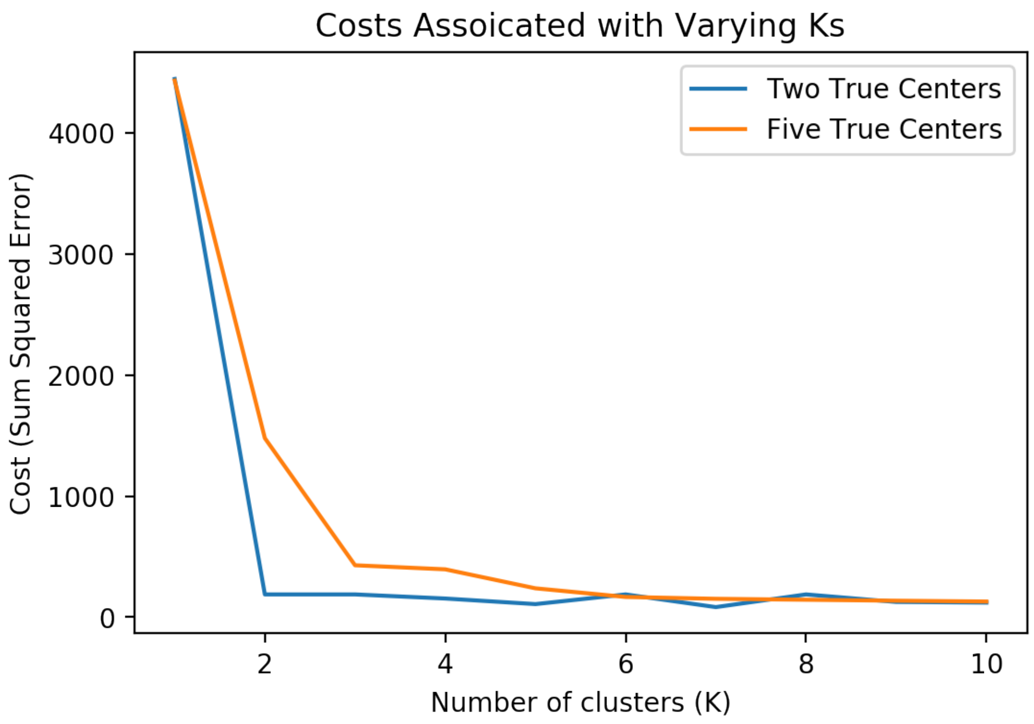
**(a)** Implement your own k-means algorithm. Demonstrate the efficacy of your algorithm on the blobs dataset from scikit-learn with 2 and 5 cluster centers. For each implementation rerun the k-means algorithm for values of k from 1 to 10 and for each plot the "elbow curve" where you plot the sum of square error. For each case, where is the elbow in the curve? Explain why.

```
In [24]: import numpy as np
         import matplotlib.pyplot as plt
         from sklearn.datasets import make_blobs
         import random

         class Kmeans:
             # k = number of clusters
             # means = centroid
             # clusters = assignments to clusters for each point (predictions)
             # last_cost = cost of the last iteration of clustering
             def __init__(self):
                 self.k = None
                 self.means = []
                 self.clusters = []
                 self.last_cost = None
                 pass

             def fit(self, X, k):
```

```python
        self.k = k
        # Select K and randomly initialize k-means values
        self.means = self.first_means(X)

        # Assign observations to the nearest means
        self.clusters = self.assign_clusters(X)
        self.last_cost = self.cost(X, self.means)

        # Update the means to the be means of the centroids
        new_means = self.calc_new_means(X)
        new_cost = self.cost(X, new_means)
        cost_diff = self.last_cost - new_cost

        # Repeat until convergence
        while cost_diff > 1:
            # Reassign cluster means
            self.means = new_means
            # Assign points to new clusters centroids and calculate costs
            self.clusters = self.assign_clusters(X)
            self.last_cost = self.cost(X, self.means)
            # Find new means and difference in cost function
            new_means = self.calc_new_means(X)
            new_cost = self.cost(X, new_means)
            cost_diff = self.last_cost - new_cost
        pass

        # Assign means and cost of last iteration
        self.means = new_means
        self.last_cost = new_cost

    def cost(self, X, means):
        # Calcuate the total distance between assigned points and centroids
        cost_total = 0
        for i in range(self.k):
            # For each cluster find the dist from each point to the center
            slicer = np.where(self.clusters == i)
            assigned_points = X[slicer,:].reshape(X[slicer,:].shape[1],2)
            dists = np.linalg.norm(means[i,:] - assigned_points, axis = 1)
            cost_total += np.dot(dists, dists)
            pass
        return cost_total

    def first_means(self, X):
        first_means = []
        # Initialize k-means values
        for i in range(self.k):
            # For each K randomize a point between min and max of each feature
            # Adjustments (adj) make sure initial means aren't too close to edge
            adj = (max(X[:,0]) - min(X[:,0]))/4
            x0 = random.uniform(min(X[:,0]) + adj,
                                max(X[:,0]) - adj)
            adj = (max(X[:,1]) - min(X[:,1]))/4
            x1 = random.uniform(min(X[:,1]) + adj,
                                max(X[:,1]) - adj)
            first_means.append(x0)
            first_means.append(x1)
            pass
        # Reshape means into nparray
        first_means = np.array(first_means).reshape(self.k,2)
        return first_means

    def assign_clusters(self, X):
        # Assign observations to the nearest means
        self.clusters = []
        for i in range(len(X)):
            # For each point, calculate distance to each mean
            euc_dist = np.linalg.norm(X[i,:] - self.means, axis=1)
            # Assign point to nearest mean
            self.clusters.append(np.argmin(euc_dist))
            pass
        #print(self.clusters)
        return np.array(self.clusters)

    def calc_new_means(self, X):
        # Update the means to the be means of the centroids
        new_means = []
        for i in range(self.k):
            slicer = np.where(self.clusters == i)
            # If there are points assigned to the cluster, find the new mean
            if (np.array(slicer).shape[1] > 0):
                x0 = np.mean(X[slicer, 0])
                x1 = np.mean(X[slicer, 1])
                pass
            else:
                adj = (max(X[:,0]) - min(X[:,0]))/4
                x0 = random.uniform(min(X[:,0]) + adj,
                                    max(X[:,0]) - adj)
                adj = (max(X[:,1]) - min(X[:,1]))/4
                x1 = random.uniform(min(X[:,1]) + adj,
                                    max(X[:,1]) - adj)
            new_means.append(x0)
            new_means.append(x1)
            pass
        # Reshape means into nparray
        new_means = np.array(new_means).reshape(self.k,2)
        #print(new_means)
```

```
            return new_means
        pass
```

```
In [29]: # TWO CENTERS
         X, y = make_blobs(centers = 2)

         # Iterate over 10 K values
         cost_list = []
         for i in range(1,11):
             kmeans = Kmeans()
             kmeans.fit(X, i)
             # Is this supposed to be MSE?
             cost_list.append(kmeans.last_cost)
             pass

         # Plot the costs
         plt.figure(dpi = 200)
         plt.plot(range(1,11), cost_list, label = 'Two True Centers')
         plt.xlabel('Number of clusters (K)')
         plt.ylabel('Cost (MSE)')

         # FIVE CENTERS
         X, y = make_blobs(centers = 5)

         # Iterate over 10 K values
         cost_list = []
         for i in range(1,11):
             kmeans = Kmeans()
             kmeans.fit(X, i)
             # Is this supposed to be MSE?
             cost_list.append(kmeans.last_cost)
             pass

         # Plot the costs
         plt.plot(range(1,11), cost_list, label = 'Five True Centers')
         plt.xlabel('Number of clusters (K)')
         plt.ylabel('Cost (Sum Squared Error)')
         plt.title('Costs Assoicated with Varying Ks')
         plt.legend()
         plt.show()
```



In our visualization, the line corresponding to dataset with two true centers appears to have an elbow at $K = 2$. Similarly, the line of corresponding to the dataset with five true centers appears to have an elbow at approximately $K = 5$.

This is most likely due to the fact that fitting a smaller number of clusters than there are true centers will inevitably produce some amount of error. In this case there will always be points "incorrectly" assigned to a cluster because there simply aren't enough clusters to represent each true center. As we add more clusters, the error will reduce as we get closer to the true fit.

If the true number of clusters is allowed in our fit, the error will, again, most likely reduce and hopefully most of our data points will be correctly assigned to their true center.

However, once we fit more clusters than there are true centers, there will only be slight reductions in error. Additional clusters will most likely not reduce error significantly because the bulk of the variation in features is already captured in the true number of clusters.

Because of this, the elbow of our graph typically represents a good estimate of the true number of centers or clusters in our dataset.

**(b)** Briefly explain in 1-2 sentences each (and at a very high level) how the following clustering techniques work and what distinguishes them from other clustering methods: (1) k-means, (2) agglomerative clustering, (3) Gaussian mixture models, (4) DBSCAN, and (5) spectral clustering

**K-means clustering** is a centroid-based clustering technique. In essence, the k-means algorithm attempts to find a user-specified number of clusters ($K$) by repeatedly assigning points to the nearest centroid, updating the centroid to be the mean of the assigned points and repeating this process until convergence.

**Agglomerative Hierarchical Clustering** is a hierarchical, or bottom-up cluster method. This method starts with each point as a separate cluster and repeatedly merges the two closest clusters until a user-specified number of clusters remains.

**Gaussian Mixture Models** is a distribution-based clustering method that can perform either hard or fuzzy clustering assignments. The GMM algorithm assigns clusters by calculating the maximum likelihood that a point comes from a Gaussian distribution. Essentially, a Gaussian mixture model approach is a K-means approach that does not assume the clusters have identical covariance matrices where all features are independent.

**DBSCAN** is a density-based clustering technique in which cluster assignments are determined by the number of surrounding points within a given distance. The user does not specify the number of clusters using DBSCAN.

**Spectral Clustering** is a clustering method that defines similarity as affinity as opposed to distance. The algorithm constructs an affinity matrix, reduces the dimensionality of the matrix and defines a user-specified number of clusters in the lower dimensional space.

**(c)** For each of the clustering algorithms in (b) run each of them on the five datasets below. Tune the parameters in each model to achieve better performance. Plot the final result as a 4-by-5 subplot showing the performance of each method on each dataset. Which methods work best/worst on each dataset and why?

- Aggregation.txt
- Compound.txt
- D31.txt
- jain.txt

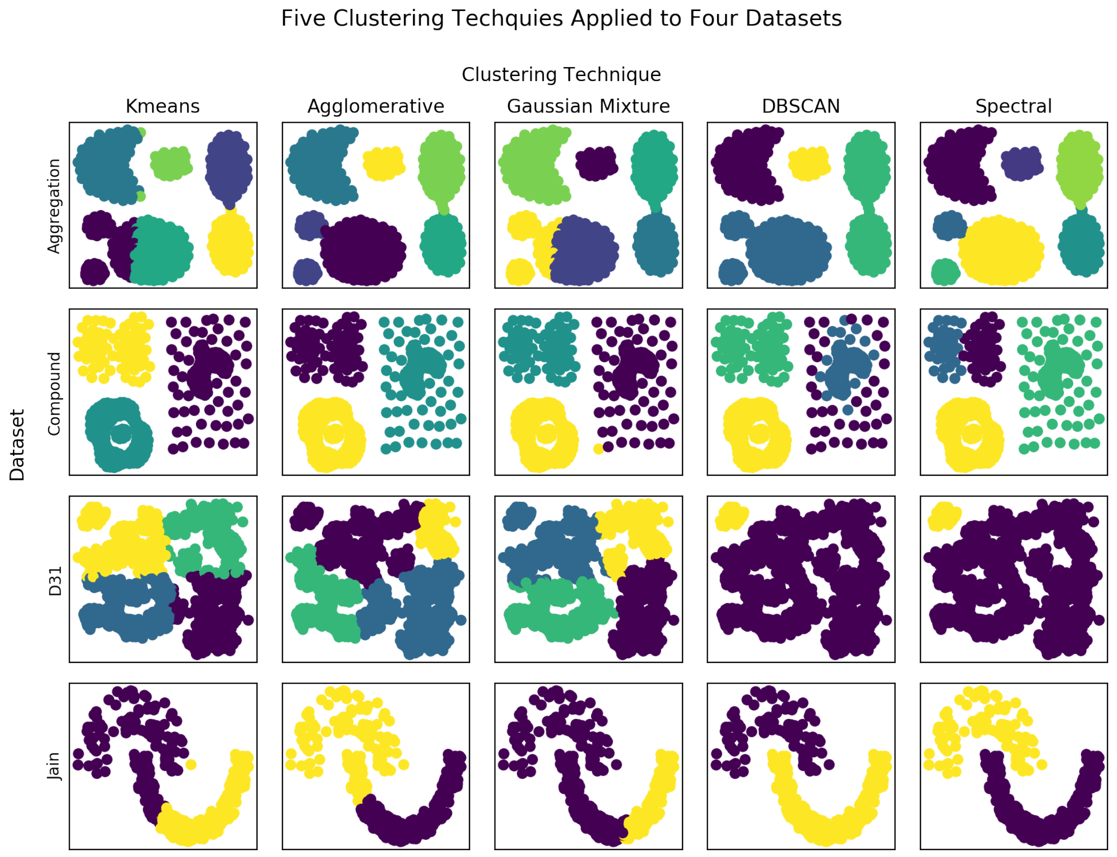
Each file has three columns: the first two are $x_1$ and $x_2$, then the third is a suggested cluster label (ignore this third column - do NOT include this in your analysis). The data are from https://cs.joensuu.fi/sipu/datasets/.

```
In [34]: from sklearn.cluster import KMeans
         from sklearn.cluster import AgglomerativeClustering
         from sklearn.mixture import GaussianMixture
         from sklearn.cluster import DBSCAN
         from sklearn.cluster import SpectralClustering
         import numpy as np
         import matplotlib.pyplot as plt

         # Load data
         agg = np.loadtxt('Assignment Code/Assignment 4/data/Aggregation.txt',
                          delimiter='\t', unpack=False)
         compound = np.loadtxt('Assignment Code/Assignment 4/data/Compound.txt',
                               delimiter='\t', unpack=False)
         d31 = np.loadtxt('Assignment Code/Assignment 4/data/D31.txt',
                          delimiter='\t', unpack=False)
         jain = np.loadtxt('Assignment Code/Assignment 4/data/jain.txt',
                           delimiter='\t', unpack=False)

         # Remove third column
         agg = agg[:,:-1]
         compound = compound[:,:-1]
         d31 = d31[:,:-1]
         jain = jain[:,:-1]
```

```
        # Create models
        # Kmeans
        kmeans_agg = KMeans(n_clusters=6).fit(agg)
        kmeans_comp = KMeans(n_clusters=3).fit(compound)
        kmeans_d31 = KMeans(n_clusters=4).fit(d31)
        kmeans_jain = KMeans(n_clusters=2).fit(jain)
        # Agglomerative
        aggl_agg = AgglomerativeClustering(n_clusters=6).fit(agg)
        aggl_comp = AgglomerativeClustering(n_clusters=3).fit(compound)
        aggl_d31 = AgglomerativeClustering(n_clusters=4).fit(d31)
        aggl_jain = AgglomerativeClustering(n_clusters=2).fit(jain)
        # Gaussian
        gauss_agg = GaussianMixture(n_components=6).fit(agg)
        gauss_comp = GaussianMixture(n_components=3).fit(compound)
        gauss_d31 = GaussianMixture(n_components=4).fit(d31)
        gauss_jain = GaussianMixture(n_components=2).fit(jain)
        # DBSCAN
        dbscan_agg = DBSCAN(eps=3).fit(agg)
        dbscan_comp = DBSCAN(eps=2).fit(compound)
        dbscan_d31 = DBSCAN(eps=2).fit(d31)
        dbscan_jain = DBSCAN(eps=2.5, min_samples=20).fit(jain)
        # Spectral
        spec_agg = SpectralClustering(n_clusters=7,
                                      assign_labels="discretize").fit(agg)
        spec_comp = SpectralClustering(n_clusters=4,
                                       assign_labels="discretize").fit(compound)
        spec_d31 = SpectralClustering(n_clusters=2,
                                      assign_labels="discretize").fit(d31)
        spec_jain = SpectralClustering(n_clusters=2).fit(jain)
```

```
In [35]: # Ploting the results
         fig, axs = plt.subplots(4, 5)
         fig.set_dpi(200)
         fig.set_figheight(8)
         fig.set_figwidth(10)
         # Kmeans
         axs[0,0].scatter(agg[:,0], agg[:,1], c = kmeans_agg.labels_)
         axs[1,0].scatter(compound[:,0], compound[:,1], c = kmeans_comp.labels_)
         axs[2,0].scatter(d31[:,0], d31[:,1],c = kmeans_d31.labels_)
         axs[3,0].scatter(jain[:,0], jain[:,1], c = kmeans_jain.labels_)
         # Agglomerative
         axs[0,1].scatter(agg[:,0], agg[:,1], c = aggl_agg.labels_)
         axs[1,1].scatter(compound[:,0], compound[:,1], c = aggl_comp.labels_)
         axs[2,1].scatter(d31[:,0], d31[:,1],c = aggl_d31.labels_)
         axs[3,1].scatter(jain[:,0], jain[:,1], c = aggl_jain.labels_)
         # Gaussian
         axs[0,2].scatter(agg[:,0], agg[:,1], c = gauss_agg.predict(agg))
```

```
axs[1,2].scatter(compound[:,0], compound[:,1], c = gauss_comp.predict(compound))
axs[2,2].scatter(d31[:,0], d31[:,1],c = gauss_d31.predict(d31))
axs[3,2].scatter(jain[:,0], jain[:,1], c = gauss_jain.predict(jain))
# DBSCAN
axs[0,3].scatter(agg[:,0], agg[:,1], c = dbscan_agg.labels_)
axs[1,3].scatter(compound[:,0], compound[:,1], c = dbscan_comp.labels_)
axs[2,3].scatter(d31[:,0], d31[:,1],c = dbscan_d31.labels_)
axs[3,3].scatter(jain[:,0], jain[:,1], c = dbscan_jain.labels_)
# Spectral
axs[0,4].scatter(agg[:,0], agg[:,1], c = spec_agg.labels_)
axs[1,4].scatter(compound[:,0], compound[:,1], c = spec_comp.labels_)
axs[2,4].scatter(d31[:,0], d31[:,1],c = spec_d31.labels_)
axs[3,4].scatter(jain[:,0], jain[:,1], c = spec_jain.labels_)

# Format axes
axs[0,0].set_title('Kmeans')
axs[0,1].set_title('Agglomerative')
axs[0,2].set_title('Gaussian Mixture')
axs[0,3].set_title('DBSCAN')
axs[0,4].set_title('Spectral')
axs[0,0].set_ylabel('Aggregation')
axs[1,0].set_ylabel('Compound')
axs[2,0].set_ylabel('D31')
axs[3,0].set_ylabel('Jain')
for i in range(4):
    for j in range(5):
        axs[i,j].set_xticks([], [])
        axs[i,j].set_yticks([], [])
        pass
    pass
fig.suptitle('Five Clustering Techquies Applied to Four Datasets', fontsize = 14)
fig.text(0.5, 0.9, 'Clustering Technique', ha='center', fontsize = 12)
fig.text(0.01, 0.5, 'Dataset', va='center', rotation='vertical', fontsize = 12)
plt.tight_layout(rect=[0.03, 0.03, 1, .90])
plt.show()
```



For the **Aggregation** dataset, spectral clustering appeared to work the best while a k-means approach appeared to produce the worst results. Spectral clustering, although slow, generally produces good results primarily due to its encoding of similarity and dimensionality reduction techniques. K-means, however, struggles where there are nonlinear boundaries and in diversity of cluster variance and therefore did not perform as well.

For the **Compound** dataset, spectral clustering again appeared to work the best and was the only method to clearly delineate the two clusters in the top left corner. DBSCAN, however was the only clustering technique able to delineate what appears to be two overlapping clusters which differ in density. Because DBSCAN is a density-based clustering method, this is clearly playing to DBSCAN's strong suits. Although k-means, agglomerative clustering, and Gaussian mixture modeling produced similar results, GMM produced the worst results. This is most likely because GMM struggles when non-normal distributions.

For the **D31** dataset, again spectral clustering and DBSCAN produced the best results for similar reasons as previously discussed. Because these clusters had highly non-linear boundaries, small distances between clusters, and non-normal distributions, k-means, agglomerative clustering, and Gaussian mixture modeling did poorly respectively.

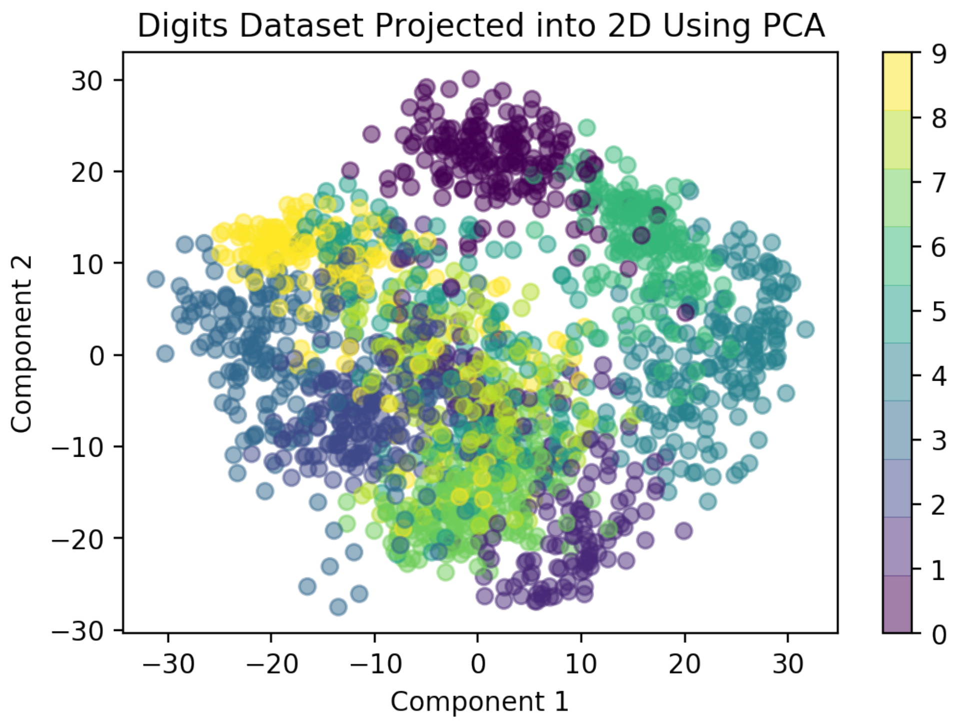
Finally, for the **Jain** dataset, again spectral clustering and DBSCAN produced the best results for similar reasons as previously discussed - the clusters have similar affinity and differing densities. Because these clusters had highly non-linear boundaries, small distances between clusters, and non-normal distributions, again k-means, agglomerative clustering, and Gaussian mixture modeling did poorly respectively.

### 1.3 2

#### 1.3.1 [25 points] Visualizing and clustering digits with PCA and t-SNE

**(a)** Load the scikit-learn digits dataset. Apply PCA and reduce the data (with the associated cluster labels 0-9) into a 2-dimensional space. Plot the resulting 2-dimensional representation of the data.

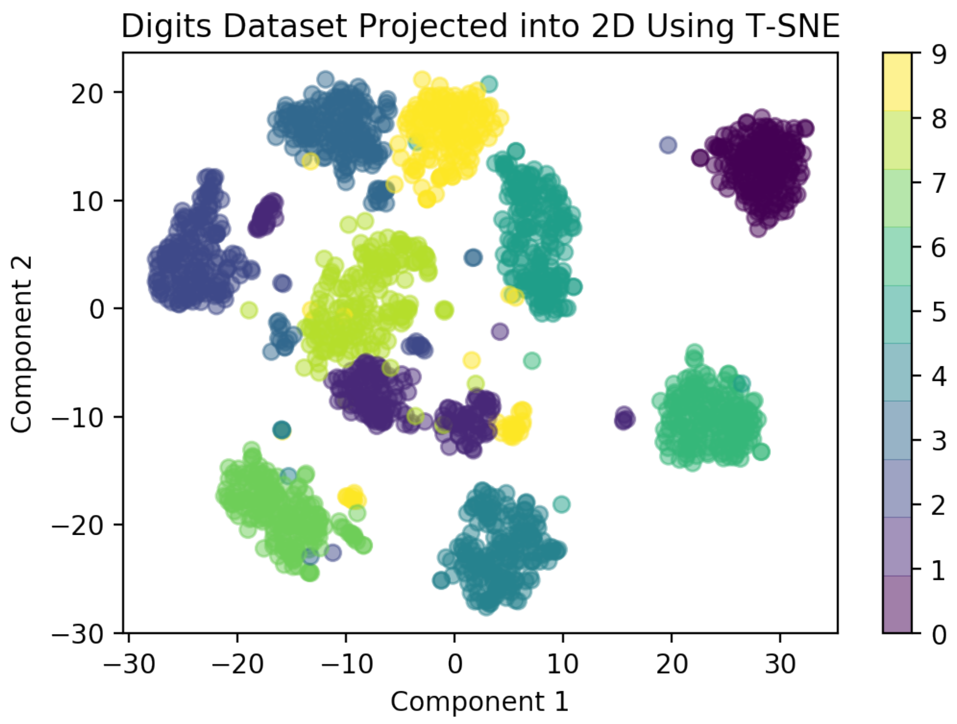
```
In [23]: from sklearn import datasets
         import matplotlib.pyplot as plt
         from sklearn.decomposition import PCA

         #Load the digits dataset
         digits = datasets.load_digits()
         X = digits.data

         # Perform PCA on digits dataset (into 2D)
         plt.figure(dpi = 200)
         pca = PCA(n_components=2).fit(X)
         X_pca = pca.fit_transform(X)
         plt.scatter(X_pca[:, 0], X_pca[:, 1], c=digits.target,
                     alpha=0.5, cmap=plt.cm.get_cmap('viridis', 10))
         plt.xlabel('Component 1')
         plt.ylabel('Component 2')
         plt.colorbar();
         plt.title('Digits Dataset Projected into 2D Using PCA')
         plt.show()
```



**(b)** t-distributed stochastic neighborhood embedding (t-SNE) is a nonlinear dimensionality reduction technique that is particularly adept at embedding the data into lower 2 or 3 dimensional spaces. Apply t-SNE to the digits dataset and plot it in 2-dimensions (with associated cluster labels 0-9). You may need to adjust the parameters to get acceptable performance. You can read more about how to use t-SNE effectively here. A video introducing this method can be found here for those who are interested.

*NOTE: An important note on t-SNE is that it is an example of transductive learning. This means that the lower dimensional representation of the data is only applicable to the specific input data - you can't just add a new sample an plot it in the sample 2-dimensional space without entirely rerunning the algorithm and finding a new representation of the data.*

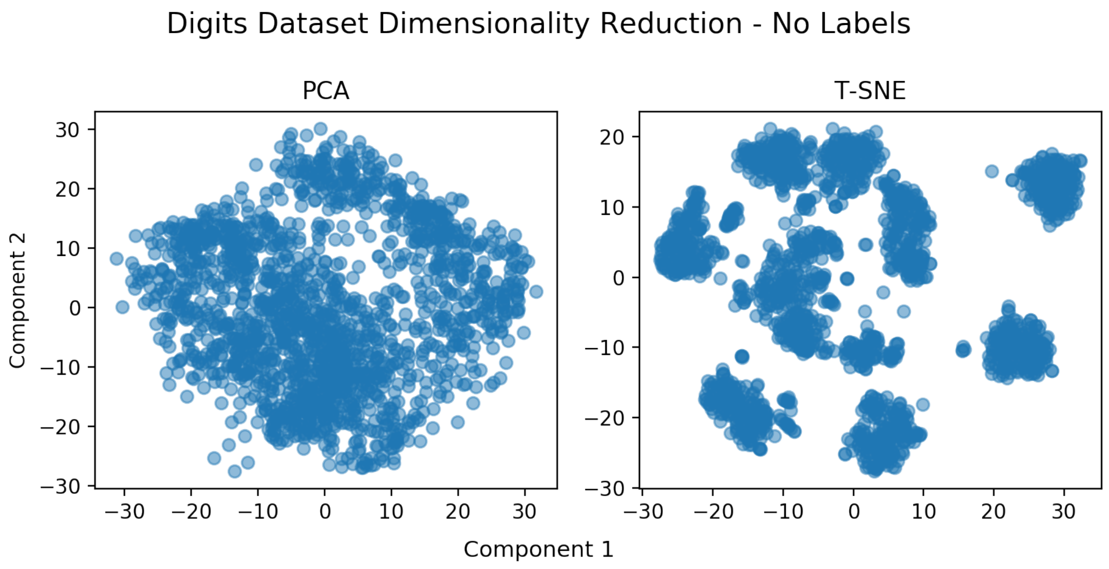
```
In [24]: from sklearn.manifold import TSNE

         # Perform T-SNE on digits dataset (into 2D)
         tsne = TSNE(n_components=2, perplexity=100)
         X_tsne = tsne.fit_transform(X)
         plt.figure(dpi = 200)
         plt.scatter(X_tsne[:, 0], X_tsne[:, 1], c=digits.target,
                     alpha=0.5, cmap=plt.cm.get_cmap('viridis', 10))
         plt.xlabel('Component 1')
         plt.ylabel('Component 2')
         plt.colorbar();
         plt.title('Digits Dataset Projected into 2D Using T-SNE')
         plt.show()
```



**(c)** Compare/contrast the performance of these two techniques. Which seemed to cluster the data best and why? Given the comparative clustering performance that you observed and the note on t-SNE above, what are the pros and cons of PCA and t-SNE? Note: You typically will not have labels available.

T-SNE appears to have clustered the data more successfully than PCA. Specifically, T-SNE created more defined clusters with less overlap compared to PCA. Looking at the clustering without labels below (as we would typically be forced to in most research settings), T-SNE would create a much clearer delineation between classes. That said, T-SNE has several disadvantages. One, T-SNE is much more computationally expensive than PCA which could be a problem for much larger or higher dimensional datasets. Secondly, once T-SNE is run, adding a new sample to the training set would require running the algorithm again and findings an entirely new representation of the data. This is not true for PCA.

```
In [25]: # Plot results of both demsionality reduction techniques without labels
         fig, axs = plt.subplots(1, 2)
         fig.set_dpi(200)
         fig.set_figheight(4)
         fig.set_figwidth(8)
         axs[0].scatter(X_pca[:, 0], X_pca[:, 1], alpha = .5)
         axs[1].scatter(X_tsne[:, 0], X_tsne[:, 1], alpha = .5)
         axs[0].set_title('PCA')
         axs[1].set_title('T-SNE')
         axs[0].set_ylabel('Component 2')
         fig.text(0.5, 0.05, 'Component 1', ha='center', fontsize = 11)
         plt.tight_layout(rect=[0.03, 0.07, 1, .90])
         fig.suptitle('Digits Dataset Dimensionality Reduction - No Labels', fontsize = 14)
         plt.show()
```



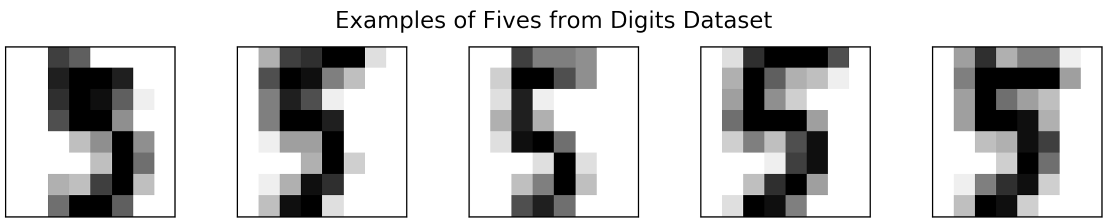
### 1.4 3

#### 1.4.1 [30 points] PCA for compression

From the digits dataset, extract all the 5's. Your going to create a compressed version of one of an image.

**(a)** Plot a number of examples of the original images.

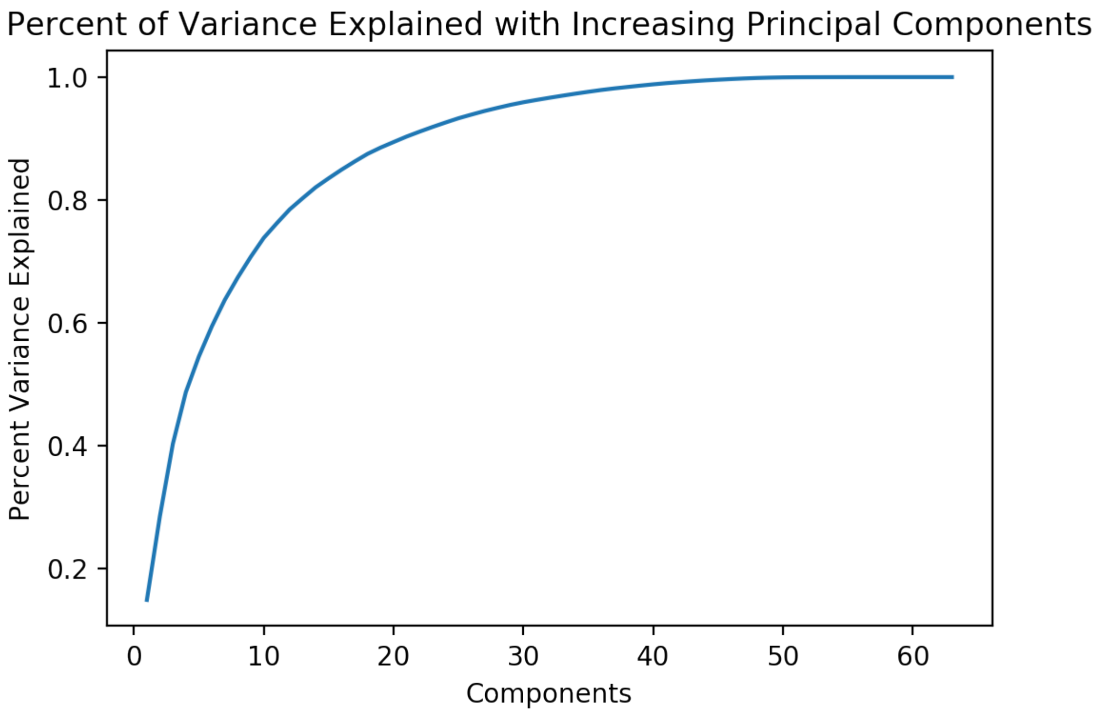
```
In [26]: from sklearn import datasets
         import matplotlib.pyplot as plt
         from sklearn.decomposition import PCA

         #Load the digits dataset
         digits = datasets.load_digits()
         X = digits.data
         fives = X[digits.target == 5]

         # Ploting examples of 5s
         fig, axs = plt.subplots(1, 5)
         fig.set_dpi(200)
         fig.set_figheight(2)
         fig.set_figwidth(10)
         for i in range(5):
             axs[i].imshow(fives[i].reshape(8, 8), cmap='binary')
             axs[i].set_xticks([], [])
             axs[i].set_yticks([], [])
             pass
         fig.suptitle('Examples of Fives from Digits Dataset',
                      fontsize = 14)
         plt.tight_layout(rect=[0, 0.03, 1, .90])
         plt.show()
```



**(b)** Perform PCA on the data. Create a plot showing the fraction of variance explained as you incorporate from 1 to $N$ components.

```
In [28]: # Plot % variance explained with increasing components
         var_explained = []
         components = range(1,fives.shape[1])
         for i in components:
             pca = PCA(n_components=i).fit(X)
```

```
        var_explained.append(sum(pca.explained_variance_ratio_))
        pass

    plt.figure(dpi = 200)
    plt.plot(components, var_explained)
    plt.xlabel('Components')
    plt.ylabel('Percent Variance Explained')
    plt.title('Percent of Variance Explained with Increasing Principal Components')
    plt.show()
```

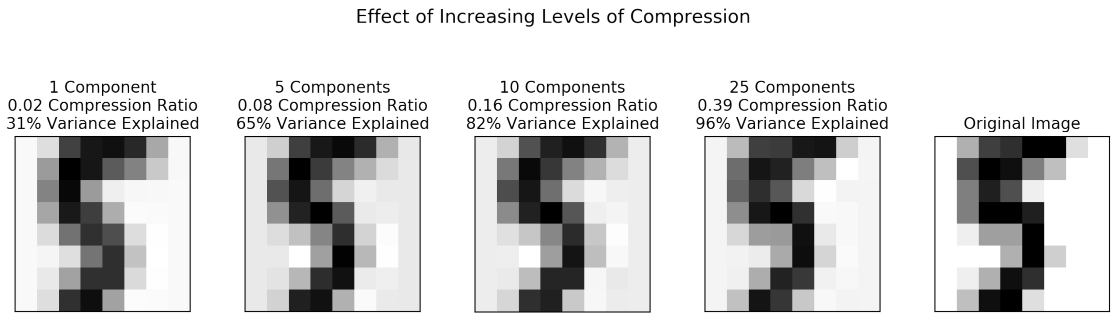


As we increase the number of components, we also increase the percent of variance explained by our principal components until we reach complete variance explained by including $N$ components. However, as we increase the number of components, we reach a point of diminishing returns. Each additional component explains less variance than the one before it.

**(c)** Select an image (from your dataset of 5's) that you will "compress" using PCA. Use the principal components extracted in (b) for data compression: choose the top $k$ principal components and represent the data using a subset of the total principal components. Plot the original image, and compressed versions with different levels of compression (i.e. using different numbers of the top principal components): use $k = 1, 5, 10, 25$.

```
In [48]: # Plot varying levels of compression
         fig, axs = plt.subplots(1, 5)
         fig.set_dpi(200)
         fig.set_figheight(3.5)
         fig.set_figwidth(12)
```

```
# Iterate over different components and appoximate image
components = [1,5,10,25]
for i, c in enumerate(components):
    pca = PCA(n_components=c).fit(fives)
    fives_lowD = pca.fit_transform(fives)
    approximation = pca.inverse_transform(fives_lowD)
    axs[i].imshow(approximation[1].reshape(8, 8), cmap='binary')
    axs[i].set_xticks([], [])
    axs[i].set_yticks([], [])
    if c == 1 :
        axs[i].set_title('{0} Component\n{1:0.2f} Compression Ratio\n{2:0.0f}% Varian
           c/fives.shape[1],
           100*sum(pca.explained_variance_ratio_)), fontsize = 12)
    else:
        axs[i].set_title('{0} Components\n{1:0.2f} Compression Ratio\n{2:0.0f}% Varia
           c/fives.shape[1], 100*sum(pca.explained_variance_ratio_)),
           fontsize = 12)
    pass

# Add original image
axs[4].imshow(fives[1].reshape(8, 8), cmap='binary')
axs[4].set_xticks([], [])
axs[4].set_yticks([], [])
axs[4].set_title('Original Image')

fig.suptitle('Effect of Increasing Levels of Compression',
             fontsize = 14)
plt.tight_layout(rect=[0, 0.03, 1, .8])
plt.show()
```



**(d)** How many principal components are required to well-approximate the data in (c)? How much compression is achieved in each case (express compression as the ratio of $k$ to the original dimension of the data $D$, so it ranges from 0 to 1). Comment on each case.

In the case of just 1 component (using a 0.02 compression ratio) we are actually able to maintain a surprising amount of detail in the image. In fact, we are able to explain almost a third of the variation in just this one component. In the case of 5 components (using a 0.08 compression ratio) we gain approximately another third of the variance and increase the level of detail of the image displayed. Looking to the case with 10 components (using a 0.16 compression ratio) we maintain 82% of variance and increase the detail in the image ever so slightly. in the case of 25 components (using a 0.39 compression ratio) we maintain almost all of the variance and increase the clarity of the image even more slightly than before. The original image can be considered the a 64 component compression or 100% variance explained.

To the naked eye, it seems that even one component (compression ratio 0.02) is enough to maintain the distinguishable features of the number 5 in an image. This could easily be considered a good appoximation of our data. However, if you wanted to be more conservative, increasing the number of components to five (compression ratio 0.08) we maintain over half of the variance in our data and increase the clarity of our image a noticeable amount. Depending on one's analysis going forward, either of these compression levels would be sufficient. There does not appear to be an advantage of increasing the compression ratio in this context.

## 1.5 4

### 1.5.1 [15 points] Build and test your own Neural Network for classification

There is no better way to understand how one of the core techniques of modern machine learning works than to build a simple version of it yourself. In this exercise you will construct and apply your own neural network classifier.

(a) Create a neural network class that follows the scikit-learn classifier convention by implementing fit, predict, and predict_proba methods. Your fit method should run backpropagation on your training data using stochastic gradient descent. Assume the activation function is a sigmoid. Choose your model architecture to have two input nodes, two hidden layers with five nodes each, and one output node.

```
In [30]: import numpy as np

         class neuralNet():

             def __init__(self, eta = 0.05):
                 self.input_size = 3 # number of features (2 + bias)
                 self.output_size = 1 # prediction
                 self.hidden_size = 5 # number of hidden nodes in each layer
                 self.epoch = 0 # counts number of interations through dataset
                 self.eta = eta # learning rate
                 self.cost_history = [] # log of cost functions after iterations

                 # Randomly initialize weights
                 # (5x2) weight matrix from input to hidden layer
                 self.W1 = np.random.randn(self.hidden_size, self.input_size)
                 # (5x5) weight matrix from hidden to output layer
                 self.W2 = np.random.randn(self.hidden_size, self.hidden_size)
                 # (1x5) weight matrix from hidden to output layer
                 self.W3 = np.random.randn(self.output_size, self.hidden_size)
                 pass
```

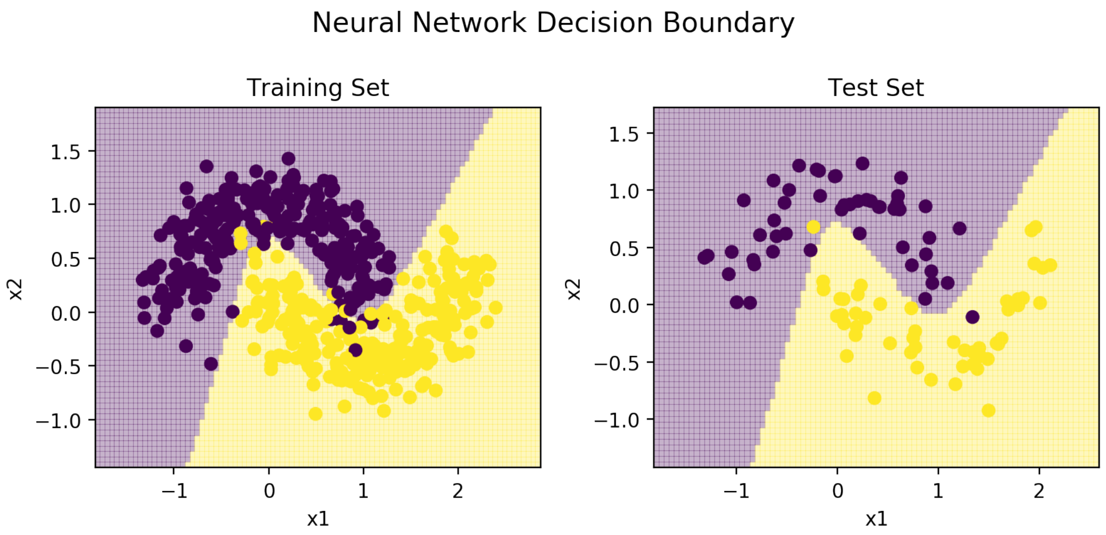
```python
def fit(self, X, y):
    # Add a 1 to input vector for bias
    X  = np.column_stack((X ,np.ones((X.shape[0],1))))

    print('|', end='') # progress bar
    # Flow through epochs until convergence
    while (self.epoch < 150):
        # Shuffle X and y and iterate through before reshuffle
        X_shuffle, y_shuffle = self.shuffle(X, y)
        for i in range(X.shape[0]):
            # Feed one input into model
            y_pred = self.feedforward(X_shuffle[i])
            # Backpropogate to update weights
            self.back_prop(X_shuffle, y_shuffle, y_pred, index = i)

            # Update cost
            new_cost = self.cost(y, self.predict_proba(X[:,:-1]))
            if self.epoch == 0:
                # If we're still going through for the first time
                # Don't stop, just update cost
                self.cost_history.append(new_cost)
                pass
            else:
                # If we've already gone through once
                # Stop iterating if cost function changes by less than 10^-4
                cost_diff = np.abs(new_cost - self.cost_history[-1])
                if cost_diff < 10e-04 :
                    # Update final cost
                    self.cost_history.append(new_cost)
                    break
                pass
            pass
        # Update progress bar and epoch number
        if self.epoch % 10 == 0: print('-', end='')
        self.epoch += 1
        pass
    print('|') # progress bar
    pass

def feedforward(self, x):
    #feedforward propagation
    self.a1 = np.dot(self.W1, x)
    self.z1 = self.sigmoid(self.a1)

    self.a2 = np.dot(self.W2, self.z1)
    self.z2 = self.sigmoid(self.a2)
```

```
        self.a3 = np.dot(self.W3, self.z2)
        y_pred = self.sigmoid(self.a3)
        return y_pred

    def back_prop(self, X, y, y_pred, index):
        # Backpropagation based on error for one sample
        # Prediction error
        pred_error = y_pred- y[index]

        # Calculate deltas for each level of as
        sig_prime_a3 = self.anti_sigmoid(self.a3)
        sig_prime_a2 = self.anti_sigmoid(self.a2).reshape(self.hidden_size,1)
        sig_prime_a1 = self.anti_sigmoid(self.a1).reshape(self.hidden_size,1)
        del_3 = pred_error * sig_prime_a3
        del_2 = self.W3.T * del_3 * sig_prime_a2
        del_1 = np.dot(self.W2.T, del_2) * sig_prime_a1

        # Update weights
        # wij = wij - n(dE/dwij)
        self.W1 += -self.eta * np.dot(del_1, X[index].reshape(1,self.input_size))
        self.W2 += -self.eta * np.dot(del_2, self.z1.reshape(1,self.hidden_size))
        self.W3 += -self.eta * np.dot(del_3, self.z2.reshape(1,self.hidden_size))
        pass

    def predict_proba(self, X_new):
        # Fit new data and predict probabilities

        # Add a 1 to input vector for bias
        X_new  = np.column_stack((X_new, np.ones((X_new.shape[0],1))))

        # Create array of predictions
        probs = []
        for x in X_new:
            prediction = float(self.feedforward(x))
            probs.append(prediction)
            pass
        return np.array(probs)

    def cost(self, y, y_pred):
        # Cost function: Sum squared error
        diff = y - y_pred
        sse = np.sum(np.dot(diff, diff)) / 2
        return sse

    def sigmoid(self, x):
        # Non-linear transformation
        return 1/(1+np.exp(-x))
    def anti_sigmoid(self, x):
        # Derivative of sigmoid function
        return self.sigmoid(x)*(1-self.sigmoid(x))

    def shuffle(self, X_orginial, y_orginial):
        # Shuffes order of predictors and output while maintaining consistency
        # between two arrays
        order = np.arange(X_orginial.shape[0])
        np.random.shuffle(order)
        X_shuffle = X_orginial[order]
        y_shuffle = y_orginial[order]
        return X_shuffle, y_shuffle
    pass
```

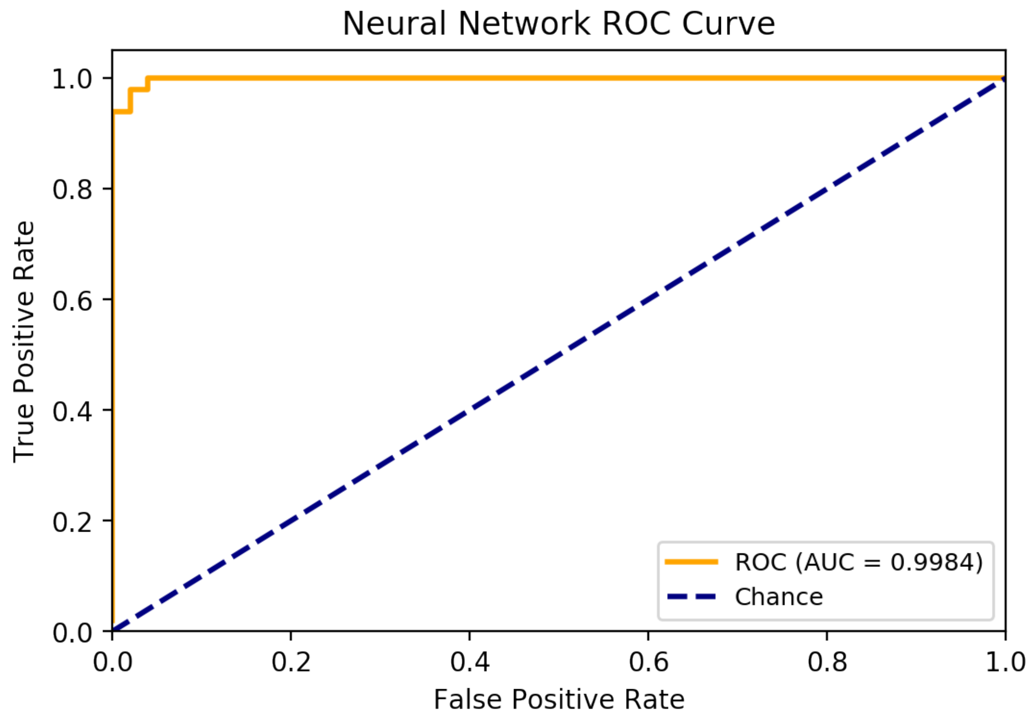
(b) Create a training and test dataset using sklearn.datasets.make_moons(N, noise=0.20), where $N_{train} = 500$ and $N_{test} = 100$. Train and test your model on this dataset. Adjust the learning rate and number of training epochs for your model to improve performance as needed. In two subplots, plot the training data on one, and the test data on the other. On each plot, also plot the decision boundary from your neural network trained on the training data. Report your performance on the test data with an ROC curve.

```
In [32]: from sklearn.datasets import make_moons

         X, y  = make_moons(500, noise = .2) # train set
         X_test, y_test  = make_moons(100, noise = .2) # test set

In [33]: nn = neuralNet(eta = .075)
         nn.fit(X, y)

|---------------|


In [49]: import matplotlib.pyplot as plt

         #plt.plot(nn.cost_history)
         #plt.show()

         fig, axs = plt.subplots(1, 2)
         fig.set_dpi(200)
         fig.set_figheight(4)
         fig.set_figwidth(8)

         #Plotting decision region
         x1_min, x1_max = X[:,0].min() - .5, X[:,0].max() + .5
         x2_min, x2_max = X[:,1].min() - .5, X[:,1].max() + .5
         xx, yy = np.meshgrid(np.arange(x1_min, x1_max, 0.05),
                              np.arange(x2_min, x2_max, 0.05))

         # Make predictions across entire decision region
```

```
Z = nn.predict_proba(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape) > .5

# Plot decision regions and original data
axs[0].pcolormesh(xx, yy, Z, alpha = .3)
axs[0].scatter(X[:,0], X[:,1], c = y)
axs[0].set_xlabel('x1')
axs[0].set_ylabel('x2')
axs[0].set_title('Training Set')

#Plotting decision region
x1_min, x1_max = X_test[:,0].min() - .5, X_test[:,0].max() + .5
x2_min, x2_max = X_test[:,1].min() - .5, X_test[:,1].max() + .5
xx, yy = np.meshgrid(np.arange(x1_min, x1_max, 0.05),
                     np.arange(x2_min, x2_max, 0.05))

# Make predictions across entire decision region
Z = nn.predict_proba(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape) > .5

# Plot decision regions and original data
axs[1].pcolormesh(xx, yy, Z, alpha = .3)
axs[1].scatter(X_test[:,0], X_test[:,1], c = y_test)
axs[1].set_xlabel('x1')
axs[1].set_ylabel('x2')
axs[1].set_title('Test Set')

fig.suptitle('Neural Network Decision Boundary', fontsize = 14)
plt.tight_layout(rect=[0, 0.03, 1, .9])
plt.show()
```



```
In [42]: from sklearn.metrics import roc_curve, auc
         plt.figure(dpi = 200)
         y_pred = nn.predict_proba(X_test)
         fpr, tpr, thresholds = roc_curve(y_test, y_pred)
         roc_auc = auc(fpr, tpr)
         plt.plot(fpr, tpr, lw = 2, color = 'orange',
                  label='ROC (AUC = {0:0.4f})'.format(roc_auc))

         # Format ROC Plot
         plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--',
                  label = 'Chance')
         plt.xlim([0.0, 1.0])
         plt.ylim([0.0, 1.05])
         plt.xlabel('False Positive Rate')
         plt.ylabel('True Positive Rate')
         plt.legend(loc="lower right", fontsize = 9)
         plt.title('Neural Network ROC Curve')
         plt.show()
```



The decision boundary produced by my neural network produces a very high AUC (.9984) and successfully detected a non-linear decision boundary that explains the majority of the variance in the dataset. Nevertheless, additional work needs be done in order to tune hyperparameters such as number of epochs and learning rate in order to fully optimize my algorithm. Moreover, adding additional layers or nodes to hidden units may increase performance.

(c) Suggest at least two ways in which you neural network implementation could be improved.

There are many ways in which I could improve my neural network. For example, I could make the number of hidden layers and nodes more flexible to user input as opposed to hard coding them. By making these hyperparameters flexible, the algorithm could be optimized through tweaking. Moreover, I could improve my neural networks to use adaptive learning rate methodology such as Adam or RMSprop as opposed to using a static $\eta$.