Assignment 4 - Unsupervised Learning and Neural Networks

AKSHAY PUNWATKAR

Netid: AP509

Instructions for all assignments can be found here (https://github.com/kylebradbury/ids705/blob/master/assignments/_Assignment%20Instructions.ipynb), which is also linked to from the course syllabus (https://kylebradbury.github.io/ids705/index.html).

Learning objectives

Through completing this assignment you will be able to...

1. Apply clustering techniques to a variety of datasets with diverse distributional properties, gaining an understanding of their strengths and weaknesses and how to tune model parameters.
2. Apply PCA and t-SNE for performing dimensionality reduction and data visualization
3. Understand how PCA represents data in lower dimensions and understand the concept of data compression.
4. Build, tune the parameters of, and apply feedforward neural networks to data
5. Develop a detailed understanding of the math and practical implementation considerations of neural networks, one of the most widely used machine learning tools.

1

[35 points] Clustering

Clustering can be used to reveal structure between samples of data and assign group membership to similar groups of samples. This exercise will provide you with experience building a basic clustering algorithm to provide insight into the structure of these techniques, then compare a number of clustering techniques on a distinctive datasets to experience the pros and cons of these approaches.

(a) Implement your own k-means algorithm. For a measure of dissimilarity use the sum of squared error of the Euclidean distance from each point to the cluster mean. Initialize your means by selecting a set of $k$ points at random from your dataset and using those values as the initial means. You may need to run your algorithm multiple times with different initializations (picking the clustering with the lower dissimilarity measure) to get the best results. You may use the template below to assist you in your implementation.

In [93]:

```
def kmeans(X, k, max_steps=100, convergence_threshold=0.01):
    '''kmeans

        Input:
            X: matrix of input data where each row represents a sample
            k: number of means to use
            max_steps: maximum number of iterations to run the algorithm
            convergence_threshold: if the means change less than this
                                   value in an iteration, declare convergence
        Output:
            means: a matrix listing the k means
            cluster_assignment: a list of the cluster assignments for each sam
ples
            dissimilarity: sum of squared error of the Euclidean distance from
each point to the cluster mean
    '''
```

(b) Demo your algorithm. Create some data to cluster by using the `blobs` module from `scikit-learn` to construct two datasets: one with 2 cluster centers and the other with 5. Set the `random_state` keyword parameter to 0 in each to ensure the datasets are consistent with the rest of the class and generate 5,000 samples of each dataset.

- For each dataset rerun your k-means algorithm for values of k ranging from 1 to 10 and for each plot the "elbow curve" where you plot dissimilarity in each case. For your two datasets, where is the elbow in the curve and why?
- Plot the data and your k-means for the optimal value of k that you determined from the elbow curve.

(c) Ensure your understanding of how clustering methods work. Briefly explain in 1-2 sentences each (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.

(d) Apply clustering algorithms to diverse datasets. For each of the clustering algorithms in (c) run each of them on the five datasets below.

- Tune the parameters in each model to achieve better performance for each dataset.
- Plot the final result as a 4-by-5 subplot showing the resulting clustering of each method on each dataset.
- Which method works best or worst on each dataset and why? (This can be 1-2 sentences for each dataset).

The datasets are:

- 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/ (https://cs.joensuu.fi/sipu/datasets/)](https://cs.joensuu.fi/sipu/datasets/).

Note: for k-means, use the `scikit-learn` module rather than your own

ANSWER

(A)

Defining K-Means Algorithm

In [1]:

```
import numpy as np
from numpy.linalg import norm
from collections import defaultdict
import matplotlib.pyplot as plt
import seaborn as sns
from tqdm import tqdm
import random
```

In [2]:

```
def kmeans(X, k, max_steps=100, convergence_threshold=0.01):

    # Random initialization of k centroids from the data
    old_centroid = X[np.random.choice(X.shape[0], size=k, replace=False), :]

    # Initializing arrays to store new centroids
    new_centroid = np.zeros([k,X.shape[1]])

    # Initializing dictionary to store cluster assignment

    cluster_assigned = []
    tot_diss = 0

    for i in range(max_steps):

        #reset assgined cluster and values
        cluster_assigned = []
        cluster = defaultdict(list)
        clust_diss = np.zeros(k)
        tot_diss = 0

        '''
        Calculating distance of points from the centroids and
        assigning them to the cluster with minimum distance.
        Minimum distance is the Squared Euclidean Distance
        '''
        for point in X:
            dist = norm(point-old_centroid,axis=1)**2
            cluster[np.argmin(dist)].append(point)
            cluster_assigned.append(np.argmin(dist))

        '''
        Calculating new Centroids based on the points in the cluster by
        taking mean of all the points in the cluster
        '''
        for key,val in cluster.items():
            new_centroid[key] = np.mean(val,axis=0)
            for point in val :
                clust_diss[key] = clust_diss[key] + norm(point-new_centroid[ke
y])**2
            #clust_diss[key] = clust_diss[key]/len(val)
            tot_diss = tot_diss + clust_diss[key]

        # cheking convergence
        if np.all(new_centroid-old_centroid < 0.01):
            return new_centroid,cluster_assigned,round(tot_diss,3)
        else:
            old_centroid = np.copy(new_centroid)

    return new_centroid,cluster_assigned, round(tot_diss,3)
```

(B)

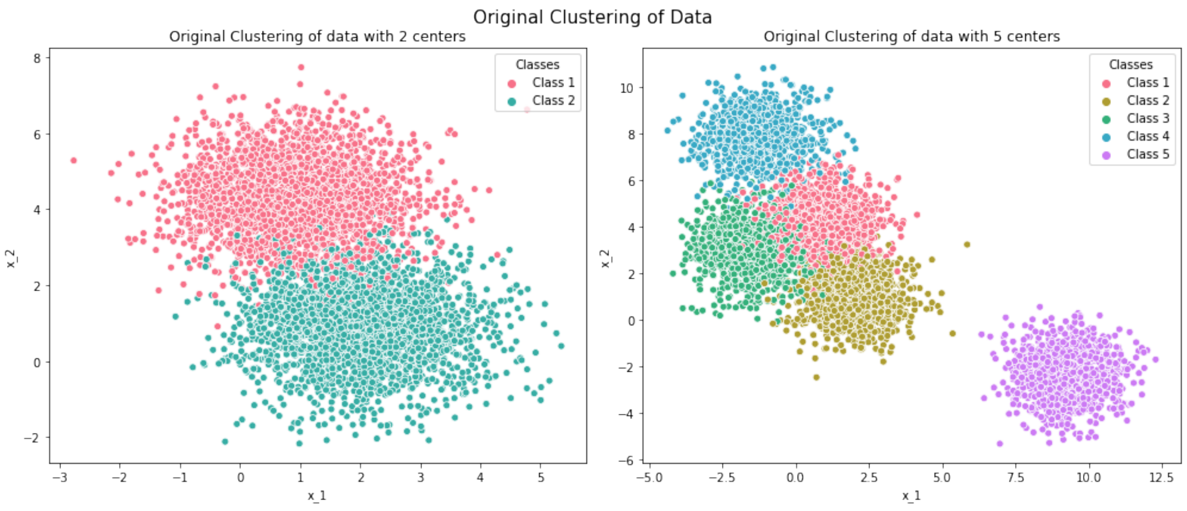
Testing the K-Means Algorithm

In [3]:

```
# Making the dataset using Blobs
from sklearn.datasets import make_blobs
data_2, y_2 = make_blobs(n_samples=5000, centers=2, n_features=2,random_state=
0)
data_5, y_5 = make_blobs(n_samples=5000, centers=5, n_features=2,random_state=
0)
```

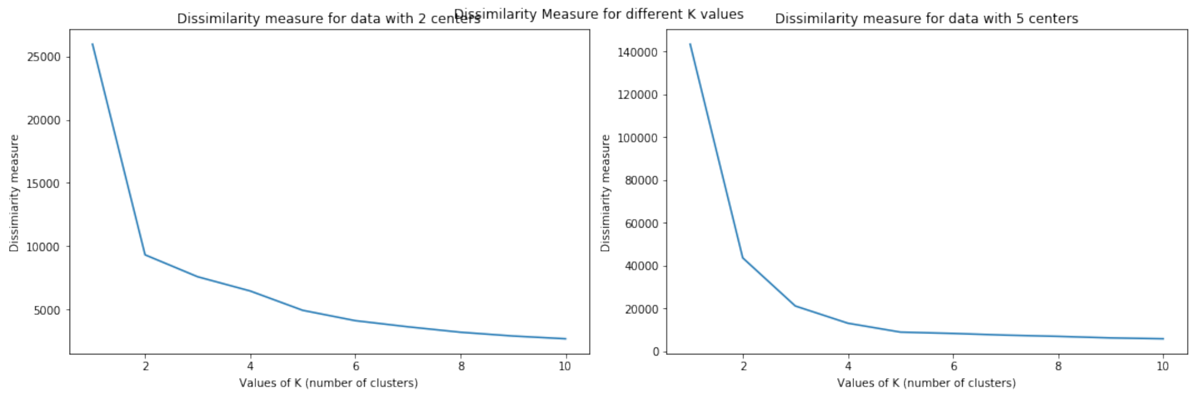
In [55]:

```
fig, axs = plt.subplots(1,2, figsize=(14,6))

labels = ["Class 1","Class 2","Class 3","Class 4", "Class 5"]

sns.scatterplot(x=data_2[:,0],y=data_2[:,1],hue=y_2,
                ax=axs[0],palette=sns.color_palette("husl", 2))
axs[0].set_title("\nOriginal Clustering of data with 2 centers")
axs[0].set_xlabel("x_1")
axs[0].set_ylabel("x_2")
h, l = axs[0].get_legend_handles_labels()
axs[0].legend(h,labels[0:2] , title="Classes", loc='upper right')

sns.scatterplot(x=data_5[:,0],y=data_5[:,1],hue=y_5,
                ax=axs[1],palette=sns.color_palette("husl", 5))
axs[1].set_title("\nOriginal Clustering of data with 5 centers")
axs[1].set_xlabel("x_1")
axs[1].set_ylabel("x_2")
h, l = axs[1].get_legend_handles_labels()
axs[1].legend(h,labels[0:6] , title="Classes", loc='upper right')

plt.suptitle("Original Clustering of Data", size= 15)
plt.tight_layout()
```



In [59]:

```
#running k-means clustering for k = 1 to 10 for both the datasets

fig, axs = plt.subplots(1,2, figsize=(15,5))

for j in (range(1)):
    result_2 = defaultdict(list)
    result_5 = defaultdict(list)
    diss_2 = np.zeros(10)
    diss_5 = np.zeros(10)
    for i in range(1,11):
        result_2[i] = kmeans(data_2,i)
        diss_2[i-1] = result_2[i][2]
        result_5[i] = kmeans(data_5,i)
        diss_5[i-1] = result_5[i][2]
    axs[0].plot(range(1,11),diss_2)
    axs[1].plot(range(1,11),diss_5)


axs[0].set_title("Dissimilarity measure for data with 2 centers")
axs[0].set_xlabel("Values of K (number of clusters)")
axs[0].set_ylabel("Dissimiarity measure")
axs[1].set_title("Dissimilarity measure for data with 5 centers")
axs[1].set_xlabel("Values of K (number of clusters)")
axs[1].set_ylabel("Dissimiarity measure")
fig.suptitle("Dissimilarity Measure for different K values")
plt.tight_layout()
```

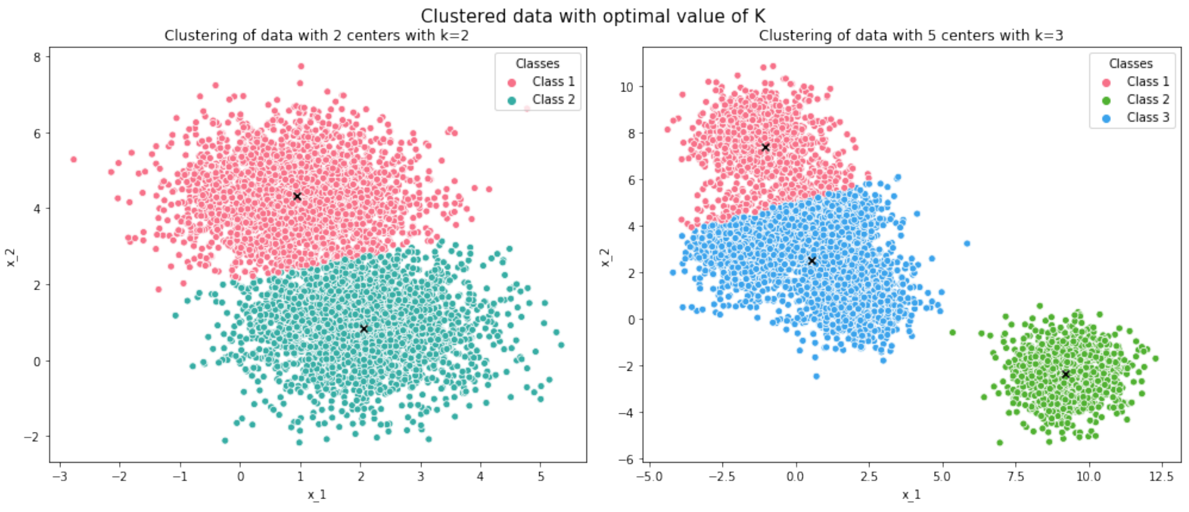


Reasoning

- For the first data (with 2 centers), the elbow (point of sudden change of dissimilarity) occurs at k=2. This could be attributed to the number of centers around which the original dataset was created, which was 2 in this case. Although the data without any clustering appears to be a single cluster, the two different clusters could be seen based on the density of points which is more towards the end and less near the boundary of the two clusters and same can be said based on the dissimalrity plot.
- Similarly, for the second data (with 5 centers), the elbow occurs at k=3. Although the data was created with five centers, after plotting the data it's clear that the data was not well separated, classes 3,4 & 5 seems to clusters very close, as comapred to class 1 (though close but little separated) and class 5 (which is clearly a separated cluster). Due to this reason the dissimialrity seems to bend and drop near k=2 and k=3. Hence, k=3 seems an optimal value, as the clusters 1 and 2 (classes 2,3,4) seems to be separated and cluster 3 (class 5) appears very well separated.

In [62]:

```
fig, axs = plt.subplots(1,2, figsize=(14,6))

sns.scatterplot(x=data_2[:,0],y=data_2[:,1],hue=result_2[2][1],
                ax=axs[0],palette=sns.color_palette("husl", 2))
axs[0].scatter(result_2[2][0][:,0],result_2[2][0][:,1],marker='x',c='black')
axs[0].set_title("\nClustering of data with 2 centers with k=2")
axs[0].set_xlabel("x_1")
axs[0].set_ylabel("x_2")
h, l = axs[0].get_legend_handles_labels()
axs[0].legend(h,labels[0:2] , title="Classes", loc='upper right')

sns.scatterplot(x=data_5[:,0],y=data_5[:,1],hue=result_5[3][1],
                ax=axs[1],palette=sns.color_palette("husl", 3))
axs[1].scatter(result_5[3][0][:,0],result_5[3][0][:,1],marker='x',c='black')
axs[1].set_title("\nClustering of data with 5 centers with k=3")
axs[1].set_xlabel("x_1")
axs[1].set_ylabel("x_2")
h, l = axs[1].get_legend_handles_labels()
axs[1].legend(h,labels[0:4] , title="Classes", loc='upper right')

plt.suptitle("Clustered data with optimal value of K\n", size= 15)
plt.tight_layout()
```



(C)

Different Clustering Algorithms

1. K-means:

K-means clustering algorithm cluster points into k clusters based on the proximity (e.g. Euclidean distance) of the point to the cluster centroid. Where cluster centroid is the mean of the points in the cluster. K-means assume the clusters have similar (circular) shapes and variance.

2. Agglomerative clustering:

Agglomerative is a type of *Hierarchical clustering* in which clusters are build in a bottom-up approach, which means clustering starts assuming all points as an individual cluster and then agglomerate multiple small cluster into a large cluster till all the points converge into one single cluster.

This clustering approach refers to a collection of closely related clustering techniques that produce a hierarchical clustering by starting with each point as a singleton cluster and then repeatedly merging the two closest clusters until a single, allencompassing cluster remains

3. Gaussian mixture models (GMM):

Gaussian Mixture Models (GMMs) assume that there are a certain number of Gaussian distributions in the data, and each of these distributions represent a cluster. Hence, a GMM tends to group the data points belonging to a single distribution together thus creating clusters

4. DBSCAN:

This is a density-based clustering algorithm that produces a partitional clustering, in which the number of clusters is automatically determined by the algorithm. Points in low-density regions are classified as noise and omitted; thus, DBSCAN does not produce a complete clustering.

5. Spectral clustering:

In spectral clustering, the data points are treated as nodes of a graph. Thus, clustering is treated as a graph partitioning problem. The nodes are then mapped to a low-dimensional space that can be easily segregated to form clusters. An important point to note is that no assumption is made about the shape/form of the clusters.

(D)

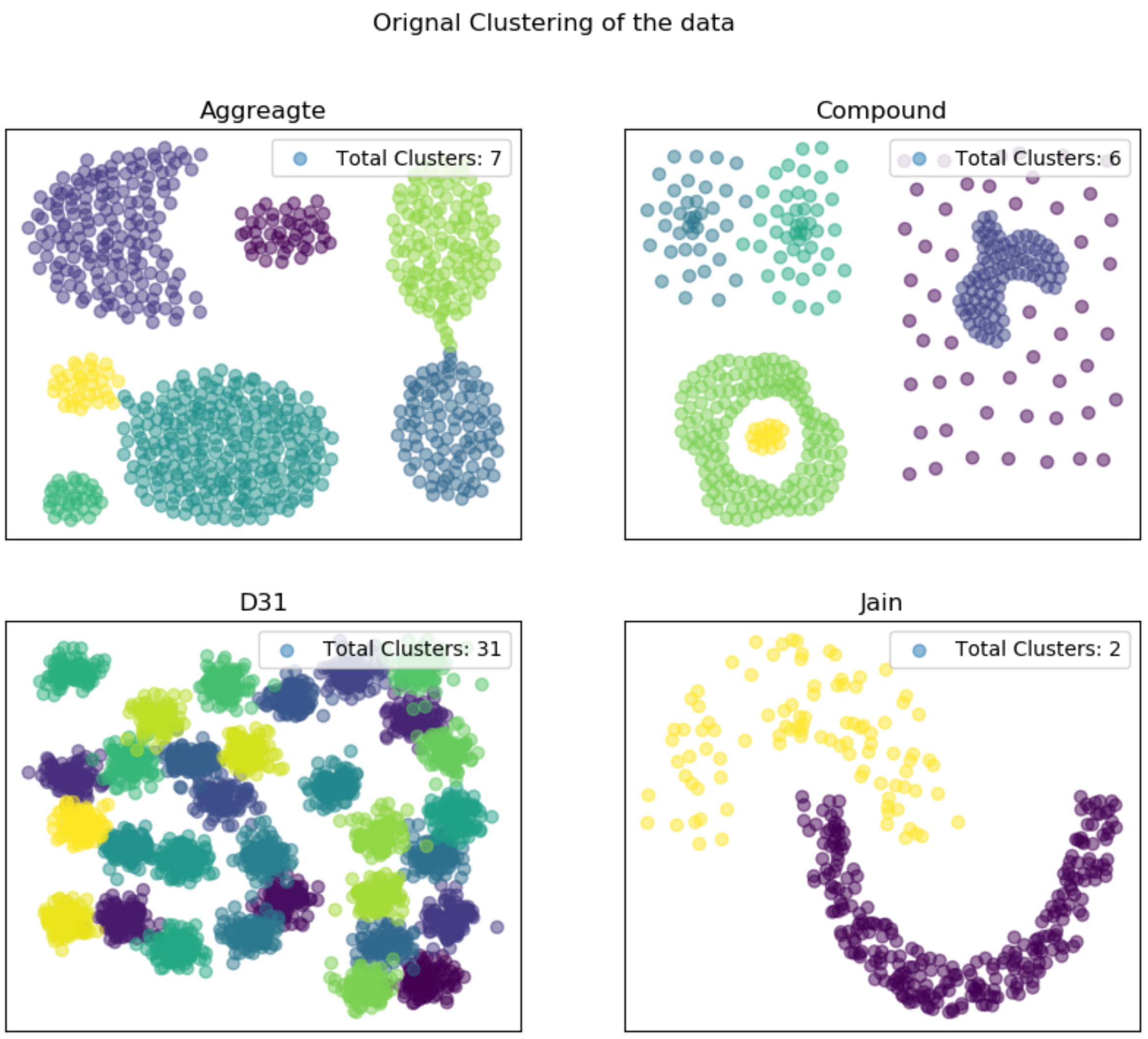
Clustering of different datasets

In [77]:

```
agg_data = np.loadtxt("Aggregation.txt") #7
comp_data = np.loadtxt("Compound.txt") #6
d31_data = np.loadtxt("D31.txt") #31
jain_data = np.loadtxt("jain.txt") #2

dataset = [agg_data[:,0:2],comp_data[:,0:2],d31_data[:,0:2],jain_data[:,0:2]]
label = [agg_data[:,2],comp_data[:,2],d31_data[:,2],jain_data[:,2]]
titles = ["Aggreagte","Compound","D31", "Jain"]
n_clust = [7,6,31,2]
```

In [429]:

```
fig, axs = plt.subplots(2,2, figsize=(10,8))
n = 0
for i in range(2):
    for j in range(2):
        axs[i,j].scatter(dataset[n][:,0],dataset[n][:,1], c=label[n], alpha =
0.5)
        axs[i,j].set_title(titles[n])
        axs[i,j].get_yaxis().set_ticks([])
        axs[i,j].get_xaxis().set_ticks([])
        c = "Total Clusters: "+str(len(np.unique(label[n])))
        axs[i,j].legend([c], loc = "upper right", fontsize=10)

        n += 1
fig.suptitle("Orignal Clustering of the data")
plt.show()
```



In [4]:

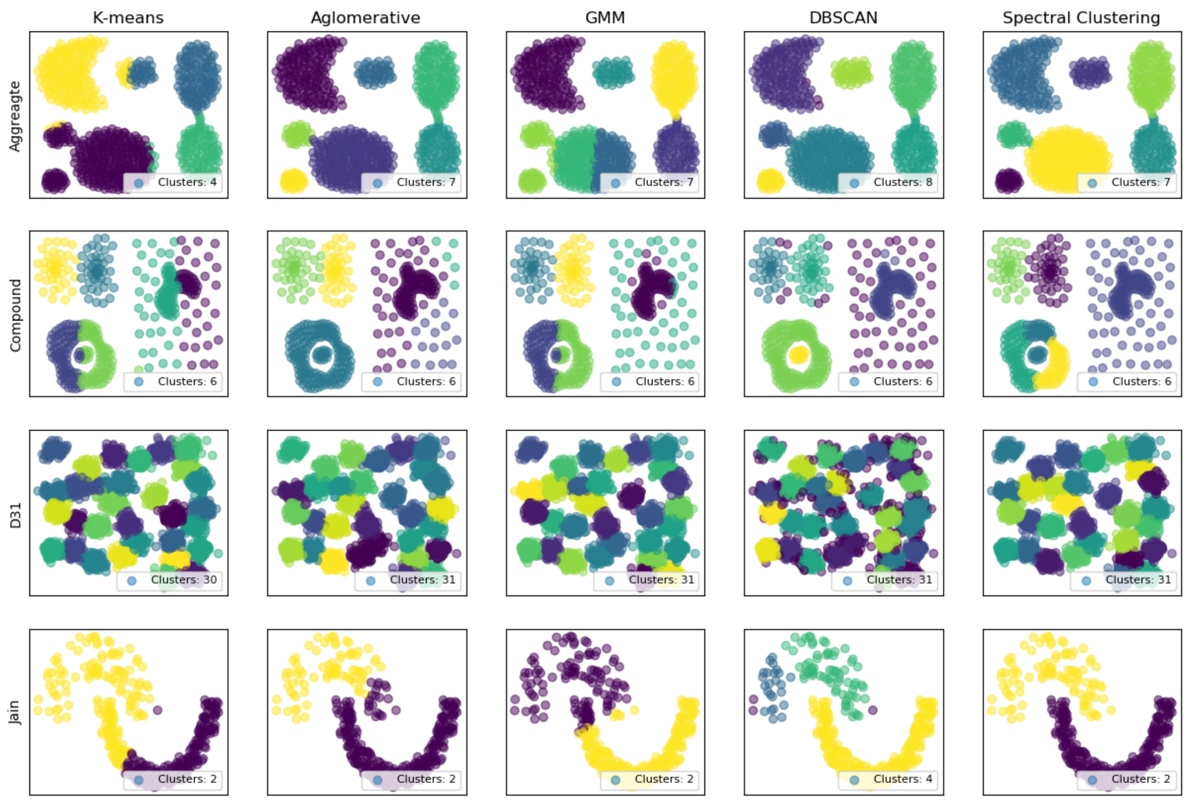
```
from sklearn.cluster import KMeans
from sklearn.cluster import AgglomerativeClustering
from sklearn.mixture import GaussianMixture
from sklearn.cluster import DBSCAN
from sklearn.cluster import SpectralClustering
```

Clustering with selected hyper-parameters

In [437]:

```
fig, axs = plt.subplots(4,5, figsize=(15,10))

methods = ["K-means","Aglomerative","GMM","DBSCAN","Spectral Clustering"]

# Setting up hyper-parameter
#for K-means
clust_ = [4,6,30,2]

#for agglomerative and Spectaral
```

```
n_clust = [7,6,31,2]


#for GMM
cov =['spherical', 'spherical', 'diag', 'diag']

#for DBSCAN
eps_val =[1.5,1.55,1,2.51]
min_sam = [8,5,40,4]

for i in range(4):

    kmeans_clust = KMeans(n_clusters=clust_[i] , random_state=0).fit(dataset[i
])

    aggcust_agg = AgglomerativeClustering(n_clusters=n_clust[i],linkage="avera
ge").fit(dataset[i])

    Gmm = GaussianMixture(n_components=n_clust[i],covariance_type=cov[i]).fit_
predict(dataset[i])

    dbsc = DBSCAN(eps=eps_val[i], min_samples=min_sam[i]).fit(dataset[i]).fit(
dataset[i])

    spect = SpectralClustering(n_clusters = n_clust[i],assign_labels="discreti
ze",
                                random_state=0).fit(dataset[i])

    clusters_assigned = [kmeans_clust.labels_, aggcust_agg.labels_, Gmm,
                         dbsc.labels_, spect.labels_]
    cluster_count = [len(np.unique(x)) for x in clusters_assigned]

    for j in range(5):
        c_ = clusters_assigned[j]
        axs[i,j].scatter(dataset[i][:,0],dataset[i][:,1],c=c_,alpha=0.5)
        axs[0,j].set_title(methods[j])
        axs[i,0].set_ylabel(titles[i])
        axs[i,j].get_yaxis().set_ticks([])
        axs[i,j].get_xaxis().set_ticks([])
        c = "Clusters: "+str(cluster_count[j])
        axs[i,j].legend([c], loc = "lower right",fontsize=8)

plt.xticks([])
plt.yticks([])
plt.show()
```



What works for what ?

1. For Aggregate Data

- **Method that works best** : Spectaral Clustering, Agglomerative Clustering & DBSCAN
  Since the data is non-uniformly clustered but well separated, spectaral clustering works best. Also, Due to the bottoms up clustering approach, agglomerative clustering performs very well given the non-uniform shapes and distribution of clusters. DBSCAN work well in indentifying cluster of different shapes and variances. On the other hand, since the clusters had non-uniform shape and variance, it did not fit the assumptions for K-means, and an absence of Gaussian Distribution (underlying assumption for GMM), neither K-means nor GMM resulted in good clustering

2. For Compound Data

- Method that works best : **DBSCAN**
  DBSCAN works very well in identifying cluster with different shapes and variances. However, the algorithm had some challenges in correctly clustering areas with high variance in cluster density (which can be seen on the edge of circular clusters). The data couldn't fit the assumptions (eg. similar shapes and variance for k-means, guassian distribution for GMM)for other clustering methods, due to which other four algorithms couldn't identify all the clusters accurately.

3. For D31 Data

- **Method that works best** : All but DBSCAN
  Since the data had uniform shapes (almost circular) and variance, all the methods worked very well in clustering the data accurately. DBSCAN, however classified several points as noise since those points appeared to be in the high variation zone in a setting of closely packed clusters.

4. For Jain Data

- **Method that works best** : Spectaral Clustering, Agglomerative Clustering & GMM Clustering
  Spectral Clustering is very useful when the structure of the individual clusters is highly non-convex or more generally when a measure of the center and spread of the cluster is not a suitable description of the complete cluster, which is true for the crest shaped Jain data. GMM clustering was able to identify the two cluster, however with few mis-clustered points due to it's assumption of gaussian distribution. Aggolmerative clustering was also able to cluster the data correctly but mis-clustered several points due to different density of the two clusters and less separation between the cluster near the edge. K- means due to it's limitations of clustering different shapes could not cluster accurately. DBSCAN struggled with the variable density of the two moons.

2

[20 points] Dimensionality reduction and visualization of digits with PCA and t-SNE

(a) Reduce the dimensionality of the data with PCA for data visualization.

- 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 data with labels in this two dimensional space (labels can be colors, shapes, or using the actual numbers to represent the data - definitely include a legend in your plot).

(b) Create a plot showing the cumulative fraction of variance explained as you incorporate from $1$ through all $D$ principal components of the data (where $D$ is the dimensionality of the data).

- What fraction of variance in the data is UNEXPLAINED by the first two principal components of the data?
- Briefly comment on how this may impact how well-clustered the data are. *You can use the `explained_variance_` attribute of the PCA module in `scikit-learn` to assist with this question*

(c) Reduce the dimensionality of the data with t-SNE for data visualization. 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 (https://distill.pub/2016/misread-tsne/)](https://distill.pub/2016/misread-tsne/).

(d) Compare/contrast the performance of these two techniques. Which seemed to cluster the data best and why? *Note: You typically will not have labels available in most problems.*

ANSWER

(A)

Dimensionality Reduction using PCA

In [14]:

```
from sklearn.datasets import load_digits
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.decomposition import PCA
```

In [ ]:

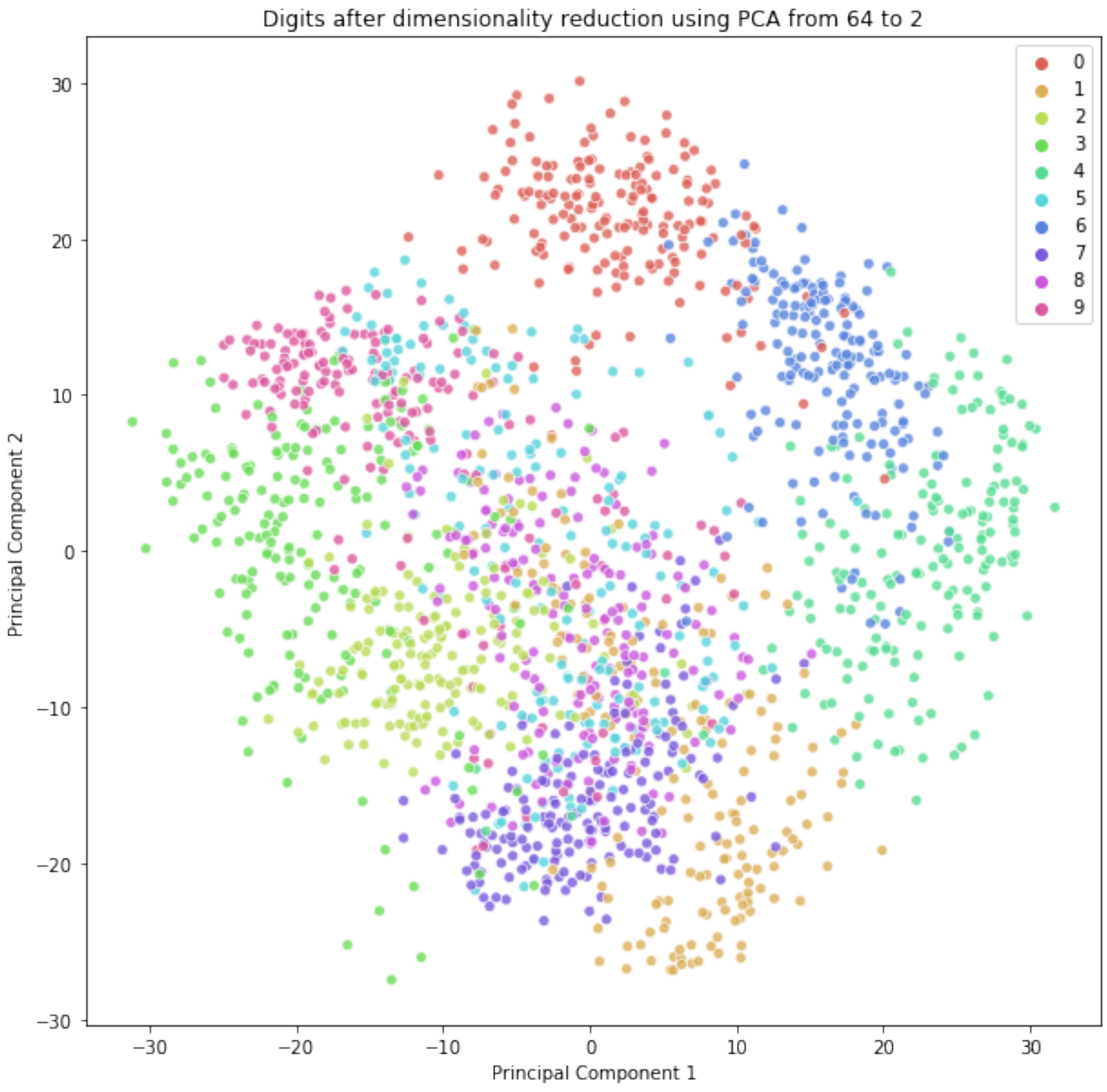
```
digits = load_digits()
print(digits.data.shape)
```

In [ ]:

```
pca = PCA(n_components=2)
transf_data = pca.fit_transform(digits.data)
print(np.shape(transf_data))
```

In [17]:

```
plt.figure(figsize=(10,10))
sns.scatterplot(
    x=transf_data[:,0], y=transf_data[:,1],
    hue=digits.target,
    palette=sns.color_palette("hls", 10),
    legend="full",
    alpha=0.8
)
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.title("Digits after dimensionality reduction using PCA from 64 to 2")
plt.show()
```



(B)

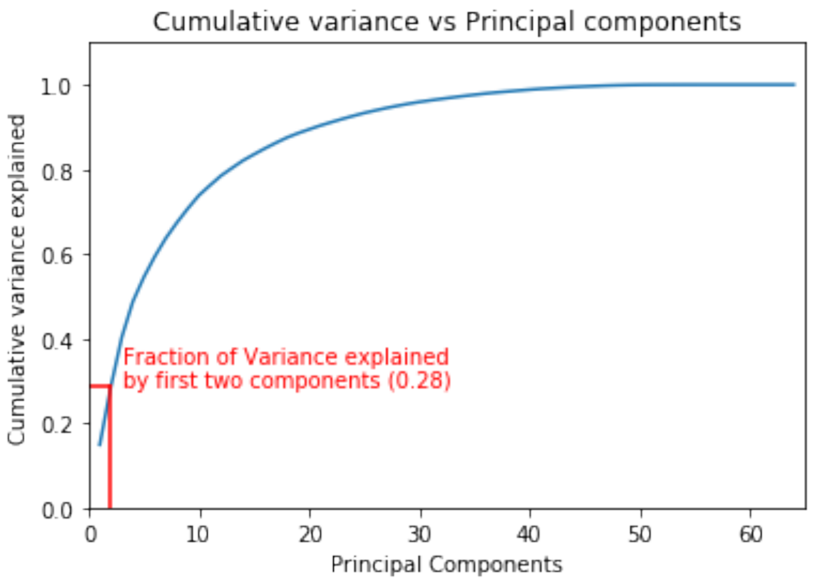
Cumulative variance with different principal components

In [18]:

```
ex_var = list()
for i in range(1,65):
    pca = PCA(n_components=i)
    trans_data = pca.fit_transform(digits.data)
    variance_ = pca.explained_variance_
    ex_var.append(sum(variance_))
```

In [19]:

```
frac_var = [x/ex_var[-1] for x in ex_var ]
plt.plot(range(1,65),frac_var)
plt.xlabel("Principal Components")
plt.ylabel("Cumulative variance explained")
plt.title("Cumulative variance vs Principal components")
plt.plot([2,2],[0,frac_var[1]], c= 'r')
plt.plot([-2,2],[frac_var[1],frac_var[1]], c= 'r')
plt.ylim(0,1.1)
plt.xlim(0,65)
plt.annotate(xy=(10,frac_var[1]),
             s="Fraction of Variance explained \nby first two components (0.28
)",
             xytext=(3,frac_var[1]), c='r')
plt.show()
```



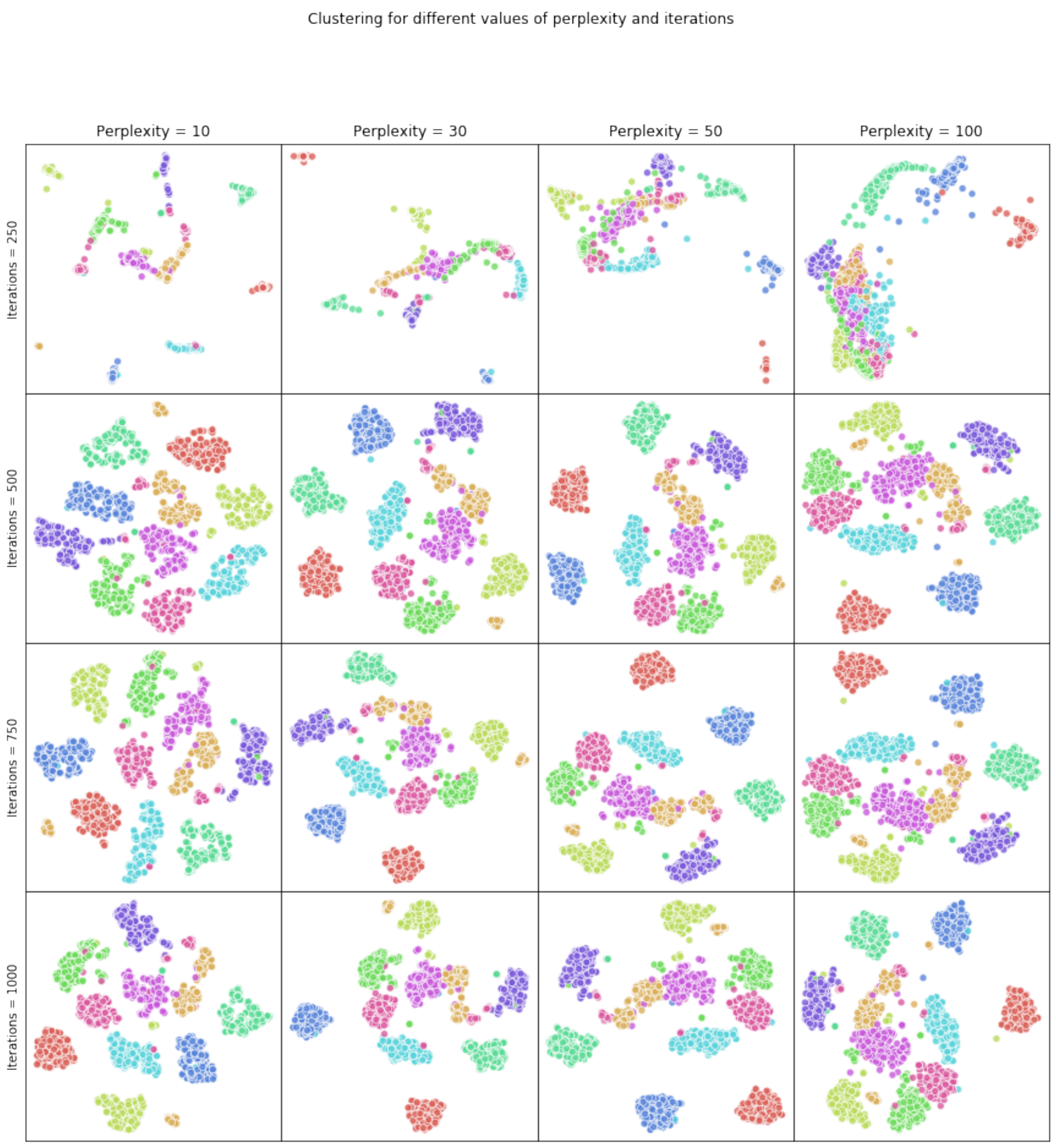
- 0.72 is fraction of variance that is **unexplained** by first two principal components.
- Since, only 28% of variance can be explained by the first two principal components, clustering could be highly impacted. The loss of information in terms of low amount of variance captured by 2 components of the original data resulted in dispersed (very low density) and overlapping clusters.

(C)

Hyper-Parameter Selection for TSNE

In [20]:

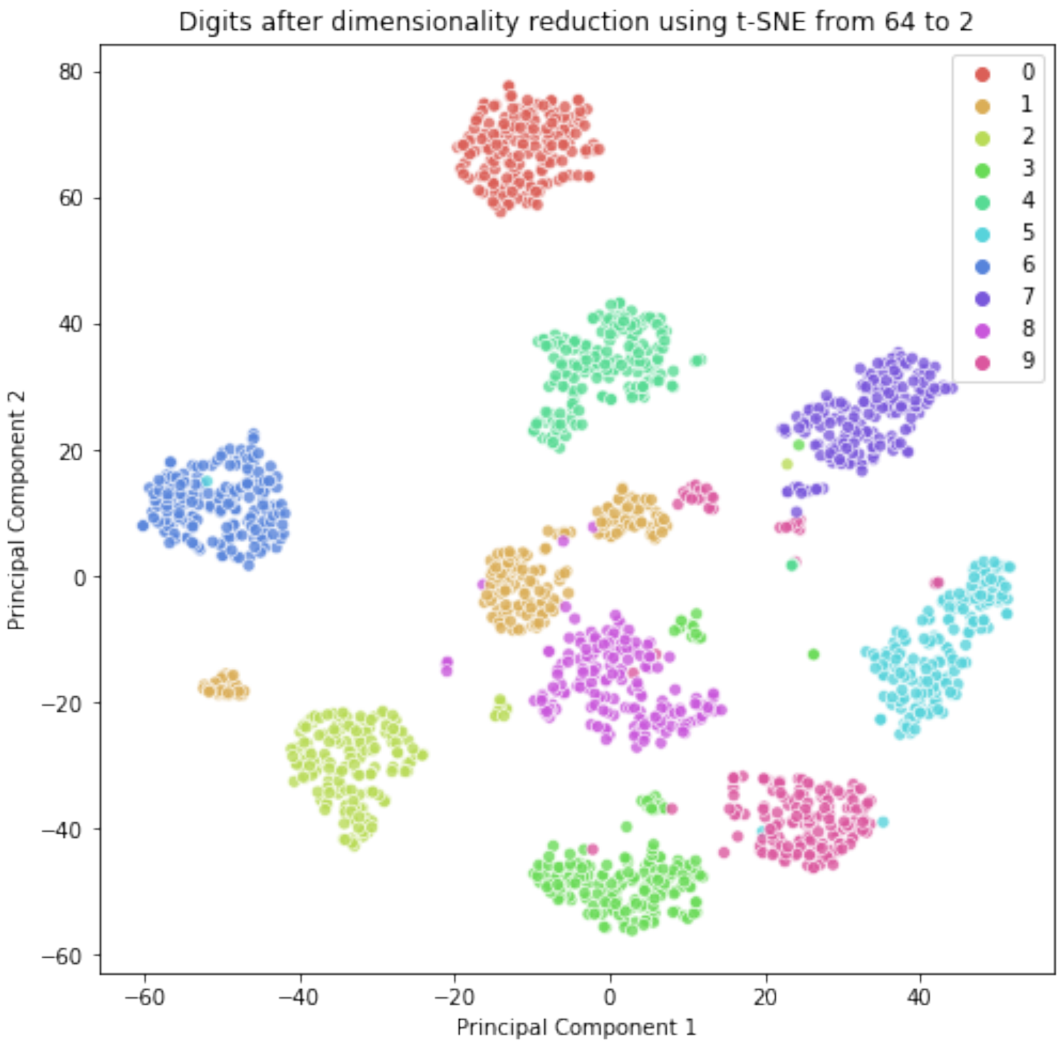
```
from sklearn.manifold import TSNE
fig, axs = plt.subplots(4,4, figsize=(15,15),gridspec_kw={'hspace': 0, 'wspace
': 0})
n_iter = [250,500,750,1000]
perpex = [10,30,50,100]
k=0
for i in range(4):
    for j in range(4):

        X_embedd = TSNE(n_components=2,perplexity=perpex[j],n_iter=n_iter[i]).
fit_transform(digits.data)
        sns.scatterplot(x=X_embedd[:,0], y=X_embedd[:,1], hue=digits.target,
                        palette=sns.color_palette("hls", 10),alpha=0.8, ax=axs
[i,j],legend=None)
        axs[i,0].set_ylabel("Iterations = %s"%(n_iter[i]))
        axs[0,j].set_title("Perplexity = %s"%(perpex[j]))
        axs[i,j].get_xaxis().set_ticklabels([])
        axs[i,j].get_xaxis().set_ticks([])
        axs[i,j].get_yaxis().set_ticklabels([])
        axs[i,j].get_yaxis().set_ticks([])

        k +=1

plt.suptitle("Clustering for different values of perplexity and iterations")
plt.show()
```



TSNE using selected Hyper-Parameters

In [175]:

```
X_embedded = TSNE(n_components=2,n_iter=1000,perplexity=30).fit_transform(digi
ts.data)
plt.figure(figsize=(8,8))
sns.scatterplot(
    x=X_embedded[:,0], y=X_embedded[:,1],
    hue=digits.target,
    palette=sns.color_palette("hls", 10),
    legend="full",
    alpha=0.8
)
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.title("Digits after dimensionality reduction using t-SNE from 64 to 2")
plt.show()
```



(D)

t-Distributed Stochastic Neighbor Embedding (t-SNE) is a non-linear technique for dimensionality reduction that is particularly well suited for the visualization of high-dimensional datasets. However, PCA is linear algorithm which is not able to interpret complex polynomial relationship. Moreover, PCA it is a mathematical technique, but t-SNE is a probabilistic one.

For the given digits dataset 64 dimension, **t-SNE seems to be much more effective** in dimnensionality reduction to 2 dimension as compared to PCA. It quite visible that the clustering after t-SNE is much better and separated, while that after PCA seems to have a lot of over lap and very less separation. This is because PCA, concentrate on placing dissimilar data points far apart in a lower dimension representation. But in order to represent high dimension data on low dimension, non-linear manifold, it is essential that similar data points must be represented close together, which is something t-SNE does not PCA.

3

[45 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. You may use numpy if you wish but no other libraries.

(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.

To guide you in the right direction with this problem, please find a skeleton of a neural network class below. You absolutely MAY use additional methods beyond those suggested in this template, but I see these methods as the minimum required to implement the model cleanly.

One of the greatest challenges of this implementations is that there are many parts and a bug could be present in any of them. I would strongly encourage you to create unit tests for most modules. Without doing this will make your code extremely difficult to bug. You can create simple examples to feed through the network to validate it is correctly computing activations and node values. Also, if you manually set the weights of the model, you can even calculate backpropagation by hand for some simple examples (admittedly, that unit test would be challenging, but a unit test is possible). You can also verify the performance of your overall neural network by comparing it against the `scikit-learn` implementation and using the same architecture and parameters as your model.

(b) Apply your neural network. Create a training and validation 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 **plotting your learning curves** (training and validation error for each epoch of stochastic gradient descent, where an epoch represents having trained on each of the training samples one time). 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 subplot, and the validation data on the other subplot. 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 and compare against the `scikit-learn MLPClassifier` trained with the same parameters.

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

In [63]:

```
from sklearn.utils import shuffle
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.metrics import roc_curve, auc, roc_auc_score
from sklearn.metrics import accuracy_score
from sklearn.neural_network import MLPClassifier
from matplotlib.colors import ListedColormap
import time
```

ANSWER

(A)

In [64]:

```
class myNeuralNetwork(object):

    def __init__(self, n_in, n_layer1, n_layer2, n_out,learning_rate=0.1):

        '''__init__
        Class constructor: Initialize the parameters of the network including
        the learning rate, layer sizes, and each of the parameters
        of the model (weights, placeholders for activations, inputs,
        deltas for gradients, and weight gradients). This method
        should also initialize the weights of your model randomly
            Input:
                n_in:          number of inputs
                n_layer1:      number of nodes in layer 1
                n_layer2:      number of nodes in layer 2
                n_out:         number of output nodes
                learning_rate: learning rate for gradient descent
            Output:
                none
        '''
        np.random.seed(0)
        self.W1 = np.random.randn(n_layer1, n_in)
        self.b1 = np.zeros((n_layer1,1))
        self.W2 = np.random.randn(n_layer2, n_layer1)
        self.b2 = np.zeros((n_layer2,1))
        self.W3 = np.random.randn(n_out, n_layer2)
        self.b3 = np.zeros((n_out,1))
        self.lr = learning_rate

    def forward_propagation(self, X):
        '''forward_propagation
        Takes a vector of your input data (one sample) and feeds
        it forward through the neural network, calculating activations and
        layer node values along the way.
            Input:
                x: a vector of data represening 1 sample [n_in x 1]
            Output:
                y_hat: a vector (or scaler of predictions) [n_out x 1]
                (typically n_out will be 1 for binary classification)
        '''
        self.Z1 = np.dot(self.W1,X.T)+self.b1
        self.A1 = self.sigmoid(self.Z1)
        self.Z2 = np.dot(self.W2,self.A1)+self.b2
        self.A2 = self.sigmoid(self.Z2)
        self.Z3 = np.dot(self.W3,self.A2)+self.b3
        self.A3 = self.sigmoid(self.Z3)
        return self.A3

    def compute_loss(self, X, y):
        '''compute_loss
        Computes the current loss/cost function of the neural network
        based on the weights and the data input into this function.
        To do so, it runs the X data through the network to generate
        predictions, then compares it to the target variable y using
        the cost/loss function
            Input:
                X: A matrix of N samples of data [N x n_in]
                y: Target variable [N x 1]
            Output:
                loss: a scalar measure of loss/cost
        '''
        y_hat = self.forward_propagation(X)
        cost = 0.5*sum((y-y_hat)**2)
        return cost[0]

    def backpropagate(self, X, Y):
        '''backpropagate
        Backpropagate the error from one sample determining the gradients
        with respect to each of the weights in the network. The steps for
        this algorithm are:
            Input:
                x: A vector of 1 samples of data [n_in x 1]
                y: Target variable [scalar]
            Output:
                Gradients
        '''
        dZ3 = np.multiply((self.A3-Y),self.sigmoid_derivative(self.Z3))
        self.dW3 = np.dot(dZ3,self.A2.T)
        self.db3 = dZ3[0]

        dZ2 = np.multiply(np.dot(self.W3.T,dZ3),self.sigmoid_derivative(self.Z
2))
        self.dW2 = np.dot(dZ2,self.A1.T)
        self.db2 = dZ2[0]
        dZ1 = np.multiply(np.dot(self.W2.T,dZ2), self.sigmoid_derivative(self.
Z1))
        self.dW1 = np.dot(dZ1,X)
        self.db1 = dZ1[0]

    def stochastic_gradient_descent_step(self):
        '''stochastic_gradient_descent_step
        Using the gradient values computer by backpropagate, update each
        weight value of the model according to the familiar stochastic
        gradient descent update equation.

        Input: none
        Output: none
        '''

        self.W1 = self.W1 - self.lr*self.dW1
        self.b1 = self.b1 - self.lr*self.db1
        self.W2 = self.W2 - self.lr*self.dW2
        self.b2 = self.b2 - self.lr*self.db2
        self.W3 = self.W3 - self.lr*self.dW3
        self.b3 = self.b3 - self.lr*self.db3

    def fit(self, X, y, max_epochs=100, learning_rate=0.1, status= True,get_va
lidation_loss=False, X_t=None, y_t=None):
        '''fit
            Input:
                X: A matrix of N samples of data [N x n_in]
                y: Target variable [N x 1]
            Output:
                training_loss:   Vector of training loss values at the end of
each epoch
                validation_loss: Vector of validation loss values at the end o
f each epoch
                                 [optional output if get_validation_loss==True
]
        '''
        self.lr = learning_rate
        y_ = y.reshape(1,-1)
        m =  y_.shape[1]
        self.train_cost_list = []
        self.val_cost_list = []
        for i in range(max_epochs):
            cost_in_epoch = []
            validation_cost = []
            for j in range(m):
                #X, y_[0,:] = shuffle(X, y_[0,:],random_state=0)
                y_hat = self.forward_propagation(X[j].reshape(1,-1))
                cost = self.compute_loss(X[j].reshape(1,-1),y_[0,j])
                self.backpropagate(X[j].reshape(1,-1), y_[0,j])
                self.stochastic_gradient_descent_step()
                cost_in_epoch.append(cost)
            self.train_cost_list.append(np.mean(cost_in_epoch))

            if get_validation_loss:
                y__ = y_t.reshape(1,-1)
                n =  y__.shape[1]
                val_cost = []
                for k in range(n):
                    y_vald_hat = self.forward_propagation(X_t[k].reshape(1,-1)
)
                    cost_vald = self.compute_loss(X_t[k].reshape(1,-1),y__[0,k
])
                    val_cost.append(cost_vald)
                self.val_cost_list.append(np.mean(val_cost))
            #print('Epoch %s | Cost : %s'%(i+1,cost))
        if status:
            print("\n Training Completed with learning rate %s and %s epochs"%
(round((self.lr),2),max_epochs))

    def predict_proba(self, X):
        '''predict_proba
        Compute the output of the neural network for each sample in X, with th
e last layer's
        sigmoid activation providing an estimate of the target output between
0 and 1
            Input:
                X: A matrix of N samples of data [N x n_in]
            Output:
                y_hat: A vector of class predictions between 0 and 1 [N x 1]
        '''
        return(self.forward_propagation(X))


    def predict(self, X, decision_thresh=0.5):
        '''predict
        Compute the output of the neural network prediction for
        each sample in X, with the last layer's sigmoid activation
        providing an estimate of the target output between 0 and 1,
        then thresholding that prediction based on decision_thresh
        to produce a binary class prediction
            Input:
                X: A matrix of N samples of data [N x n_in]
                decision_threshold: threshold for the class confidence score
                                    of predict_proba for binarizing the output
            Output:
                y_hat: A vector of class predictions of either 0 or 1 [N x 1]
        '''
        proba = self.forward_propagation(X)
        pred = np.ones(len(proba[0]))
        pred = [0 if x < decision_thresh else 1 for x in proba[0] ]
        return np.array(pred)

    def sigmoid(self, X):
        '''sigmoid
        Compute the sigmoid function for each value in matrix X
            Input:
                X: A matrix of any size [m x n]
            Output:
                X_sigmoid: A matrix [m x n] where each entry corresponds to th
e
                           entry of X after applying the sigmoid function
        '''
        return 1/(1+np.exp(-X))

    def sigmoid_derivative(self, X):
        '''sigmoid_derivative
        Compute the sigmoid derivative function for each value in matrix X
            Input:
                X: A matrix of any size [m x n]
            Output:
                X_sigmoid: A matrix [m x n] where each entry corresponds to th
e
                           entry of X after applying the sigmoid derivative fu
nction
        '''
        return np.exp(-X)/((1+np.exp(-X))**2)
```

In [75]:

```
# nn = myNeuralNetwork(n_in=2,n_layer1=5,n_layer2=5,n_out=1)
# x = np.array([[1,4],[21,23],[5,1],[50,90],[40,21]])
# y_ = np.array([1,0,1,0,0]).reshape(1,-1)
```

In [53]:

```
# out_a3 = nn.forward_propagation(x)
# nn.compute_loss(y=y_, y_hat=out_a3)
# nn.backpropagate(x,y_)
# nn.stochastic_gradient_descent_step()
```

(B)

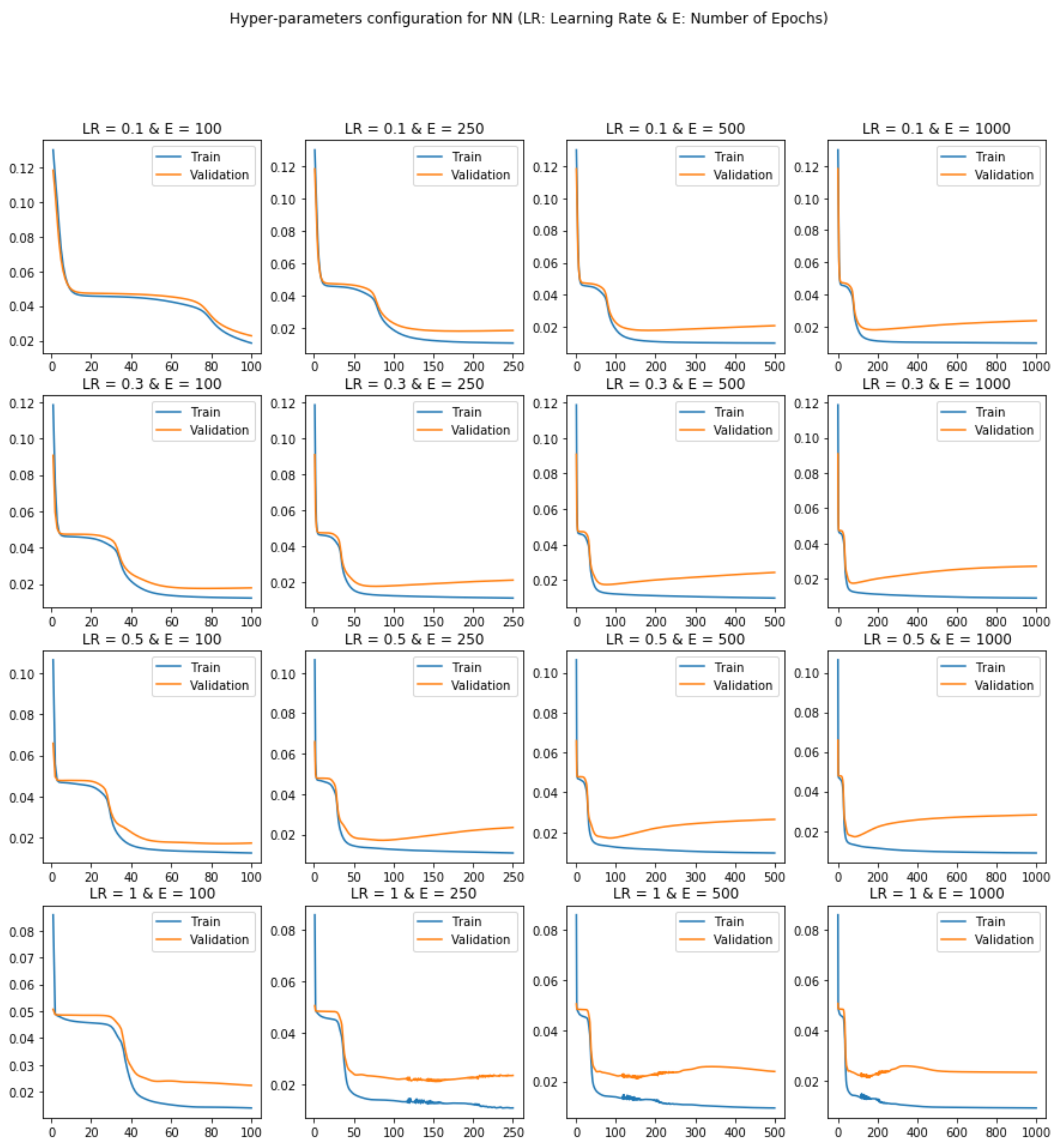
In [65]:

```
X_train,y_train = make_moons(500, noise=0.20)
X_test, y_test = make_moons(100, noise=0.20)
```

Hyper-Parameter selection

In [67]:

```
fig, axs = plt.subplots(4,4, figsize=(15,15))

max_e = [100,250,500,1000]
lern_rate = [0.1,0.3,0.5,1]
for i in range(4):
    for j in range(4):
        nn = myNeuralNetwork(n_in=2,n_layer1=5,n_layer2=5,n_out=1)
        nn.fit(X_train,y_train,max_epochs=max_e[j],learning_rate=lern_rate[i],
                  get_validation_loss = True, X_t = X_test, y_t = y_test, status=
False)
        x_len = range(1,max_e[j]+1)
        axs[i,j].plot(x_len,nn.train_cost_list,label="Train")
        axs[i,j].plot(x_len,nn.val_cost_list,label="Validation")
        axs[i,j].set_title("LR = %s & E = %s"%(lern_rate[i],max_e[j]))
        axs[i,j].legend()
fig.suptitle("Hyper-parameters configuration for NN (LR: Learning Rate & E: Nu
mber of Epochs)")
plt.show()
```



- Based on different configurations of learning-rate and epochs, it seems stable miminium error/cost (less than 0.02 in this case) is being achieved optimally with the **learning rate** of **0.1** and near about **250 epochs** taking into consideration both curve of training and validation, and minimizing the chances of overfitting. Hence, the combination of **0.1** and **250** will be used as the final set of hyper parameters to train the neural network

Final Model

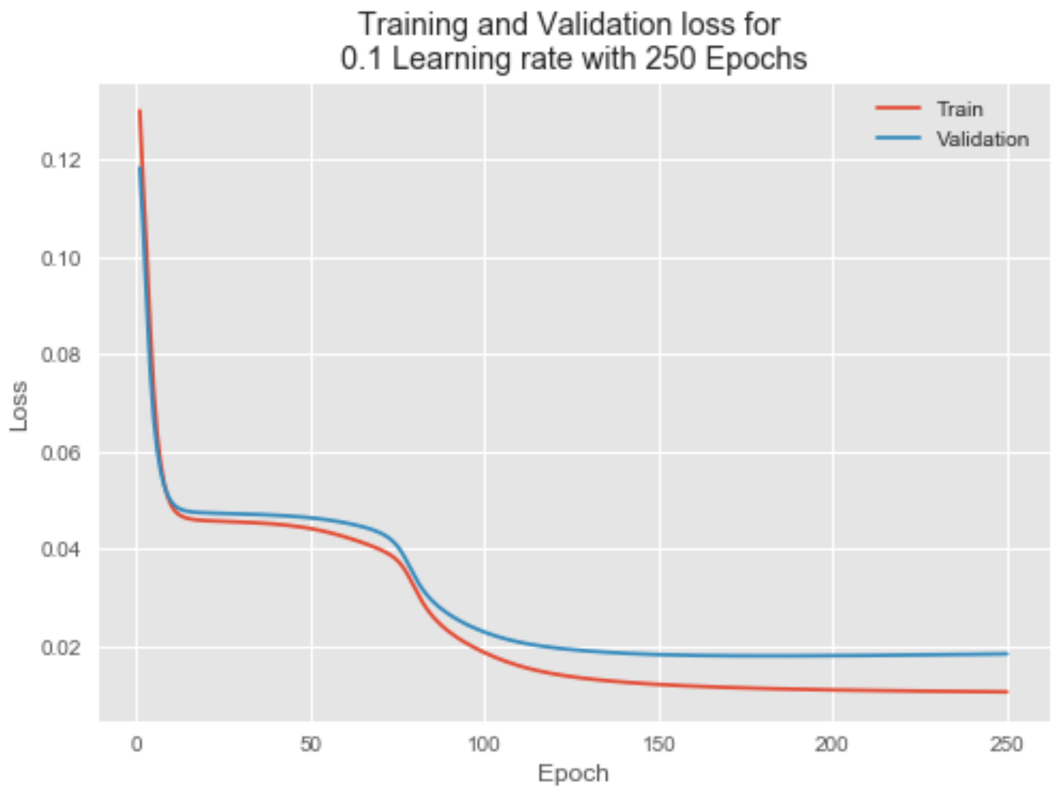
In [104]:

```
# Training the final nn model with the selected hyper paramters.
nn = myNeuralNetwork(n_in=2,n_layer1=5,n_layer2=5,n_out=1)

start_time = time.time()
nn.fit(X_train,y_train,max_epochs=250,learning_rate=0.1,
               get_validation_loss = True, X_t = X_test, y_t = y_test,status=F
alse)

print("--- %s seconds ---" % (time.time() - start_time))

plt.plot(range(1,251),nn.train_cost_list,label="Train")
plt.plot(range(1,251),nn.val_cost_list,label="Validation")
plt.title("Training and Validation loss for \n0.1 Learning rate with 250 Epoch
s")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
plt.plot()
```

```
--- 11.871925830841064 seconds ---
```

Out[104]:

```
[]
```



In [105]:

```
# Insample accuracy
y_pred_insample = nn.predict(X_train, decision_thresh=0.5)
train_accuracy = accuracy_score(y_true=y_train, y_pred = y_pred_insample)*100

# Testing accuracy
y_pred = nn.predict(X_test, decision_thresh=0.5)
test_accuracy = accuracy_score(y_true=y_test, y_pred = y_pred)*100
nn_pred_score_test = nn.predict_proba(X_test)
```

In [106]:

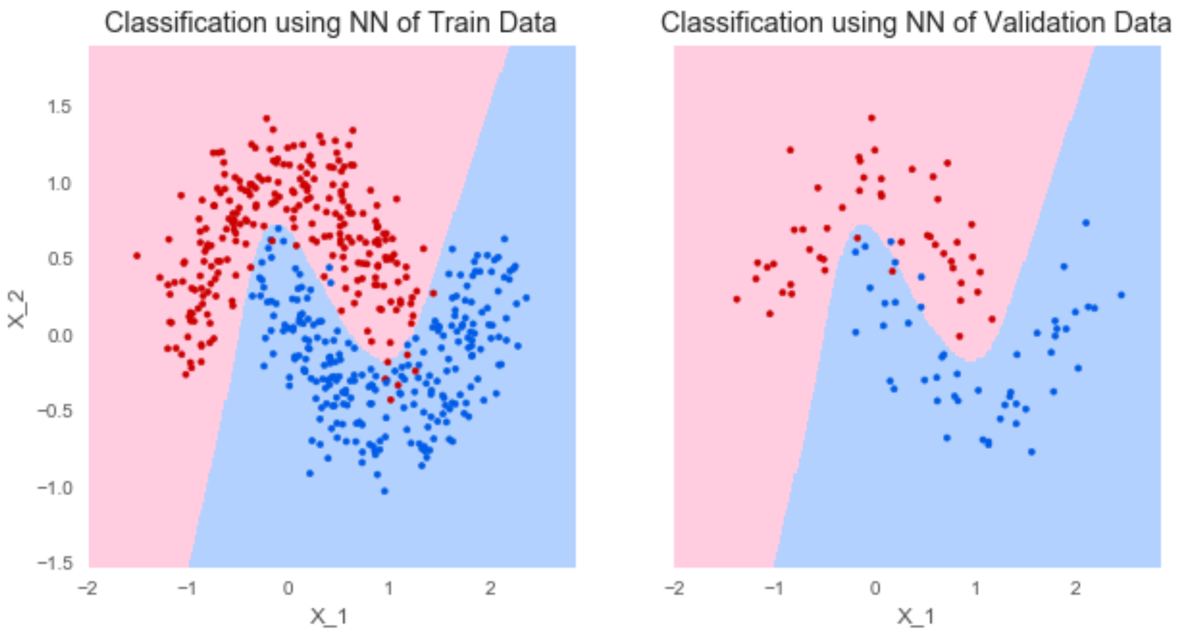
```
print("Training Accuracy : %s"%(round(train_accuracy,2)))
print("Testing Accuracy : %s"%(round(test_accuracy,2)))
```

```
Training Accuracy : 97.8
Testing Accuracy : 94.0
```

Plotting Decision boundary for Test and Train data using Neural Network

In [107]:

```
cmap_light = ListedColormap(['#FFCCE0', '#AAFFAA', '#b3d1ff'])
cmap_bold = ListedColormap(['#cc0000', '#00FF00', '#005ce6'])

plt.style.use('ggplot')
fig, axs = plt.subplots(1,2, figsize=(10,5), sharex=True, sharey=True)

x_min, x_max = X_train[:,0].min() - 0.5, X_train[:,0].max() + 0.5
y_min, y_max = X_train[:,1].min() - 0.5, X_train[:,1].max() + 0.5
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02), np.arange(y_min, y_max, 0.
02))
pred = nn.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

axs[0].pcolormesh(xx, yy, pred, cmap=cmap_light)
axs[0].scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cmap_bold, s=10)
axs[0].set_xlabel("X_1")
axs[0].set_ylabel("X_2")

axs[1].pcolormesh(xx, yy, pred, cmap=cmap_light)
axs[1].scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cmap_bold, s=10)
axs[1].set_xlabel("X_1")

axs[0].set_title("Classification using NN of Train Data")
axs[1].set_title("Classification using NN of Validation Data")
plt.show()
```



- Based on the generated decision boundary, it appears the model performed extremely well with the selected model configuration. Also, both the train and test data had > **94 % accuracy**

MLP Classifier

In [114]:

```
start_time = time.time()
clf = MLPClassifier(solver='sgd', alpha=0, batch_size = 1,
                    hidden_layer_sizes=(5, 5),activation = 'logistic',
                    learning_rate = 'constant', learning_rate_init =0.1)
clf.fit(X_train,y_train)
print("--- %s seconds ---" % (time.time() - start_time))
```

```
--- 4.936761140823364 seconds ---
```

In [117]:

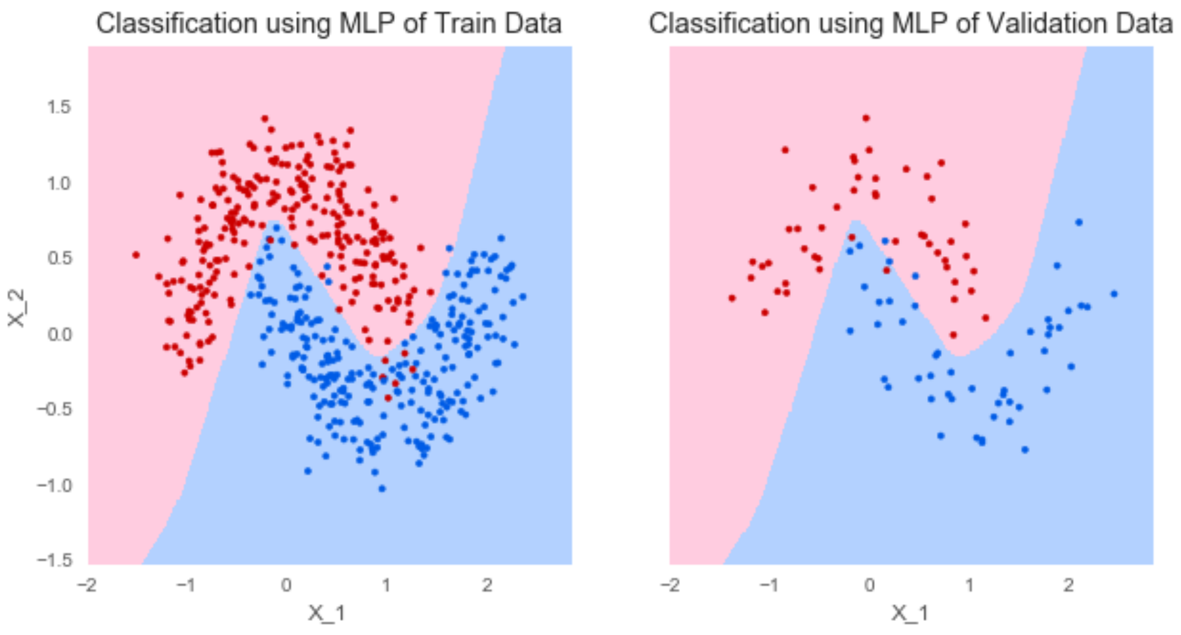
```
# Insample accuracy
mlp_y_pred_insample = clf.predict(X_train)
mlp_train_accuracy = accuracy_score(y_true=y_train, y_pred = mlp_y_pred_insamp
le)*100

# Testing accuracy
mlp_y_pred = clf.predict(X_test)
mlp_test_accuracy = accuracy_score(y_true=y_test, y_pred = mlp_y_pred)*100
mlp_Predscore_test = clf.predict_proba(X_test)

print("MLP Training Accuracy : %s"%(round(mlp_train_accuracy,2)))
print("MLP Testing Accuracy : %s"%(round(mlp_test_accuracy,2)))
```

```
MLP Training Accuracy : 96.8
MLP Testing Accuracy : 95.0
```

Plotting Decision boundary for Test and Train data using MLP Classifier

In [110]:

```
fig, axs = plt.subplots(1,2, figsize=(10,5), sharex=True, sharey=True)

mlp_pred = clf.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

axs[0].pcolormesh(xx, yy, mlp_pred, cmap=cmap_light)
axs[0].scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cmap_bold, s=10)
axs[0].set_xlabel("X_1")
axs[0].set_ylabel("X_2")

axs[1].pcolormesh(xx, yy, mlp_pred, cmap=cmap_light)
axs[1].scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cmap_bold, s=10)
axs[1].set_xlabel("X_1")

axs[0].set_title("Classification using MLP of Train Data")
axs[1].set_title("Classification using MLP of Validation Data")
plt.show()
```

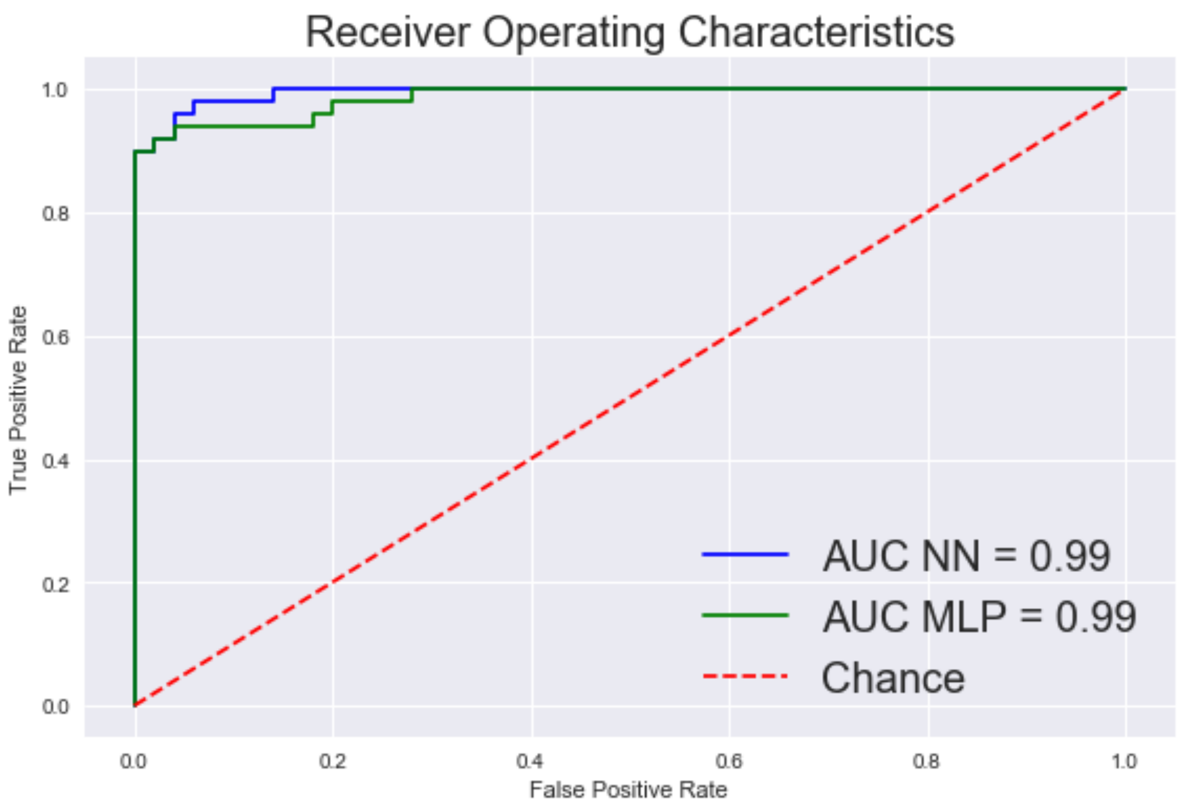


- MLP seems to have generated a linear decision boundary with the same model configuration as custom neural network. However, even MLP provided high accuracy > 95% for both train and test data.

ROC Curve for Neural Network and MLP Classifier

In [118]:

```
fpr, tpr, _ = roc_curve(y_test, nn_pred_score_test[0], pos_label=1)
auc_score = auc(fpr, tpr)

fpr_mlp, tpr_mlp, _ = roc_curve(y_test, mlp_Predscore_test[:,1], pos_label=1)
auc_score_mlp = auc(fpr_mlp,tpr_mlp)

plt.style.use('seaborn')
plt.plot(fpr, tpr, 'b',label="AUC NN = %s"%(round(auc_score,2)))
plt.plot(fpr_mlp, tpr_mlp, 'g',label="AUC MLP = %s"%(round(auc_score_mlp,2)))
plt.plot([0, 1], [0, 1],'r--', label = "Chance")
plt.xlim([-0.05, 1.05])
plt.ylim([-0.05, 1.05])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.title("Receiver Operating Characteristics", size=20)
plt.legend(fontsize=20)
plt.tight_layout()
```



- On Comparing the ROC of Neural Network and Multi-Layer Perceptron (MLP), it's clear that both the model performed extremely well with an **AUC of 0.99 for Neural network** and an **AUC of 0.99 for MLP** for the **TEST data**.
- However, it was also evident that MLP had faster execution time (almost 2 times faster) as compared to neural network in this case.

(C)

Two ways in which this neural network implementation could be improved:

1. Currently the network is following a "Stochastic gradient descent" approach, which although being good takes a lot of time and computationally expensive. This issue could be resolved if we the network could peform mini-batch/batch gradient descent. This is improve the run time and at the same time, with batch size as an hyper parameter provide flexibility for the cases which essentially needs stochastic process.
2. The current neural network is static in terms of configuration, where number of hidden layer and nodes cannot be changed. The neural network could be improved if the configuration could be made dynamic, which would allow us to change the the number of hidden layers and nodes dynamically.
3. Also, the current network is using "Sigmoid" activation function, which is not advisable for neural network due to vanishing gradient problem. Other activation functions such as "tanh" or "relu" could be used in the layers for better model performance.