Assignment 4 - Unsupervised Learning and More Supervised Learning

Frank Xu

Netid: *hx44*

https://github.com/Frank-Xu-Huaze/machine-learning-course/blob/master/4/Assignment4.ipynb (https://github.com/Frank-Xu-Huaze/machine-learning-course/blob/master/4/Assignment4.ipynb)

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.

(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

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

ANSWER

In [18]:

```
# import libraries
%config InlineBackend.figure_format = 'retina'
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time
import warnings
import keras
import time
from sklearn.datasets import make_blobs
np.random.seed(124)
```

In [2]:

```
# (a)
class Kmeans:
    def __init__(self, k = 3):
        self.k = k
        self.label = None
        self.inertia = None
        assert self.k > 0 # assert k to be larger than 0
        pass

    def labels(self, x, c):
        # utility function to get an array of clustered labels given x and c
        dimension = len(x.T)
        diff = np.repeat(x, self.k, axis = 0).reshape(len(x) * self.k, dimension
) - np.array(list(c)*len(x))
        dis = np.sum(diff**2, axis = 1).reshape(len(x), self.k)
        return np.array([min([(v,i) for i,v in enumerate(row)])[1] for row in di
s])

    def single_cluster(self, x, max_iter = 50, verbose = 0): # single-time clust
er
        assert max_iter > 0 # assert max_iter to be larger than 0
        if self.k > len(x): # if number of clusters is bigger than number of X,
use number of X instead
            self.k = len(x)
            print('Warning: Assigned class number larger than total number of X,
use number of X instead.')
        c = x[np.random.choice(x.shape[0], self.k, replace=False)] # initializin
g centoids by randomly choose k points from X
        label_, label = np.zeros(len(x)), np.zeros(len(x)) # initializing predic
ted labels by 0s
        for i in range(max_iter):
            label = self.labels(x, c) # get labels for current iteration
            if np.sum(np.abs(label_ - label)) < 2 and verbose: # early stopping
till no labels change between iterations
                print('Class change less than 2 between iterations, number of it
erations = {}'.format(i+1))
                break
            label_ = label
            c = np.array([np.mean(x[label == i], axis = 0) for i in range(len(c)
)]) # update centroids
        self.label, self.inertia = label, np.sum((x - c[label]) ** 2) # store la
bel and inertia for each cluster trials
        return self

    def fit(self, x, trials = 8, max_iter = 50, verbose = 0): # function to run
single-time cluster multiple times to get best result
        t0 = time.time()
        for i in range(trials):
            temp = self.single_cluster(x = x, max_iter = max_iter, verbose = ver
bose)
            if i == 0: # store the initial result
                best_label, best_inertia = temp.label, temp.inertia
            else: # update if better
                if temp.inertia < best_inertia:
                    best_label, best_inertia = temp.label, temp.inertia
            pass
        if verbose:
            print('Clustering time: {0:0.3f}s'.format(time.time()-t0))
        self.label, self.inertia = best_label, best_inertia
        return self
```

In [3]:

```
warnings.filterwarnings("ignore", category=RuntimeWarning)
K = np.arange(1, 11)
for centers in range(2,6):
    X, y = make_blobs(n_samples = 500, centers = centers, n_features = 2)
    inertia = []
    for k in K:
        inertia.append(Kmeans(k).fit(X).inertia)
        pass
    fig = plt.figure(figsize=(20, 6))
    ax1, ax2, ax3 = plt.subplot(1,3,1), plt.subplot(1,3,2), plt.subplot(1,3,3)
    ax1.scatter(X.T[0], X.T[1], c = y)
    ax2.scatter(X.T[0], X.T[1], c = Kmeans(centers).fit(X).label)
    ax3.plot(K, inertia)
    ax1.set_xlabel('feature 1'); ax1.set_ylabel('feature 2'); ax2.set_xlabel('fe
ature 1'); ax2.set_ylabel('feature 2');
    ax3.set_xlabel('k'); ax3.set_ylabel('Sum of Squares Error');
    fig.text(.5, .01,
             'Figure {}: Generated {}-blobs with original labels (left), with cl
ustered labels (middle), and elbow curve for k from 1 to 10 (right)'.format(cent
ers-1, centers),
             ha='center')
    plt.show()
```

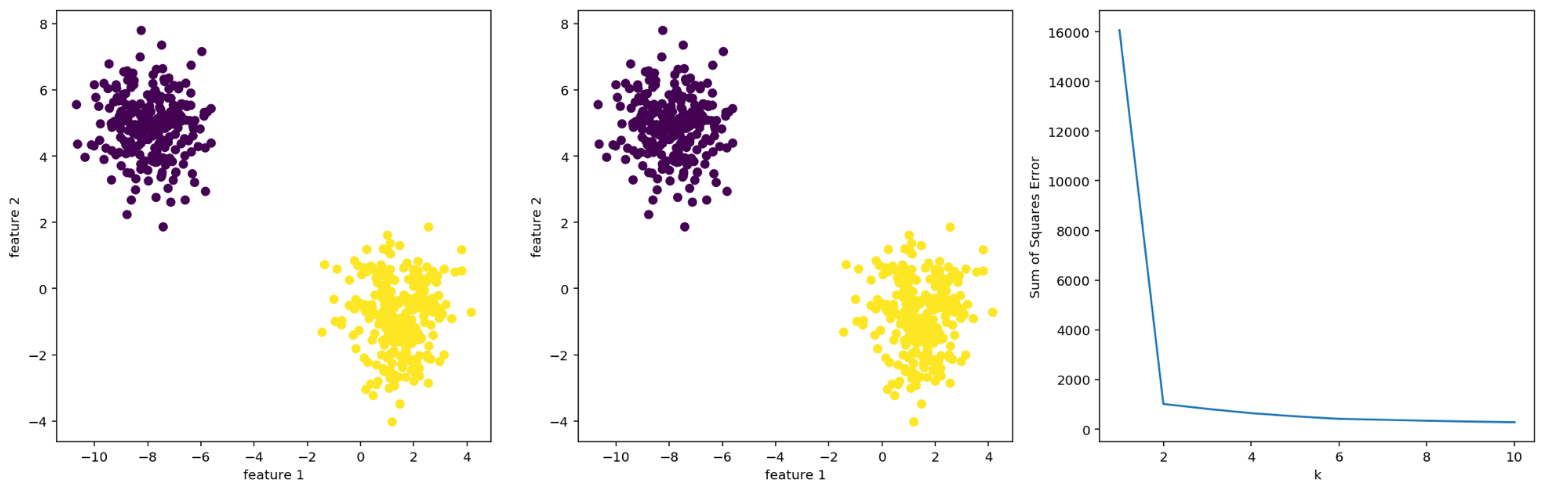

Figure 1: Generated 2-blobs with original labels (left), with clustered labels (middle), and elbow curve for k from 1 to 10 (right)

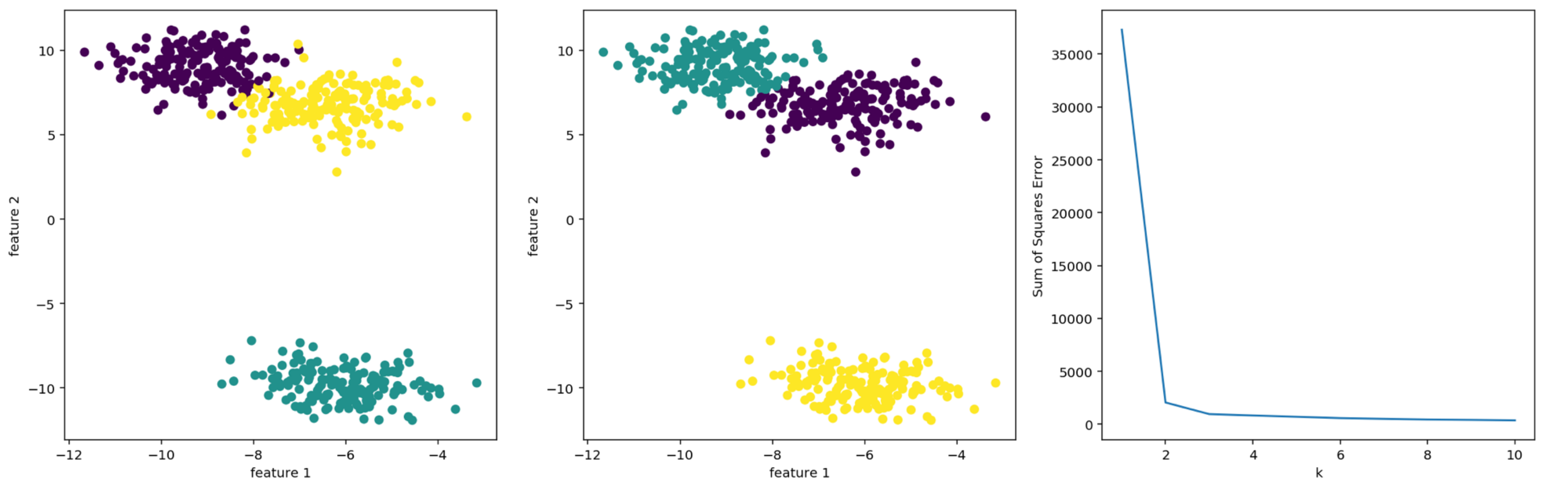

Figure 2: Generated 3-blobs with original labels (left), with clustered labels (middle), and elbow curve for k from 1 to 10 (right)

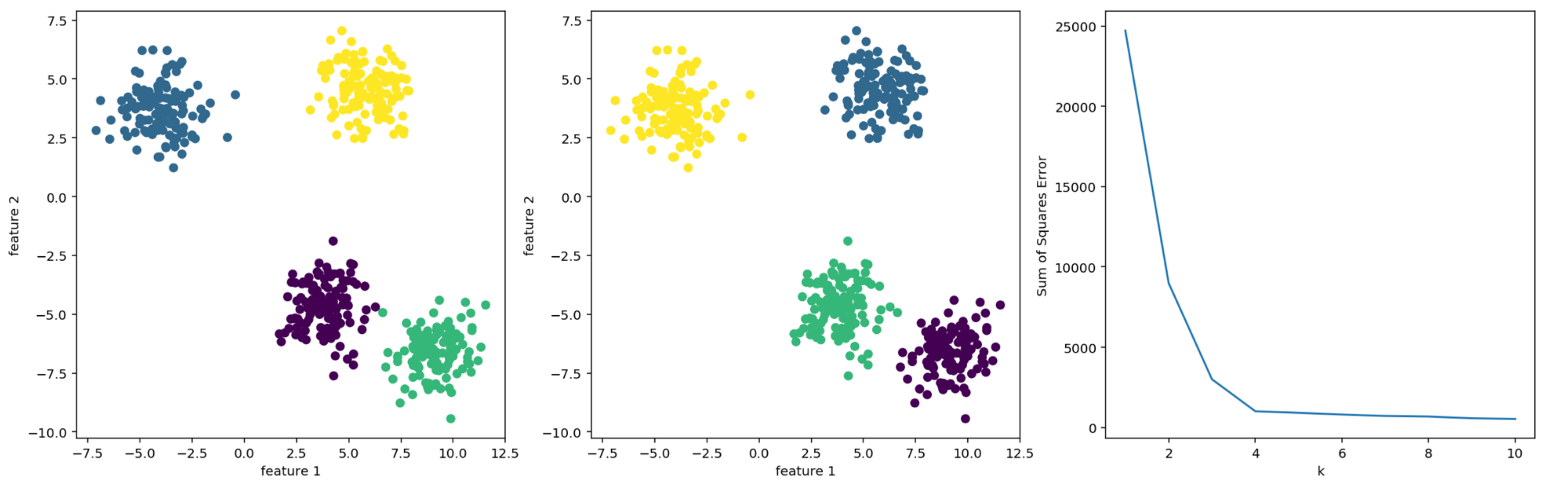

Figure 3: Generated 4-blobs with original labels (left), with clustered labels (middle), and elbow curve for k from 1 to 10 (right)

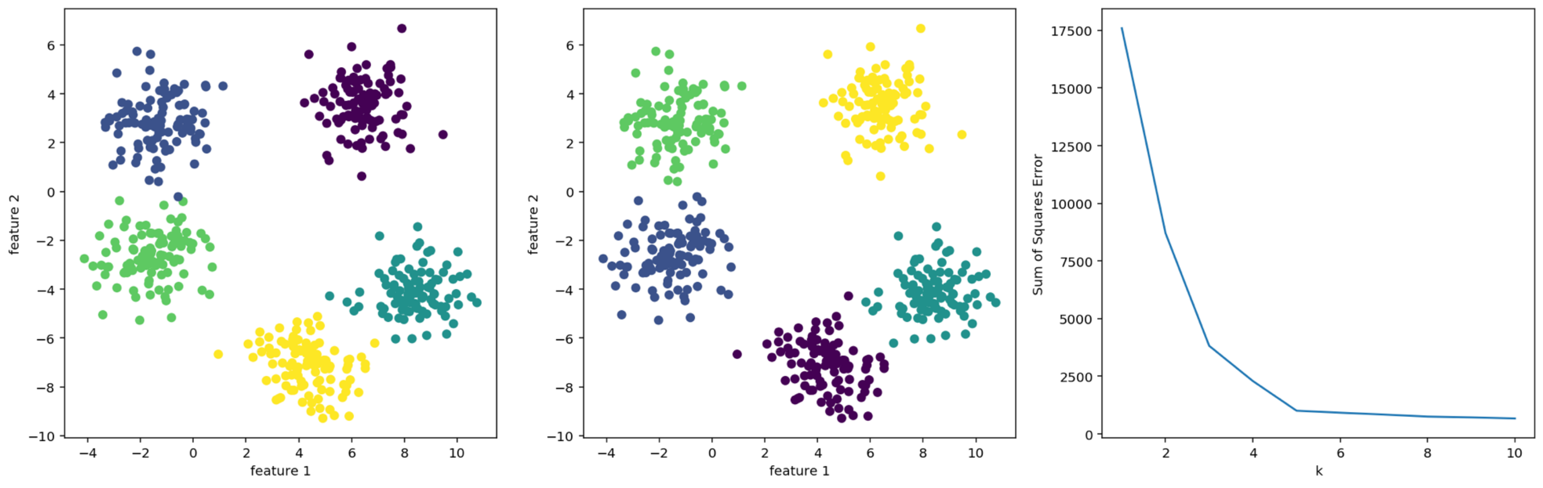

Figure 4: Generated 5-blobs with original labels (left), with clustered labels (middle), and elbow curve for k from 1 to 10 (right)

(a)

For a 2-blobs data, k = 2 is the elbow point, because the 2 clusters are well separated.

For a 3-blobs data (as shown above), k = 2 is still the elbow point, because two of those clusters actually merged together, so 2 clusters actually makes more sense than 3.

For a 4-blobs data, k = 4 is the elbow point, because the 4 clusters are well separated.

For a 5-blobs data, k = 5 is the elbow point, because the 5 clusters are well separated.

(b)

(1) k-means: It is computational simpler than most of the clustering algorithms. Struggles when cluster variance varied and non-linear. You have to set number of k in advance, and the result is not so consistent. Hard boundaries in comparison to GMM.

(2) agglomerative clustering: It is one of the most sophisticated clustering algorithms (slower). Struggles when clusters are not well-separated. Works for non-linear. You can choose where to cut the dendrogram after went over all clusters. The result is consistent.

(3) Gaussian mixture models: It is computational simpler than most of the clustering algorithms. Struggles when non-linear. Have to set number of clusters in advance, overall consistent. Soft boundaries (mixed membership) compared with K-means.

(4) DBSCAN: Automatically choose number of clusters for you. Struggles with siginificant variation in cluster density. Make outliers (so the border points might not be accurate). Works with non-linear.

(5) spectral clustering: It is one of the most sophisticated clustering algorithms (slower). Uses a different distance metric with similarity. Needs to assign number of clusters beforehand. Works with non-linear.

In [4]:

```
# (c)
# loading data
agg = pd.read_csv('Aggregation.txt', sep = '\t', header = None).iloc[:,0:2].valu
es
com = pd.read_csv('Compound.txt', sep = '\t', header = None).iloc[:,0:2].values
d31 = pd.read_csv('D31.txt', sep = '\t', header = None).iloc[:,0:2].values
jai = pd.read_csv('jain.txt', sep = '\t', header = None).iloc[:,0:2].values
```

In [5]:

```
from sklearn.cluster import KMeans, AgglomerativeClustering, DBSCAN, SpectralClu
stering
from sklearn.mixture import GaussianMixture

# storing predicted cluster labels
agg_lab = [
    KMeans(n_clusters = 7).fit(agg).labels_,
    AgglomerativeClustering(n_clusters = 7, linkage='average').fit(agg).labels_,
    GaussianMixture(n_components=7, covariance_type='full').fit_predict(agg),
    DBSCAN(eps=2, min_samples=8).fit(agg).labels_,
    SpectralClustering(n_clusters=7).fit(agg).labels_]

com_lab = [
    KMeans(n_clusters = 6).fit(com).labels_,
    AgglomerativeClustering(n_clusters = 6, linkage='ward').fit(com).labels_,
    GaussianMixture(n_components=6, covariance_type='full').fit_predict(com),
    DBSCAN(eps=1.5, min_samples=4).fit(com).labels_,
    SpectralClustering(n_clusters=6).fit(com).labels_]

d31_lab = [
    KMeans(n_clusters = 31).fit(d31).labels_,
    AgglomerativeClustering(n_clusters = 31, linkage='ward').fit(d31).labels_,
    GaussianMixture(n_components=31, covariance_type='full').fit_predict(d31),
    DBSCAN(eps=0.3, min_samples=1).fit(d31).labels_,
    SpectralClustering(n_clusters=31).fit(d31).labels_]

jai_lab = [
    KMeans(n_clusters = 2).fit(jai).labels_,
    AgglomerativeClustering(n_clusters = 2, linkage='complete').fit(jai).labels_
,
    GaussianMixture(n_components=2, covariance_type='tied').fit_predict(jai),
    DBSCAN(eps=2.5, min_samples=20).fit(jai).labels_,
    SpectralClustering(n_clusters=2).fit(jai).labels_]

# some utility lists for plotting
dataset = [agg, com, d31, jai]
labels = [agg_lab, com_lab, d31_lab, jai_lab]
methods = ['KMeans', 'Agglomerative', 'GMM', 'DBSCAN', 'Spectral']
dataname = ['Aggregation', 'Compound', 'D31', 'Jain']
```

In [6]:

```
# plotting
for i, lab in enumerate(labels):
    fig = plt.figure(figsize=(20, 3.6))
    data = dataset[i]
    for idx, label in enumerate(lab):
        plt.subplot(1,5,idx+1)
        plt.scatter(data.T[0], data.T[1], c = label, cmap = 'Paired', s = 8)
        plt.xticks([]); plt.yticks([])
        plt.xlabel('x1'); plt.ylabel('x2')
        if i == 0:
            plt.title(methods[idx], size=15)
        if idx == 0:
             plt.text(-.6, .5, dataname[i], transform=plt.gca().transAxes, size=
15, horizontalalignment='left')
fig.text(.5, .01, 'Figure 5: Five different clustering applications on four give
n datasets', ha='center')
plt.show()
```

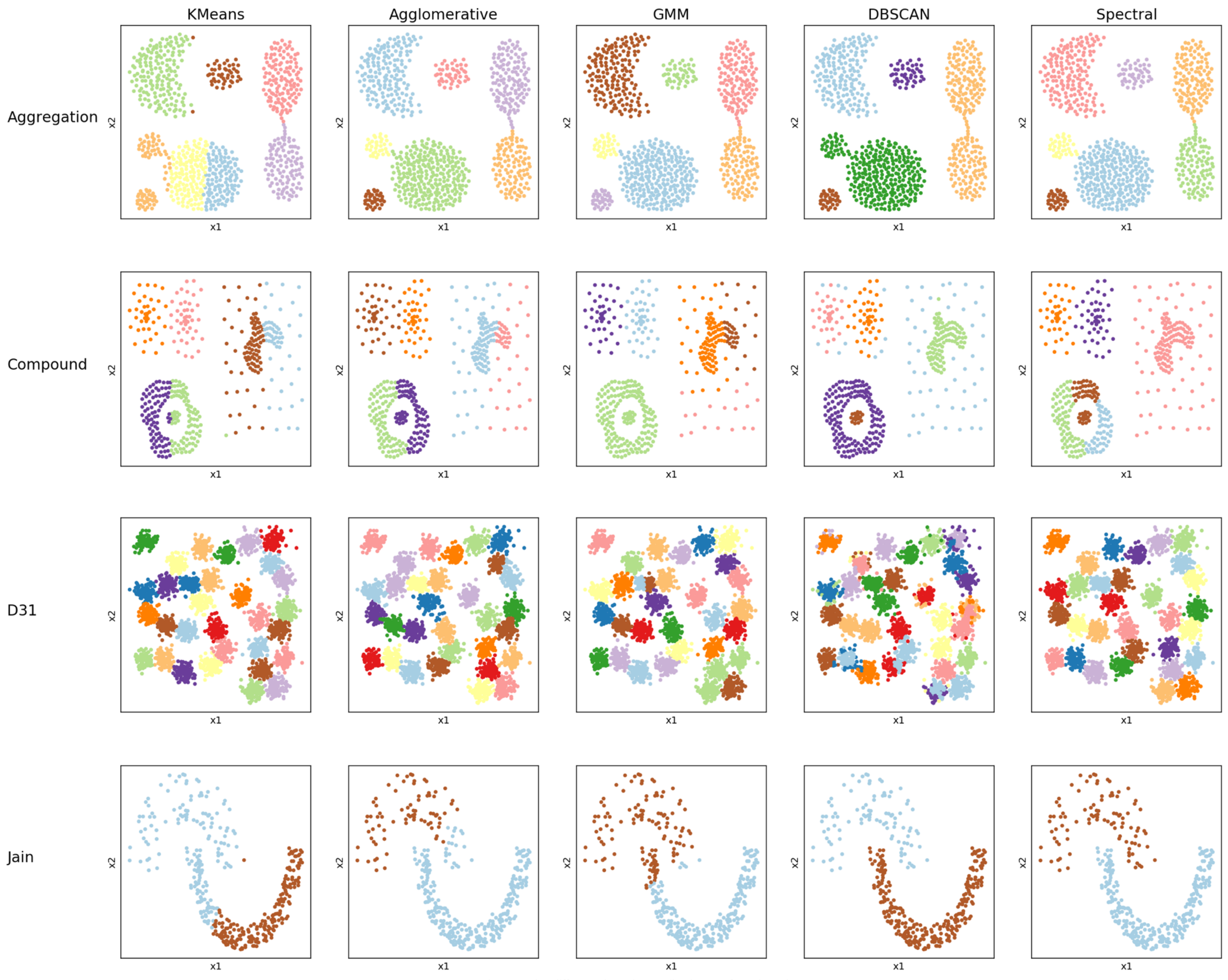

Figure 5: Five different clustering applications on four given datasets

(c)

Given the performance of five different clustering applications above:

For the ***Aggregation*** dataset, ***Agglomerative*** and ***Spectral*** gave the best output. The ***difference in variance*** killed KMeans and GMM, and the ***connection between groups*** greatly affected DBSCAN.

For the ***Compound*** dataset, ***DBSCAN*** gave the best output. The ***Non-linearity*** killed KMeans and GMM, while the ***outliers*** greatly affected Spectral and Agglomerative. DBSCAN is great in dealing with outliers.

For the ***D31*** dataset, ***all methods other than DBSCAN*** gave the best output. The dataset is well separated, linear, and consistent in variance. However, DBSCAN can have trouble in telling points on the boarder.

For the ***Jain*** dataset, ***DBSCAN and Spectral*** gave the best output. The ***Non-linearity*** killed KMeans and GMM, and the ***not so well separated*** points affected Agglomerative.

2

[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.

(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 (https://distill.pub/2016/misread-tsne/). A video introducing this method can be found here (https://www.youtube.com/watch?v=RJVL80Gg3lA&list=UUtXKDgv1AVoG88PLl8nGXmw) 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.

(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.*

ANSWER

In [7]:

```
# (a)
from sklearn import datasets
digits = datasets.load_digits()
x_train = digits.data
y_train = digits.target
```

In [8]:

```
from sklearn.decomposition import PCA
pca = PCA(n_components=2).fit_transform(x_train)

fig = plt.figure(figsize = (10,8))
plt.scatter(pca.T[0], pca.T[1], c = y_train, cmap=plt.get_cmap('jet', 10), edgec
olor='none', s = 10)
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.colorbar()
fig.text(.5, .01, 'Figure 6: 2-components PCA graph on the MNIST training datase
t', ha='center')
plt.show()
```

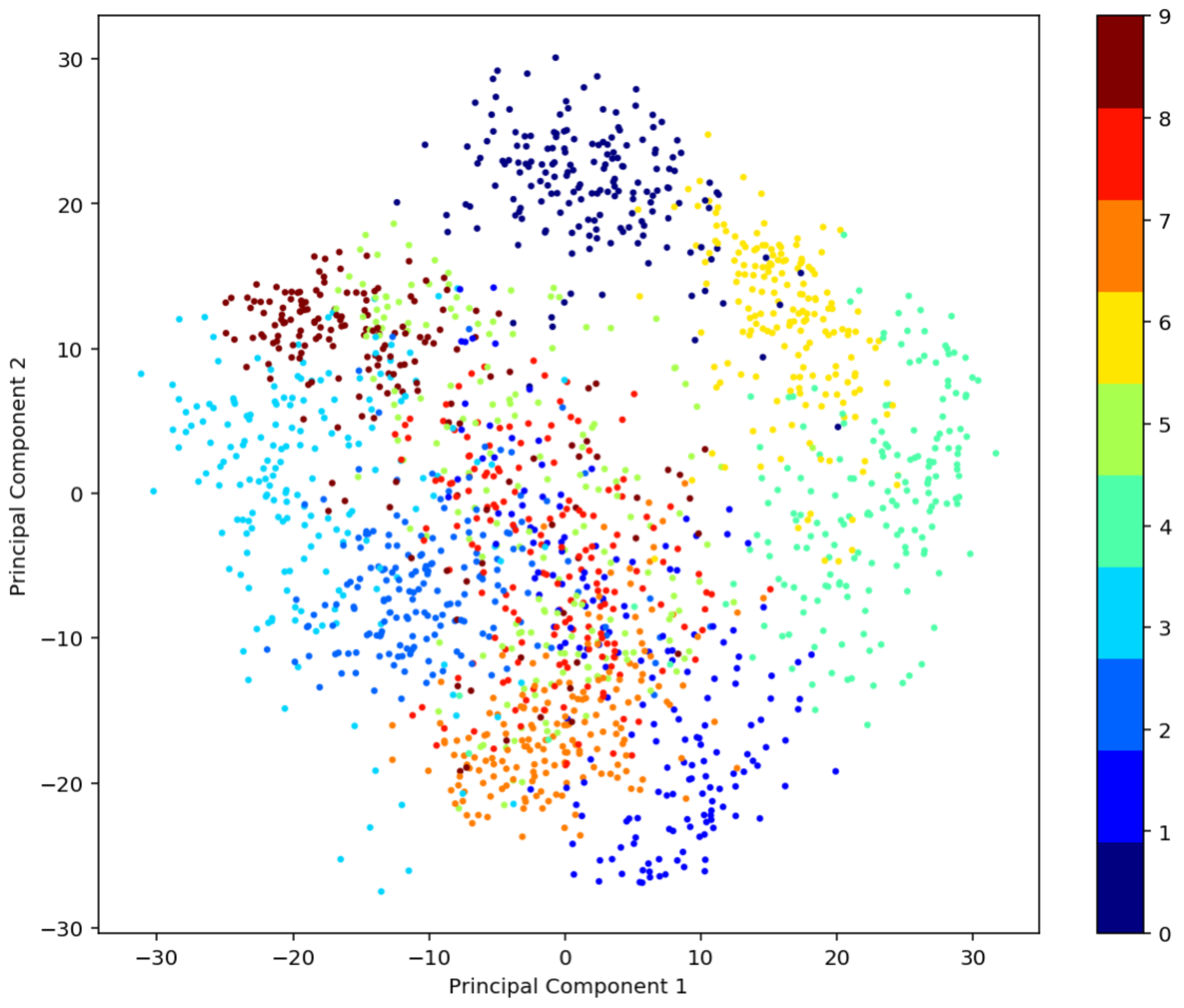

Figure 6: 2-components PCA graph on the MNIST training dataset

In [9]:

```
# (b)
from sklearn.manifold import TSNE
tsne = TSNE(n_components=2, init='random', angle = 0.6, n_iter = 500).fit_transf
orm(x_train)
fig = plt.figure(figsize = (10,8))
plt.scatter(tsne.T[0], tsne.T[1], c = y_train, cmap=plt.get_cmap('jet', 10), edg
ecolor='none', s = 10)
plt.xlabel('T-SNE Component 1')
plt.ylabel('T-SNE Component 2')
plt.colorbar()
fig.text(.5, .01, 'Figure 7: 2-components T-SNE graph on the MNIST training data
set', ha='center')
plt.show()
```

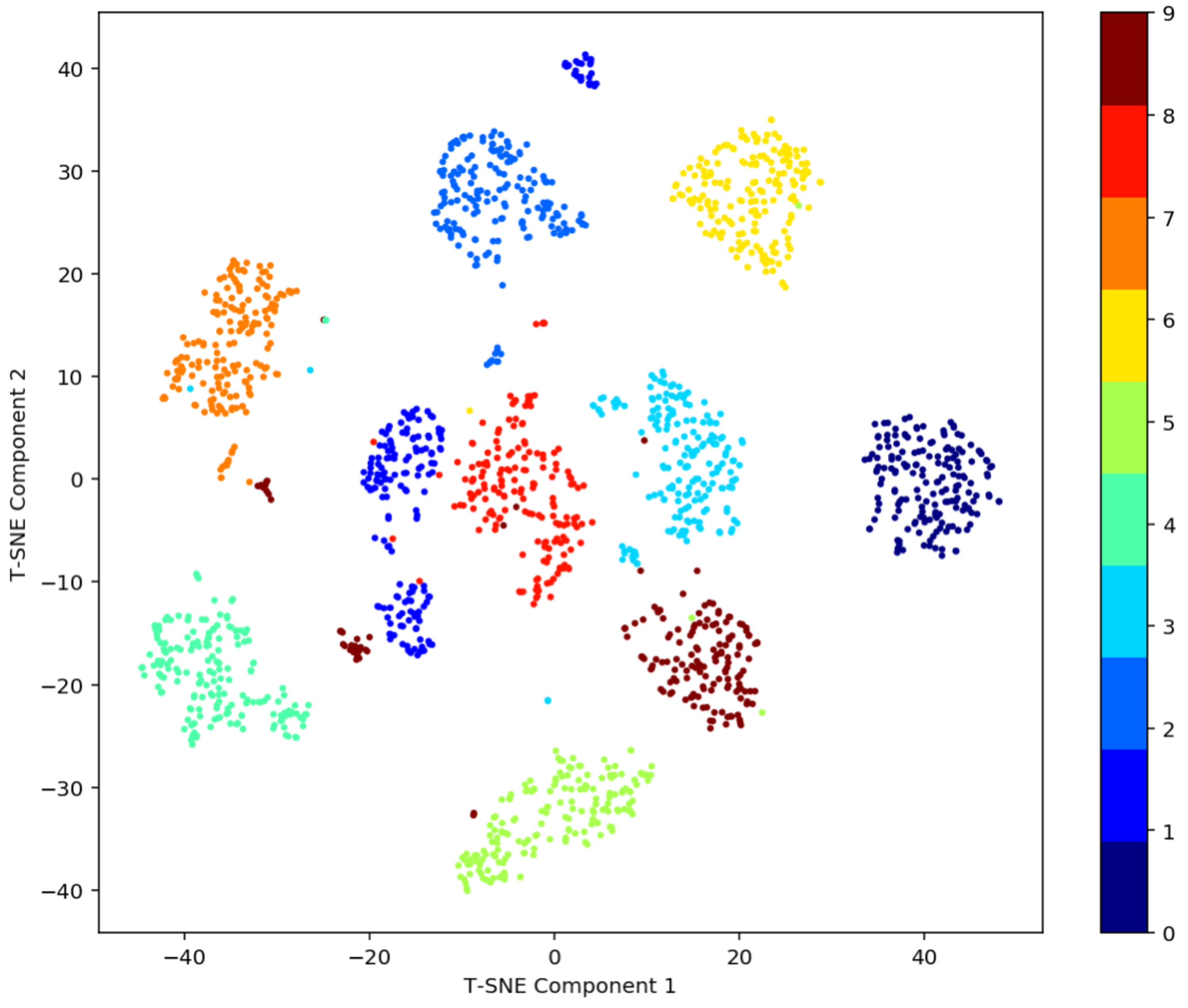

Figure 7: 2-components T-SNE graph on the MNIST training dataset

(c)

Compared with PCA, T-SNE seems to cluster the data better: most of the labels in T-SNE are clearly separated and well clustered, except for 9 and 1, which is a little bit scattered. However in PCA, most digits are kind of mixed with each other, we can see the majority and centroids for different digits but the boarders are not clear enough.

Given the observation, the pro of PCA is that it is extremely fast (since it is basically just calculating the eigenvectors), and the con is that it is not well separated in comparison to T-SNE. When we don't have the labels (colors), we cannot tell most clusters.

The pro of T-SNE, however, is the very well-separated clusters, we can observe the clusters even without the labels given. The con can be it takes too much time, it takes O(NlogN) time according to document.

3

[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.

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

(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 $k = 1, 5, 10, 25$.

(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.

ANSWER

In [10]:

```
# (a)
x_5 = x_train[y_train == 5]
x_5_3d = x_5.reshape(x_5.shape[0], 8, 8)
fig = plt.figure(figsize=(18, 3))
for i in range(0,10):
    plt.subplot(1, 10, i+1)
    plt.imshow(x_5_3d[i], cmap='gray')
    plt.xlabel("pixels")
    plt.ylabel("pixels")
    fig.text(.5, .01, 'Figure 8: 10 examples from number 5', ha='center')
plt.show()
```

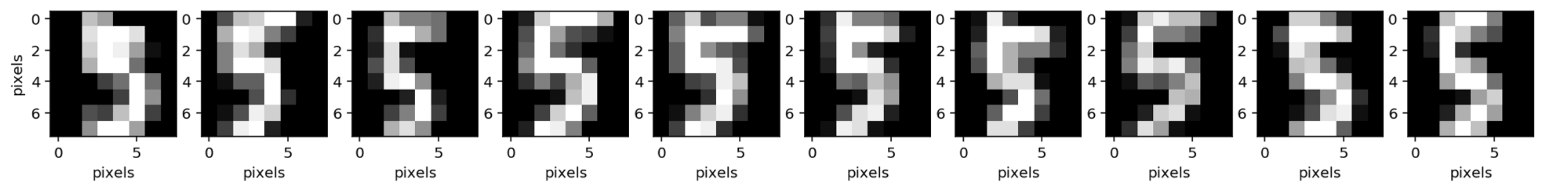

Figure 8: 10 examples from number 5

In [11]:

```
# (b)
var = []
for n in range(1, 65):
    pca = PCA(n_components=n).fit(x_5)
    var.append(sum(pca.explained_variance_ratio_))
fig = plt.figure(figsize=(8, 6))
plt.plot(range(1, 65), var, label='Fraction of Variance Explained')
plt.ylabel('Fraction of Variance Explained')
plt.xlabel('Number of Principal components')
plt.legend(loc='lower right')
fig.text(.5, .01, 'Figure 9: Fraction of Variance Explained as a Function of the
Number of Components', ha='center')
plt.show()
```

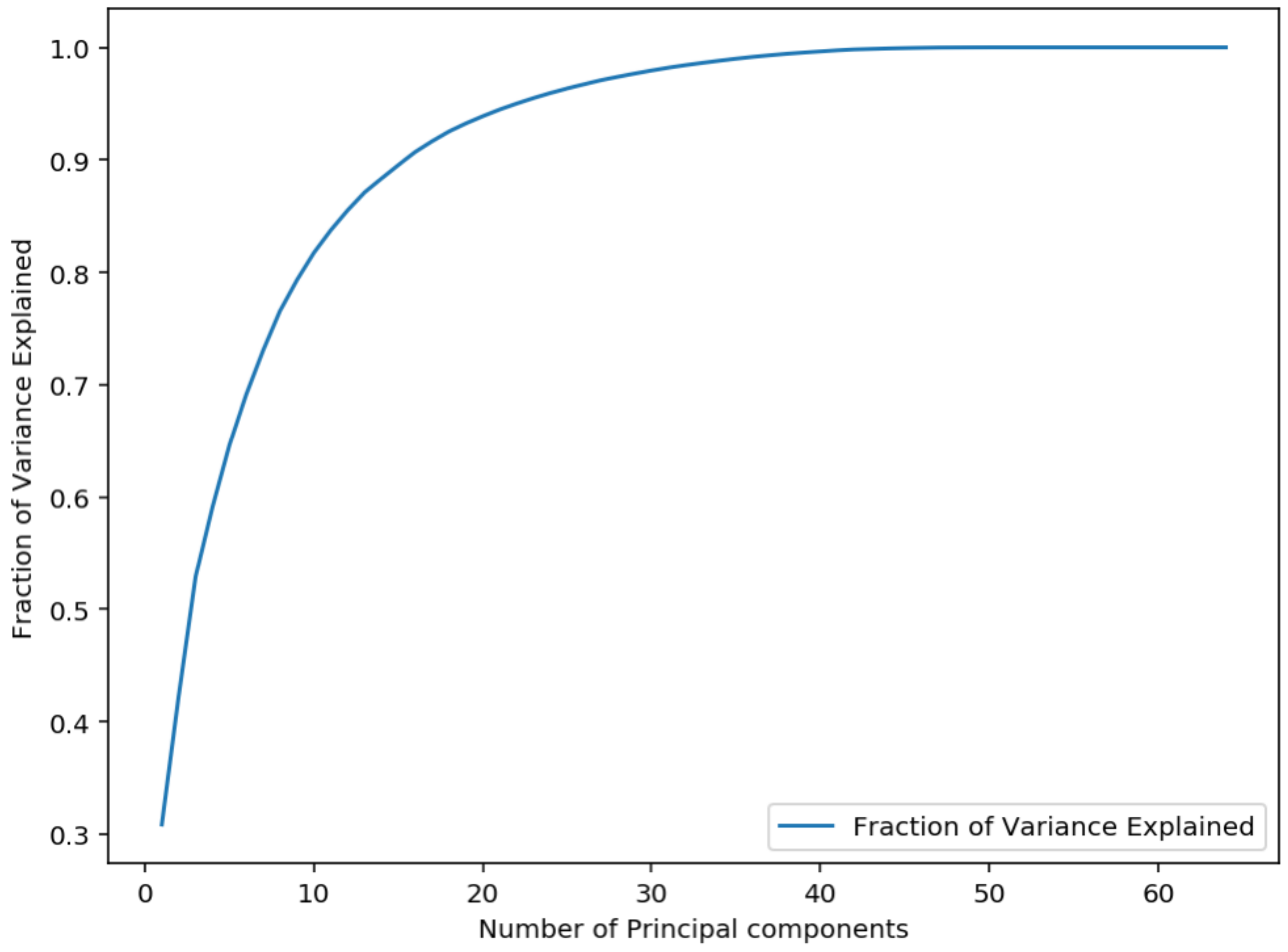

Figure 9: Fraction of Variance Explained as a Function of the Number of Components

In [12]:

```
# (c)
x = x_5_3d[-1]
fig = plt.figure(figsize=(18, 3.7))
plt.subplot(1,5,1)
plt.imshow(x, cmap = 'gray')
plt.title('Original')

for idx, n in enumerate([1,5,10,25]):
    pca = PCA(n_components=n)
    compressed = pca.fit_transform(x_5)
    x_trans = pca.inverse_transform(compressed).reshape(x_5.shape[0], 8, 8)
    plt.subplot(1,5,idx+2)
    plt.imshow(x_trans[-1], cmap = 'gray')
    plt.title('Compressed with {} components'.format(n))
fig.text(.5, .01, 'Figure 10: An example of digit 5 under PCA by different numbe
r of components', ha='center')
plt.show()
```

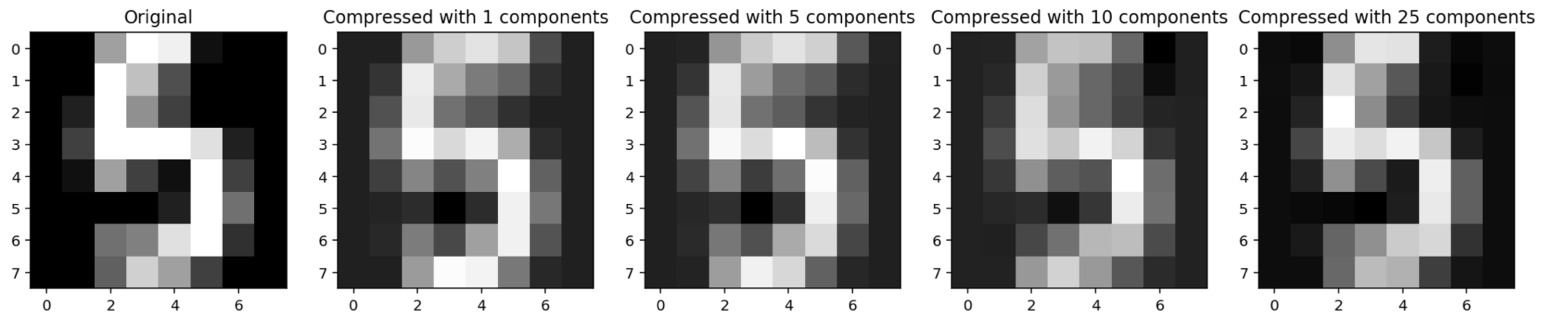

Figure 10: An example of digit 5 under PCA by different number of components

(d)

According to the graphs shown above, you can see the trend in the 10-components plot, but 25-componemts are required to well-approximate the original plot.

compression rate for 1 component: $1 - 1/64 = 0.984$

compression rate for 5 component: $1 - 5/64 = 0.922$

compression rate for 10 component: $1 - 10/64 = 0.844$

compression rate for 25 component: $1 - 25/64 = 0.610$

4

[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.

(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.

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

ANSWER

In [13]:

```
# (a)
class NN:
    def __init__(self):
        # initializing weights for different layers
        self.w1 = np.random.randn(5,2)
        self.w2 = np.random.randn(5,5)
        self.w3 = np.random.randn(1,5)

    def sig(self, a): # sigmoid funcion
        return 1 / (1 + np.exp(-a))
    def sig_deriv(self, z): # the derivative of sigmoid function
        return z * (1 - z)

    def forward(self, X): # forward propagation to get all outputs
        self.a1 = np.dot(self.w1, X.T)
        self.z1 = self.sig(self.a1)
        self.a2 = np.dot(self.w2, self.z1)
        self.z2 = self.sig(self.a2)
        self.a3 = np.dot(self.w3, self.z2)
        self.z3 = self.sig(self.a3)
        return self

    def backward(self, X, y): # backward propagation to get all derivatives
        self.da3 = (self.z3 - y) * self.sig_deriv(self.z3)
        self.dw3 = np.dot(self.da3, self.z2.T)
        self.da2 = np.dot(self.w3.T, self.da3) * self.sig_deriv(self.z2)
        self.dw2 = np.dot(self.da2, self.z1.T)
        self.da1 = np.dot(self.w2.T, self.da2) * self.sig_deriv(self.z1)
        self.dw1 = np.dot(self.da1, X)
        return self

    def fit(self, X, y, lr = 0.05, epoch = 1500):
        self.error = []
        for _ in range(epoch): # loop the fitting process within given epoch num
bers
            # random sort the index for SGD
            indexes = np.random.choice(np.arange(0, len(y)), size = len(y), repl
ace = False)
            for i in indexes:
                self.forward(X[i].reshape(1,-1)).backward(X[i].reshape(1,-1), y[
i].reshape(1,-1))
                self.w1 -= lr * self.dw1
                self.w2 -= lr * self.dw2
                self.w3 -= lr * self.dw3
                # update weight after fitting each random chosen data
            self.error.append(0.5*np.sum(np.square(self.forward(X).z3 - y))/len(
y))
        return self

    def predict_proba(self, X): # return a probability prediction given data X
        self.forward(X)
        return np.squeeze(self.z3)

    def predict(self, X): # return a classification prediction given data X
        self.forward(X)
        return np.squeeze(self.z3)>0.5
```

In [14]:

```
# (b)
# loading data
from sklearn.datasets import make_moons
x_train, y_train = make_moons(500, noise=0.20)
x_test, y_test = make_moons(100, noise=0.20)

# training with learning rate of 0.05 and epoch of 2000.
epoch = 2000
t0 = time.time()
nn = NN().fit(x_train, y_train, lr = 0.05, epoch = epoch)
print('Training time:', time.time() - t0, 's')

# plotting training error
fig = plt.figure(figsize=(8, 6))
plt.plot(np.arange(epoch), nn.error)
plt.xlabel('Epoches'); plt.ylabel('Error')
fig.text(.5, .01, 'Figure 11: Plot of training error and epoches', ha='center')
plt.show()
```

```
Training time: 34.69738960266113 s
```

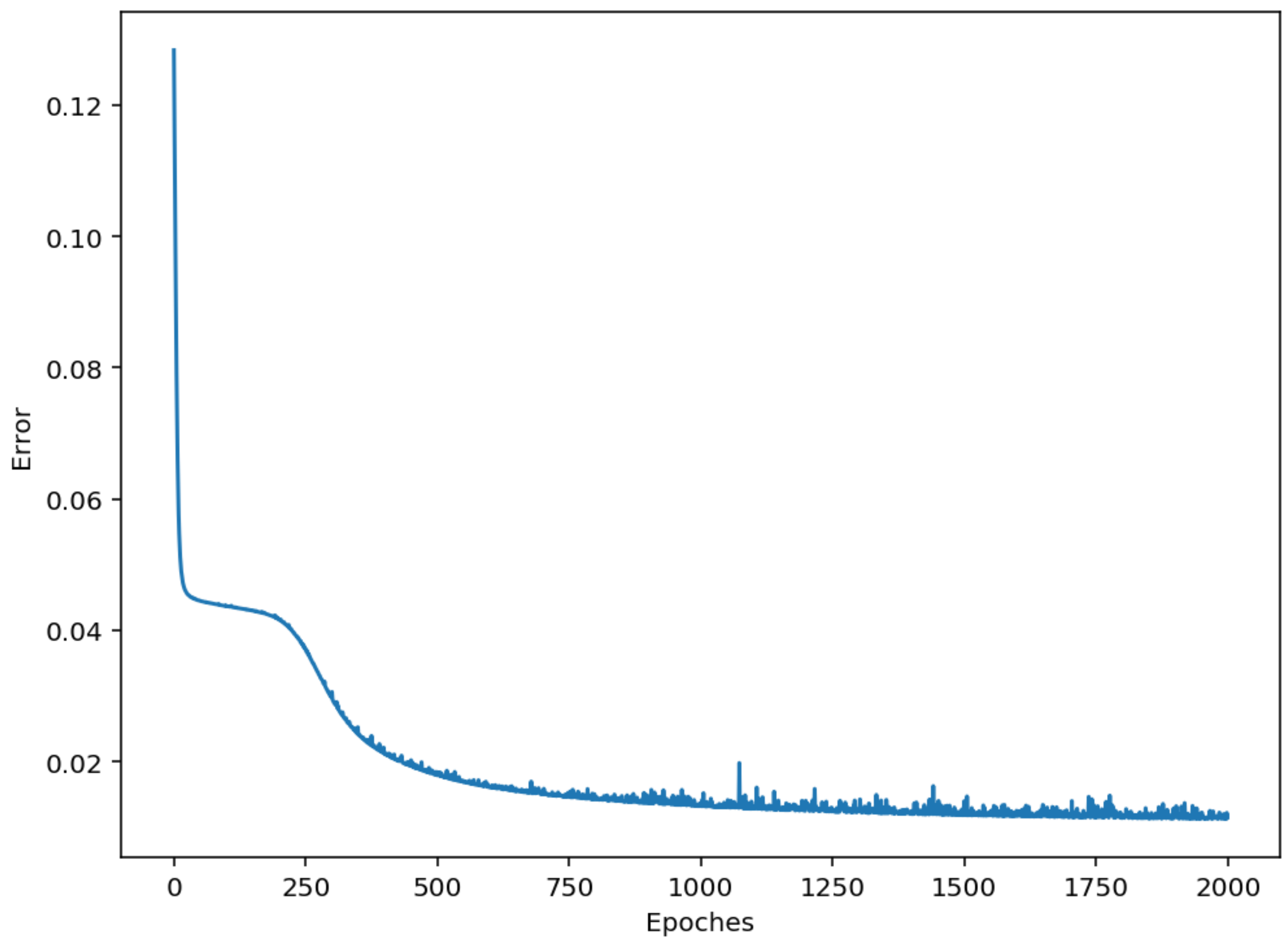

Figure 11: Plot of training error and epoches

In [15]:

```
# utility function to plot boundaries and scatter plot
def plot_scatter_boundary(X, y):
    x1_min, x1_max = X.T[0].min() - 0.5, X.T[0].max() + 0.5
    x2_min, x2_max = X.T[1].min() - 0.5, X.T[1].max() + 0.5
    x1, x2 = np.meshgrid(np.arange(x1_min, x1_max, 0.01), np.arange(x2_min, x2_m
ax, 0.01))
    z = nn.predict(np.array([x1.ravel(), x2.ravel()]).T).reshape(x1.shape)
    plt.contourf(x1, x2, z, cmap = 'tab10', zorder = 1)
    plt.contour(x1, x2, z, colors = "Black", linewidths = 0.5, zorder = 2)
    plt.scatter(X[y == 1].T[0], X[y == 1].T[1], c = 'paleturquoise', s = 20, alp
ha = 0.8, label = 'y = 1', zorder = 3)
    plt.scatter(X[y == 0].T[0], X[y == 0].T[1], c = 'blue', s = 20, alpha = 0.8,
label = 'y = 0', zorder = 4)
    plt.legend(loc="upper left")
    plt.xlabel("x1"); plt.ylabel("x2")

# plotting
fig = plt.figure(figsize=(16, 6))
plt.subplot(1,2,1)
plot_scatter_boundary(x_train, y_train)
plt.subplot(1,2,2)
plot_scatter_boundary(x_test, y_test)
fig.text(.5, .01, 'Figure 12: Decision Boundary and scatter plots for training d
ata (left) and testing data (right)', ha='center')
plt.show()
```

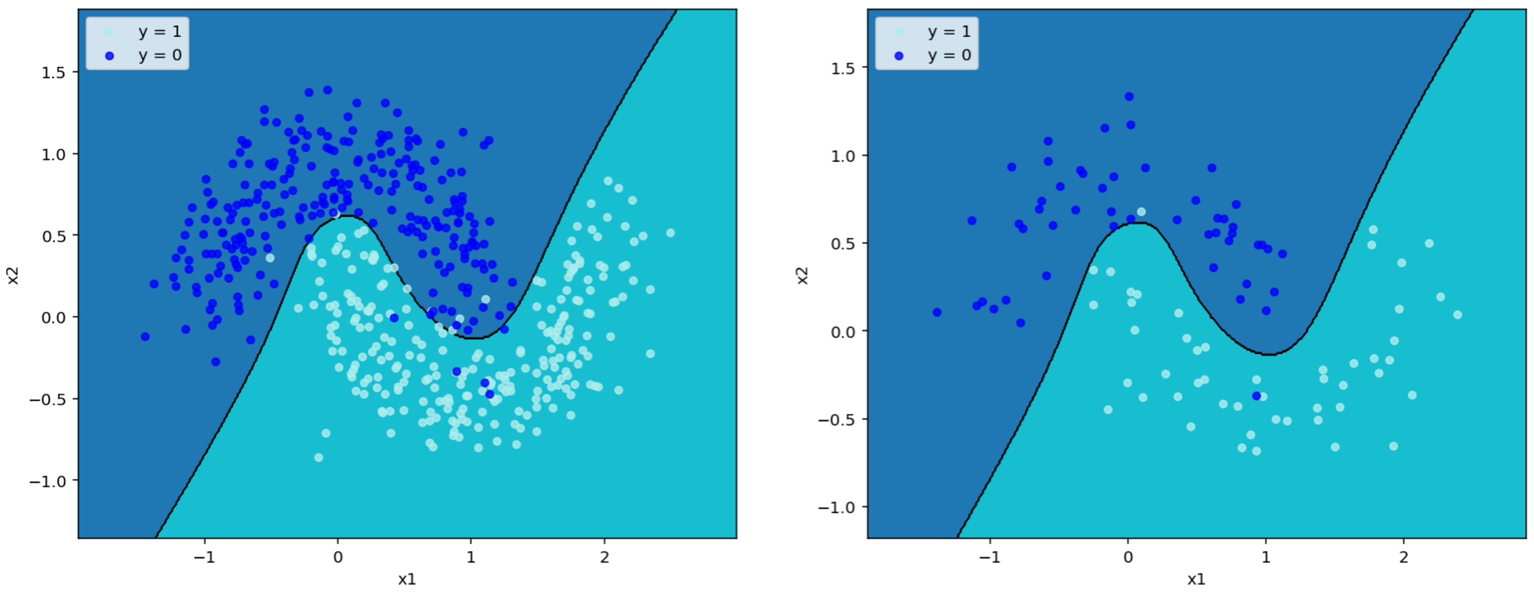

Figure 12: Decision Boundary and scatter plots for training data (left) and testing data (right)

In [16]:

```
from sklearn.metrics import roc_curve, auc, precision_recall_curve, f1_score
# utility function for plotting ROC curves
def plot_curves(label, preds, lab = ''):
    fpr, tpr, _ = roc_curve(label, preds)
    roc_auc = auc(fpr, tpr)
    precision, recall, _ = precision_recall_curve(label, preds)
    f1 = f1_score(label, preds>0.5)
    plt.subplot(1,2,1)
    plt.plot(fpr, tpr, color = 'darkorange', lw = 2, label = lab + 'ROC curve (a
rea = {0:0.3f})'.format(roc_auc))
    plt.plot([0, 1], [0, 1], color = 'navy', lw = 2, linestyle='--')
    plt.xlim([-0.01, 1.0]);plt.ylim([0.0, 1.01])
    plt.xlabel('False Positive Rate');plt.ylabel('True Positive Rate')
    plt.legend(loc="lower right")
    plt.subplot(1,2,2)
    plt.step(recall, precision, color='b', alpha=0.2, where='post', label = lab
+ 'Precision-Recall curve: F1={0:0.3f}'.format(f1))
    plt.fill_between(recall, precision, alpha=0.2, color='b', step = 'post')
    plt.xlabel('Recall')
    plt.ylabel('Precision')
    plt.ylim([0.0, 1.01])
    plt.xlim([0.0, 1.0])
    plt.legend(loc="lower left")

# plotting
fig = plt.figure(figsize=(13, 6))
plot_curves(y_test, nn.predict_proba(x_test))
fig.text(.5, .01, "Figure 13: ROC and PRC curves of neural network's prediction
on testing data", ha='center')
plt.show()
```

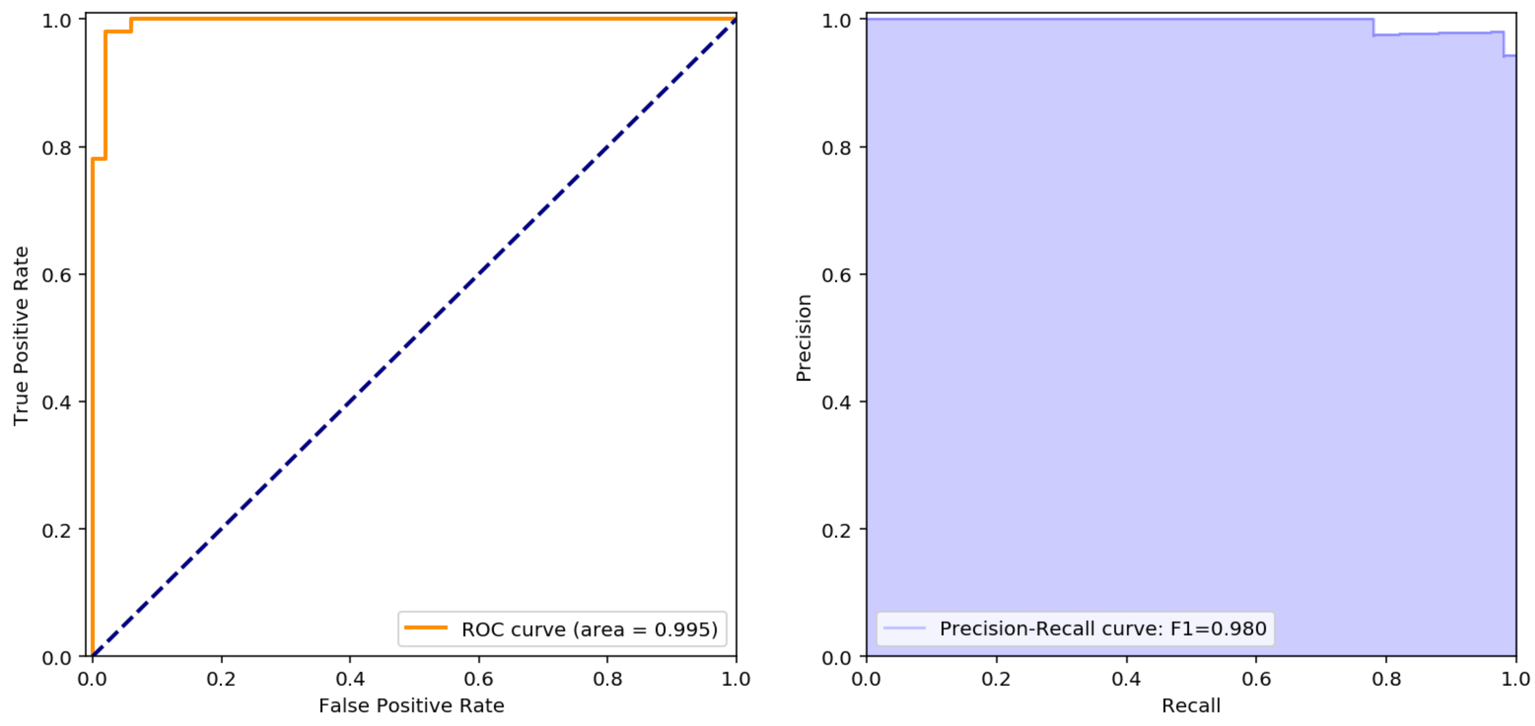

Figure 13: ROC and PRC curves of neural network's prediction on testing data

The plots above show the model fits the data pretty well, basically captures the trend of the training/testing data, as well as reaches an auc of around 0.990, F1 score of around 0.980 for testing data.

(c)

1. My model is specified on this case: it initialized its weights according to the structure of [2,5,5,1] as asked in the question. It could be tricky if we want to implement a different structured model according to needs. However I did write another more sophisticated model which enables user to choose model structure, just add an arguement of a list refering the model structure (e.g. structure = [2,5,5,1] means it has 2 input nodes, 2 layers of hidden layers with 5 nodes each, 1 output node). The code is attached below. Since this is actually finished before the code above, much optimization and changes can be done to make it more clear, but it functions well.

2. Similar to 1, my model is also specified in weight updating methods: SGD. Adding a batch option to model could enable users to choose batch/minibatch/stochastic gradient descent.

2. My model is built on numpy operations and python for loops, which is much slower than some functions in packages available. More optimizations could be done using C/Cpp to speed up the whole process.

In [17]:

```
class NN2:
    def __init__(self, structure = [2,5,5,1]):
        self.weights = []
        self.input_num = structure[0]
        for i, j in zip(structure[1:], structure[0:-1]):
            self.weights.append(np.random.random_sample(size = (i,j)))
        self.structure = structure
        pass

    def reg(self, x, w):
        return np.dot(w, x.T)

    def sig(self, a):
        return 1 / (1 + np.exp(-a))

    def sig_deriv(self, a):
        temp = self.sig(a) * (1 - self.sig(a))
        return temp.reshape(temp.shape[0],1)

    def fp(self, x):
        x = np.array(x)
        self.output, self.output_raw = [x], [x]
        for layer in range(len(self.structure) - 1):
            self.output_raw.append(self.reg(x, self.weights[layer]))
            x = self.sig(self.reg(x, self.weights[layer]))
            self.output.append(x)
        return self

    def bp(self, y):
        y = np.array(y)

        errors, self.delta = [], []
        weights = [np.sum(i, axis = 1) for i in self.weights]
        for idx, layer in enumerate(reversed(range(1,len(self.structure)))):
            errors.append([self.output[-1] - y] if idx == 0 else np.dot(self.weights[layer].T, delta))
            delta = np.array(errors[idx] * self.sig_deriv(self.output_raw[layer]))
            self.delta.append(delta)
        self.delta.reverse()
        self.deriv = [np.dot(j, np.array([i])) for i, j in zip(self.output_raw, self.delta)]
        return self

    def fit(self, X, y, lr = 0.02, epoch = 1000):
        self.error = []
        for _ in range(epoch):
            length = len(y)
            indexes = np.random.choice(np.arange(0, length), size = length, replace = False)
            for i in indexes:
                self.fp(X[i]).bp(y[i])
                self.weights = [(weight - lr * deriv) for weight, deriv in zip(self.weights, self.deriv)]
            prediction = []
            for row in X:
                prediction.append(self.fp(row).output[-1])
            self.error.append(0.5*sum(np.square(y - np.array(prediction).T[0]))/len(y))
        return self

    def predict_proba(self, x):
        prediction = []
        for row in x:
            prediction.append(self.fp(row).output[-1])
        return np.array(prediction).reshape(len(x),)

    def predict(self, x):
        return self.predict_proba(x) > 0.5
```