Part 1. Data Preprocessing

Python Code with comment:

```
# In[1]
import pandas as pd
import numpy as np
import warnings
from sklearn import preprocessing
from sklearn.utils import shuffle
from sklearn.exceptions import ConvergenceWarning
from pandas.core.common import SettingWithCopyWarning
import matplotlib.pyplot as plt
warnings.simplefilter(action="ignore", category=ConvergenceWarning)
warnings.simplefilter(action="ignore", category=SettingWithCopyWarning)
warnings.filterwarnings('ignore')

from sklearn.impute import SimpleImputer


#input dataset pima_indians_diabetes.csv
dataset1 = pd.read_csv('FoodNutrients.csv')
#print(dataset1)

#drop some useless feature columns
dataset1 = dataset1.drop(['Public Food Key', 'Food Name'], axis=1)
dataset1 = dataset1.drop([0])
#print(dataset1)

#shuffle data
dataset1 = shuffle(dataset1)
print(dataset1)

#split the dataset to feature data and labels
data = np.array(dataset1.iloc[0:,1:])
labels = np.array(dataset1.iloc[0:,0])
labels = labels//1000
#print(data)
print(labels)

#use mean value to replace Nah value
#print(data.mean())
imp_mean = SimpleImputer(missing_values=np.nan, strategy='mean')
imp_mean.fit(data)
data = imp_mean.transform(data)
print(data)
```

```
#split data into train(75%) and test(25%) sets
from sklearn.model_selection import train_test_split
train_data, test_data, train_labels, test_labels = train_test_split(dat
a, labels)


#scaler the data, so that some extrem values will not affact the trainn
ing too much
norm_train = preprocessing.StandardScaler().fit(train_data)
train_data = norm_train.transform(train_data)
#print(train_data)
#print(train_labels)
norm_test = preprocessing.StandardScaler().fit(test_data)
test_data = norm_test.transform(test_data)
#print(test_data)
```

Analysis:

After importing the dataset, some columns and row with low effect to feature are removed.
Deleted column: "Public Food Key" and "Food Name"
Reason: A classification column is enough to present the labels of food.
Deleted row: the unit row (0th row)
Reason: no effect to data

Due to large amount of blanked data, we cannot simply drop the Nah data. The mean values of the column are filled to the Nah value. The reason to choose mean is it will have less effect to the overall training model.

As the data is sorted default, a shuffle process is necessary before training and testing to make sure each train and test sample contains all the kinds of labels.

A data normalization process is applied to reduce data redundancy and improve data integrity.

The labels of the dataset are simplified by only using the first two digits. This will only divide the data to the food categories instead of specific food. If we use the specific food as the label, this will cause each label to contain only a couple of data, and this will easily cause the model overfit.

The dataset are then split to 75% of training and 25% of testing.

Part 2. SVM

Code:

```
# In[2]
#SVM
from sklearn import svm

def rbf_kernal(gamma):
    rbf_svc = svm.SVC(kernel='rbf', gamma=gamma)
    rbf_svc.fit(train_data, train_labels)
    train_predict = rbf_svc.predict(train_data)
    # print(train_predict)
    train_error = np.sum(train_predict != train_labels) / len(train_lab
els)
    # print(train_error)

    test_predict = rbf_svc.predict(test_data)
    test_error = np.sum(test_predict != test_labels) / len(test_labels)
    # print(test_error)
    if gamma != 'scale':
        print("gamma: %0.3f | train error: %0.3f | test error: %0.3f" %

              (gamma, train_error, test_error))
    else:
        print("gamma: " + gamma, "| train error: " + str(round(train_er
ror, 3)),
              "| test error: " + str(round(test_error, 3)))
    return test_error, train_error

def polynomial_kernal(gamma, degree):
    poly_svc = svm.SVC(kernel='poly', gamma=gamma, degree=degree)
    poly_svc.fit(train_data, train_labels)
    train_predict = poly_svc.predict(train_data)
    #print(train_predict)
    train_error = np.sum(train_predict != train_labels) / len(train_lab
els)
    #print(train_error)

    test_predict = poly_svc.predict(test_data)
    test_error = np.sum(test_predict != test_labels) / len(test_labels)
    #print(test_error)
    print("degree: %0.2f | train error: %0.3f | test error: %0.3f" %
              (degree, train_error, test_error))
    return test_error, train_error
```

```
def plot_train_vs_test_error(C_list, train_error, test_error, method, x
):
    plt.plot(C_list, train_error, "o", label="train error")
    plt.plot(C_list, test_error, "o", label="test error")
    plt.title(method)
    plt.xscale("log")
    plt.xlabel(x)
    plt.ylabel('error')
    plt.legend()
    plt.show()

#compare the effect of gamma
gamma_list = [0.001, 0.01, 0.1, 0.5, 1.0, 5.0]
rbf_train_performance = []
rbf_test_performance = []
for gamma in gamma_list:
    test_error, train_error = rbf_kernal(gamma)
    rbf_test_performance.append(test_error)
    rbf_train_performance.append(train_error)

plot_train_vs_test_error(gamma_list, rbf_train_performance, rbf_test_pe
rformance, "rbf performance", "gamma")

#investigate the effect of degree
degree_list = [0.01, 0.1, 1.0, 2.0, 5.0, 10.0, 20.0]
poly_train_performance = []
poly_test_performance = []
for degree in degree_list:
    test_error, train_error = polynomial_kernal("scale", degree)
    poly_test_performance.append(test_error)
    poly_train_performance.append(train_error)

plot_train_vs_test_error(degree_list, poly_train_performance, poly_test
_performance, "polynomial performance", "degree")


#apply C to the model
def get_error_C(C):
    rbf_svc = svm.SVC(C=C, kernel='rbf')
    rbf_svc.fit(train_data, train_labels)

    test_predict = rbf_svc.predict(test_data)
    test_error = np.sum(test_predict != test_labels) / len(test_labels)
    # print(test_error)
    return test_error


C_list = [0.01, 0.1, 1.0, 10.0, 100.0]
test_performance = []
for c in C_list:
    test_error = get_error_C(c)
    test_performance.append(test_error)
    print("C: %0.2f | test error: %0.3f" % (c, test_error))



#use 10-fold cross validation to retrain the dataset
from sklearn.model_selection import cross_val_score, KFold

def get_cv_error(X, y, C, fold):
    rbf_svm = svm.SVC(C=C, kernel='rbf')
    kfold = KFold(n_splits=fold, shuffle=False)
    scores = cross_val_score(rbf_svm, X, y, cv=kfold)
    return 1 - scores.mean()

C_list1 = [0.01, 0.1, 1.0, 10.0, 100.0]
CV_error = []
for c in C_list1:
    error = get_cv_error(data, labels, c, 10)
    #error = get_error_corss_validate(data, labels, c, 10)
    CV_error.append(error)
    print("C: %0.2f | CV error: %0.3f" % (c, error))



#plot the correctness of simple split and crocess validation
for index, i in enumerate(test_performance):
    test_performance[index] = (1 - i) * 100
C_V = []
for i in CV_error:
    C_V.append((1 - i) * 100)
plt.scatter(np.array([0.01, 0.1, 1, 10, 100]).astype(str), test_perform
ance, c='y');
plt.scatter(np.array([0.01, 0.1, 1, 10, 100]).astype(str), C_V, c='b');
plt.legend(("simple split","cross validation"), loc="lower right");
plt.xlabel("C");
plt.ylabel("Predict Accuracy (%)");
plt.title("Performance On The Test Set");
```

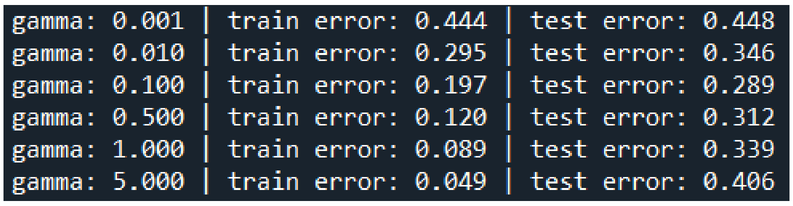
Output:

To do the SVM model, both RBF and polynomial kernel function are applied.

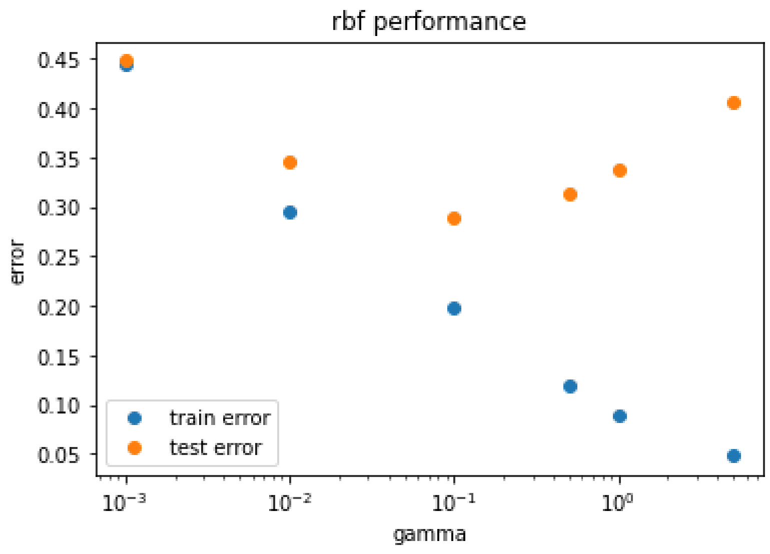
RBF and Polynomial:

A list of different gamma value is input to determine a proper value.

Train and Test error of rbf:



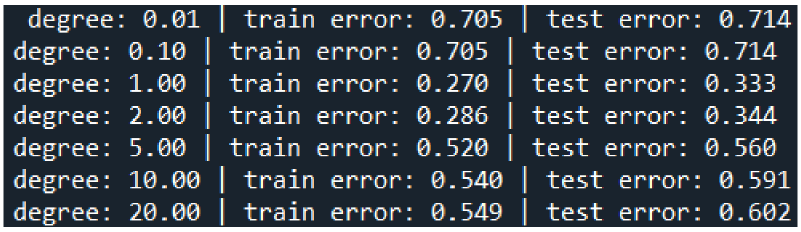
```
gamma: 0.001 | train error: 0.444 | test error: 0.448
gamma: 0.010 | train error: 0.295 | test error: 0.346
gamma: 0.100 | train error: 0.197 | test error: 0.289
gamma: 0.500 | train error: 0.120 | test error: 0.312
gamma: 1.000 | train error: 0.089 | test error: 0.339
gamma: 5.000 | train error: 0.049 | test error: 0.406
```



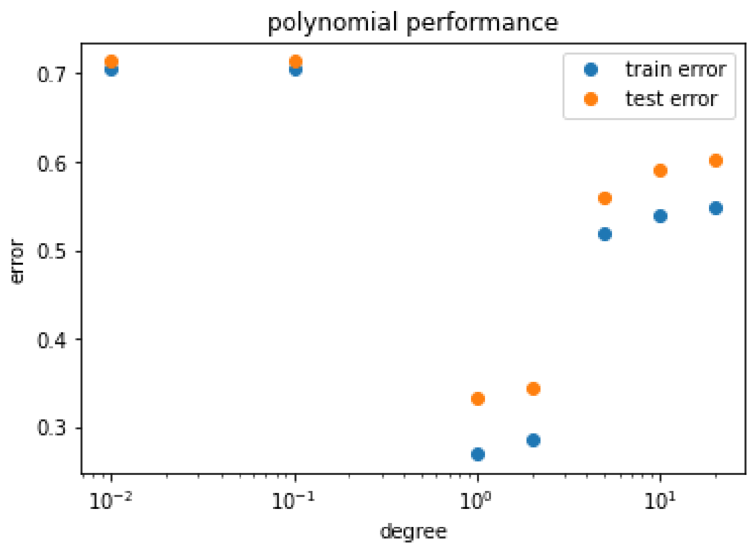
When gamma is larger than 0.1, the rbf model tended to be overfitted. The lowest test error is 28.9% with gamma equals to 0.1.

A list of different degree value is input to determine a proper value.

Train and test error of polynomial:



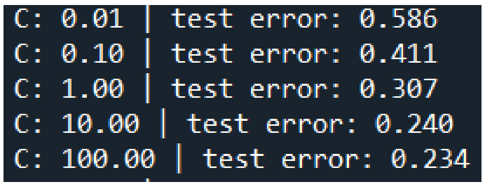
```
 degree: 0.01 | train error: 0.705 | test error: 0.714
degree: 0.10 | train error: 0.705 | test error: 0.714
degree: 1.00 | train error: 0.270 | test error: 0.333
degree: 2.00 | train error: 0.286 | test error: 0.344
degree: 5.00 | train error: 0.520 | test error: 0.560
degree: 10.00 | train error: 0.540 | test error: 0.591
degree: 20.00 | train error: 0.549 | test error: 0.602
```



When degree is larger than 1, the polynomial model tended to be overfitted. The lowest test error is 33.3% with degree equals to 1.

Through the output, it is simple to find that the rbf model has lower overall train and test error than polynomial model. This is because rbf model always has strong tolerance to input noise and it is easy to design. Polynomial kernel function has more parameters and it is very hard to set proper input parameters. Therefore, the RBF kernel is chosen to do the further training.

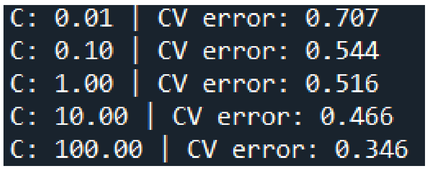
Test error after applying C:



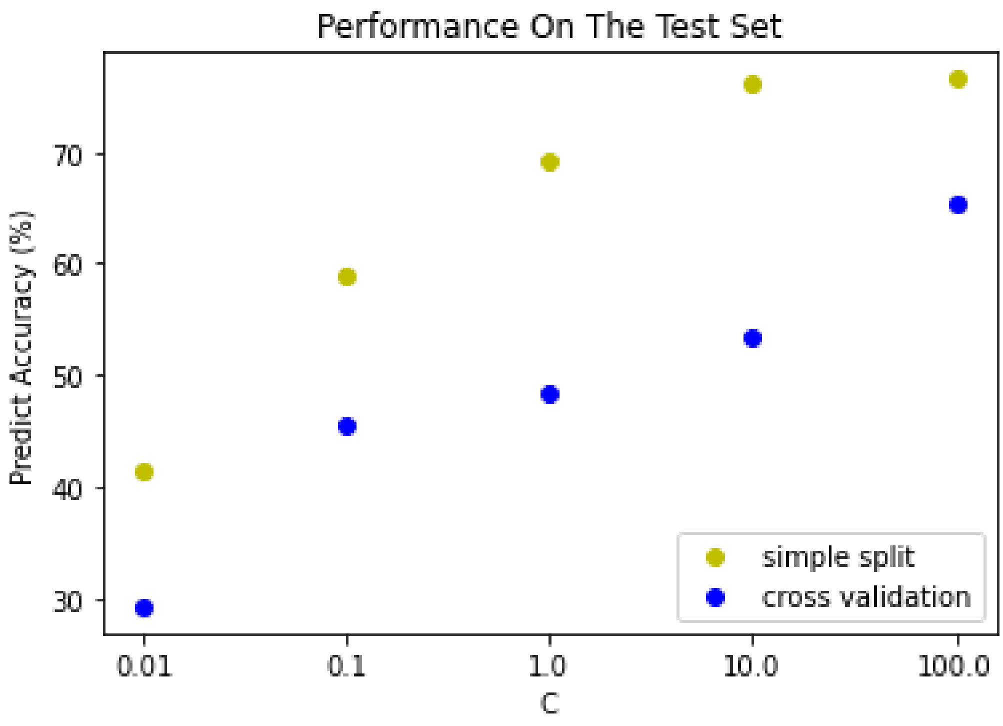
```
C: 0.01 | test error: 0.586
C: 0.10 | test error: 0.411
C: 1.00 | test error: 0.307
C: 10.00 | test error: 0.240
C: 100.00 | test error: 0.234
```

With the increasing of C, the test error decreased significantly. This is caused by the overfitting of the model. The proper value of C should be about 1.

Cross Validation:



```
C: 0.01 | CV error: 0.707
C: 0.10 | CV error: 0.544
C: 1.00 | CV error: 0.516
C: 10.00 | CV error: 0.466
C: 100.00 | CV error: 0.346
```



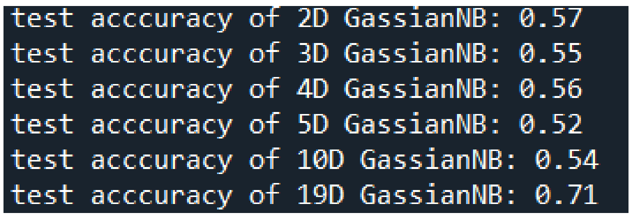
A 10-fold cross validation is applied to redo the training and test of the above rbf model. We can find the testing errors after applying CV increased.

Part 3. Gaussian Naïve Bayes

Code:

```
# In[4]

#gaussian naive bayes classification
from sklearn.naive_bayes import GaussianNB
def GaussianNB_(X, labels, dimension):

    z, eig_vals = PCA_kk(X, labels, dimension)

    #divide date to train(75%) and test(25%)
    x_train, x_test, train_label, test_label = train_test_split(z, labe
ls)
    #fit gaussianNB model
    clf = GaussianNB()
    classified = clf.fit(x_train, train_label)
    predict_labels = classified.predict(x_test)
    #calculate the correctness of test data
    score = classified.score(x_test, test_label)
    #plot test data
    species = np.unique(predict_labels)
    plt.figure(figsize = (10, 8))
    for i in species:
        plt.scatter(x_test[np.where(predict_labels == i), 0],
        x_test[np.where(predict_labels == i), 1],
        label = '%s'%(i))
    plt.legend()
    plt.xlabel('PC1')
    plt.ylabel('PC2')
    plt.title("test acccuracy of %dD GassianNB: %0.2f" %(dimension, sco
re))
    plt.show()
    print("test acccuracy of %dD GassianNB: %0.2f" %(dimension, score))

dimension_list = [2, 3, 4, 5, 10, feature_number]
for i in dimension_list:
    GaussianNB_(data, labels, i)
```

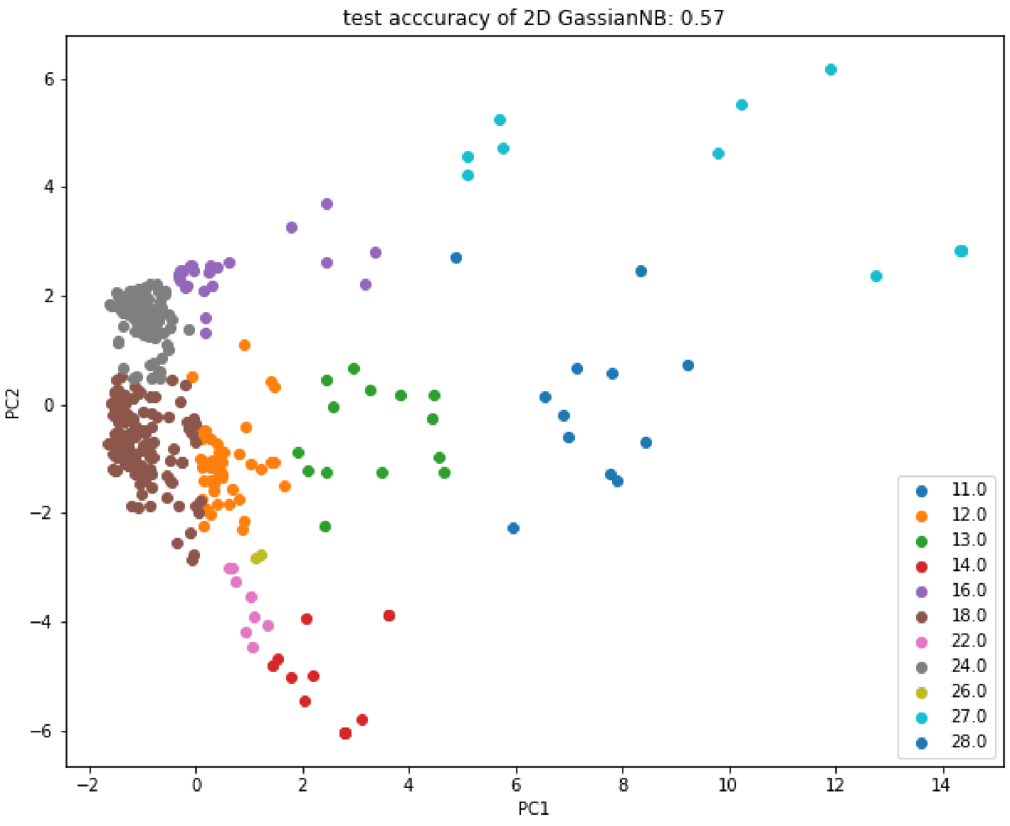
Output:



```
test acccuracy of 2D GassianNB: 0.57
test acccuracy of 3D GassianNB: 0.55
test acccuracy of 4D GassianNB: 0.56
test acccuracy of 5D GassianNB: 0.52
test acccuracy of 10D GassianNB: 0.54
test acccuracy of 19D GassianNB: 0.71
```

With the increasing of the dimension of input data, the test accuracy of the GassianNB model increases. This is because, more feature column will contain more data feature and it is closer to the original dataset.

Some more Plot:



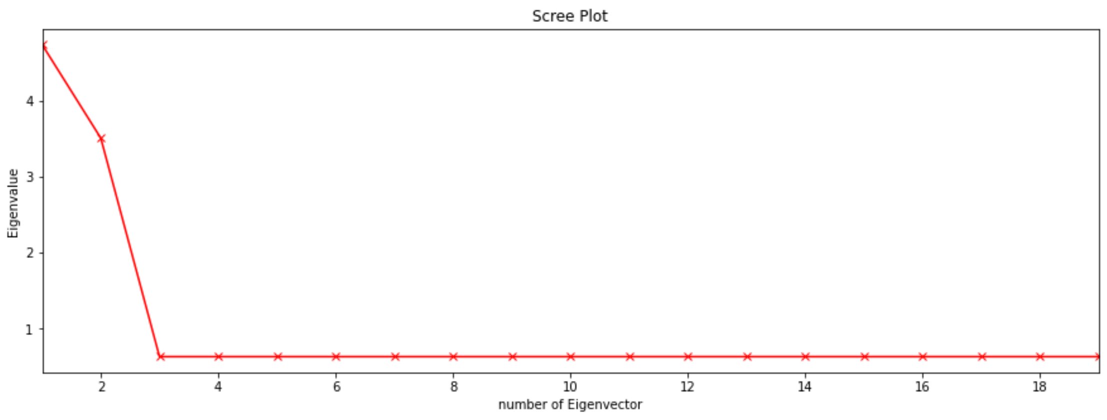
Part 4. PCA and t-SNE

Code:

```
# In[3]
from sklearn.decomposition import PCA
from sklearn.preprocessing import scale

#reduce the dimension of data
def PCA_kk(data, target, k):
    data = scale(data)
    pca = PCA(n_components=k)
    pca.fit(data)
    scores = pca.transform(data)
    cov_matrix = pca.get_covariance()
    eig_vals, eig_vecs = np.linalg.eig(cov_matrix)
    #print(eig_vals)
    #scores_df = pd.DataFrame(scores, columns=['PC1', 'PC2'])
    #print(scores_df)
    #label = pd.DataFrame(target, columns=['labels'])
    #df = pd.concat([scores_df, label], axis=1)
    #print(df)
    return scores, eig_vals


#plot PCA (2D)
def plot_PCA_2D():
    scores, eig = PCA_kk(data, labels, 2)
    species = np.unique(labels)
    plt.figure(figsize = (10, 8))
    for i in species:
        plt.scatter(scores[np.where(labels == i), 0],
        scores[np.where(labels == i), 1],
        label = '%s'%(i))
    plt.legend()
    plt.xlabel('PC1')
    plt.ylabel('PC2')
    plt.title('Plot of the data projected onto the first two PC')
    plt.show()

def scree_plot(eig_vals, feature_num):
    eig_vecs_number = np.linspace(1, feature_num, feature_num)
    plt.figure(figsize = (15,5))
    plt.plot(eig_vecs_number,eig_vals,'x-r')
    plt.title('Scree Plot')
    plt.xlabel('number of Eigenvector')
    plt.ylabel('Eigenvalue')
    plt.xlim((1,feature_num))
    plt.show()


z, eig_vals = PCA_kk(data, labels, 2)
feature_number = data.shape[1]
scree_plot(eig_vals, feature_number)
plot_PCA_2D()
# In[5]
from sklearn.manifold import TSNE

def T_SNE(data, label):
    data = scale(data)
    tsne = TSNE(n_components=2, perplexity=30, init='pca', random_state
=0)
    result = tsne.fit_transform(data)
    #print(result)
    species = np.unique(label)
    plt.figure(figsize = (10, 8))
    for i in species:
        plt.scatter(result[np.where(labels == i), 0],
        result[np.where(labels == i), 1],
        label = '%s'%(i))
    plt.legend()
    plt.xlabel('PC1')
    plt.ylabel('PC2')
    plt.title('Plot of the data projected onto the first two PC by appl
ying t-sne')
    plt.show()

T_SNE(data, labels)
```

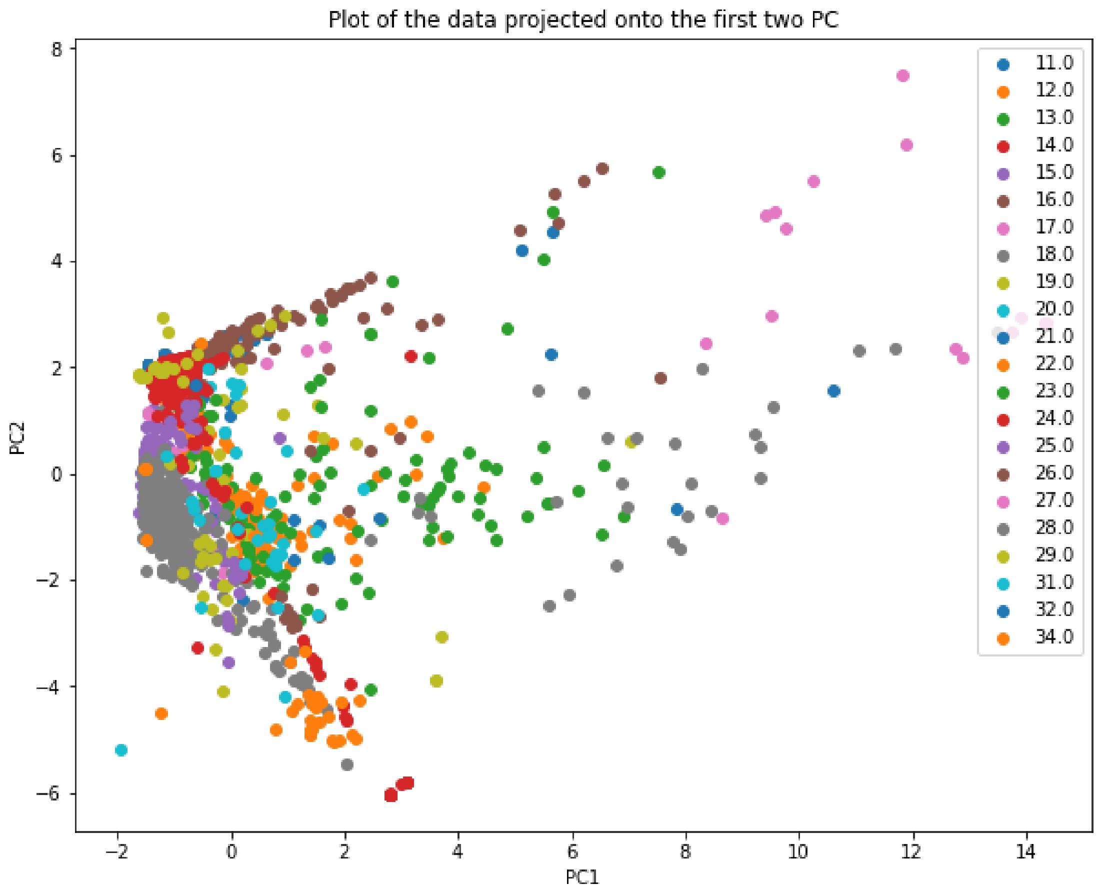
Output:

Scree Plot:

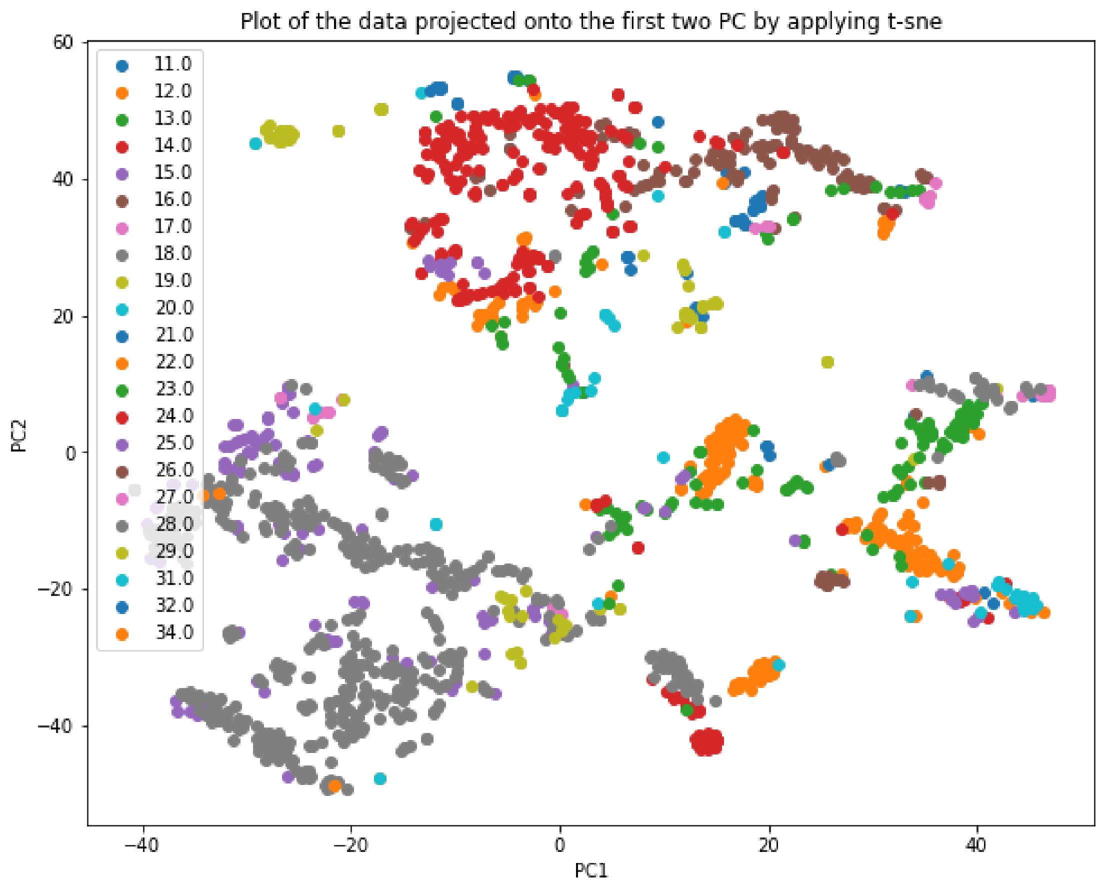


From the scree plot, we can find that the first two or three PC contain most of the feature of the dataset.

PCA plot on first 2 PC:



From the PCA plot, it is very hard to see the classification of each class after the reduction of dimension, this is because the data points are too crowded and the 2D model lose too many features. A better way to do the dimension reduction is t-SNE.



After using the t-SNE, it is much easily to divide each class in 2D.

Part 5. K means

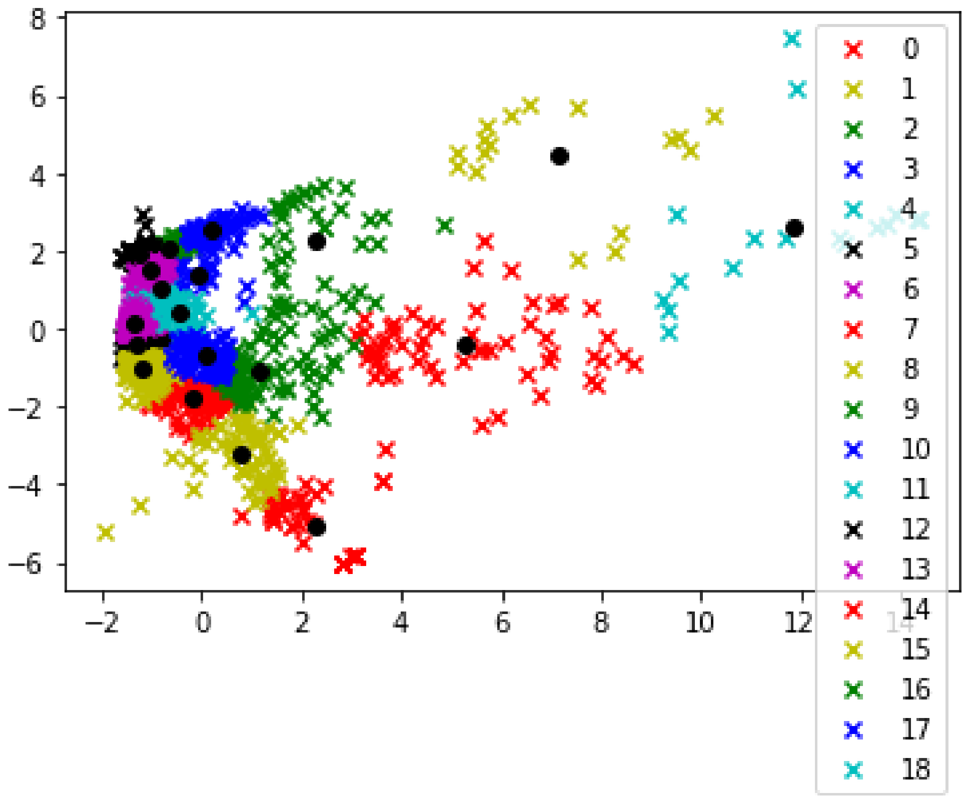
Code:

```
# In[6]

import pandas as pd
import numpy as np
import random
from matplotlib import pyplot as plt
from sklearn.cluster import KMeans
import math

#calculate the distance between center and each data point
def getEuclideanDistance(point1, point2):
    dimension = len(point1)
    dist = 0.0
    for i in range(dimension):
        dist += (point1[i] - point2[i]) ** 2
    return math.sqrt(dist)

def k_means(dataset, k, iteration):
    #set the initial center for each cluster
    index = random.sample(list(range(len(dataset))), k)
    vectors = []
    for i in index:
        vectors.append(dataset[i])
    #create a list to store the cluster for each data point, the initia
l value is -1
    labels = []
    for i in range(len(dataset)):
        labels.append(-1)
    #calculate k means for iteration
    while(iteration > 0):
        #set an empty list for each cluster
        C = []
        for i in range(k):
            C.append([])
        #calculate the distance between data and each center
        for labelIndex, item in enumerate(dataset):
            classIndex = -1
            minDist = 1e6
            for i, point in enumerate(vectors):
                dist = getEuclideanDistance(item, point)
                if(dist < minDist):
```

```python
                    classIndex = i
                    minDist = dist
            C[classIndex].append(item)
            labels[labelIndex] = classIndex

        # calculate new centers
        for i, cluster in enumerate(C):
            clusterCenter = []
            dimension = len(dataset[0])
            for j in range(dimension):
                clusterCenter.append(0)
            for item in cluster:
                for j, coordinate in enumerate(item):
                    clusterCenter[j] += coordinate / len(cluster)
            vectors[i] = clusterCenter
            #print(clusterCenter)

        iteration -= 1

    return C, labels, vectors

C, labels, vectors = k_means(z, feature_number, 20)

#plot clustered dataset
colValue = ['r', 'y', 'g', 'b', 'c', 'k', 'm']
for i in range(len(C)):
    X = []
    Y = []
    for j in range(len(C[i])):
        X.append(C[i][j][0])
        Y.append(C[i][j][1])
    plt.scatter(X, Y, marker='x', color=colValue[i%len(colValue)], labe
l=i)

#plot the latest cluster centers
for i in range(len(vectors)):
    c_X = []
    c_Y = []
    c_X.append(vectors[i][0])
    c_Y.append(vectors[i][1])
    plt.scatter(c_X, c_Y, marker='o', color='black')

plt.legend(loc='upper right')
plt.show()
```

```
#score of applying k means
def k_means_score(z, labels):
    kmeans = KMeans(n_clusters=20, random_state=0).fit(z)
    score = kmeans.score(z, labels)
    print("The trainning score of k means is: ", score)

k_means_score(z, labels)
```

Output:



Use k means method to cluster the dataset after the dimension is reduced to 2D. We can find the classes divided by the k means method is completely different with the original dataset.

Conclusion:

In the report, two classification method (SVM and GaussianNB) are applied to classify the data through training and testing, both simple split and cross validation are applied. Both two method has similar test error, which is about 30%. The PCA and t-SNE methods are applied to reduce the dimension of data to make the data visible in 2D. However, due to the limitation of PCA, the data points in 2D are too crowed and its very hard to observe the classes. Therefore, a t-SNE method is used to avoid the influence of “crowded”. Finally, k means method is applied to cluster the data and compare with the original classes, it shows that the classes divided by k mean method is almost wrong.