```python
# STAT4080 Data Programming with Python (online) - Project
# k nearest neighbours on the TunedIT data set

# Import packages
from scipy.spatial.distance import pdist, squareform
from pandas import Series, DataFrame
import pandas as pd
import numpy as np
import numpy.random as npr
from scipy.stats import pearsonr


# For the project we will study the method of k nearest neighbours applied to a
# music classification data set.These data come from the TunedIT website
# http://tunedit.org/challenge/music-retrieval/genres
# Each row corresponds to a different sample of music from a certain genre.
# The original challenge was to classify the different genres (the original
# prize for this was hard cash!). However we will just focus on a sample of the
# data (~4000 samples) which is either rock or not. There are 191
# characteristics (go back to the website if you want to read about these)
# The general tasks of this exercise are to:
# - Load the data set
# - Standardise all the columns
# - Divide the data set up into a training and test set
# - Write a function which runs k nearest neighbours (kNN) on the data set.
#   (Don't worry you don't need to know anything about kNN)
# - Check which value of k produces the smallest misclassification rate on the
#   training set
# - Predict on the test set and see how it does


# Q1 Load in the data using the pandas read_csv function. The last variable
# 'RockOrNot' determines whether the music genre for that sample is rock or not
# What percentage of the songs in this data set are rock songs (to 1 d.p.)?
song = pd.read_csv("./tunedit_genres.csv")
rock_ratio = song['RockOrNot'].mean()
# Ans:
print("About {:.4f} of songs are rock songs.".format(rock_ratio))


# Q2 To perform a classification algorithm, you need to define a classification
# variable and separate it from the other variables. We will use 'RockOrNot' as
# our classification variable. Write a piece of code to separate the data into a
# DataFrames X and a Series y, where X contains a standardised version of
# everything except for the classification variable ('RockOrNot'), and y contains
# only the classification variable. To standardise the variables in X, you need
# to subtract the mean and divide by the standard deviation
y = song.pop("RockOrNot").values
X = song.values
labels = song.columns
X_col = X.shape[1]
for i in range(X_col):
    mean_val = X[:, i].mean()
    std_val = X[:, i].std()
    X[:, i] = (X[:, i] - mean_val)/std_val


# Q3 Which variable in X has the largest correlation with y?
corr_list = list()
# Using Pearson correlation coefficient to determine which values of a column in X
# have the largest correlation with y
```

```python
for i in range(X_col):
    corr_list.append(pearsonr(X[:, i], y)[0])

# find which values of a column in X have the largest negative correlation or
# positive correlation
index_var = 0
if max(corr_list) > abs(min(corr_list)):
    index_var = np.argmax(corr_list)
else:
    index_var = np.argmin(corr_list)

# Ans:
print("The variable in X has the largest correlation with y is {:s}".format(
    labels[index_var]))


# Q4 When performing a classification problem, you fit the model to a portion of
# your data, and use the remaining data to determine how good the model fit was.
# Write a piece of code to divide X and y into training and test sets, use 75%
# of the data for training and keep 25% for testing. The data should be randomly
# selected, hence, you cannot simply take the first, say, 3000 rows. If you select
# rows 1,4,7,8,13,... of X for your training set, you must also select rows
# 1,4,7,8,13,... of y for training set. Additionally, the data in the training
# set cannot appear in the test set, and vice versa, so that when recombined,
# all data is accounted for. Use the seed 123 when generating random numbers
# Note: The data may not spilt equally into 75% and 25% portions. In this
# situation you should round to the nearest integer.

# Ans:
song = pd.read_csv("./tunedit_genres.csv")
# create a spliting function to split the original data set into 4 part(i.e. X_train, X_test, y_train, y_test)
# input: DataFrame, the size of training data set, random seed, the name of the classification label.
# output: X_train, X_test, y_train, y_test


def split_df(df, train_size, seed, target):
    np.random.seed(seed)
    # randomly shuffle the data set
    rand_index = np.random.permutation(df.index)
    rand_df = df.reindex(rand_index)
    # set the last training row of a data frame according to the train_size
    last_train_row = int(len(rand_df)*train_size)
    # get the training data set and testing data set
    train_df = rand_df.iloc[0:last_train_row]
    test_df = rand_df.iloc[last_train_row:]
    # spliting the data frame into X_train, X_test, y_train, y_test
    y_train = train_df.pop(target).values
    X_train = train_df.values
    y_test = test_df.pop(target).values
    X_test = test_df.values

    return X_train, X_test, y_train, y_test


X_train, X_test, y_train, y_test = split_df(song, 0.75, 123, "RockOrNot")

# Q5 What is the percentage of rock songs in the training dataset and in the
# test dataset? Are they the same as the value found in Q1?

# Ans:
print("About {:.4f} of the songs in training dataset are rock song.".format(
    y_train.mean()))
print("About {:.4f} of the songs in testing dataset are rock song.".format(
    y_test.mean()))
# They are not as same as the value found in Q1.


# Q6 Now we're going to write a function to run kNN on the data sets. kNN works
# by the following algorithm:
# 1) Choose a value of k (usually odd)
# 2) For each observation, find its k closest neighbours
# 3) Take the majority vote (mean) of these neighbours
# 4) Classify observation based on majority vote

# We're going to use standard Euclidean distance to find the distance between
# observations, defined as sqrt( (xi - xj)^T (xi-xj) )
# A useful short cut for this is the scipy functions pdist and squareform

# The function inputs are:
# - DataFrame X of explanatory variables
# - binary Series y of classification values
# - value of k (you can assume this is always an odd number)

# The function should produce:
# - Series y_star of predicted classification values


def kNN(X, y, k):
    # Find the number of obsvervation
    n = X.shape[0]
    # Set up return values
    y_star = list()
    # Calculate the distance matrix for the observations in X
    dist = squareform(pdist(X, 'euclidean'))
    # Loop through each observation to create predictions
    for i in range(n):
        d = sorted(list(dist[i]))
        sum = 0
        # Find the y values of the k nearest neighbours
        for j in range(1, k+1):
            y_index = list(dist[i]).index(d[j])
            sum += y[y_index]
        # Now allocate to y_star
        if (sum/k) > 0.5:
            y_star.append(1)
        else:
            y_star.append(0)
    return y_star


# Q7 The misclassification rate is the percentage of times the output of a
# classifier doesn't match the classification value. Calculate the
# misclassification rate of the kNN classifier for X_train and y_train, with k=3.
test1 = kNN(X_train, y_train, 3)
mis_count = 0
for i in range(len(test1)):
    if (test1[i] != y_train[i]):
        mis_count += 1
# Ans:
print("The misclassification rate is about {:.4f} ".format(
    mis_count/len(test1)))

# Q8 The best choice for k depends on the data. Write a function kNN_select that
# will run a kNN classification for a range of k values, and compute the
# misclassification rate for each.

# The function inputs are:
# - DataFrame X of explanatory variables
# - binary Series y of classification values
# - a list of k values k_vals

# The function should produce:
# - a Series mis_class_rates, indexed by k, with the misclassification rates for
# each k value in k_vals


def kNN_select(X, y, k_vals):
    mis_class_rates = list()
    for i in range(len(k_vals)):
        test1 = kNN(X, y, k_vals[i])
        mis_count = 0
        for j in range(len(test1)):
            if (test1[j] != y[j]):
                mis_count += 1
        rate = mis_count/len(test1)
        mis_class_rates.append(rate)
    mis_class_rates = pd.Series(mis_class_rates, index=k_vals)
    return mis_class_rates


# Q9 Run the function kNN_select on the training data for k = [1, 3, 5, 7, 9]
# and find the value of k with the best misclassification rate. Use the best
# value of k to report the mis-classification rate for the test data. What is
# the misclassification percentage with this k on the test set?
k = [1, 3, 5, 7, 9]
temp = kNN_select(X_train, y_train, k)
test2 = kNN(X_test, y_test, temp.idxmin())
mis_count = 0
for i in range(len(test2)):
    if (test2[i] != y_test[i]):
        mis_count += 1
# Ans:
print("The best k value for test data is {0:d} with about {1:.4f} misclassification rate.".format(
    temp.idxmin(),  mis_count/len(test2)))

# Q10 Write a function to generalise the k nearest neighbours classification
# algorithm. The function should:
# - Separate out the classification variable for the other variables in the dataset,
#   i.e. create X and y.
# - Divide X and y into training and test set, where the number in each is
#   specified by 'percent_train'.
# - Run the k nearest neighbours classification on the training data, for a set
#   of k values, computing the mis-classification rate for each k
# - Find the k that gives the lowest mis-classification rate for the training data,
#   and hence, the classification with the best fit to the data.
# - Use the best k value to run the k nearest neighbours classification on the test
#   data, and calculate the mis-classification rate
# The function should return the mis-classification rate for a k nearest neighbours
# classification on the test data, using the best k value for the training data
# You can call the functions from Q6 and Q8 inside this function, provided they
# generalise, i.e. will work for any dataset, not just the TunedIT dataset.


def kNN_classification(df, class_column, seed, percent_train, k_vals):
    # df            - DataFrame to
    # class_column  - column of df to be used as classification variable, should
    #                 specified as a string
    # seed          - seed value for creating the training/test sets
    # percent_train - percentage of data to be used as training data
    # k_vals        - set of k values to be tests for best classification
    # Divide into training and test
    X_train, X_test, y_train, y_test = split_df(
        df, percent_train, seed, class_column)
    # Compute the mis-classification rates for each for the values in k_vals
    temp = kNN_select(X_train, y_train, k_vals)
    # Find the best k value, by finding the minimum entry of mis_class_rates
    k = temp.idxmin()
    # Run the classification on the test set to see how well the 'best fit'
    # classifier does on new data generated from the same source
    y_star = kNN(X_test, y_test, k)
    # Calculate the mis-classification rates for the test data
    mis_count = 0
    for i in range(len(y_star)):
        if (y_star[i] != y_test[i]):
            mis_count += 1
    mis_class_test = mis_count/len(y_star)
    return mis_class_test


# Test your function with the TunedIT data set, with class_column = 'RockOrNot',
# seed = the value from Q4, percent_train = 0.75, and k_vals = set of k values
# from Q8, and confirm that it gives the same answer as Q9.

k = [1, 3, 5, 7, 9]
song = pd.read_csv("./tunedit_genres.csv")
kNN_classification(song, "RockOrNot", 123, 0.75, k)
# The mis-classification rate for test set is 0.395, which is as same as Q9's.

# Now test your function with another dataset, to ensure that your code
# generalises. You can use the house_votes.csv dataset, with 'Party' as the
# classifier. Select the other parameters as you wish.
# This dataset contains the voting records of 435 congressman and women in the
# US House of Representatives. The parties are specified as 1 for democrat and 0
# for republican, and the votes are labelled as 1 for yes, -1 for no and 0 for
# abstained.
# Your kNN classifier should return a mis-classification for the test data (with
# the best fit k value) of ~8%.
df = pd.read_csv("./house_votes.csv")
mis_rate = []
seed = 123
for i in range(10):
    rate = kNN_classification(df, "Party", seed, 0.75, k)
    seed -= 1
    mis_rate.append(rate)
pd.Series(mis_rate).mean()
# Because the misclassification rate is significantly influenced by the seed value and the way to split the data set,
# Then try 10 times with different seed values to get an average misclassification, which is about 8%

```