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

# Import packages
from pandas import Series, DataFrame
import pandas as pd
import numpy as np
import numpy.random as npr

# 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.)?
tg=pd.read_csv('tunedit_genres.csv')
rockp=tg.RockOrNot.value_counts()[1]/tg.shape[0]
# Ans: 48.8%

# 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
tg_std=(tg-tg.mean())/tg.std()
X=tg_std.drop('RockOrNot',axis=1)
y=tg.RockOrNot

# Q3 Which variable in X has the largest correlation with y?
maxcor=X.corrwith(y).max
maxindex=X.corrwith(y).idxmax
mincor=X.corrwith(y).min
minindex=X.corrwith(y).idxmin
# Ans:'PAR_SFM_M'


# 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.
train_size = 2999
np.random.seed(123)
train_select = np.random.permutation(range(len(y)))
X_train = X.iloc[train_select[:train_size],:].reset_index(drop=True)
X_test = X.iloc[train_select[train_size:],:].reset_index(drop=True)
y_train = y[train_select[:train_size]].reset_index(drop=True)
y_test = y[train_select[train_size:]].reset_index(drop=True)
# Ans: I use 2999 as the training set size and 1000 for testing.


# 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?
rockp_y_train=y_train.value_counts()[1]/len(y)
rockp_y_test=y_test.value_counts()[1]/len(y)
# Ans: the percentage of rock songs in the training dataset is 37%
#the percentage of rock songs in the training dataset is 11.8%
#37%+11.8%=48.8% so when add the two percentages, it equals to the percentage 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

from scipy.spatial.distance import pdist, squareform

from sklearn.metrics.pairwise import euclidean_distances

def kNN(X,y,k):
    # Find the number of obsvervation
    n = len(y)
    # Set up return values
    y_star = np.zeros(n)
    # Calculate the distance matrix for the observations in X
    Xa=np.array(X)
    dist = euclidean_distances(Xa, Xa)
    # Make all the diagonals very large so it can't choose itself as a closest neighbour
    for i in range(n):
        dist[i][i]=1000
    # Loop through each observation to create predictions
    for i in range(n):
        # Find the y values of the k nearest neighbours
        y_nearest = y.iloc[np.argsort(dist[i])[:k]]
        # Now allocate to y_star
        y_star[i] = y_nearest.value_counts().index[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.
from sklearn.metrics import accuracy_score
y_star=kNN(X_train,y_train,3)
misrate=1-accuracy_score(y_train, y_star)
# Ans: 4.7%

# 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):
    misrate=[]
    for ki in k_vals:
        misrate.append(1-accuracy_score(y,kNN(X,y,ki)))
    mis_class_rates=Series(misrate,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]
mis_class_rates=kNN_select(X_train,y_train,k)
mis_class_test=kNN_select(X_test,y_test,k)
# Ans: The best value of k is k=1, the misclassification percentage on the
#      test data is 5%

# 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

    # Separate X and y
    df_std=(df-df.mean())/df.std()
    X=df_std.drop(class_column,axis=1)
    y=df[class_column]
    # Divide into training and test
    train_size = int(len(y)*percent_train)
    np.random.seed(seed)
    train_select = np.random.permutation(range(len(y)))
    X_train = X.iloc[train_select[:train_size],:].reset_index(drop=True)
    X_test = X.iloc[train_select[train_size:],:].reset_index(drop=True)
    y_train = y[train_select[:train_size]].reset_index(drop=True)
    y_test = y[train_select[train_size:]].reset_index(drop=True)
    # Compute the mis-classification rates for each for the values in k_vals
    mis_class_rates=kNN_select(X_train,y_train,k_vals)
    # Find the best k value, by finding the minimum entry of mis_class_rates
    k=mis_class_rates.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_class_test=1-accuracy_score(y_test, 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.
mis_class_test=kNN_classification(tg,'RockOrNot',123,0.75,[1, 3, 5, 7, 9])
# 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%.
house=pd.read_csv('house_votes.csv')
kNN_classification(house,'Party',123,0.75,[1,2,3,4,5,6,7,8])
```