```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 math
import numpy.random as npr
from scipy.spatial.distance import pdist, squareform

# 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
```

```
29
30
31 # Q1 Load in the data using the pandas read_csv function. The last variable
32 # 'RockOrNot' determines whether the music genre for that sample is rock or not
33 # What percentage of the songs in this data set are rock songs (to 1 d.p.)?
34
35 data = pd.read_csv('tunedit_genres.csv')
36 percentage = sum(data.iloc[:, -1]) / len(data)
37 print(percentage)
38
39 # Ans: 0.4883720930232558
40
41
42 # Q2 To perform a classification algorithm, you need to define a classification
43 # variable and separate it from the other variables. We will use 'RockOrNot' as
44 # our classification variable. Write a piece of code to separate the data into a
45 # DataFrames X and a Series y, where X contains a standardised version of
46 # everything except for the classification variable ('RockOrNot'), and y contains
47 # only the classification variable. To standardise the variables in X, you need
48 # to subtract the mean and divide by the standard deviation
49
50 X = data.iloc[:, :-1]
51 X = (X - X.mean()) / X.std()
52 Y = data.iloc[:, -1]
53
54 # Q3 Which variable in X has the largest correlation with y?
55 corr = data.corr()
56 for i in range(len(corr)):
```

```
57      corr.iloc[i, i] = -math.inf
58  print(corr.iloc[:, -1].idxmax())
59  # Ans: PAR_SFM_M
60
61
62  # Q4 When performing a classification problem, you fit the model to a portion of
63  # your data, and use the remaining data to determine how good the model fit was.
64  # Write a piece of code to divide X and y into training and test sets, use 75%
65  # of the data for training and keep 25% for testing. The data should be randomly
66  # selected, hence, you cannot simply take the first, say, 3000 rows. If you select
67  # rows 1,4,7,8,13,... of X for your training set, you must also select rows
68  # 1,4,7,8,13,... of y for training set. Additionally, the data in the training
69  # set cannot appear in the test set, and vice versa, so that when recombined,
70  # all data is accounted for. Use the seed 123 when generating random numbers
71  # Note: The data may not spilt equally into 75% and 25% portions. In this
72  # situation you should round to the nearest integer.
73
74  # Ans:
75  c = np.random.RandomState(123).permutation(len(X))
76  X = X.iloc[c]
77  X = X.reset_index(drop=True)
78  Y = Y.iloc[c]
79  Y = Y.reset_index(drop=True)
80  num_train = math.ceil(len(X) * 0.75)
81  X_train = X[:num_train]
82  X_test = X[num_train:]
83  Y_train = Y[:num_train]
84  Y_test = Y[num_train:]
```

```
 85 X_train = X_train.reset_index(drop=True)
 86 X_test = X_test.reset_index(drop=True)
 87 Y_train = Y_train.reset_index(drop=True)
 88 Y_test = Y_test.reset_index(drop=True)
 89
 90 # Q5 What is the percentage of rock songs in the training dataset and in the
 91 # test dataset? Are they the same as the value found in Q1?
 92
 93 # Ans:percentage_train= 0.494, percentage_test= 0.47147147147147145.
 94 #     The percentage of rock songs in training data and test data are not exactly
     the same, but very close.
 95
 96 percentage_train = sum(Y_train) / len(Y_train)
 97 percentage_test = sum(Y_test) / len(Y_test)
 98 print('percentage_train=', percentage_train)
 99 print('percentage_test=', percentage_test)
100
101
102 # Q6 Now we're going to write a function to run kNN on the data sets. kNN works
103 # by the following algorithm:
104 # 1) Choose a value of k (usually odd)
105 # 2) For each observation, find its k closest neighbours
106 # 3) Take the majority vote (mean) of these neighbours
107 # 4) Classify observation based on majority vote
108
109 # We're going to use standard Euclidean distance to find the distance between
110 # observations, defined as sqrt( (xi - xj)^T (xi-xj) )
111 # A useful short cut for this is the scipy functions pdist and squareform
```

```
112
113 # The function inputs are:
114 # - DataFrame X of explanatory variables
115 # - binary Series y of classification values
116 # - value of k (you can assume this is always an odd number)
117
118 # The function should produce:
119 # - Series y_star of predicted classification values
120
121
122 def kNN(X, y, k):
123     # Find the number of obsvervation
124     n = len(X)
125     # Set up return values
126     y_star = []
127     # Calculate the distance matrix for the observations in X
128     dist = squareform(pdist(X))
129     # Make all the diagonals very large so it can't choose itself as a closest
    neighbour
130
131     # Loop through each observation to create predictions
132     for i in range(n):
133         dist[i, i] = math.inf
134         neighbours = y[dist[i].argsort()[:k]]
135         # Find the y values of the k nearest neighbours
136         y_nearest = 1 if sum(neighbours) > k / 2 else 0
137         # y_star = y_star.append(pd.Series(y_nearest, index=[i]))
138         y_star.append(y_nearest)
```

```
139     # Now allocate to y_star
140     y_star = pd.Series(y_star)
141
142     return y_star
143
144
145 # Q7 The misclassification rate is the percentage of times the output of a
146 # classifier doesn't match the classification value. Calculate the
147 # misclassification rate of the kNN classifier for X_train and y_train, with k=3.
148
149 percentage_misclassification_train = sum(abs(kNN(X_train, Y_train, 3) ^ Y_train
    )) / len(X_train)
150 print('percentage_misclassification_train=', percentage_misclassification_train)
151
152
153 # Ans: 0.047
154
155 # Q8 The best choice for k depends on the data. Write a function kNN_select that
156 # will run a kNN classification for a range of k values, and compute the
157 # misclassification rate for each.
158
159 # The function inputs are:
160 # - DataFrame X of explanatory variables
161 # - binary Series y of classification values
162 # - a list of k values k_vals
163
164 # The function should produce:
165 # - a Series mis_class_rates, indexed by k, with the misclassification rates for
```

```
166 # each k value in k_vals
167
168 def kNN_select(X, y, k_vals):
169     mis_class_rates = pd.Series()
170     for k in k_vals:
171         percentage_misclassification = sum(abs(kNN(X, y, k) ^ y)) / len(X)
172         mis_class_rates = mis_class_rates.append(pd.Series(
    percentage_misclassification, index=[k]))
173     return mis_class_rates
174
175
176 # Q9 Run the function kNN_select on the training data for k = [1, 3, 5, 7, 9]
177 # and find the value of k with the best misclassification rate. Use the best
178 # value of k to report the mis-classification rate for the test data. What is
179 # the misclassification percentage with this k on the test set?
180 k = kNN_select(X_train, Y_train, k_vals=[1, 3, 5, 7, 9]).idxmin()
181 percentage_misclassification_test = sum(abs(kNN(X_test, Y_test, k) ^ Y_test)) /
    len(X_test)
182 print('percentage_misclassification_test=', percentage_misclassification_test)
183
184
185 # Ans: 0.05005
186
187 # Q10 Write a function to generalise the k nearest neighbours classification
188 # algorithm. The function should:
189 # - Separate out the classification variable for the other variables in the
    dataset,
190 #   i.e. create X and y.
```

```
191 # - Divide X and y into training and test set, where the number in each is
192 #   specified by 'percent_train'.
193 # - Run the k nearest neighbours classification on the training data, for a set
194 #   of k values, computing the mis-classification rate for each k
195 # - Find the k that gives the lowest mis-classification rate for the training
    data,
196 #   and hence, the classification with the best fit to the data.
197 # - Use the best k value to run the k nearest neighbours classification on the
    test
198 #   data, and calculate the mis-classification rate
199 # The function should return the mis-classification rate for a k nearest
    neighbours
200 # classification on the test data, using the best k value for the training data
201 # You can call the functions from Q6 and Q8 inside this function, provided they
202 # generalise, i.e. will work for any dataset, not just the TunedIT dataset.
203 def kNN_classification(df, class_column, seed, percent_train, k_vals):
204     # df            - DataFrame to
205     # class_column  - column of df to be used as classification variable, should
206     #                 specified as a string
207     # seed          - seed value for creating the training/test sets
208     # percent_train - percentage of data to be used as training data
209     # k_vals        - set of k values to be tests for best classification
210
211     # Separate X and y
212     Y = df[class_column]
213     X = df.drop(class_column, axis=1)
214     X = (X - X.mean()) / X.std()
215     # Divide into training and test
```

```
216     c = np.random.RandomState(seed).permutation(len(df))
217     X = X.iloc[c]
218     X = X.reset_index(drop=True)
219     Y = Y.iloc[c]
220     Y = Y.reset_index(drop=True)
221     num_train = math.ceil(len(X) * percent_train)
222     X_train = X[:num_train]
223     X_test = X[num_train:]
224     Y_train = Y[:num_train]
225     Y_test = Y[num_train:]
226     X_train = X_train.reset_index(drop=True)
227     X_test = X_test.reset_index(drop=True)
228     Y_train = Y_train.reset_index(drop=True)
229     Y_test = Y_test.reset_index(drop=True)
230
231     # Compute the mis-classification rates for each for the values in k_vals
232     misclassification_rates = kNN_select(X_train, Y_train, k_vals)
233
234     # Find the best k value, by finding the minimum entry of mis_class_rates
235     k_best = misclassification_rates.idxmin()
236     # Run the classification on the test set to see how well the 'best fit'
237     # classifier does on new data generated from the same source
238     # Calculate the mis-classification rates for the test data
239     mis_class_test = sum(abs(kNN(X_test, Y_test, k_best) ^ Y_test)) / len(X_test)
240
241     return mis_class_test
242
243
```

```
244 # Test your function with the TunedIT data set, with class_column = 'RockOrNot',
245 # seed = the value from Q4, percent_train = 0.75, and k_vals = set of k values
246 # from Q8, and confirm that it gives the same answer as Q9.
247 print(kNN_classification(data, 'RockOrNot', 123, 0.75, [1, 3, 5, 7, 9]))
248
249 # Now test your function with another dataset, to ensure that your code
250 # generalises. You can use the house_votes.csv dataset, with 'Party' as the
251 # classifier. Select the other parameters as you wish.
252 # This dataset contains the voting records of 435 congressman and women in the
253 # US House of Representatives. The parties are specified as 1 for democrat and 0
254 # for republican, and the votes are labelled as 1 for yes, -1 for no and 0 for
255 # abstained.
256 # Your kNN classifier should return a mis-classification for the test data (with
257 # the best fit k value) of ~8%.
258
```