Practical 6

Text Analytics: Classification

Otto Hermann
otto.hermann@ucdconnect.ie
16203034

COMP47600

University College Dublin
24th October 2017

Q1: kNN

systematically vary the size of the split

I modified *main*() to include inputs for *split* and *k*.

```
def main_modified(split, k): # add split, k variables
# prepare data
trainingSet=[]
testSet=[]
loadDataset('iris.csv', split, trainingSet, testSet)
while len(trainingSet) < k:
  loadDataset('iris.csv', split, trainingSet, testSet) # added to ensure a set is
       created; see comments below
# generate predictions
predictions=[]
if len(trainingSet) < k:
  print("len(trainingSet) < k when split = {} and k = {}".format(split, k)) # check
       and notification of size
else:
  for x in range(len(testSet)):
    neighbors = getNeighbors(trainingSet, testSet[x], k)
    result = getResponse(neighbors)
    predictions.append(result)
  accuracy = getAccuracy(testSet, predictions)
  # return
  return [split, k, accuracy] # return information relevant to the assignment
```

I also included a while loop to prevent errors: without the addition of the while loop, it is possible that *traingSet*, the contents of which are materially determined by a pseudo-random value compared to *split* in *loadDataset*, could have fewer than *k* elements. A quick fix (but subject to potential issues) is the while loop I put in: it works for this exercise, but might not be appropriate for a different data set or if other circumstances changed.

plot the accuracy in a graph for these parameter changes

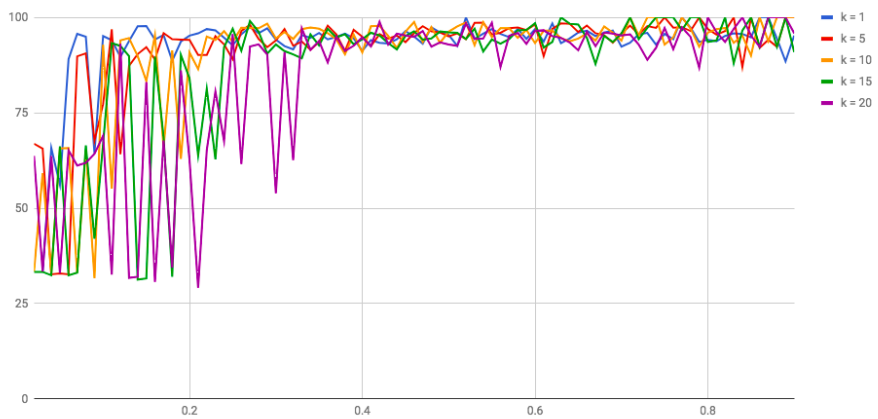

Figure 1: y = Accuracy, x = Split

By the time *split* is in the low 30s, the importance of the value of *k* seems to reduce to randomness (although from *split* about 70 upwards there seems

to be an increase in volatility of accuracy), while before *split* is close to 10, $k = 5, 10$ provide the most accuracy. It appears that the trade-off between accuracy and k is optimized, for this data set and implementation of kNN, at between $40 \leq split \leq 50$.

k-fold algorithm and implementation

My implementation of a k-fold algorithm that performs a k-fold cross validation on this data set can be found in q1_mod.py.

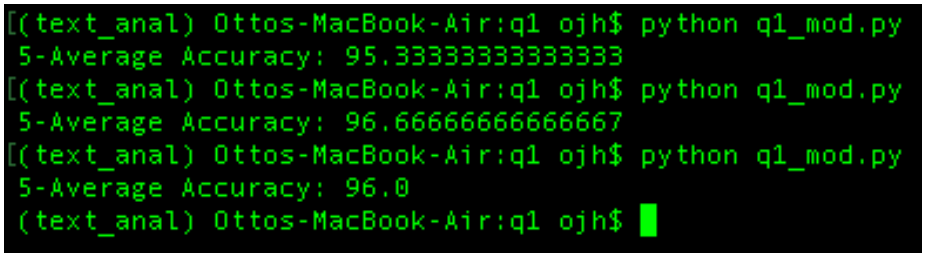

Figure 2: k = 5 output examples

Here is the folding function, which shuffles the data (we could alternatively stratify it), and iteratively populates k sets of data, which results in a k sets of data available for k-fold cross validation:

```
def get_data_for_k_folding(data_list, k):
# data
if len(data_list) < k: # if k is too large, we cannot proceed
  return False
random.shuffle(data_list) # shuffle the data; we could stratify the data if required
k_data_lists = [[] for j in range(k)] # return object: list of k lists filled with
     source data
# populate k_data_lists
k_index = 0
for j in range(len(data_list)): # iterate over each entry in data_list
  k_data_lists[k_index].append(data_list[j]) # add the data
  k_index += 1
  if k_index == k:
    k_index = 0
# return
return k_data_lists
```

The rest of the implementation does the following: for each of the k data sets created, run the k-nearest neighbour algorithm with that data set as test and the others combined as training; take the resulting accuracy scores, sum them, and divide by k to get the average accuracy.

Q2: Bayes Classifiers

New Features

```
def last_n_letters(word, n):
    if n < len(word):
        return {'last_n_letters': word[-n:]}
    else:
```

```
        return {'last_n_letters': word}

def first_n_letters(word, n):
    if n < len(word):
        return {'first_n_letters': word[:-n]}
    else:
        return {'first_n_letters': word}
```

Full code in q2.py. I extended the given formula to allow for n last letters and compared it to one allowing for n first letters. I also modified the code to use all of the data: 50% in training and 50% in test; this could easily be modified to change either the split or volume of total data used.

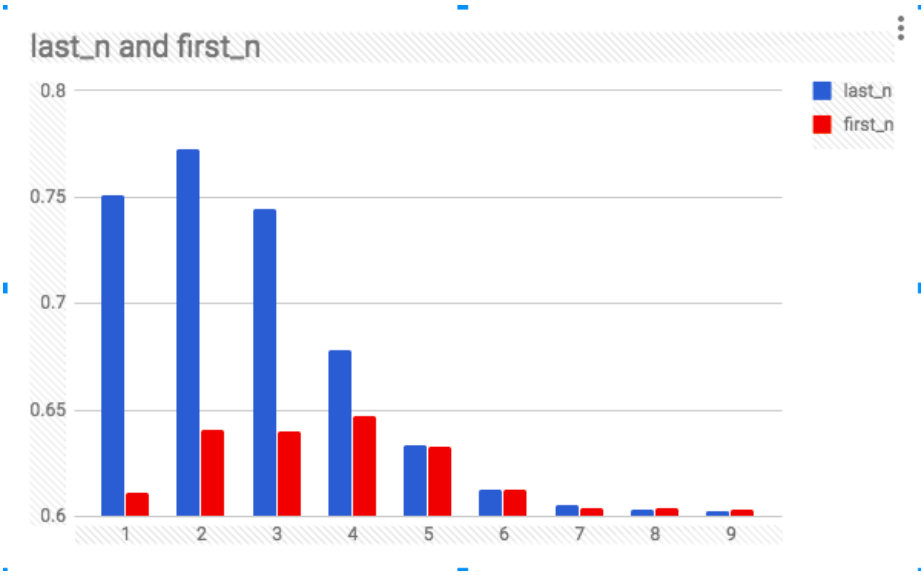

Figure 3: Comparing features with variable n

Note that given that I shuffle the data, each run will be slightly different, however, all runs demonstrate the same behaviour. In this data set, using 2 of the last letters is the most accurate predictor, both on an accuracy score of 77% and on finding the highest average odds of individual names e.g. 'na' is 93.5:1 a female name. It would be interesting to run this type of analysis, but to stratify the names by geography, culture, or other metrics e.g. I would expect Latin American names with genderised suffixes to be highly predictive of gender, but the phenomena would not be as predictive with, say, Germanic names.

Q3: SVMs

Modified code

```
def get_data():
  return datasets.load_digits()

def get_classifier(free_parameter, cost_function_parameter):
```

```
  return svm.SVC(gamma = free_parameter, C = cost_function_parameter)

def get_training_data(digits, number_left_out):
  return [digits.data[:-number_left_out], digits.target[:-number_left_out]]

def get_fitted_svm(free_parameter, cost_function_parameter, digits, number_left_out):
  classifier = get_classifier(free_parameter, cost_function_parameter)
  training_data = get_training_data(digits, number_left_out)
  X, y = training_data[0], training_data[1]
  classifier.fit(X, y)
  return classifier

def get_prediction(classifier, digits, number_left_out):
  return classifier.predict(digits.data[-number_left_out])

def get_visualisation(digits, number_left_out):
  plt.imshow(digits.images[-number_left_out], cmap = plt.cm.gray_r, interpolation =
       "nearest")
  return plt
```

Results

- 5 digits: successfully predicted the number
- 6 digits: successfully predicted the number
- 7 digits: I cannot tell what the image is, but it vaguely resemble an eight, which was predicted
- 8 digits: I cannot tell what the image is, but it vaguely resemble an eight, which was predicted
- 9 digits: successfully predicted the number
- 10 digits: successfully predicted the number

Discussion of C

- higher values make a "hard margin" and allows fewer errors when fitting the model (compared to lower C)
- higher/"harder" C could lead to overfitting the training data
- lower/"softer" C could allow for a more generalisable model

Discussion of λ

- if too large, then the radius of the area of influence of the area of support vecors only includes the support vector itself and will lead to overfitting
- if too small, then the model will not be able to capture the complexity of the data; the region of influence of any support vector would be all the training data, so it would behave like a linear model i.e. not capture the non-linear nature and complexity of the data

Conclusion: the tricky part of implementing an SVM is optimising the levels of C and λ