q1

October 24, 2019

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        import pandas as pd
        import string
        from sklearn.model_selection import KFold
        from sklearn.metrics import classification_report
        from sklearn.metrics import f1_score, precision_score, recall_score
        from statistics import mean
        from collections import Counter
        from nltk import ngrams
        from nltk.corpus import stopwords
        from nltk.tokenize import word_tokenize, sent_tokenize
        from copy import deepcopy
        import operator
```

```
In [2]: import nltk
        nltk.download('stopwords')
        nltk.download('punkt')
        nltk.download('averaged_perceptron_tagger')
```

```
[nltk_data] Downloading package stopwords to /home/mayank/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package punkt to /home/mayank/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /home/mayank/nltk_data...
[nltk_data]   Package averaged_perceptron_tagger is already up-to-
[nltk_data]       date!
```

```
Out[2]: True
```

```
In [3]: def get_ngrams(data, n, common_n):
            tokens = [token for token in data.split(" ") if token != ""]
            output = list(ngrams(tokens, n))

            ngram_counts = Counter(output)
            ng_counts = ngram_counts.most_common(common_n)
            return output

In [4]: stop_words = set(stopwords.words('english'))

        def get_postag(txt):
            tokenized = sent_tokenize(txt)

            wordsList = nltk.word_tokenize(tokenized[0])
            wordsList = [w for w in wordsList if not w in stop_words]
            tagged = nltk.pos_tag(wordsList)

            return tagged

In [5]: data = []
        one_grams = []
        uni = []
        bi = []
        tri = []
        pos = []
        file = open('./traindata.txt')

        for line in file:
            line = line.split(':')
            row = []
            row.append(line[0])
            row.append(' '.join(line[1].split(' ')[1:]).translate(str.maketrans('', '', string

            length = len(row[1].split(' '))
            unigram = get_ngrams(row[1], 1, 500)
            bigram = get_ngrams(row[1], 2, 300)
            trigram = get_ngrams(row[1], 3, 200)
            postag = get_postag(row[1])

            row.append(length)
            row.append(unigram)
            uni.extend(unigram)
            row.append(bigram)
            bi.extend(bigram)
            row.append(trigram)
            tri.extend(trigram)
            row.append(postag)
            pos.extend(postag)

            data.append(row)

        len(data)
        print(data[0])
```

```
['DESC', 'How did serfdom develop in and then leave Russia', 9, [('How',), ('did',), ('serfdom
```

```
In [6]: def top_grams(grams, top_n):
            return Counter(grams).most_common(top_n)
```

```
In [7]: unigram_counts = top_grams(uni, 500)
        bigram_counts = top_grams(bi, 300)
        trigram_counts = top_grams(tri, 200)
        pos_counts = top_grams(pos, 500)
        lengthAvg = mean([row[2] for row in data])
        print(lengthAvg)
```

```
9.031548055759353
```

```
In [8]: def is_numeric(value):
            return isinstance(value, int) or isinstance(value, float)
```

```
In [9]: header = ['Label', 'Text', 'Length', 'Unigram', 'Bigram', 'Trigram']
        class Question:
            def __init__(self, column, value):
                self.column = column
                self.value = value

            def match(self, example):
                val = example[self.column]

                if is_numeric(val):
                    return val <= self.value

                return self.value in val

            def __repr__(self):
                condition = "contains"
                return "Does %s %s %s?" % (
                    header[self.column], condition, str(self.value))
```

```
In [10]: def class_counts(rows):
             counts = {}
             for row in rows:
                 label = row[0]
                 if label not in counts:
                     counts[label] = 0
                 counts[label] += 1
             return counts
```

```
In [11]: def gini(rows):
             counts = class_counts(rows)
             impurity = 1
             for lbl in counts:
                 prob_of_lbl = counts[lbl] / float(len(rows))
                 impurity -= prob_of_lbl**2
             return impurity
```

```
In [12]: def info_gain(left, right, current_uncertainty):
             p = float(len(left)) / (len(left) + len(right))
             return current_uncertainty - p * gini(left) - (1 - p) * gini(right)
```

```
In [13]: class Leaf:
             def __init__(self, rows):
                 self.predictions = class_counts(rows)
```

```
In [14]: class Decision_Node:
             def __init__(self,
                          question,
                          true_branch,
                          false_branch):
                 self.question = question
                 self.true_branch = true_branch
                 self.false_branch = false_branch
```

```
In [15]: questions = []

         for x in unigram_counts:
             questions.append(Question(3, x[0]))

         for x in bigram_counts:
             questions.append(Question(4, x[0]))

         for x in trigram_counts:
             questions.append(Question(5, x[0]))

         for x in pos_counts:
             questions.append(Question(6, x[0]))

         questions.append(Question(2, lengthAvg))

         print(len(questions))
         # print(questions[1500])
```

```
1501
```

```
In [16]: def partition(rows, question):
             trueRows = []
             falseRows = []
```

```
        for r in rows:
            if question.match(r):
                trueRows.append(r)
            else:
                falseRows.append(r)

        return trueRows, falseRows
```

```
In [17]: def findBestSplit(rows, questions):
             best_gain = 0
             best_question = None
             current_uncertainty = gini(rows)

             for q in questions:
                 trueRows, falseRows = partition(rows, q)
                 if len(trueRows) == 0 or len(falseRows) == 0:
                     continue

                 gain = info_gain(trueRows, falseRows, current_uncertainty)

                 if gain >= best_gain:
                     best_gain, best_question = gain, q

             return best_gain, best_question
```

```
In [18]: def formTree(rows, questions):
             gain, question = findBestSplit(rows, questions)

             if gain == 0:
                 return Leaf(rows)

             trueRows, falseRows = partition(rows, question)
             questions.remove(question)

             trueBranch = formTree(trueRows, questions)
             falseBranch = formTree(falseRows, questions)

             return Decision_Node(question, trueBranch, falseBranch)
```

```
In [19]: def classifyRow(node, row):
             if isinstance(node, Leaf):
                 return node.predictions

             if node.question.match(row):
                 return classifyRow(node.true_branch, row)
             else:
                 return classifyRow(node.false_branch, row)
```

```
In [20]: def train(data, questions):
             return formTree(data, deepcopy(questions))
```

```
In [21]: def classify(root, rows):
             predictions = []
             for r in rows:
                 predictions.append(max(classifyRow(root, r).items(), key=operator.itemgetter(
             return predictions
```

```
In [22]: def getDataInIndex(data, index):
             l = []
             for i in range(len(data)):
                 if i in index:
                     l.append(data[i])
             return l
```

```
In [23]: def getActualLabels(act_data):
             act_labels = []
             for d in act_data:
                 act_labels.append(d[0])
             return act_labels
```

```
In [24]: kfold = KFold(10, True, 1)
         precision = []
         recall = []
         f_score = []
         i = 0

         for trainInd,testInd in kfold.split(data):
             train_data = getDataInIndex(data, trainInd)
             test_data = getDataInIndex(data, testInd)

             root = train(train_data, questions)

             prediction = classify(root, test_data)

             actual = getActualLabels(test_data)
             predicted = prediction

         #     print(classification_report(actual, predicted))
             precision.append(precision_score(actual, predicted, average='macro'))
             recall.append(recall_score(actual, predicted, average='macro'))
             f_score.append(f1_score(actual, predicted, average='macro'))

             print("Training...")

         print("Precision Score = " + str(mean(precision)))
         print("Recall Score = " + str(mean(recall)))
         print("F Score = " + str(mean(f_score)))
```

```
Training...
Training...
Training...
Training...
Training...
Training...
Training...
Training...
Training...
Training...
Precision Score = 0.8028907609403665
Recall Score = 0.7574562257221622
F Score = 0.7722368576621397
```