q1

November 1, 2019

```
In [9]: 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, accuracy_score, co
        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
        from math import log2
```

```
In [10]: import nltk
         nltk.download('stopwords')
         nltk.download('punkt')
         nltk.download('averaged_perceptron_tagger')
```

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

```
Out[10]: True
```

```
In [11]: def get_ngrams(data, n):
             tokens = [token for token in data.split(" ") if token != ""]
             output = list(ngrams(tokens, n))
             return output
```

```
In [12]: 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 [13]: def buildData(filePath):
             data = []
             uni = []
             bi = []
             tri = []
             pos = []
             file = open(filePath)

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

                 length = len(row[1].split(' '))
                 unigram = get_ngrams(row[1], 1)
                 bigram = get_ngrams(row[1], 2)
                 trigram = get_ngrams(row[1], 3)
                 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)

             return data, uni, bi, tri, pos
```

```
In [14]: data, uni, bi, tri, pos = buildData('./traindata.txt')
```

```
In [15]: def top_grams(grams, top_n):
             return Counter(grams).most_common(top_n)
```

```
In [16]: 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 [17]: def is_numeric(value):
             return isinstance(value, int) or isinstance(value, float)
```

```
In [18]: 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 [19]: 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 [20]: 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 [21]: def misclassifcation_error(rows):
             counts = class_counts(rows)
             max_prob = 0
             for lbl in counts:
                 prob_of_lbl = counts[lbl] / float(len(rows))
                 if prob_of_lbl > max_prob:
                     max_prob = prob_of_lbl
             return 1 - max_prob
```

```
In [22]: def entropy(rows):
             counts = class_counts(rows)
             impurity = 0
             for lbl in counts:
                 prob_of_lbl = counts[lbl] / float(len(rows))
                 impurity -= prob_of_lbl*log2(prob_of_lbl)
             return impurity
```

```
In [23]: def info_gain(left, right, current_uncertainty, func):
             p = float(len(left)) / (len(left) + len(right))
             return current_uncertainty - p * func(left) - (1 - p) * func(right)
```

```
In [24]: class Leaf:
             def __init__(self, rows):
                 self.predictions = class_counts(rows)
```

```
In [25]: 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 [26]: 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 [27]: 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 [28]: def findBestSplit(rows, questions, func):
             best_gain = 0
             best_question = None
             current_uncertainty = func(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, func)

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

             return best_gain, best_question
```

```
In [29]: def formTree(rows, questions, func):
             gain, question = findBestSplit(rows, questions, func)

             if gain == 0:
                 return Leaf(rows)

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

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

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

```
In [30]: 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 [31]: def train(data, questions, func):
             return formTree(data, deepcopy(questions), func)
```

```
In [32]: def classify(root, rows):
             predictions = []
             for r in rows:
                 predictions.append(max(classifyRow(root, r).items(), key=operator.itemgetter(
             return predictions
```

```
In [33]: def getDataInIndex(data, index):
             l = []
             for i in range(len(data)):
                 if i in index:
                     l.append(data[i])
             return l
```

```
In [34]: 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, gini)
```

```
    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
```

0.1 Part 2

- All
- Unigram, Bigram, Trigram, POS
- Unigram, Bigram, Trigram

```
In [35]: classes = ['ABBR', 'DESC', 'ENTY', 'HUM', 'LOC', 'NUM']
```

```
In [54]: def getReport(traindata, testdata, uniFlag=True, biFlag=True, triFlag=True, posFlag=T
             allQuestions = []

             if uniFlag:
                 for x in unigram_counts:
                     allQuestions.append(Question(3, x[0]))

             if biFlag:
                 for x in bigram_counts:
```

```
            allQuestions.append(Question(4, x[0]))

    if triFlag:
        for x in trigram_counts:
            allQuestions.append(Question(5, x[0]))

    if posFlag:
        for x in pos_counts:
            allQuestions.append(Question(6, x[0]))

    if lenFlag:
        allQuestions.append(Question(2, lengthAvg))

    print("No of questions = " + str(len(allQuestions)))
    print("Training...")
    root = train(traindata, allQuestions, func)
    print("Predicting...")
    prediction = classify(root, testdata)
    actual = getActualLabels(testdata)
    print("Prediction done...")
    matrix = confusion_matrix(actual, prediction)
    acc = matrix.diagonal()/matrix.sum(axis=1)
    accuracy_report = dict(zip(classes, acc))

    return accuracy_report, root, prediction, actual
```

```
In [55]: testdata = buildData('./testdata.txt')[0]
         len(testdata)
```

```
Out[55]: 500
```

```
In [38]: print(getReport(traindata=data, testdata=testdata)[0])
```

```
No of questions = 1501
Training...
Predicting...
Prediction done...
{'ABBR': 0.6666666666666666, 'DESC': 0.9710144927536232, 'ENTY': 0.723404255319149, 'HUM': 0.84
```

```
In [39]: print(getReport(traindata=data, testdata=testdata, func=entropy)[0])
```

```
No of questions = 1501
Training...
Predicting...
Prediction done...
{'ABBR': 0.6666666666666666, 'DESC': 0.9710144927536232, 'ENTY': 0.5, 'HUM': 0.8615384615384616
```

```
In [40]: print(getReport(traindata=data, testdata=testdata, func=misclassifcation_error)[0])
```

```
No of questions = 1501
Training...
Predicting...
Prediction done...
{'ABBR': 0.6666666666666666, 'DESC': 0.8260869565217391, 'ENTY': 0.7978723404255319, 'HUM': 0.8
```

```
In [41]: print(getReport(traindata=data, testdata=testdata, lenFlag=False)[0])
```

```
No of questions = 1500
Training...
Predicting...
Prediction done...
{'ABBR': 0.6666666666666666, 'DESC': 0.9710144927536232, 'ENTY': 0.723404255319149, 'HUM': 0.84
```

```
In [44]: print(getReport(traindata=data, testdata=testdata, lenFlag=False, func=entropy)[0])
```

```
No of questions = 1500
Training...
Predicting...
Prediction done...
{'ABBR': 0.6666666666666666, 'DESC': 0.9710144927536232, 'ENTY': 0.5, 'HUM': 0.8615384615384616
```

```
In [45]: print(getReport(traindata=data, testdata=testdata, lenFlag=False, func=misclassifcatio
```

```
No of questions = 1500
Training...
Predicting...
Prediction done...
{'ABBR': 0.6666666666666666, 'DESC': 0.8260869565217391, 'ENTY': 0.7978723404255319, 'HUM': 0.8
```

```
In [46]: print(getReport(traindata=data, testdata=testdata, lenFlag=False, posFlag=False)[0])
```

```
No of questions = 1000
Training...
Predicting...
Prediction done...
{'ABBR': 0.6666666666666666, 'DESC': 0.9782608695652174, 'ENTY': 0.6276595744680851, 'HUM': 0.8
```

```
In [47]: print(getReport(traindata=data, testdata=testdata, lenFlag=False, posFlag=False, func=
```

```
No of questions = 1000
Training...
Predicting...
Prediction done...
{'ABBR': 0.6666666666666666, 'DESC': 0.427536231884058, 'ENTY': 0.648936170212766, 'HUM': 0.876
```

```
In [48]: print(getReport(traindata=data, testdata=testdata, lenFlag=False, posFlag=False, func=
```

```
No of questions = 1000
Training...
Predicting...
Prediction done...
{'ABBR': 0.6666666666666666, 'DESC': 0.8188405797101449, 'ENTY': 0.7340425531914894, 'HUM': 0.8
```

1 Error Analysis

```
In [51]: def getWrongPrediction(prediction, actual, dataset):
             data_list = []

             for i in range(len(prediction)):
                 if prediction[i] !=actual[i] :
                     data_list.append(dataset[i])

             return data_list
```

```
In [56]: _ , root_gini, prediction_gini, actual_gini  = getReport(traindata=data, testdata=test
         wrong_data = getWrongPrediction(prediction_gini, actual_gini, testdata)
```

```
No of questions = 1501
Training...
Predicting...
Prediction done...
```

```
In [58]: len(wrong_data)
```

```
Out[58]: 88
```

```
In [61]: _ , root_entropy, prediction_entropy, actual_entropy  = getReport(traindata=data, test
         wrong_data_en = getWrongPrediction(prediction_entropy, actual_entropy, wrong_data)
         len(wrong_data_en)
```

```
No of questions = 1501
Training...
Predicting...
Prediction done...
```

```
Out[61]: 78
```

```
In [63]: _ , root_mis, prediction_mis, actual_mis  = getReport(traindata=data, testdata=wrong_d
         wrong_data_mis = getWrongPrediction(prediction_entropy, actual_entropy, wrong_data)
         len(wrong_data_mis)
```

```
No of questions = 1501
Training...
Predicting...
Prediction done...
```

```
Out[63]: 78
```