[1]:

```
import re
import nltk
import numpy as np
from pyspark.ml.feature import StopWordsRemover
from pyspark.ml.feature import CountVectorizer
from pyspark.ml.feature import Tokenizer
from pyspark.ml.feature import NGram
from pyspark.sql import Row
from pyspark.sql.types import *
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.classification import LogisticRegressionWithLBFGS
from pyspark.mllib.classification import NaiveBayes, NaiveBayesModel
from pyspark.mllib.evaluation import MulticlassMetrics
```

[2]:

```
data = sc.wholeTextFiles("F:/Data3/*.txt",use_unicode=True)
```

[3]:

```
def parse_text(text):
        regex = re.compile(r'[\ufeff\n\r\t]')
        t = regex.sub(" ", text[1])
        q = re.sub(r'\d', ' ', t)
        return (text[0],q)

def parse_text1(text):
        text = text.replace('\ufeff', ' ')
        text = re.sub(r'[-!$%^*&>:;,.?#/@"()\[\]]',' ',text.lower().replace('\r\n', ' '))
        text = re.sub(r'\d', ' ', text)
        text = re.sub(r'twain|chapter',' ',text)
        text = ' '.join(text.split())
        return text

def filter_test(text, lst):
        file = text[0]
        bg = text[1]
        res=[]
        for i in lst:
                for j in bg:
                        if i==j:
                                res.append(i)
        return (file,res)

def token_to_pos(ch):
        tokens = nltk.word_tokenize(ch[1])
        return [p[1] for p in nltk.pos_tag(tokens)]

def Syn(ch):
        chapters_pos = token_to_pos(ch)
        pos_list = ['NN', 'NNP', 'DT', 'IN', 'JJ', 'NNS']
        fvs_syntax = [[ch.count(pos) for pos in pos_list] for ch in chapters_pos]
        fvs_syntax = fvs_syntax[0]

        result=[]

        for i in fvs_syntax:
                for j in chapters_pos:
                        result.append(float(format(float(i/len(j)),'.4f')))

        r1=result[0]
        r2=result[1]
        r3=result[2]
        r4=result[3]
        r5=result[4]
        r6=result[5]

        return (ch[0],r1,r2,r3,r4,r5,r6)

def LexicalFeatures(chapters):
        sentence_tokenizer = nltk.data.load('tokenizers/punkt/english.pickle')
        word_tokenizer = nltk.tokenize.RegexpTokenizer(r'\w+')

        tokens = nltk.word_tokenize(chapters[1].lower())
        words = word_tokenizer.tokenize(chapters[1].lower())
        sentences = sentence_tokenizer.tokenize(chapters[1])
        vocab = set(words)

        words_per_sentence = np.array([len(word_tokenizer.tokenize(s)) for s in sentences])
        fvs_lexical_0 = format(words_per_sentence.mean(),'.4f')
        fvs_lexical_1 = format(words_per_sentence.std(),'.4f')
        fvs_lexical_2 = format(len(vocab) / float(len(words)),'.4f')
        fvs_punct_0 = format(tokens.count(',') / float(len(sentences)),'.4f')
        fvs_punct_1 = format(tokens.count(';') / float(len(sentences)),'.4f')
        fvs_punct_2 = format(tokens.count(':') / float(len(sentences)),'.4f')

        return (chapters[0],float(fvs_lexical_0), float(fvs_lexical_1),float(fvs_lexical_2),float(fvs_punct_0),\
            float(fvs_punct_1),float(fvs_punct_2))
```

[4]:
```
data_parse = data.map(lambda x: (x[0].split("/")[-1],parse_text1(x[1])))
```

[5]:
```
textDF = spark.createDataFrame(data_parse, ["label", "text"])
tokenizer = Tokenizer(inputCol="text", outputCol="words")
wordsDataFrame = tokenizer.transform(textDF)
remover = StopWordsRemover(inputCol="words", outputCol="stwords")
DF = remover.transform(wordsDataFrame)

pair = DF.rdd.flatMap(lambda df: df[3]).map(lambda token: (token,1)).reduceByKey(lambda v1,v2 : v1+v2).sortBy(lambda x: x[1],Fal
se)
stp = pair.map(lambda x: x[0]).collect()[0:20]

filrdd = DF.rdd.map(lambda df: (df[0],df[3])).map(lambda l: filter_test(l,stp))
FDF = spark.createDataFrame(filrdd, ["label", "finalword"])

cv = CountVectorizer(inputCol="finalword", outputCol="features")
model = cv.fit(FDF)
result = model.transform(FDF)
finaldata = result.select("label","features")

word_df = finaldata.rdd.map(lambda x: Row(Doc_Name=x[0],f1=float(x[1].toArray()[0]),f2=float(x[1].toArray()[1]),\
                            f3=float(x[1].toArray()[2]),f4=float(x[1].toArray()[3]),f5=float(x[1].toArray()[4]),\
                            f6=float(x[1].toArray()[5]),f7=float(x[1].toArray()[6]),f8=float(x[1].toArray()[7]),\
                            f9=float(x[1].toArray()[8]),f10=float(x[1].toArray()[9]),f11=float(x[1].toArray()[10]),\
                            f12=float(x[1].toArray()[11]),f13=float(x[1].toArray()[12]),f14=float(x[1].toArray()[13]),\
                            f15=float(x[1].toArray()[14]),f16=float(x[1].toArray()[15]),f17=float(x[1].toArray()[16]),\
                            f18=float(x[1].toArray()[17]),f19=float(x[1].toArray()[18]),f20=float(x[1].toArray()[19]))).toDF()
```

[6]:
```
bigram = NGram(n=2, inputCol="words", outputCol="bigrams")
bigramDF = bigram.transform(wordsDataFrame)

bipair = bigramDF.rdd.flatMap(lambda df: df[3]).map(lambda token: (token,1)).reduceByKey(lambda v1,v2 : v1+v2).sortBy(lambda x:
1],False)
bistp = bipair.map(lambda x: x[0]).collect()[0:10]

bifilrdd = bigramDF.rdd.map(lambda df: (df[0],df[3])).map(lambda l: filter_test(l,bistp))
biFDF = spark.createDataFrame(bifilrdd, ["label", "finalword"])


bicv = CountVectorizer(inputCol="finalword", outputCol="features")
bimodel = cv.fit(biFDF)
biresult = bimodel.transform(biFDF)
bifinaldata = biresult.select("label","features")

bigram_df = bifinaldata.rdd.map(lambda x: Row(Doc_Name=x[0],bg1=float(x[1].toArray()[0]),bg2=float(x[1].toArray()[1]),\
                                bg3=float(x[1].toArray()[2]),bg4=float(x[1].toArray()[3]),bg5=float(x[1].toArray()[4]),\
                                bg6=float(x[1].toArray()[5]),bg7=float(x[1].toArray()[6]),bg8=float(x[1].toArray()[7]),\
                                bg9=float(x[1].toArray()[8]),bg10=float(x[1].toArray()[9]))).toDF()
```

[7]:
```
trigram = NGram(n=3, inputCol="words", outputCol="trigrams")
trigramDF = trigram.transform(wordsDataFrame)

tripair = trigramDF.rdd.flatMap(lambda df: df[3]).map(lambda token: (token,1)).reduceByKey(lambda v1,v2 : v1+v2).sortBy(lambda
x: x[1],False)
tristp = tripair.map(lambda x: x[0]).collect()[0:10]

trifilrdd = trigramDF.rdd.map(lambda df: (df[0],df[3])).map(lambda l: filter_test(l,tristp))
triFDF = spark.createDataFrame(trifilrdd, ["label", "finalword"])

tricv = CountVectorizer(inputCol="finalword", outputCol="features")
trimodel = cv.fit(triFDF)
triresult = trimodel.transform(triFDF)
trifinaldata = triresult.select("label","features")

trigram_df = trifinaldata.rdd.map(lambda x: Row(Doc_Name=x[0],tg1=float(x[1].toArray()[0]),tg2=float(x[1].toArray()[1]),\
                                  tg3=float(x[1].toArray()[2]),tg4=float(x[1].toArray()[3]),tg5=float(x[1].toArray()[4]),\
                                  tg6=float(x[1].toArray()[5]),tg7=float(x[1].toArray()[6]),tg8=float(x[1].toArray()[7]),\
                                  tg9=float(x[1].toArray()[8]),tg10=float(x[1].toArray()[9]))).toDF()
```

[8]:
```
qgram = NGram(n=4, inputCol="words", outputCol="qgrams")
qgramDF = qgram.transform(wordsDataFrame)
```

```
qpair = qgramDF.rdd.flatMap(lambda df: df[3]).map(lambda token: (token,1)).reduceByKey(lambda v1,v2 : v1+v2).sortBy(lambda x: x[1
False)
qstp = qpair.map(lambda x: x[0]).collect()[0:10]

qfilrdd = qgramDF.rdd.map(lambda df: (df[0],df[3])).map(lambda l: filter_test(l,qstp))
qFDF = spark.createDataFrame(qfilrdd, ["label", "finalword"])

qcv = CountVectorizer(inputCol="finalword", outputCol="features")
qmodel = cv.fit(qFDF)
qresult = qmodel.transform(qFDF)
qfinaldata = qresult.select("label","features")

qgram_df = qfinaldata.rdd.map(lambda x: Row(Doc_Name=x[0],qg1=float(x[1].toArray()[0]),qg2=float(x[1].toArray()[1]),\
                              qg3=float(x[1].toArray()[2]),qg4=float(x[1].toArray()[3]),qg5=float(x[1].toArray()[4]),\
                              qg6=float(x[1].toArray()[5]),qg7=float(x[1].toArray()[6]),qg8=float(x[1].toArray()[7]),\
                              qg9=float(x[1].toArray()[8]),qg10=float(x[1].toArray()[9]))).toDF()
```

[9]:

```
data2 = data.map(lambda x: (x[0].split("/")[-1],x[1].lower()))

data_new = data2.map(parse_text)

lexical=data_new.map(LexicalFeatures)

syntactic=data_new.map(Syn)

lexical_df=spark.createDataFrame(lexical, ['Doc_Name','wrds_sent_mean', 'wrds_sent_std','Lex_diver','Commas_sent','Semicolon_sen
t','Colons_sent'])
syntactic_df=spark.createDataFrame(syntactic, ['Doc_Name','NN', 'NNP', 'DT', 'IN', 'JJ', 'NNS'])
```

[10]:

```
lexical_df.createOrReplaceTempView("lexical_df")
syntactic_df.createOrReplaceTempView("syntactic_df")
word_df.createOrReplaceTempView("word_df")
bigram_df.createOrReplaceTempView("bigram_df")
trigram_df.createOrReplaceTempView("trigram_df")
qgram_df.createOrReplaceTempView("qgram_df")
```

[11]:

```
final_df=spark.sql(" SELECT CASE WHEN ld.Doc_Name == 'Austen1.txt' THEN '0' \
                                 WHEN ld.Doc_Name == 'Austen2.txt' THEN '0' \
                                 WHEN ld.Doc_Name == 'Austen3.txt' THEN '0' \
                                 WHEN ld.Doc_Name == 'Austen4.txt' THEN '0' \
                                 WHEN ld.Doc_Name == 'Austen5.txt' THEN '0' \
                                 WHEN ld.Doc_Name == 'Dickens1.txt' THEN '1' \
                                 WHEN ld.Doc_Name == 'Dickens2.txt' THEN '1' \
                                 WHEN ld.Doc_Name == 'Dickens3.txt' THEN '1' \
                                 WHEN ld.Doc_Name == 'Dickens4.txt' THEN '1' \
                                 WHEN ld.Doc_Name == 'Dickens5.txt' THEN '1' \
                                 WHEN ld.Doc_Name == 'Twain1.txt' THEN '2' \
                                 WHEN ld.Doc_Name == 'Twain2.txt' THEN '2' \
                                 WHEN ld.Doc_Name == 'Twain3.txt' THEN '2' \
                                 WHEN ld.Doc_Name == 'Twain4.txt' THEN '2' \
                                 WHEN ld.Doc_Name == 'Twain5.txt' THEN '2' END Author,\
                                 ld.wrds_sent_mean, ld.wrds_sent_std, ld.Lex_diver, ld.Commas_sent, ld.Semicolon_sent, ld.Colons
_sent, \
                                 sd.NN, sd.NNP, sd.DT, sd.IN, sd.JJ, sd.NNS, wd.f1, wd.f2,  wd.f3, wd.f4, wd.f5, wd.f6, wd.f7, w
d.f8, \
                                 wd.f9, wd.f10, wd.f11, wd.f12, wd.f13, wd.f14, wd.f15, wd.f16, wd.f17, wd.f18, wd.f19, wd.f20,
 bd.bg1, \
                                 bd.bg2, bd.bg3, bd.bg4, bd.bg5, bd.bg6, bd.bg7, bd.bg8, bd.bg9, bd.bg10, td.tg1, td.tg2, td.tg
3, td.tg4, \
                                 td.tg5, td.tg6, td.tg7, td.tg8, td.tg9, td.tg10, qd.qg1, qd.qg2, qd.qg3, qd.qg4, qd.qg5, qd.qg
6, qd.qg7, \
                                 qd.qg8, qd.qg9, qd.qg10,ld.Doc_Name \
                      FROM lexical_df ld, syntactic_df sd, word_df wd, bigram_df bd, trigram_df td, qgram_df qd \
                      WHERE ld.Doc_Name = sd.Doc_Name and ld.Doc_Name = wd.Doc_Name \
                        and ld.Doc_Name = bd.Doc_Name and ld.Doc_Name = td.Doc_Name and ld.Doc_Name = qd.Doc_Name ")

final_df.createOrReplaceTempView("final_df")
```

[12]:

```
training_df=spark.sql("SELECT Author, wrds_sent_mean, wrds_sent_std, Lex_diver, Commas_sent, Semicolon_sent, Colons_sent, NN, NN
P, DT, IN, JJ, NNS, \
                           f1, f2,  f3, f4, f5, f6, f7, f8, f9, f10, f11, f12, f13, f14, f15, f16, f17, f18, f19, f20, bg1, bg2,
 bg3, bg4, bg5, \
                           bg6, bg7, bg8, bg9, bg10, tg1, tg2, tg3, tg4, tg5, tg6, tg7, tg8, tg9, tg10, qg1, qg2, qg3, qg4, qg5,
 qg6, qg7, qg8, \
                           qg9, qg10 FROM final_df WHERE Doc_Name not in ('Austen2.txt', 'Dickens3.txt', 'Twain5.txt')")
```

```
test_df=spark.sql("SELECT Author, wrds_sent_mean, wrds_sent_std, Lex_diver, Commas_sent, Semicolon_sent, Colons_sent, NN, NNP, D
T, IN, JJ, NNS, \
                            f1, f2,  f3, f4, f5, f6, f7, f8, f9, f10, f11, f12, f13, f14, f15, f16, f17, f18, f19, f20, bg1, bg2,
 bg3, bg4, bg5, \
                            bg6, bg7, bg8, bg9, bg10, tg1, tg2, tg3, tg4, tg5, tg6, tg7, tg8, tg9, tg10, qg1, qg2, qg3, qg4, qg5,
 qg6, qg7, qg8, \
                            qg9, qg10 FROM final_df WHERE Doc_Name in ('Austen2.txt', 'Dickens3.txt', 'Twain5.txt')")
```

[13]:
```
training_df.repartition(1).write.format("com.databricks.spark.csv").options(header='true', inferschema='true').save("F:/BookTrai
ning.csv")
test_df.repartition(1).write.format("com.databricks.spark.csv").options(header='true', inferschema='true').save("F:/BookTest.cs
v")
```

[14]:
```
trainingdf = spark.read.format('com.databricks.spark.csv').options(header='true', inferschema='true').load('F:/Book_Training.cs
v')
testdf = spark.read.format('com.databricks.spark.csv').options(header='true', inferschema='true').load('F:/Book_Test.csv')
```

[15]:
```
training= trainingdf.rdd.map(lambda row: LabeledPoint(row[0], row[1:]))
test= testdf.rdd.map(lambda row: LabeledPoint(row[0], row[1:]))
```

[16]:
```
## Logistic Regression
modelLR = LogisticRegressionWithLBFGS.train(training, numClasses=3)
predictionAndLabelsLR = training.map(lambda l: (float(modelLR.predict(l.features)), l.label))
metricsLR = MulticlassMetrics(predictionAndLabelsLR)
print('Confusion Matrix for Logistic Regression : \n\n' + str(metricsLR.confusionMatrix().toArray()) +
'\n\nSummary Of Statistics For Logistic Regression\n\nTotal Accuracy  : ' + str(format(metricsLR.accuracy,'.2f')) +
'\n\nTotal Precision : ' + str(format(metricsLR.precision(),'.2f')) + '\t Precision 1: ' + str(format(metricsLR.precision(0),'.2
f')) + '\t Precision 2: ' + str(format(metricsLR.precision(1),'.2f')) + '\t Precision 3: ' + str(format(metricsLR.precision(2),'.
 +
'\n\nTotal Recall    : ' + str(format(metricsLR.recall(),'.2f')) + '\t Recall 1   : ' + str(format(metricsLR.recall(0),'.2f')) +
 '\t Recall 2   : ' + str(format(metricsLR.recall(1),'.2f')) + '\t Recall 3   : ' + str(format(metricsLR.recall(2),'.2f')) +
'\n\nTotal fMeasure  : ' + str(format(metricsLR.fMeasure(),'.2f')))
```

```
C:\spark-2.0.1-bin-hadoop2.7\python\pyspark\mllib\evaluation.py:237: UserWarning: Deprecated in 2.0.0. Use accuracy.
  warnings.warn("Deprecated in 2.0.0. Use accuracy.")
```

```
Confusion Matrix for Logistic Regression :

[[ 4.  0.  0.]
 [ 0.  4.  0.]
 [ 0.  0.  4.]]

Summary Of Statistics For Logistic Regression

Total Accuracy  : 1.00

Total Precision : 1.00   Precision 1: 1.00       Precision 2: 1.00       Precision 3: 1.00

Total Recall    : 1.00   Recall 1   : 1.00       Recall 2   : 1.00       Recall 3   : 1.00

Total fMeasure  : 1.00
```

```
C:\spark-2.0.1-bin-hadoop2.7\python\pyspark\mllib\evaluation.py:249: UserWarning: Deprecated in 2.0.0. Use accuracy.
  warnings.warn("Deprecated in 2.0.0. Use accuracy.")
C:\spark-2.0.1-bin-hadoop2.7\python\pyspark\mllib\evaluation.py:262: UserWarning: Deprecated in 2.0.0. Use accuracy.
  warnings.warn("Deprecated in 2.0.0. Use accuracy.")
```

[17]:
```
print('\nPredictions using test data:\n')
predictionAndLabelsLR_test = test.map(lambda l: (float(modelLR.predict(l.features)), l.label))
predictionAndLabelsLR_test.collect()
```

```
Predictions using test data:
```

[17]:
```
[(1.0, 1.0), (2.0, 2.0), (0.0, 0.0)]
```

[18]:
```
## Naive Bayes
modelNB = NaiveBayes.train(training)
predictionAndLabelsNB = training.map(lambda l: (float(modelNB.predict(l.features)), l.label))
metricsNB = MulticlassMetrics(predictionAndLabelsNB)
print('Confusion Matrix for Naive Bayes : \n\n' + str(metricsNB.confusionMatrix().toArray()) +
'\n\nSummary Of Statistics For Naive Bayes\n\nTotal Accuracy  : ' + str(format(metricsNB.accuracy,'.2f')) +
'\n\nTotal Precision : ' + str(format(metricsNB.precision(),'.2f')) + '\t Precision 1: ' + str(format(metricsNB.precision(0),'.2
f')) + '\t Precision 2: ' + str(format(metricsNB.precision(1),'.2f')) + '\t Precision 3: ' + str(format(metricsNB.precision(2),'.
 +
'\n\nTotal Recall    : ' + str(format(metricsNB.recall(),'.2f')) + '\t Recall 1   : ' + str(format(metricsNB.recall(0),'.2f')) +
 '\t Recall 2   : ' + str(format(metricsNB.recall(1),'.2f')) + '\t Recall 3   : ' + str(format(metricsNB.recall(2),'.2f')) +
'\n\nTotal fMeasure  : ' + str(format(metricsNB.fMeasure(),'.2f')))
```

```
C:\spark-2.0.1-bin-hadoop2.7\python\pyspark\mllib\evaluation.py:237: UserWarning: Deprecated in 2.0.0. Use accuracy.
  warnings.warn("Deprecated in 2.0.0. Use accuracy.")
```

```
Confusion Matrix for Naive Bayes :

[[ 4.  0.  0.]
 [ 1.  3.  0.]
 [ 0.  0.  4.]]

Summary Of Statistics For Naive Bayes

Total Accuracy  : 0.92

Total Precision : 0.92   Precision 1: 0.80      Precision 2: 1.00      Precision 3: 1.00

Total Recall    : 0.92   Recall 1   : 1.00      Recall 2   : 0.75      Recall 3   : 1.00

Total fMeasure  : 0.92
```

```
C:\spark-2.0.1-bin-hadoop2.7\python\pyspark\mllib\evaluation.py:249: UserWarning: Deprecated in 2.0.0. Use accuracy.
  warnings.warn("Deprecated in 2.0.0. Use accuracy.")
C:\spark-2.0.1-bin-hadoop2.7\python\pyspark\mllib\evaluation.py:262: UserWarning: Deprecated in 2.0.0. Use accuracy.
  warnings.warn("Deprecated in 2.0.0. Use accuracy.")
```

```
[19]: print("\nPredictions using test data:\n")
      predictionAndLabels_testNB = test.map(lambda l: (float(modelNB.predict(l.features)), l.label))
      predictionAndLabels_testNB.collect()
```

```
      Predictions using test data:


[19]: [(1.0, 1.0), (2.0, 2.0), (0.0, 0.0)]
```