Clustering

Data Preparation

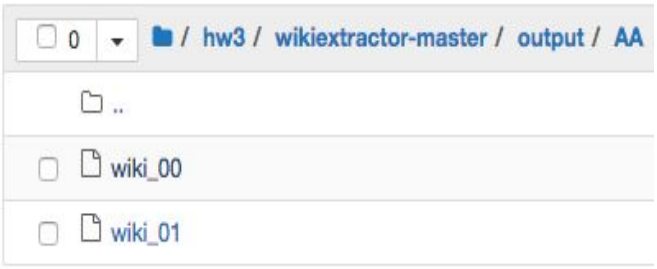
We select three wiki datasets(acewiki, elwikinews, hewikinews) from wiki dumps xml files. And we transform them from xml file into json file. In order to reduce the volume of the datasets, we just select the first json file to do clustering.

```
jb4076@big-data-analytics:~/hw3/wikiextractor-master$ python WikiExtractor.py
./acewiki.xml -o ./output --json

jb4076@big-data-analytics:~/hw3/wikiextractor-master$ python WikiExtractor.py
./elwikinews.xml -o ./output2 --json

jb4076@big-data-analytics:~/hw3/wikiextractor-master$ python WikiExtractor.py
./hewikinews.xml -o ./output3 --json
```

Then, we could get json files:



Then, we do clustering algorithms in jupyter notebook as shown in next page:

In [1]:

```
import findspark
findspark.init()

from pyspark import SQLContext
from pyspark import SparkContext
sc= SparkContext()
sqlContext = SQLContext(sc)
```

In [2]:

```
from pyspark.sql.functions import *
```

In [3]:

```
df_1 = sqlContext.read.json('/home/jb4076/hw3/wikiextractor-master/output/AA/wiki_**').\
       withColumn('label', lit(1)).sample(False, 0.16, 0)
df_2 = sqlContext.read.json('/home/jb4076/hw3/wikiextractor-master/output2/AA/wiki_**').\
       withColumn('label', lit(2)).sample(False, 0.4, 0)
df_3 = sqlContext.read.json('/home/jb4076/hw3/wikiextractor-master/output3/AA/wiki_**').\
       withColumn('label', lit(3)).sample(False, 0.5, 0)
```

In [4]:

```
print(df_1.count())
print(df_2.count())
print(df_3.count())
```

```
1204
1215
1212
```

Now, we sampled around 1200 rows of every data file and combine them into one dataset. And we could lable them with "0", "1", "2" in order to assess model later. We would assume that correct clustering should follow labels.

In [5]:

```
df_t = df_1.union(df_2)
df = df_t.union(df_3)
```

In [6]:

```
from pyspark.ml.feature import Tokenizer, RegexTokenizer
from pyspark.sql.functions import col, udf
from pyspark.sql.types import IntegerType
```

In [7]:

```
# Assume after importing the data, there's a column called "text" which contains the whole text fo
r each document.

regexTokenizer = RegexTokenizer(inputCol="text", outputCol="words", pattern="[^A-Za-z]+", toLowerca
se=True)
tokenized_data = regexTokenizer.transform(df)
```

In [8]:

```
# Removes stop words (e.g the, who, which, at, on, I).

from pyspark.ml.feature import StopWordsRemover

stopWordsRemover = StopWordsRemover(inputCol="words", outputCol="filtered_data")
filtered_data = stopWordsRemover.transform(tokenized_data)
```

In [9]:

```
# Convert a "bag of words" into a raw feature vector.

from pyspark.ml.feature import HashingTF, IDF

hashingTF = HashingTF(inputCol="filtered_data", outputCol="raw_features", numFeatures=10)
featurized_data = hashingTF.transform(filtered_data)

# HashingTF goes through each token, calculates a hash of it, and and increases the count of the r
esult.

# numFeatures controls the number of hash bins.
# TF(t) = (Number of times term t appears in a document) / (Total number of terms in the
document).
```

In [10]:

```
# Normalize the feature vector with the document frequencies.
# IDF down-weighs columns in the feature vector with high count across all the rows.

idf= IDF(inputCol="raw_features", outputCol="features")
idfModel = idf.fit(featurized_data)
featurizedData = idfModel.transform(featurized_data)

# IDF(t) = log_e(Total number of documents / Number of documents with term t in it).
```

In [11]:

```
featurizedData.select("features").show(10)
```

```
+--------------------+
|            features|
+--------------------+
|(10,[8,9],[1.8589...|
|(10,[1,2,5],[1.94...|
|(10,[8],[1.858944...|
|(10,[4],[2.023752...|
|(10,[0,6],[2.0426...|
|(10,[0],[2.042680...|
|(10,[9],[1.777543...|
|(10,[8],[1.858944...|
|(10,[1],[1.949495...|
|(10,[0,5],[2.0426...|
+--------------------+
only showing top 10 rows
```

K-Means Clustering

In [12]:

```
from pyspark.ml.clustering import KMeans

# Read data into Dataframe.
# Fitting:

kmeans = KMeans().setK(3).setSeed(789)
model = kmeans.fit(featurizedData)
```

In [13]:

```
# Evaluation:

wssse = model.computeCost(featurizedData)
print("Within Set Sum of Squared Errors = " + str(wssse))
```

```
Within Set Sum of Squared Errors = 30232.461917520246
```

In [14]:

```
# Show the results

centers = model.clusterCenters()
print("Cluster Centers: ")
for center in centers:
    print(center)
```

```
Cluster Centers:
[1.13590594 1.17248916 2.03441557 1.12392506 1.18333985 1.79036782
 1.15402176 0.95157015 0.63000034 2.1296587 ]
[0.16476638 0.20697853 0.20293754 0.24382369 0.18765579 0.18745976
 0.18801366 0.10237735 0.13648922 0.12377244]
[0.82758123 0.54535736 0.61620969 0.52298604 0.44899978 0.365846
 0.87067417 2.08001033 2.27735664 0.58870422]
```

In [15]:

```
model_data = model.transform(featurizedData)
model_data.groupBy('prediction').count().show()
predict_error = model_data.withColumn('error', when(col('label')==col('prediction'),0).\
                    otherwise(1)).select('id','label','prediction','error')
predict_error.show(10)
```

```
+----------+-----+
|prediction|count|
+----------+-----+
|         1| 2901|
|         2|  311|
|         0|  419|
+----------+-----+

+---+-----+----------+-----+
| id|label|prediction|error|
+---+-----+----------+-----+
| 27|    1|         1|    0|
| 37|    1|         1|    0|
| 39|    1|         1|    0|
| 42|    1|         1|    0|
| 48|    1|         1|    0|
| 56|    1|         1|    0|
| 61|    1|         1|    0|
| 67|    1|         1|    0|
| 70|    1|         1|    0|
| 77|    1|         1|    0|
+---+-----+----------+-----+
only showing top 10 rows
```

In [16]:

```
print(predict_error.groupBy().sum().collect())
error_sum = predict_error.groupBy().sum().collect()[0][2]
print('Numbers of Incorrect Prediction is:', error_sum)
error_rate = error_sum/predict_error.count()
print('Prediction Error Rate is:', error_rate)
```

```
[Row(sum(label)=7270, sum(prediction)=3523, sum(error)=2828)]
Numbers of Incorrect Prediction is: 2828
Prediction Error Rate is: 0.7788488019829248
```

Bisecting k-means

In [17]:

```
from pyspark.ml.clustering import BisectingKMeans


# Trains a bisecting k-means model.
bkm = BisectingKMeans().setK(3).setSeed(328752)
```

```
model = bkm.fit(featurizedData)

# Evaluate clustering.
cost = model.computeCost(featurizedData)
print("Within Set Sum of Squared Errors = " + str(cost))

# Shows the result.
print("Cluster Centers: ")
centers = model.clusterCenters()
for center in centers:
    print(center)
```

```
Within Set Sum of Squared Errors = 30482.386216932762
Cluster Centers:
[0.05365567 0.03413864 0.1052702  0.         0.04383243 0.11908354
 0.04278283 0.08155644 0.09337556 0.11877597]
[0.47622259 0.57908102 0.45177432 0.84379446 0.53886905 0.36058451
 0.54476383 0.25999549 0.65778042 0.14304495]
[1.12413756 1.13298623 1.66006091 1.01931732 1.02173226 1.37119772
 1.21197764 1.53557489 0.98982768 1.7140601 ]
```

In [18]:

```
model_data = model.transform(featurizedData)
model_data.groupBy('prediction').count().show()
predict_error = model_data.withColumn('error', when(col('label')==col('prediction'),0).\
                      otherwise(1)).select('id','label','prediction','error')
predict_error.show(10)
```

```
+----------+-----+
|prediction|count|
+----------+-----+
|         1|  845|
|         2|  616|
|         0| 2170|
+----------+-----+

+---+-----+----------+-----+
| id|label|prediction|error|
+---+-----+----------+-----+
| 27|    1|         0|    1|
| 37|    1|         1|    0|
| 39|    1|         0|    1|
| 42|    1|         0|    1|
| 48|    1|         1|    0|
| 56|    1|         0|    1|
| 61|    1|         0|    1|
| 67|    1|         0|    1|
| 70|    1|         0|    1|
| 77|    1|         1|    0|
+---+-----+----------+-----+
only showing top 10 rows
```

In [19]:

```
print(predict_error.groupBy().sum().collect())
error_sum = predict_error.groupBy().sum().collect()[0][2]
print('Numbers of Incorrect Prediction is:', error_sum)
error_rate = error_sum/predict_error.count()
print('Prediction Error Rate is:', error_rate)
```

```
[Row(sum(label)=7270, sum(prediction)=2077, sum(error)=3003)]
Numbers of Incorrect Prediction is: 3003
Prediction Error Rate is: 0.8270448912145415
```

Latent Dirichlet allocation (LDA)

In [20]:

```
from pyspark.ml.clustering import LDA
```

```
# Trains a LDA model.
lda = LDA(k=3, maxIter=10)
model = lda.fit(featurizedData)

ll = model.logLikelihood(featurizedData)
lp = model.logPerplexity(featurizedData)
print("The lower bound on the log likelihood of the entire corpus: " + str(ll))
print("The upper bound on perplexity: " + str(lp))

# Describe topics.
topics = model.describeTopics(3)
print("The topics described by their top-weighted terms:")
topics.show(truncate=False)

# Shows the result
transformed = model.transform(featurizedData)
transformed.select('id','label','topicDistribution').show(truncate=False)
```

```
The lower bound on the log likelihood of the entire corpus: -30972.799207109038
The upper bound on perplexity: 2.2978687377582596
The topics described by their top-weighted terms:
+-----+-----------+---------------------------------------------------------------+
|topic|termIndices|termWeights                                                    |
+-----+-----------+---------------------------------------------------------------+
|0    |[0, 3, 9]  |[0.23337047102015182, 0.222418092836805, 0.19955316584566288]  |
|1    |[5, 2, 1]  |[0.22860398301359264, 0.22635468630547154, 0.19317014639715105]|
|2    |[8, 7, 2]  |[0.3946564366117858, 0.36004880246877574, 0.08128529154864185] |
+-----+-----------+---------------------------------------------------------------+

+---+-----+-----------------------------------------------------------+
|id |label|topicDistribution                                          |
+---+-----+-----------------------------------------------------------+
|27 |1    |[0.43740991600206414,0.08343877196727323,0.4791513120306627]|
|37 |1    |[0.05253885737737014,0.8962610749419417,0.05120006768068804]|
|39 |1    |[0.11705158969101756,0.12106762384360671,0.7618807864653758]|
|42 |1    |[0.7737859180793071,0.11927782394759501,0.10693625797309787]|
|48 |1    |[0.8530674166576546,0.07970876115326096,0.06722382218908447]|
|56 |1    |[0.7777479530619504,0.11515032235727224,0.10710172458077741]|
|61 |1    |[0.7452871079988144,0.1384409398187654,0.11627195218242016] |
|67 |1    |[0.11705205941961519,0.12107077485723539,0.7618771657231495]|
|70 |1    |[0.11393176587648002,0.774846878743691,0.11122135537982904] |
|77 |1    |[0.4679411116814213,0.46646516978835567,0.06559371853022312]|
|91 |1    |[0.7777480129736791,0.11515026916420847,0.1071017178621124] |
|103|1    |[0.11393174492076402,0.7748479748105846,0.11122028026865138]|
|109|1    |[0.07389356956129829,0.8582174548121965,0.0678889756265052] |
|111|1    |[0.07070511150111634,0.4803215906435651,0.4489732978553186] |
|118|1    |[0.11705220767926268,0.12107173285437209,0.7618760594663653]|
|125|1    |[0.76488259542826,0.12110579860385391,0.11401160596788609]  |
|130|1    |[0.7648797278758998,0.1211059670017336,0.1140143051223668]  |
|133|1    |[0.11037525409853526,0.11414306535460805,0.7754816805468567]|
|144|1    |[0.8504394573166216,0.08237985164358891,0.06718069103978952]|
|165|1    |[0.11536587917495554,0.7747002492247366,0.1099338716003079] |
+---+-----+-----------------------------------------------------------+
only showing top 20 rows
```

Gaussian Mixture Model (GMM)

In [21]:

```
from pyspark.ml.clustering import GaussianMixture

gmm = GaussianMixture().setK(3).setSeed(562546)
model = gmm.fit(featurizedData)

print("Gaussians shown as a DataFrame: ")
model.gaussiansDF.show(truncate=False)
```

```
Gaussians shown as a DataFrame:
+-----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
-+-----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
```

```
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
-----------------+
|mean
|cov
|
+----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
-+----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
-----------------+
|
[0.6443357353880437,0.8633835436482681,0.5810253689057424,0.7919558256743872,0.733988738869104,0.34
72254693156,0.7670840226266286,0.2927705229281981,0.30386310914971326,0.4697093912161421]
|1.3630004414177601      -0.0061458756146789756  ... (10 total)
-0.0061458756146789756  1.7597626120939909      ...
0.011918689599287435    0.18876684389648432     ...
-0.16084516624049378    -0.13069799902972942    ...
-0.042574647254994324   0.015728561799562783    ...
0.05507072788185395     0.05797870108531074     ...
0.020951174692179567    0.08558501270286398     ...
0.05593361519467195     0.06752001376097545     ...
0.1377042439196595      0.09624165474772647     ...
0.017822115463077787    0.1617627703490934      ...|
|
[0.9511643878843243,0.501707722741529,1.596669500031005,0.8786695168824157,0.6616621696200714,1.737
78352708,0.8327237733153777,1.8812209318674067,1.2160561648521082,1.6730402769183705]
|2.909407121352349    0.450881281108931      ... (10 total)
0.450881281108931    1.157204070953002      ...
0.04269131688441191  0.17261666268140793    ...
0.2673949197719086   0.23044530171456098    ...
0.33655194338796346  0.13448418248388694    ...
-0.2701952976748173  -0.21291461766789957   ...
0.49620439187423737  0.34174785343388975    ...
0.34238462402056175  -0.004779229696471566  ...
0.18694917248573067  0.2203153217579023     ...
0.12671974437060848  0.25190702316609515    ...                                          |
|
[0.017423853042825885,0.012401223077563596,0.12273877380920128,0.011362349201335404,0.012621572686
6,0.11558096954590599,0.015479900383203635,0.08494707987057123,0.23629542118482072,0.07265536141569
|0.14834222543887227   0.05315650766979548   ... (10 total)
0.05315650766979548   0.07248455428836922   ...
0.048692795763230726  0.04300511073530505   ...
0.05742532994971028   0.04282921472950737   ...
0.04861728145300933   0.026000074838424477  ...
0.052776753037960394  0.03668262257529149   ...
0.05509261240125136   0.04677006509070215   ...
0.05879858052278138   0.04005280608670427   ...
0.07526871559336595   0.047167065379905594  ...
0.038659000728421995  0.04546156793397379   ...                                          |
+----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
-+----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
-----------------+
```

In [22]:

```
model_data = model.transform(featurizedData)
model_data.groupBy('prediction').count().show()
predict_error = model_data.withColumn('error', when(col('label')==col('prediction'),0).\
                    otherwise(1)).select('id','label','prediction','error')
predict_error.show(10)
```

```
+----------+-----+
|prediction|count|
+----------+-----+
```

```
|          1|  422|
|          2| 2013|
|          0| 1196|
+-----------+-----+

+---+-----+----------+-----+
| id|label|prediction|error|
+---+-----+----------+-----+
| 27|    1|         2|    1|
| 37|    1|         0|    1|
| 39|    1|         2|    1|
| 42|    1|         0|    1|
| 48|    1|         0|    1|
| 56|    1|         0|    1|
| 61|    1|         2|    1|
| 67|    1|         2|    1|
| 70|    1|         0|    1|
| 77|    1|         0|    1|
+---+-----+----------+-----+
only showing top 10 rows
```

In [23]:

```
print(predict_error.groupBy().sum().collect())
error_sum = predict_error.groupBy().sum().collect()[0][2]
print('Numbers of Incorrect Prediction is:', error_sum)
error_rate = error_sum/predict_error.count()
print('Prediction Error Rate is:', error_rate)
```

```
[Row(sum(label)=7270, sum(prediction)=4448, sum(error)=2502)]
Numbers of Incorrect Prediction is: 2502
Prediction Error Rate is: 0.6890663729000276
```

We could see that both K-means clustering and Hierachical K-means have large Within Set Sums of Square error and large predition error. I think this is because that assuming that text data features' means are more similar if they come from the same wiki file.

And we could see that GMM models are more reliable, because the prediction error rate is reduced significantly in this model.

One interesting results are from the LDA models. The model supports a variety of other methods for the analysis of documents with respect to the trained set of topics. For instance, it enables the calculation of the top documents for each topic or it can return the top k weighted topics for a document. We can also see the distribution of topics for each document.

Maybe we could try another mothod to generate data to get better clustering results.

In [24]:

```
# word2Vec = Word2Vec(VectorSize = 20, minCount = 0, inputCol = "filtered_data", outputCol = "raw_
features")
# model = word2Vec.fit(filtered_data)
# featurizedData = model.transform(filtered_data)
```