# DS222: Assignment 1
# Naive Bayes using Map Reduce

**Souvik Karmakar**
Systems Engineering
Sr. No.: 14700
`email: souvikk@iisc.ac.in`

## 1 Introduction

In this assignment we implement Naive Bayes algorithm on the DBPedia data set, in two frameworks:

- Local Naive Bayes : where the entire data set is loaded into memory to be be further processed for training and testing purposes.
- MapReduce[1] Framework: here the data set in loaded onto to the Turing cluster in HDFS file system, to be further processed in hadoop distributed framework.

All code implementations where done in python 3.5 environment.

### 1.1 Dataset

We use DBPedia data set for testing the two algorithms. The *full* data set consists of 50 labels and has 2,14,997 train documents, 61,497 validation documents and 29,997 test documents. Each document may belong to more than one class. A smaller data set with 49 labels is also provided. It has 10,497 documents in train file, 2997 in validation file and 1497 in test file.

## 2 Literature

Naive Bayes Classifier is a generative model that uses Bayes theorem for classification. The name naive is used because of the strong assumption that the individual features are independent of each other. Given a set of features $v = (x_1, ...., x_n)$ and the underlying training data we try to estimate the individual class posterior probabilities

$$p(C_k|x_1, ...., x_n)$$

In case of large set of features, estimating the conditional probability becomes infeasible. This is where the Bayes theorem and strong assumption of independence comes into play. The final form of Naive Bayes is derived as follows:

$$p(C_k|v) = \frac{p(C_k)p(s|C_k)}{p(v)}$$

Thus, the above equation can be written as,

$$posterior = \frac{prior * likelihood}{evidence}$$

$$\begin{aligned} p(C_k|x_1, .., x_n) &= p(x_1|x_2, .., x_n, C_k)p(x_2, ..x_n, C_k) \\ &= p(C_k)p(x_1|C_k)p(x_2|C_k)...p(x_n|C_k) \\ &= p(C_k)\prod_{i=1}^{N} p(x_i|C_k) \end{aligned}$$

Substituting the above into the original Naive Bayes equation gives the final form as follows:

$$p(C_k|x_1, ..., x_n) = \frac{1}{p(v)}p(C_k)\prod_{i=1}^{N} p(x_i|C_k)$$

where the evidence $p(v) = \sum_k pC_k)p(v|C_k)$ is a scaling factor and can be treated as constant[2].

### 2.1 Naive Bayes for Document Classification

For document classification we use individual unique words as the independent features. In this assignment we have used multinomial naive Bayes together with Laplace smoothing. Laplace smoothing is used to mitigate the problem of zero probability, encountered when a particular word in document is not present in the given class vocabulary.

### 2.2 Experiments

#### 2.2.1 Local Naive Bayes

In the local Naive Bayes implementation we do the training as follows: We prepare 50 dictionaries for the 50 classes, to keep track of the unique words and there individual counts that have appeared in the individual documents. We also keep count of the no of times that every document has appeared and also the total document count. This is used for prior probability calculation.

**Testing Naive Bayes**

1. For every document do the following:
2. Extract words separated by spaces, and remove stop words
3. Calculate likelihood of individual classes using the formula:
4. $$\hat{P}(w|c) = \frac{Freq_{cw} + 1}{(\sum_{w' \in V} Freq_{cw'}) + B'}$$
5. Above is probability of individual words given a class, which is used in the likelihood formula. We use Laplace smoothing to account for words that don't appear in a particular class vocabulary. $B'$ is the size of vocabulary.

## 2.3 Naive Bayes on Hadoop MapReduce framework

### 2.3.1 Train

For training we use a single mapper and reducer. The codes are written in python. We also use a custom jar file prepared from a java class **CustomMultiOutputFormat**. The class inherits hadoops **MultipleTextOutputFormat** class. The functions are overriden such that the output from reducer is directed to different folders according to the key supplied. The use will become much clear one the reducer is explained.

**Mapper**
The mapper takes a single line as input and splits the document to extract class and individual words. The words are printed in the following format:
*(class, word, 1)*
Once the program finishes printing the words from a particular document, the class is printed in the following format:
*(doc_count_unique, class, 1)*
here **doc_count_unique** is a special tag and is used in the reducer to keep count of individual classes.

**Reducer**
The reducer program after aggregation of class and word counts prints 3 types of lines:
*('class_word_count', current_class, current_word, word_count)* which maps unique word count in individual classes.
*('doc_count', current_doc, current_doc_count)* which is mapped from the 'doc_count_unique' tag to count the individual document count.
*('class_total_word_count', c, class_word_count)* which prints the total no of words in each class.
Three separate folders *'class_word_count', 'doc_count', 'class_total_word_count'* are created by the java class with their respective counts in part file. The java class also removes these tags from the final output to the file.

### 2.3.2 Test

In this section we use a single mapper and reducer to the required job.

**Mapper**
This program works similar to the first mapper but only prints a single type of line :
*(document_no, class, current_word, word_count)*
we keep track of the individual document number to calculate the accuracy.

**Reducer**
In the reducer, we load the three files from the training output and convert them into 3 dictionaries with following (key, value) pairs:

- (class, word): word count, used for individual word probabilities
- class: document count, used for prior probability
- class : total word count, used for individual word probabilities

We calculate the probabilities of individual document and store them in a results dictionary. The procedure is as follows:

- for every input line *(document_no, class, current_word, word_count)*, store in results, calculate the word probability for the 50 classes and store it in dictionary which is stored in the results dictionary. The results dictionary uses (document_no, class) as the key
- for upcoming lines keep updating the results dictionary,

- once input is finished calculate the find the max probability class and calculate accuracy.

## 3 Observation and Results

In this section we report the various observations and results of out experiments.

### 3.1 Time

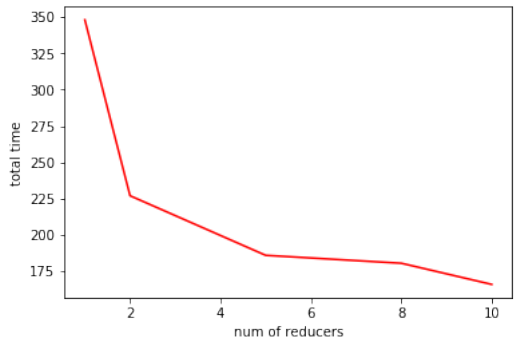

Figure 1: Total Time vs number of reducers

Initially we observe a huge fall in process time with increase in reducers, followed by a slow decrease in process time. Hence the decrease is non-linear with the number of reducers.

### 3.2 Accuracy

In this section we report the accuracy of different methods and experiments.

Table 1: Accuracy

| Experiment | Dev Acc(%) | Test Acc(%) |
|---|---|---|
| Local NB(very small) | 85 | 84 |
| Local NB'(full) | 80.2 | 79.7 |
| MR single Node (very small) | 81.4 | 80.5 |
| MR single Node (full) | 78.5 | 78.2 |

We also observe a decrease in accuracy with increase an in number of reducers as shown

## 4 Acknowledgements

I would like to thank Sai Suman for his help in running the programs on cluster. I would also like to thank my batch mates Rishi, Karan, Chaikesh, Shubham, for their help in solving this assignment.

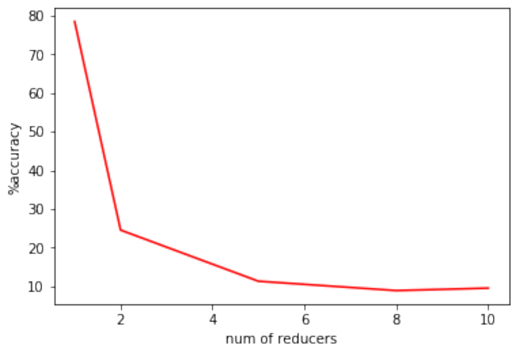

Figure 2: Accuracy vs Number of reducers

## References

[1] Dean, Jeffrey and Ghemawat, Sanjay, "MapReduce: Simplified Data Processing on Large Clusters" Commun. ACM, January 2008.

[2] https://en.wikipedia.org/wiki/Naive_Bayes_classifier