DS222: MLLD 2018
Assignment 1 : Naive Bayes classifier implementation

Sourabh Balgi
Sr. No. - 14318
EE Department
M. Tech. (Systems Engineering)
sourabhbalgi@gmail.com

Abstract

Naive Bayes Classification is one of the simplest, yet effective and most commonly-used machine learning classifier. Since the variables are considered independent, it's easier for performing computations in case of large number of variables. This document summarize the results and observations of dealing with very large datasets DBPedia and distributed computing using HADOOP mapreduce. As a part of the assignment, we implement two variants of Naive-Bayes classifier one in-memory and the other using HADOOP mapreduce.

1 Introduction

Naive Bayes classifier is a defacto method to go for when a large number of data is not available. It's one of the easiest classifiers as it makes no assumptions on the interdependence of the features in the corpus. Considering the availability of the huge corpus of data now-a-days, It's much required to go for simple methods which can be implemented without much feature engineering and usage of massive computational resources. With large data, in memory computation becomes much harder and we need to explore distributed computing. One such tools we are exploring in this assignment is HADOOP.

1.1 Dataset

We use DBPedia 2015 dataset which consists of 50 classes based on the labels obtained from mapping based type of the documents.

The columns are separated by space-tab and the list of labels are comma separated. Each line is a unique document with labels and document data.

2 Naive Bayes classifier

Naive Bayes involves maximizing the posterior probability assuming Independence of the conditional density. The given problem instance to be classified, represented by a vector $s = (w_1, ..., w_n)$ representing some n words w_i in a sentence (independent variables), it assigns to this instance probabilities

$$p(L_m|w_1, ..., w_n)$$

for each of m possible outcomes or labels L_m.

The problem with the above formulation is that if the number of features n is large or if a feature can take on a large number of values, then basing such a model on probability tables is infeasible. We therefore reformulate the model to make it more tractable. Using **Bayes' theorem**, the conditional probability can be decomposed as,

$$p(L_m|s) = \frac{p(L_m)p(s|L_m)}{p(s)}$$

Thus, the above equation can be written as,

$$posterior = \frac{prior * conditional}{\Sigma prior * conditional}$$

2.1 Naive Bayes Algorithm

To eliminate zeros, we use add-one or Laplace smoothing, which simply adds one to each count

$$\hat{P}(w|class) = \frac{C_w + 1}{\sum_{w' \in V}(C_{w'} + 1)} = \frac{C_w + 1}{(\sum_{w' \in V} T_{w'}) + |V|}$$

where *C* indicates actual count of word *w*, *V* is set of vocabulary and —*V*— is the size of vocabulary

Algorithm 1: Naive Bayes Classifier : Training

```
1 Pre-process the training data using stopword
    removal, lemmatizer and tokenizer.
2 Group the documents belonging to the same
    class and count the number of documents for
    each class to obtain the prior probability
    using
        $prior = \frac{class\_count}{total\_class\_count}$   (1)
3 Find the conditional density of each class
    with add-one smoothing for numerical
    stability and unknown vocabulary handling.
        $cond_{w,m} = \frac{wordcount_m + 1}{totalword_m + vocabsize}$   (2)
```

Algorithm 2: Naive Bayes Classifier : Inference

```
1 Pre-process the training data using stopword
    removal, lemmatizer and tokenizer.
2 For each document, find the posterior
    densities for all the classes using Bayes rule.
        $p(L_m|s) = \frac{p(L_m)p(s|L_m)}{p(s)}$   (3)
3 Consider the class with the maximum
    posterior probability as the predicted class.
4 Find and report accuracy using
        $Accuracy = \frac{total\_correct\_prediction}{total\_number\_of\_documents}$   (4)
```

2.2 Naive Bayes on Hadoop MapReduce framework

We implement the mapreduce using Hadoop streaming. Hadoop streaming is a utility that comes with the Hadoop distribution. This utility allows us to create and run Map/Reduce jobs with any executable or script as the mapper and/or the reducer. In this assignment, both the mapper and the reducer are python scripts that read the input from standard input and emit the output to standard output. We control the number of reducers used by the mapreduce per job to observe the run-times

3 Hadoop Approach

3.1 Map Reduce model description

3.1.1 Training Phase :

1. **Get Model Parameters** : In this job, we get the counts of individual classes and counts of individual words present in the training corpus in map phase. The reducer phase sums these individual counts and emits the total used by the next Job to find the prior and conditional densities. This first mapreduce can be executed using different number of reducer tasks as the counting of words and labels can be done in a distributed and parallel way.
   We then use a final mapreduce to combine the outputs from the multiple reducers of the previous job to get prior probabilities of each class and class conditional densities of words in each class. These constitutes the model parameters used by the inference phase mapreduce jobs.

3.1.2 Inference Phase :

This involves 3 separate mapreduce phases.

1. **Get document word count** : We consider each document to be associated with a unique document id i.e. line number as the key and the words in this document in the mapper phase. In the reduce phase, we sum all the key and emit the key - (doc id, word) : value - (word count) This phase is used to get only the word counts in each documents and hence can be paralleled to have multiple reducer tasks.

2. **Calculate posterior probability** : We use the outputs from **Get model parameters** Job i.e. probabilities (prior and conditional) and Get document word count Job i.e. word count in the test document.

3. **Calculate accuracy** : We use the outputs from **Calculate posterior probability** Job. We sum the prior and conditional probabilities according to documents and find the label corresponding to the maximum posterior probability and emit as predicted label for the document.

4 Observations and Results

- **Total number of documents in full dataset** : Train - 214997, Devel - 61497 and Test - 29997

- **Vocab Size of full dataset for different types of lemmatizers** :

  1. No Stemming (only stopword removal) : 260294
  2. Porter Stemmer + stopword removal : 220836
  3. Snowball Stemmer + stopword removal : 221040
  4. Lancaster Stemmer + stopword removal : 187163

  Inference : It was observed that without lemmatizer, the vocab size was relatively bigger. Lancaster stemmer was observed to be most aggresive form of lemmatizer reducing the vocab to the lowest. However both these method did not provide better results in terms of accuracy. The train, devel and test accuracies were much better for Porter stemmer and Snowball stemmer. Since both these provided almost same accuracies, we choose Porter stemmer as the best method for further experimentation as it has a relatively lower vocab size compared to other methods, thus reducing the model parameters and execution time.

- **Runtime for full dataset with different lemmatizers** :
  Inference : Almost all the methods required similar times. The Lancaster model had the least number of model params as the vocab size is the smallest. But considering both model accuracy and runtime, Porter stemmer was chosen for further investigation in Mapreduce phase.

Table 1: **Runtime for full dataset with different Lemmatizers**

Lemmatizer (#Model params	**Train** Model params	**Train** Inference	**Devel** Inference	**Test** Inference
No Stemming (1076964)	1 min 45 sec	19 min 3 sec	4 min 22 sec	2 min 4 sec
Porter Stemmer (877270)	5 min 31 sec	19 min 14 sec	5 min 24 sec	2 min 18 sec
Snowball Stemmer (875119)	4 min 14 sec	19 min 40 sec	5 min 18 sec	2 min 20 sec
Lancaster Stemmer (757303)	4 min 46 sec	19 min 42 sec	5 min 26 sec	2 min 13 sec

- **In-Memory Accuracy** :
  Model accuracy for In-Memory implementation for different configurations on full dataset.

Table 2: **In-Memory Accuracy**

Lemmatizer	**Train %**	**Devel %**	**Test %**
No Stemming	85.6	72.4	70.7
Porter Stemmer	85.03	73.04	73.42
Snowball Stemmer	85.04	73.09	73.38
Lancaster Stemmer	84.4	72.45	73.54

- **Hadoop Mapreduce implementation** :

  1. **Vocab Size** : 220836 (Same as In-memory Porter stemmer implementation as Porter stemmer is used here too.)
  2. **Model parameter** : 877270 (Same as In-memory Porter stemmer implementation as Porter stemmer is used here too). Includes both class prior probabilities and conditional densities of words per each class including unknown(UNK) word.

- **Hadoop Mapreduce Accuracy** :
  Model accuracy for Hadoop Mapreduce implementation for different configurations on full dataset.

Table 3: Hadoop Mapreduce Accuracy

Train %	**Devel %**	**Test %**
43.6	30.03	28.02

- **Mapreduce run-time for training with different number of Reducer tasks** :
Run-time observations from the Job log for different number of reducer task for model training. ***It was observed with that the reducer started in parallel even before the mapper had completed 100% task, typically at 83% map phase. The reducers started the task when the mapper was just in 83% stage***. Thus it was experimentally observed and confirmed that the mapper and reducer phases can be run in parallel for better execution times.

Table 4: **Mapreduce run-time for training with different # of Reducer tasks** (only reducer time considered)

Number of reducers tasks (R)	**Time for initial reducers**	**Time for final reducers**	**Total time to get model params**
1	1 min 2 sec	0 min 47 sec	1 min 49 sec
2	0 min 37 sec	0 min 46 sec	1 min 23 sec
5	0 min 19 sec	0 min 39 sec	0 min 58 sec
8	0 min 17 sec	0 min 40 sec	0 min 57 sec
10	**0 min 12 sec**	**0 min 42 sec**	**0 min 54 sec**

- **Hadoop Mapreduce observations** :
  1. Mapper takes almost same time with any number of reducers.
  2. Time taken reduces as the number of reducers increase.
  3. Each reducer file creates separate output file with prefix **'part-'** in the output folder.
  4. The successful completion of the Job is indicated by **'_SUCCESS'** file in the output folder on hdfs.
  5. In case a job fails, the Tasker reassigns the job with multiple attempts to successfully complete the job. This robustness to failure was also observed as some of the jobs failed near completion were restarted.
  6. The number of reducers has to be chosen according to our input file size, our algorithm for parallel computation and execution time.

- **Project Github link** :
DS222_Assignment1

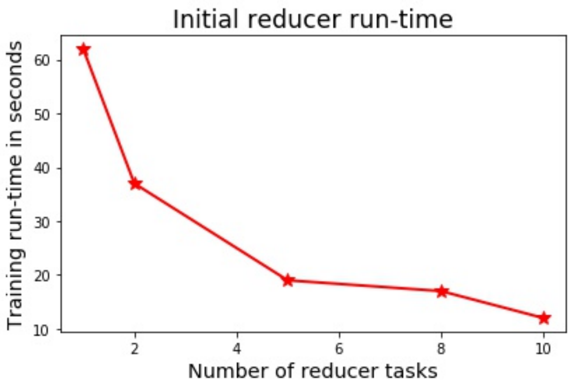

Figure 1: **Initial reducer run-time vs Number of reducers**

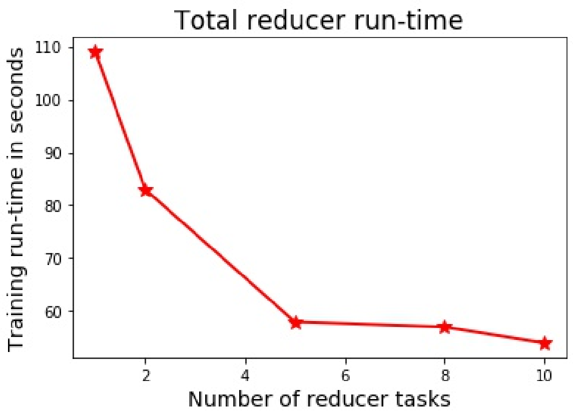

Figure 2: **Total reducer run-time (Initial+Final) vs Number of reducers**

References

[1] Chapter 4, Speech and Language Processing, Daniel Jurafsky James H. Martin, Draft of August 1, 2018.