DS222 Assignment 1
Naive Bayes Classifier : Local & Mapreduce Implementation

Nilesh Agrawal (SR No.: 15007) [1]

Abstract

As a part of this assignment, we implemented Naive Bayes Classifier for classification of DBPedia documents into one of movies classes, in local Java and also the Hadoop Map Reduce framework, and analysed for its scalability characteristics. Naive Bayes classifiers are highly scalable, requiring a number of parameters linear in the number of variables (features/predictors) in a learning problem. Maximum-likelihood training can be done by evaluating a closed-form expression, which takes linear time, rather than by expensive iterative approximation as used for many other types of classifiers.

1. Introduction

In machine learning, naive Bayes classifiers are a family of simple "probabilistic classifiers" based on applying Bayes' theorem with strong (naive) independence assumptions between the features. Naive Bayes model is easy to build and particularly useful for very large data sets. Along with simplicity, Naive Bayes is known to outperform even highly sophisticated classification methods.

1.1. Naive Bayes Formulation

Bayes theorem provides a way of calculating posterior probability $P(y|x)$ from $P(y)$, $P(x)$ and $P(x|y)$. Look at the equation below:

$$\mathsf{P}(Y|X) = \mathsf{P}(X|Y)\mathsf{P}(Y)$$

where,

- $\mathsf{P}(Y|X)$ = given test example X, what is its probability of belonging to class Y. This is also known as posterior probability. This is conditional probability that is to be found for the given test example X for each of the given training classes.
- $\mathsf{P}(X|Y)$ = given class Y, what is the probability of example X belonging to class Y. This is also known as likelihood as it implies how much likely does example X belongs to class Y. This is conditional probability too as we are finding probability of X out of total instances of class Y only i.e we have restricted/conditioned our search space to class Y while finding the probability of X. We calculate this probability using the counts of words that are determined during the training phase.

Now let us assume we have to predict what is the probability of test instance $x_i = x_i^1 x_i^2 ... x_i^d$ for class label y_j then

$$\begin{aligned}\mathsf{P}(Y = y_j|X = x_i) &= \mathsf{P}(x_i^1|y_j).\mathsf{P}(x_i^2|y_j)....\mathsf{P}(x_i^d|y_j) \\ &= \prod_{k=1}^{d} \mathsf{P}(x_i^k|y_j)\end{aligned}$$

where $\mathsf{P}(x_i^k|y_j)$ =

$$\frac{counts\ of\ word\ "k"\ in\ class\ j\ +\ 1}{counts\ of\ word\ in\ class\ j\ +|\mathsf{V}|}$$

Here **j**, represents a class and **k** represents a feature.

2. Implementation

2.1. Local Naive Bayes Implementation

We have used Java as a programming language to implement local Naive Bayes and used it to classify DBPedia documents. In this approach we have counted occurence/frequency of every unique label-feature pair in training set and stored it in Hashtable[1]. And then we have used this training frequency count to predict for each testing instance[2], what is the probability it belongs to any of the class labels.

$$Y_{New} \leftarrow \arg\max_{y_j} \mathsf{P}(Y = y_j) \prod_{k=1}^{d} \mathsf{P}(x_i^k|y_j)$$

[1]Department of Computer science and Automation, Indian Institute of Science, Bangalore. Correspondence to: Nilesh Agrawal <anilesh@iisc.ac.in>.

Algorithm 1 Local Naive Bayes: Training

Input: $(x_i, y_i)pairs$
for each example i, x_i, y_i in train **do**
  $C(Y = ANY) + +\ C(Y = y_i) + +$
  for each word $x_d \in x_i^d$ **do**
    $C(Y = y_i \wedge X = x_d") + +$
    $C(Y = y_i \wedge X = ANY") + +$
  end for
end for

Algorithm 2 Local Naive Bayes: Testing

Input: $(x_i, y_i)pairs$
for each example i, x_i, y_i in test **do**
  for each class y_j in Y **do**
    Compute $\mathsf{P}(y_j|x_i) = \mathsf{P}(Y = y_j) \prod_{k=1}^{d} \mathsf{P}(x_i^k|y_j)$
  end for
  $Y_{New} \leftarrow \arg\max_{y_j} \mathsf{P}(y_j|x_i)$
end for

2.2. MapReduce Naive Bayes Implementation

We have used Java Mapreduce framework to develop scalable Naive Bayes classifier. In this we have written 1 mapreduce jobs for training [3], and 3 mapreduce jobs for testing and classifying documents, the third job is just to calculate accuracy, so testing can be done in 2 jobs.

Algorithm 3 Mapreduce Naive Bayes: Training

Input: $(x_i, y_i)pairs$
Function map(key, value)
  X, Y = value.split(\t)
  for each class y_j in Y **do**
    emit(Y@ANY, 1), emit(Y@y_j, 1)
    for each word x_i in X **do**
      emit(X@ANY@Y@y_j, 1)
      emit(X@x_j@Y@y_j, 1)
      emit(XVOCAB@x_j, 1)
    end for
  end for
EndFunction
Function reduce(key, value[])
  keys[] = key.split("@")
  for each val in value **do**
    sum += val
  end for
  if keys[0]=XVOCAB **then**
    X_{vocab}++
  else
    emit(key,sum)
  end if
EndFunction

3. Analysis

In this section, we report accuracy for train,development and test set for both implementations, number of parameters in model, wall clock timings for both implementations and scalability analysis with different number of reducers.

3.1. Accuracy

Below table shows[3.3], accuracy for all three sets in local as well as Mapreduce implementation. It is observed that both local as well as mapreduce implementations gives same accuracy for all 3 sets.

	Train	**Development**	**Test**
Local	83.32%	70.06%	74.24%
Mapreduce	83.32%	70.06%	74.24%

Table 1. Accuracy for all 3 Datasets for both Implementations

3.2. Parameters in model

No. of paramters in Naive Bayes model is given by,

$$|V| * (K - 1) + (K - 1) \tag{1}$$

where, $|V|$: Vocabulary size and
K : No. of Classes
In our model $|V|$ = 484127 and K = 50. So total number of parameters in our model is $484127 * 49 + 49 =$ **23722272**.

3.3. Running Time

Below table shows[3.3], running time for training and testing on local as well as Mapreduce implementation.

	Training	**Testing**
Local	2.99s	14.26s
Mapreduce(R=1)	3m52s	8m42s
Mapreduce(R=2)	3m25s	5m26s
Mapreduce(R=5)	3m8s	5m1s
Mapreduce(R=8)	3m6s	5m7s
Mapreduce(R=10)	2m59s	4m58s

Table 2. Running Time for Training & Testing for both Implementations

As we can see running time decreases with increasing number of reducers, this is because more reduce tasks are spinned up and reducer work is done parallely across these tasks thus exhibiting **weak scalability**. It is clearly evident from above table that local implementation runs fast as all the computations can be done in memory, while for Mapreduce implementation the shuffle cost between mapper and

reducer takes a lot of time. But if the dataset is large, local implementation will eventually fail as it runs on single machine. But the running time doesn't decreases linearly with increasing number of reducers as more reducers incur more shuffling cost.

3.4. Scalability Analysis

The result of weak scaling by varying the number of reducers for training and testing is shown in Figure 1 and Figure 2 respectively. We can see from below figures, the running time doesn't decreases linearly with number of reducers, but it decreases as number of reducers grow and then it becomes flatten. This is due to increase in message passing because of shuffle phase as number of reducer grows, and thus spending more time in message transfer than actual compute.

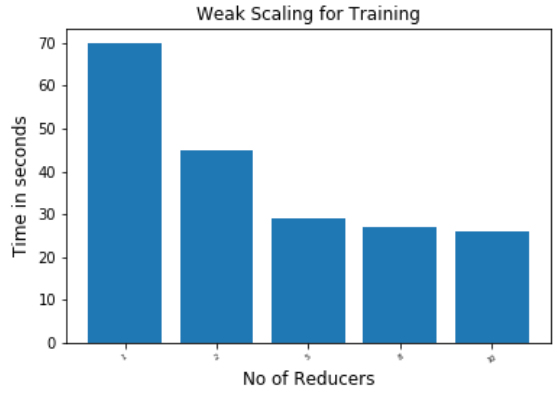

Figure 1. Weak Scaling for Training

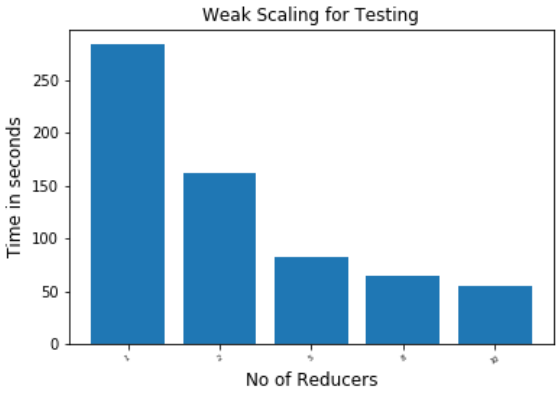

Figure 2. Weak Scaling for Testing