Language Modeling - A text prediction implementation using PySpark

Muhammad Manjur Rahaman Patowary, Mohammad W. Ullah

Big Data Programming Project, The School of Computing Science, Simon Fraser University

Keywords: Language Modeling, Text Prediction, Kneser-Ney Smoothing, Stupid Back-off, PySpark

1. Problem Definition

Text prediction is a real world application of the usage of language modeling, where a set of probable next words that are most likely to appear are suggested based on phrase, which has been entered by an user. Our training data is consisted of two corpus from blog and Reddit text covering a wide range of topics and context. In this project we are using, N-gram language modeling using two text prediction algorithms; Stupid Backoff and Kneser-Ney smoothing to do the prediction. We used PySpark as the engine at back-end and a web front-end with Flask to provide an user interface. To test the system scalability and faster querying we have used Cassandra and Parquet file system.

2. Methodology

The problem of text prediction in the context of big data comes in two fold.

1. To device statistical text prediction algorithm on big data platform using its abstractions.
2. Dealing with massive amount of text data which is a fundamental requirement for any good text prediction system.

2.1. Statistical Algorithm

For this part, we used two well known statistical language models, namely Stupid Backoff and Kneser-Ney smoothing. Both models started with cleaning the corpus. In the cleaning step we kept common English words (stopwords), also no stemming and lemmatization was done to keep the word context.

2.1.1. Stupid Backoff

After cleaning the text corpus, we generated uni/bi/tri/quadgrams and saved that into parquets file format. Ngrams help us calculating a word probability based on what came before. This is essentially taking the advantage of conditional probability as calculating full joint probability from a given corpora is computationally infeasible. For n tokens, it would take $2^{(n-1)}$ parameters.

We loaded the parquets into DataFrame and estimated each Ngram probabiliy based on maximum likelihood estimation. The Stupid Backoff algorithm follows the following schema:

$$S(w_i|w_{i-k+1}^{i-1}) = \begin{cases} \frac{f(w_{i-k+1}^{i})}{f(w_{i-k+1}^{i-1})} & \text{if } f(w_{i-k+1}^{i}) > 0 \\ \alpha S(w_i|w_{i-k+2}^{i-1}) & \text{otherwise} \end{cases}$$

There is no normalized probability and no discounting. This approach is very simple and applicable for larger size language model for example Google N-gram corpus. This model essentially back-off to lower-gram each time we fail to find a word with probability in higher-gram. This returns us a list of Ngram with probability score associated with it. We take the maximum of that and present it to the user. For the inherent simplicity Stupid Backoff has its own advantage as the size of text grows the prediction gets better. But in real world scenario often we will not have as much data as we would like, Stupid Backoff does not work well if the corpus size is not large enough. Finally, the algorithm implementation was connected with FLASK to show the output in a web-front.

2.1.2. Kenser-Ney Smoothing

Kenser-Ney (KN) smoothing is the most sophisticated statistical language modeling algorithm that takes into account both backoff and interpolation. This method is particularly useful for lower order Ngrams and for small corpus size. As before, after cleaning the corpus we generated uni/bi/trigrams and KN probability was calculated and saved in csv file. The KN smoothing used the following recursive formulation:

$$P_{KN}(w_i|w_{i-n+1}^{i-1}) = \frac{max(c_{KN}(w_{i-n+1}^{i}) - d, 0)}{c_{KN}(w_{i-n+1}^{i-1})} + \lambda(w_{i-n+1}^{i-1}) P_{KN}(w_i|w_{i-n+2}^{i-1}) \tag{1}$$

We have used trigram as our highest order Ngram hence the recursion follows the following loop:

$$P_{KN}^{1}(w_i) = \frac{w_i}{N_s} \tag{2a}$$

$$P_{KN}^{2}(w_i) = \frac{max(c(w_{i-1}, w_i) - d, 0)}{c(w_{i-1})} + \lambda(w_{i-1}) \times P_{KN}^{1}(w_i) \tag{2b}$$

$$P_{KN}^{3}(w_i) = \frac{max(c(w_{i-2}w_{i-1}, w_i) - d, 0)}{c(w_{i-2}w_{i-1})} + \lambda(w_{i-2}w_{i-1}) \times P_{KN}^{2}(w_i) \tag{2c}$$

Here,

$$\lambda(w_{i-1}) = \frac{d}{c(w_{i-1})} |w : c(w_{i-1}, w) > 0| \tag{3}$$

The KN-smoothing starts with unigram probability (equation 2a) which calculates the maximum likelihood (ML). Then it goes to bigram probability (equation 2a) that can be divided into three parts. The

first part calculates the probability given the previous context with a discount (d). We used constant discount where $d = 0.75$. The second part consists of λ that is calculated based on equation 3. We already know the value for d and $c(w_{i-1})$; $|w : c(w_{i-1}, w) > 0|$ is number of different unique word types (not their frequency). The final part is the probability from unigram. Similarly trigram KN probability was calculated using equation 2c. We should mention that no special treatment was dine for unigram probability, just simple maximum likelihood. Hence, the implementation first looks for trigram and returns a interpolated result that takes into account uni,bi and trigram probabilities. If trigram is not found and algorithm drops to bigram and finally if the given word is out of vocabulary then ML probability from unigram is returned. The final result is presented at a DataFrame on the terminal.

2.2. Large scale data manipulation

A 1GB size of text corpus is consist of more than 170 millions of words (a rough estimate). Assuming that a production ready system will have the capability to hold corpora with many corpus like this. Its apparent how the power and technologies of big data tools could aid in. A high level overview of our system can been seen from the diagram.

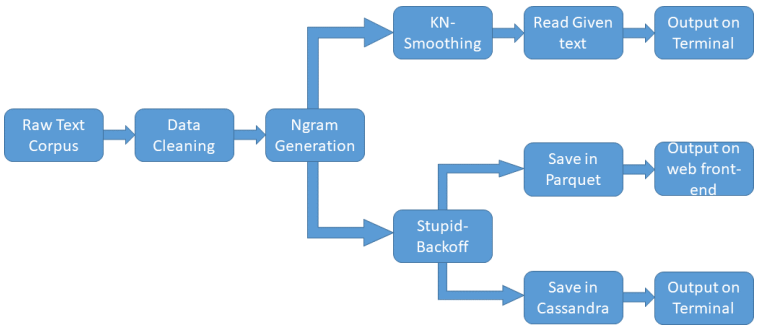

Figure 1: A high-level Flow-chat of the project

One fundamental advantage of using spark for text processing can be found in its uses of lower latency by caching partial/complete results across distributed nodes. Although, caching is suitable for small corpus but not always possible when dealing with large corpus size. All the calculation for Stupid Backoff and KN-smoothing are implemented on PySpark DataFrame to utilize the parallel performance. For our hypothetical production scenario we explored parquet and Cassandra to save the Ngrams and corresponding probabilities.

Parquet is a columnar file format with built in optimization for faster queries. One big advantage parquets columnar file format provides is that spark SQL is significantly faster with parquet. Parquet scales better, takes less disk IO, provides higher scan throughput, and provides an efficient execution of graph compared to row file format. Cassandra scales really well, and for sequential read response time is often faster. This basically means if one happens to know the query, Cassandra's database can be designed to take advantage of that. One other massive advantage of Cassandra has over parquet is that, it is platform independent.

3. Difficulties

We faced several issues during the course of the implementation of our project. The first problem was: to decide whether we want to take a neural network based approach as RNN and it's variants specially LSTM are good at finding the next probable word given a sentence, or to take a language modeling approach based on statistical algorithm. We decided on the later, as inference on LSTM is often harder to deal with and training is far more expensive, as our end goal is to provide an web application, we believe the approach we took would work out well for most cases.

Devising the algorithms (Stupid Backoff and KN-smoothing) to work efficiently with Bigdata by taking advantage of sparks abstractions (DataFrame,RDD etc) are non trivial. It is not obvious how to convert the mathematics behind these algorithms in terms of sparks DataFrame/RDDs. The complex math behind the KN-smoothing could be easily implemented in python or any other programming language. In PySpark DataFrame, we had to manage the calculations in such a way that, we can use DataFrame from beginning to end. This was a huge difficulty and it took a long time to finalize the coding. We wrote the algorithm from scratch without using any pre-build function or library (we are not aware of any such library).

We implemented stupid-backoff using a combination of try/catch block where each exception triggers into backing off to a lower Ngram. The recursion stops at one-gram. Another big challenge for us was to combine spark with flask to provide a working front-end in the final deliverable of our project. But it is worth mentioning in a real world deployment we will integrate cherry pi/celery with it for even faster querying.

4. Result

We have implemented two statistical language models for this project, namely Stupid-Backoff and KN-smoothing. As expected, KN-smoothing works better for lower-order Ngrams also with smaller corpus. In particular, we can see new words in KN-smoothing that are not visible in Stupid-Backoff. We would like to particularly mention that we have implemented the algorithm by ourselves. We have also learned during the project work and implemented a web-based front-end using Flask so that user can type on the given text box on the web-page and it will return some high probability words. We have also added a word could that will generate on the web-page.

Next word probability with respect to both algorithms, Dataset size 1 GB

Algorithm	User Input	Top predictions
Back off	I hope you are	prepared, kidding, in, proud, just
Kneser-Ney	I hope you are	also, a, adjusting, glad, joking, fit
Back off	you should not be	changing,necessary,crying
Kneser-Ney	you should not be	wasting, falling, seen, doing, here
Back off	so many people seem to think	you, when, down, which
Kneser-Ney	so many people seem to think	in, why, that, hope, something

Next word probability with respect to both algorithms, Dataset size 57 MB

Algorithm	User Input	Top predictions
Back off	you are	reproduced, sent, discouraged, cutting, deleted
Kneser-Ney	you are	expecting, crying, stressed, controlled, alive

Consistency of KN-smoothing can be easily seen from the result, where back off can be a hit or miss depending upon the corpus size. As we see KN performs much better for smaller Dataset, where Backoffs result are less than acceptable. But as the size of our Dataset grows Backoff performance improves significantly. KN still has the upper hand when it comes to accuracy, but considering the simplicity of Backoff and ease of implementation Backoff performance is reasonable.

5. Project Summary

1. Getting the data: Acquiring/gathering/downloading **1**
2. ETL: Extract-Transform-Load work and cleaning the data set. **3**
3. Problem: Work on defining problem itself and motivation for the analysis. **3**
4. Algorithmic work: Work on the algorithms needed to work with the data, including integrating data mining and machine learning techniques. **4**
5. Bigness/parallelization: Efficiency of the analysis on a cluster, and scalability to larger data sets. **3**
6. UI: User interface to the results, possibly including web or data exploration frontends. **2**
7. Visualization: Visualization of analysis results. **1**
8. Technologies: New technologies learned as part of doing the project. **3**