JFastText Text Classification

Dhayananth Dharmalingam

July 19, 2020

1 Abstract

The FastText Sentence Classification library developed by facebook for the purpose of text classification and sentence analysis. The library originally written using C++, JfastText is a Java based wrapper on top of FastText, gives access to C++ based library inside Java program. Further section discusses about JFastText regarding the integration, training, learning approach, and result.

2 Introduction

("FastText: cc" 2020) FastText library is for efficient learning of word representation and sentence analysis. The library uses Neural network algorithms to achieve best result in the context of text classification. It supports both supervised and unsupervised learning approaches based on the data-set ("FastText: Under the Hood" 2018). The library is executable in cmd. It is an open source project in Github.

The project is on much stable to integrate to production environment based on the project complexity. Still, improvements and modifications are actively performed by the developer team. ("An intro to text classification with Facebook's fastText" 2019). FastText supports training continuous bag of words (CBOW) or Skip-gram models using negative sampling, softmax or hierarchical softmax loss functions.

3 How it work

FastText achieves good performance with text classification, in special case of rare word by making use of character level information. Each word is represented as a bag of character n-grams in addition to the word itself, so for example, for the word "MATTER", with n = 3, the fastText representations for the character n-grams is <ma, mat, att, tte, ter, er>. < and > are added as boundary symbols to distinguish the ngram of a word from a word itself, so for example, if the word mat is part of the vocabulary, it is represented as <mat>.

This helps preserve the meaning of shorter words that may show up as ngrams of other words. Inherently, this also allows to capture meaning for suffixes/prefixes.

("Bag of Tricks for Efficient Text Classification") The model considered to be a bag of words and apart from ngram window , there is no internal structure of a word that is taken into account when predicting. The length of the n-gram controlled with -minn and -maxn options that passed to the command.

The length of n-grams can be controlled by the *-minn* and *-maxn* flags for minimum and maximum number of characters to use respectively. These control the range of values to get n-grams for. setting both max and min to 0 will turn of the n-gram embedding. This can be useful when character level n-gram doesn't make sense. (Ex: Testing partially different language documents (Ex: French) from model (English)).During the model learn update, FastText learns weights for each of the n-gram and the entire word token.

3.1 Bag of word process of efficient classification

The BoW feature of FastText performs with following process and steps.

- Starts with word representations that are averaged into text representation and feed them to a linear classifier (multinomial logistic regression).
- Text representation as a hidden state that can be shared among features and classes.
- Softmax layer to obtain a probability distribution over pre-defined classes.
- Hierarchial Softmax: Based on Huffman Coding Tree Used to reduce computational complexity O(kh) to O(hlog(k)), where k is the number of classes and h is dimension of text representation.

- Uses a bag of n-grams to maintain efficiency without losing accuracy. No explicit use of word order.
- Uses hashing trick to maintain fast and memory efficient mapping of the n-grams.

("Bag of Tricks for Efficient Text Classification" 2017)

A sentence/document vector is obtained by averaging the word/n-gram embeddings. For the classification task, multinomial **logistic regression** is used, where the sentence/document vector corresponds to the features. When applying FastText on problems with a large number of classes, we can use the hierarchical softmax to speed-up the computation.

4 Training

FastText initializes a couple of vectors (word2int_ and words_. word2int_) to keep track of the input information, The vectors are indexed on the hash of the word string, and stores a sequential int index to the words_ array (vector) as it's value.

Once the input and hidden vectors are initialized, multiple training threads been started. The number of threads can set by -thread option. All the training threads hold a shared pointer to the matrices for input and hidden vectors. The threads all read from the input file, updating the model with each input line that is read, (i.e. stochastic gradient descent with a batch size of 1). An input line is truncated if newline character is encountered, or if the count of words that have been reaches the maximum allowed line size. This is set via MAX_LINE_SIZE, and defaults to 1024. Both the continuous bag of words and the Skip-gram model update the weights for a context of size between a random uniform distribution between 1 and the value determined by -ws, (i.e the window size is stochastic). The target vector for the loss function is computed via a normalized sum of all the input vectors. The input vectors are the vector representation for the original word, and all the n-grams of that word. The loss is computed which sets the weights for the forward pass, that propagate their way all the way back to the vectors for the input layer in the back propagation pass.

The learning rate affects how much each particular instance affects the weights. That can be controlled with -lr option.

5 Project setup and configuration

Initially the training data prepared with label prefix as guided in the documentation.

```
(Ex: __labe__socker i play football).
```

Training data should contain line by line text with label prefix. The training data should annotated with *-input* , then based on the label, supervised model generated in a folder where it is addressed with *-output*. The project requires maven dependency of javacpp package. FastText uses native c++ compiler and this package enables a bridge between jvm and c++ native compiler.

5.1 Options

("FastText: Options" 2020)

```
Important:*
  -input  training file path
  -output output file path

Optional:
  -wordNgrams
  --max length of word ngram [1]

Training options:
  -lr
  -- learning rate [0.4]

  -lrUpdateRate
  --change the rate of updates for the
  learning rate [0]

  -epoch
  --number of epochs [50]

  -loss
  --loss function {ns, hs, softmax} [softmax]
```

6 Dataset

We employ 3108 (french) training data with 4 labels. Same data set trained with two labels to check another situation (Two Label). The training data then used for testing and evaluate the algorithm without labels.

7 Test result summary

The result shows good efficiency in the field of text classification based on training data. The model keep learning with 0.5 (default) learning update value. We tested the algorithm with same data that is used to train the model. We tested two situation, first one is with 2 labels (INFORMATION, REQUIREMENT), and second one is with 4 labels (FUNCTIONNAL , INFORMATION, PERFORMANCE, CONSRAINT). Results are evaluated with actual label and predicted label as it showing in Table [1] and Table [2]. (Total - Total number of records, TRUE - Correctly identified, FALSE - Incorrectly identified, Avg.Strengh - Average prediction Strenght ())

Table 1: *Test Results- 2 Labels - Cross validation*

SUMMARY-2 Labels			
TOTAL	TRUE	FALSE	Avg.Strength
3107	3106 99.97%	1 0.03%	0.993636421
Mean Abs.Error	0.012		
Accuracy	99.97%		

Table 2: *Test Results- 4 Labels - Cross validation*

SUMMARY-4 Labels			
TOTAL	TRUE	FALSE	Avg.Strength
3107	3061 98.52%	46 1.48%	0.984135384
Mean Abs.Error	0.046		
Accuracy	98.52%		

8 Discussion

FastText library shows better performance and accuracy when it was evaluated with same training data. Learning and classification process are much faster, Since, the training set contains 3000 of records.The result of 4 Labels are predicted with 90.34% correctly identified texts. Then we clean the data set that used for training by removing the stop words and assigned nGram length for the learning algorithm. This modification helps to achieve 3061 correctly identified labels (98.52%) from 3107 entries. The training time and classification time are much faster. As shown in the Table [5], the algorithm took 23.5s to train 3061 record with two labels, and 26.7s with four labels. Also, it took 0.28s to classify with two label model and 0.33s with four label model. FastText obtains performance on parallel with recently proposed methods inspired by deep learning, while being much faster. It can perform well on large tender document with memory efficient that helps to process the document faster and accurately.

Table 3: *Performance Summary*

Performance Summary			
Label	TOTAL	Learning time	Classification Time
2	3061	23588ms/23.5s	280ms/0.28s
4	3061	26734ms/26.7s	333.5ms/0.33s

Table 4: *Test result - Comparison*

Result comparison with baseline algorithm				
Classification technique	Correctly classified instances	Incorrectly classified instances	Mean absolute error	Accuracy
SMO	3699	584	0.26	86.3%
J48	3217	1066	0.15	75.1%
Naive Bayes	3440	843	0.11	80.3%
Random Forest	3520	763	0.18	82.1%
JFastText 2 Label	3106	1	0.012	99.97%
JFastText 4 Label	3061	46	0.046	98.52%

Table 5: *Performance comparison*

Performance - Comparison		
Classifier	Learning Time	Classification Time
SMO	9.78ms	13ms
J48	56.82ms	19ms
Naive Bayes	3.18ms	59ms
Random Forest	46.48ms	43ms
FastText (2Labels)	23588ms/23.5s	280ms/0.28s
FastText (4Labels)	26734ms/26.7s	333.5ms/0.33s

9 Comparison

Baseline algorithm shows the (average) accuracy of 81.3% with different classification techniques. The highest accuracy of 86.3% is achieved with SMO technique. The results are better, but still we can find that increased number of incorrectly identified instances in all techniques. JFastText shows the result of nearly 99% accuracy in both instances. It classified every data perfectly except one data with two label instance. Even the results are much better with 4 label instance. Absolute error is considerably very low with JFastText compared to baseline algorithm. By considering the result of both baseline and JFastText, JFastText is achieving better accuracy.

The advantage of the baseline algorithm is, it shows fast enough performance than JFastText. The training time and classification time are comparatively low than JFastText. There can be many other factors that might impact the execution time (Ex: Processing power, additional workload of classifier method -print etc). File printing method also performed along with classification process using JFastText. These kind of situation and hardware configuration might have been affected the JFastText performance. Additionally, we have options to start multi thread learning by specifying the thread count option. This could helps to improve the performance of the algorithm

References

"An intro to text classification with Facebook's fastText" (Nov. 2019). In: URL: `https://towardsdatascience.com/natural-language-processing-with-fasttext-part-1-an-intro-to-text-classification-with-fasttext-11b9771722d8`.

"Bag of Tricks for Efficient Text Classification" (Apr. 2017). In: URL: `https://www.aclweb.org/anthology/E17-2068/`.

"FastText: cc" (July 2020). In: URL: `https://fasttext.cc/`.

"FastText: Options" (July 2020). In: URL: `https://fasttext.cc/docs/en/options.html`.

"FastText: Under the Hood" (July 2018). In: URL: `https://towardsdatascience.com/fasttext-under-the-hood-11efc57b2b3/`.

Mikolov, Armand Joulin Edouard Grave Piotr Bojanowski Tomas. "Bag of Tricks for Efficient Text Classification". In: *Bag of Tricks for Efficient Text Classification* 1.1 (), pp. 1–4. URL: `https://arxiv.org/pdf/1607.01759.pdf`.