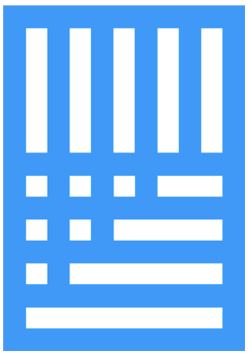

CASELAW ACCESS PROJECT
HARVARD LAW SCHOOL

FastLaw vs. FastText

Technical Report

Author:
Jonathan Besomi
Ecole Polytechnique Fédérale de Lausanne

Chair of Technology and Innovation Strategy
Ecole Polytechnique Fédérale de Lausanne

I. Introduction

Fastlaw is a law-specific word embedding[1] model trained on a large corpus of law-related text data provided by the Caselaw Access Project[2] (CAP). *Fastlaw* was computed using *Spark*, a distributed cluster-computing framework, and *FastText*, a library for efficient word representation learning.

The code to reproduce *Fastlaw* and the results shown in this document has been published on Github[3]. As this is a technical paper, the aim of this document is to provide additional details about the implementation and the respective challenges encountered during development. This document may be useful to anyone who would like to contribute to the development of *Fastlaw*.

Fastlaw has been developed as a semester project by Jonathan Besomi at the Technology and Innovation Strategy Laboratory (TIS)[4] at EPFL. It was supervised by Prof. Kenneth Younge and his PhD assistant Maximilian Hofer.

A. Fastlaw motivation

In general, when the training dataset is large enough a machine learning model does not rapidly overfit the training set. If well tuned, the model is able to perform well on its task. However, if the training dataset is too small, chances are that the model overfits quickly.

Labelling data manually is slow, and having access to labelled training data is complex. Besides that, computational power is needed and it is expensive.

Fastlaw's purpose is to replace generic word embeddings for work on supervised machine learning NLP-tasks with legal texts.

As will be highlighted in the result section, for a chosen classification task when the training size is small, *Fastlaw* outperforms *FastText*. Nevertheless, the goal of this document is not to prove or disprove that *Fastlaw* may be a better solution for certain cases, rather, it is to showcase what has been done, to let the community try the model, and to later examine the feedback.

[1] A technique where words are encoded as real-valued vectors in a high dimensional space. Semantic similarity between words translates to closeness in the vector space.

[2] https://case.law/

[3] http://github.com/jonathanbesomi/fastlaw

[4] http://tis.epfl.ch

B. CAP Research Summit

The project was presented in June 2019 at the Caselaw Access Project Research Summit at Harvard University. The summit is a "space to highlight work supported by CAP data, and to think about how this dataset may be used to to shape scholarship and access to the law in the future".

As well as explaining *Fastlaw* and its results, the focus of the presentation was on the preprocessing steps made on the CAP data. This goes from downloading the data to feeding the cleaned and tokenized text to *FastText*.

As will be shown throughout the paper, using *Spark*/*PySpark* to handle legal data has been an efficient and advisable solution because of its capacity to handle large semi-structured datasets. The presentation has underlined the fact that an analogous program may be used to clean the data and provide the CAP community with a slightly better version of it.

The PowerPoint presentation is available at the following address: http://go.epfl.ch/fastlaw-ppt.

II. Word2Law

Word2Law [5] is an analogous word representation model trained on legal corpora from various public sources. It was developed by Ilias Chalkidis and Dimitrios Kampas. The authors released two models with respectively 100 and 200 dimensions. The training corpus has 492M individual tokens. The US jurisdiction, used to construct the final word embedding has about 5000M tokens, 10 times larger.

The authors did not make the code available and limited themselves to a quantitative analysis showing the top 5 similar words for a set of 20 selected words.

Word2Law was trained with Word2Vec because it "is reported to provide better semantic representation than *FastText*" and because

they "empirically observed that legal documents have been consistent being formal by misspellings, grammatical/syntactical errors, as well as the vocabulary being formal and pertinent to the domain".

Unlike the corpora of Word2Law, *Fastlaw* does contains misspelled words since the data have been digitally recorded. *FastText* was chosen initially because it implements character embeddings (subwords) that – according to its authors – boost the model's performance, particularly when words are misspelled [2].

III. FASTLAW PIPELINE

For a better overview of the paper, we quickly describe the main phases of producing *Fastlaw*.

1) The jurisdiction *jsonl* data are downloaded from the CAP portal
2) Selected *jsonl* data are merged together and loaded into a *PySpark Dataframe*
3) Data from the *opinions*-column are selected and preprocessed
4) Preprocessed text data are saved as (multiple) *Spark*-text files
5) Exploiting Apache Hadoop File System, the text files are merged into a single *txt* file
6) *FastText* receives as argument the single *txt* file and eventually outputs *Fastlaw*

IV. GITHUB REPOSITORY

To ensure reproducibility and enable the community to use and evaluate the models, all work has been published on Github at http://github.com/jonathanbesomi/fastlaw.

The repository is composed of three main bash scripts. A substantial effort has been made to keep the code simple, concise and well documented to permit anyone to understand and, possibly, contribute to it.

It's advised to refer directly to the *readme* file in the root of the repository for details about accepted arguments and the execution of the different scripts.

1) download_bulk_data.sh: allows to download all jurisdiction data from the CAP website. The script accomplishes several tasks such as downloading a single file or a list of jurisdiction files, downloading all data, or downloading only public jurisdiction. The script may be used by other researchers who are working on the CAP dataset that doesn't necessarily need a word embedding.

2) wordembedding.sh: permits to construct the word embedding. The most important parameter is the slug (*-s*) that represents the jurisdiction data based on which the word embedding will be computed.

3) evaluation.sh: allows to assess different models of *Fastlaw* against *FastText*[5]. The evaluation is performed on a text classification task where the objective is to distinguish civil vs. criminal cases.

V. DEVELOPMENT

This section aims at introducing the data and explaining the obstacles and respective solutions encountered during the development of *Fastlaw*.

A. CAP data

All needed data can be downloaded from the CAP website[6]. The portal permits to download each jurisdiction file separately. The decompressed format is *jsonl*, where each line is a single case.

An exploratory data analysis has been conducted for the state of Illinois and published as a *kernel* on *kaggle.com*.

It's advised to go through the notebook to better grasp the content and format of the data.

A single case has multiple fields nested in a hierarchical structure. Among others, the fields are *jurisdiction*, *attorney*, *parties*, hence the different entities of the case. Of primary importance is the field *opinions* that contains legal text about the respective case. It accounts for more than 90% of the size of the dataset.

[5]wiki-news-300d-1M.vec.zip, 1 million word vectors trained on Wikipedia 2017, UMBC webbase corpus and statmt.org news dataset (16B tokens)

[6]https://case.law/

B. Spark, DataFrame and Hadoop

1) Spark: The CAP dataset is big. Once decompressed, the cumulative size of all *jsonl* jurisdiction files is *79G*.

Spark is a distributed data processing framework that allows to deal with large amounts of data and it is therefore suitable for the CAP dataset. Moreover, Spark's ability to store data in memory and rapidly run repeated queries makes it a good choice for the preprocessing step – one of the most important phases to produce a reliable word embedding model (see section V-C).

Fastlaw uses the *PySpark API* as Python is now the de facto programming language for NLP and data analysis applications.

Spark revolves around the concept of resilient distributed datasets (RDD), which are collections of elements that can be operated on in parallel. Once a Spark application is executed, the data are partitioned into multiple smaller chunks, called *partitions*.

2) DataFrame API: The CAP data comes in a structured format. Whenever analyzing semi-structured data with Spark, it is strongly suggested to make use of *Spark SQL*, a component on top of *Spark Core* that introduces a data abstraction called *DataFrames*. A *DataFrame* is a distributed collection of data organized into named columns, it is conceptually equivalent to a table in a relational database, but with richer optimizations under the hood.

3) Apache Hadoop: Spark's power lies in its ability to execute data manipulation and transformation in parallel on multiple *partitions*. *FastText*, on the other hand, accepts as input a single *txt* file. A solution, preferably an efficient one, is therefore needed to merge the text files located on different partitions into a single one that will be fed to *FastText*.

Spark does not have a built-in function capable of outputting a single text file from different partitions. Relying only on *Spark*, the naive solution would consist of coalescing all the data into a single partition and then save the content of it in a single text file. This action would be expensive and inappropriate since all data would be loaded into a single partition first.

To overcome Spark's limitation on outputting a single data file, an alternative solution that exploits Apache Hadoop Distributed File System (HDFS) has been implemented. First, the content of each partition is saved in parallel into multiple text files, then, with a parallel version of the *cat* command provided by the HDFS library, all files are merged into a single one.

C. Preprocessing of the data

The CAP data have been digitized by scanning over 40 million pages. Consequently, the text data are dirty.

The development of the final word embedding was done by trial and error. Every time the pipeline changed, the new model was intrinsically evaluated and compared against *FastText*. Section VI explains in detail what has been done to compare the different models and to assess new preprocessing steps or other changes to the final model.

All opinions text data were *tokenized* using *wordpunct_tokenize*, a function provided by NLTK[7] that handles punctuation separately.

The list below contains most of the important tasks performed during preprocessing:

1) Removed numbers from all words. For example *law5* was replaced with *law*
2) Removed special characters from all words, for example *law_* was replaced with *law*
3) Removed all accents from all words
4) Tokenized and *lowercased* all words

The developed pipeline, due to its use of *PySpark* and the respective *DataFrame* is robust and permits to perform these operations – that may be heavy and time-consuming for large amounts of data – in a quick time.

1) PySpark Dataframe Regex: The *PySpark SQL* documentation was one of the most visited websites during the development of the whole project. One of the most used functions was *regexp_replace(str, pattern, rep)*. This method replaces all substrings of the specified string that match the regular expression with the string contained in *rep*.

[7]Natural Language Toolkit, https://www.nltk.org/

The following regular expression, for instance, was used to convert any string of the form *word_* to *word*.

```
(\w+[^\\w{_}]|\w+)
```

2) Non-existing words: When looking for similar words in the word space, it was noticed that a large quantities of tokens where not officially English words. The so called *non-existing English words* are either misspelled words due to the fact that the documents have been automatically digitized or proper nouns such as the name of attorneys or the name of involved parties in a case.

An algorithm was implemented to filter out non-existing words in linear time. Here, instead of dealing with a *DataFrame*, the *opinions* data are first converted into a resilient distributed dataset. Then each token in the RDD is filtered out if it is not present in the vocabulary of English words (*all_english_words*) or it is not a punctuation symbol.

For better performance, the vocabulary is defined as a Python *set* so that the cost of checking the existence is constant. The vocabulary is defined as a broadcast variable that is read-only and shared across each node of the cluster.

```
all_english_words = sparkContext
    .broadcast(
     set(all_english_words.collect())
    )

ops = ops.filter(
     lambda w: w in all_words.value)
```

D. Google cloud

Under the hood, *PySpark* is built on top of the *Spark Java API*. Data is processed in Python but cached in the Java Virtual Machine. One execution error that appears often when dealing with *PySpark* is *java.lang.OutOfMemoryError*. This error occurs when at some point, certain data could not be loaded into memory. This can be easily fixed by properly setting *Spark*'s configuration parameters: spark.driver.memory and spark.executor.memory.

The code has been tested both locally on a personal computer and on a Google cloud instance with 120GB of memory and 32 CPUs. Because the resources are different, the *Spark* configurations are different across machines.

The get_spark() methods in *tools.py* permit to configure Spark's resources separately on different machines.

VI. MODEL EVALUATION

As explained in [4], even if recently model representations have gained popularity, "the issue of the most adequate evaluation method still remains open". There is little to no research about word embedding evaluation and there doesn't exist a clear scheme to follow.

One of the most effective ways to evaluate a word representation model is to verify how different models perform when used as the feature vectors of supervised machine learning tasks such as text classification, sentiment analysis or named entity recognition.

A. Subwords in FastText

One of the reasons *FastText* has been chosen as the word embedding library is because – according to its authors – it is able to achieve good performance on word representations with character level information (n-gram embeddings). In *FastText*, each word is represented as a bag of character n-grams in addition to the word itself. For instance the n-grams character representation of the word lawyer, with $n = 3$, is <la, law, awy, wye, yer, er>.

B. Word embeddings

Four word embeddings were constructed for evaluation. All four were trained on the US jurisdiction data (about 28G in size) and have a dimension of 300. The *FastText* model for computing word representations is skip-gram, that learns to predict a target word thanks to a nearby word.

In the word embedding with the *nocap* flag, the non-existing English words have been removed, whereas in the word embedding with the *zero* flag, the model has been trained without subwords. The *zero* flag is defined

because the *-maxn* and *-minn* are set to zero if subwords should not be considered during training.

- fastlaw_300_us_cap_ngram.vec
- fastlaw_300_us_cap_zero.vec
- fastlaw_300_us_nocap_ngram.vec
- fastlaw_300_us_nocap_zero.vec

C. Classification task

To assess the quality of *FastLaw*, a classification task that attempts to categorize civil vs. criminal cases was developed.

The main steps of the evaluation script are the following:

1) For a specific jurisdiction, *opinions* data are loaded with PySpark
2) Using PySpark, cases are labelled either criminal or civil
3) Data are preprocessed so that they can be fed into a neural network
4) Using *Keras*, different word embedding models are tested and their performance is saved in log files and plots

Since there is no field in the CAP data that signals if a specific case is civil or criminal, the data was labelled automatically. All cases that have the word *state* or *commonwealth* (target words) in the *name_abbreviation* field were labelled has criminal cases. Moreover, all target words have been removed from the *opinions* column to minimize the bias.

Because some *opinions* texts are extremely long, each *opinions* text is limited to a certain length. For the *opinions*, the average length in words is about 1500 and the same value is also the default value.

Multiple neural network structures have been tested, including a simple feed-forward model, a convolutional neural network and a recurrent neural network using LSTM. For multiple jurisdiction files, a fast-forward neural network composed of a dense layer of size 400, a global max pooler and two other dense layers of size respectively 200 and 10 performed well enough and was therefore chosen as the default model. The code, as well as the logs, have been written so that it is easy to experiment with different models.

D. Metrics

Since the task is a binary classification, it's crucial that the distribution of the target value is uniform. In the preprocessing phase the dataframe is balanced.

We want to show that when the training data is limited, using *FastText* may be convenient. The main metric used to evaluate the model is the training accuracy.

Even when the model is trained on only hundred cases, the training accuracy is always computed on multiple thousand cases (around 10k in most cases).

Also, note that large jurisdictions are limited with PySpark to *max_num_cases* cases. Some jurisdictions such as New York have millions of cases, it would be worthless and computationally expensive to deal with all cases.

E. Results

Fastlaw vas been evaluated on two states, Florida and Hawaii. The tables below summarize some keys fact about the jurisdictions and the respective law cases.

Florida	
# of total cases	333'512
# of criminal cases	149'683
# of civil cases	183'829

Hawaii	
# of total cases	19'715
# of criminal cases	12'125
# of civil cases	7'590

In addition to the four FastLaw models, FastText and a model with random weights initialization has been evaluated.

The FastText model is the wiki-news-300d-1M.vec.zip that can be downloaded from https://fasttext.cc/docs/en/english-vectors.html. It contains 1 million word vectors trained on Wikipedia 2017, UMBC webbase corpus and statmt.org news dataset (16B tokens).

For each jurisdiction, each models (6 in total) has been trained on the same initial settings over 5 runs and 5 different training size, 50, 100, 200, 500 and 1000.

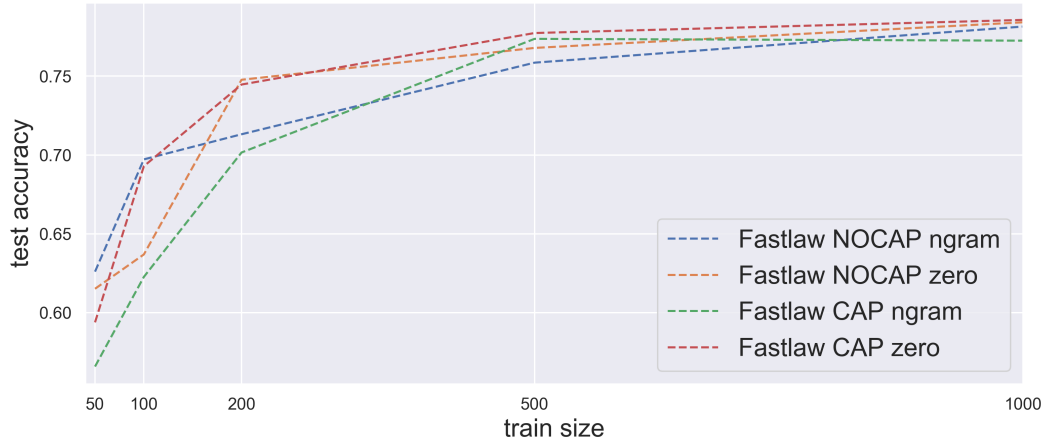

Fig. 1: Comparison of the four Fastlaw models for Florida's State

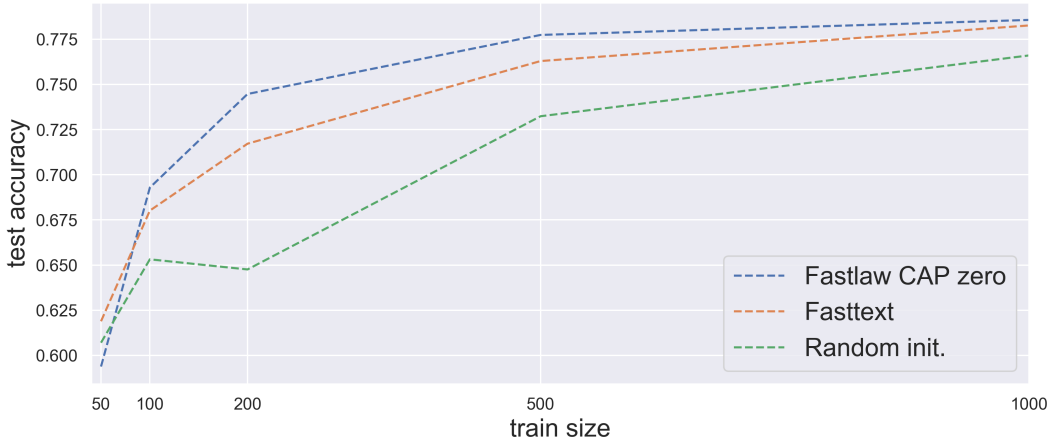

Fig. 2: Best Fastlaw models against FastText for Florida's State

The plot in figure 1 compare the four FastLaw models. Among them, the best model is the one trained on all CAP dataset but without subwords and the second best model is the NOCAP with subwords. This fact has been remarked in other experiments and it's interesting. Not enough research has been made to assert a claim, but apparently subwords work better only in case the data are not too dirty, as it's the case with the NOCAP dataset.

In figure 2, the best model of Fastlaw (CAP zero) is compared against FastText and a random word initialization. As it's expected, both models perform better than a random initialization. It can be noticed that for 100, 200 and 500 training cases, FastLaw perform slightly better than FastText.

Ultimately, figure 3 compare FastLaw CAP without subwords against FastText for Hawaii's State

VII. FUTURE WORK

The future works will vary according to the feedback the project will receive at the CAP research summit. The improvements includes, but are not limited to:

- Evaluate the model on new NLP-tasks and update the results accordingly
- Evaluate the model intrinsically (example: word similarities)
- More hyperparameters analysis (skip-gram vs. cbow, dimensions, importance of subwords)
- Improve the pre-processing steps

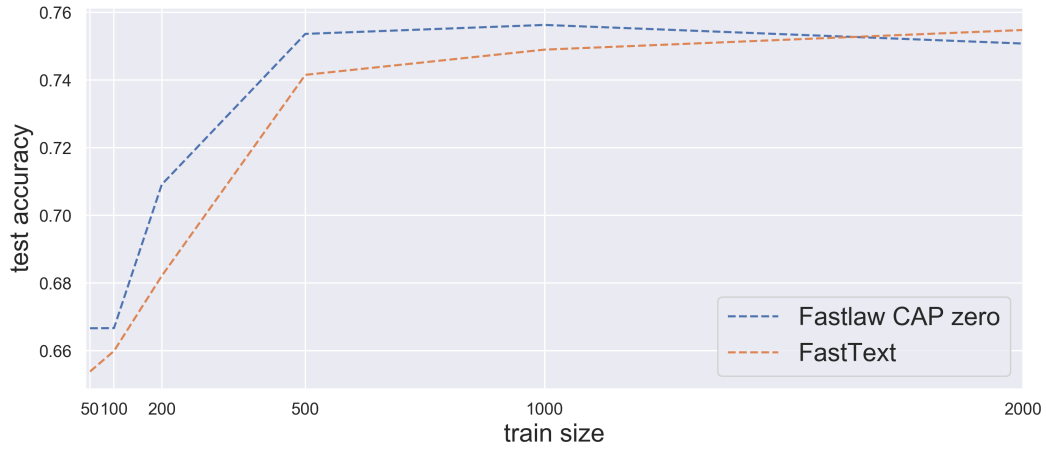

Fig. 3: FastText vs. Fastlaw CAP zero vs. Random Initialization for Hawaii's State

VIII. ACKNOWLEDGEMENTS

A special thank goes to Professor Kenneth Younge and PhD candidate Maximilian Hofer for having followed and helped me during the whole project.

Prof. Kenneth, with his large understanding of AI, business, law and management has constantly given me advice and suggestions on how to proceed with the development of the project.

Maximilian Hofer has always been available for my questions and to support me during the project. His wide knowledge on NLP and word embeddings has been extremely useful to address and orient the research.

I thank both of them for having invested many precious hours of their time to follow the project. Those 12 weeks permitted me to learn a lot both on the technical and personal side.

A warm thank you also to my friends Yann Bolliger and Pietro Carta for having proofread this document.

IX. CONCLUSION

In some cases, *Fastlaw* proved to be a valid alternative to a generic word embedding for classifying civil vs. criminal cases. This is not yet sufficient to claim that *Fastlaw* is better than a generic word embedding for any law-related NLP task, more experiments and evaluation should be made.

The new word representation has been trained on the CAP dataset. Subsequently, it may be used by other CAP researchers that need to perform similar NLP-related tasks. The CAP research summit will be a good time to discuss this.

What has been proven, in contrast, is that the use of PySpark to analyze structured *json* data is efficient and convenient. The same analysis and preprocessing of legal data with the developed tools (*PySpark*/Bash/Apache Hadoop) may be repeated for the whole CAP dataset. For instance, in order to clean the data, to analyze it in an efficient way and to produce advanced statistics.

The hope is that the effort of producing a "plug-and-play"-style Github repository will pay off, enabling new research in the field.

REFERENCES

[1] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado and Jeffrey Dean.
Distributed Representations of Words and Phrases and their Compositionality, 2013

[2] Bojanowski, Piotr and Grave, Edouard and Joulin, Armand and Mikolov, Tomas.
Enriching Word Vectors with Subword Information, 2017

[3] Mikolov, Tomas and Grave, Edouard and Bojanowski, Piotr and Puhrsch, Christian and Joulin, Armand.
Advances in Pre-Training Distributed Word Representations, 2008

[4] Amir Bakarov.
A Survey of Word Embeddings Evaluation Methods, 2018, arXiv:1801.09536

[5] Ilias Chalkidis and Dimitrios Kampas
Deep learning in law: early adaptation and legal word embeddings trained on large corpora, 2018