**Comparative Study To Solve Text Categorization**

CES Data Scientist 2017-2018 (Telecom ParisTech)

Jacques Doan-Huu

***Abstract***

In the last decade, **D**eep **L**earning (**DL**) has demonstrated outstanding performance in the field of computer vision beating indisputably traditional methods, thanks to the GPU performance leapfrog and the huge amount of labeled datasets. A bit more recently, DL also went into the **N**atural **L**anguage **P**rocessing (**NLP**) field battle to solve common NLP problems like text classification and translation, with very promising perspectives and results: in particular, word embedding and **R**ecurrent/**C**onvolutional **N**eural **N**etwork (**RNN/CNN**) architectures provide efficient technical responses to NLP challenge.

**POSOS** French startup has submitted a data challenge for which I took the opportunity to verify humbly whether DL is a suitable solution compared to traditional techniques, for a beginner like me having very few experiences on NLP/DL area and low end hardware system (DL has the bad reputation to be numerically intensive…).

***Statement of the Problem***

It’s plainly exhibited at ENS school web site ([link](https://challengedata.ens.fr/en/challenge/33/predict_the_expected_answer.html)) and it consists in classifying into **51** intents, drug related questions written in natural language (French to be precise). **POSOS** claimed to get good modeling result with 86% accuracy by utilizing DL: they don’t provide any details on the DL architecture nor any engineering clues except the possibility to extract some relevant information procured by the French drug administration (**ANSM**).

The target categories (question intent) have been intentionally anonymized into indices from 0 to 50: hiding their respective semantic is probably aimed to avoid the usage of topic-specific (and so biased) procedures. Training dataset contains only ~8000 questions: it’s pretty short to produce a good learning outcome. Moreover, the text suffers from many types of anomalies (misspelling, grammatical incorrectness, familiar acronym, …) and employs specific medical vocabulary (drug name like “mirtazapine”, …): it makes the NLP challenge harder to handle.

Here’s an example of misspelled sentence: “8 jrs avant la fin de ma plaquette d'evepar j'ai eu des saignement~~s~~ et des douleurs au bas ventre et au bas du dos dois je m'inquiéter»

***Project Motivation***

The purpose of this study is to compare and the pro and cons between DL and non-DL approaches from different perspectives:

* model accuracy/performance
* operational aspect (tooling, hardware requirement, ...)
* engineering level of difficulty to find the good hyper-parameter, architecture and processing logic

In fact, these aspects are somehow inter-related: typically, model accuracy may be unexpectedly suboptimal due to operational reasons (memory shortage to complete the processing job, …).

The idea is not to get a fined-tuned model with DL or other techniques, but an attempt to explore comparatively the end to end methodology to tackle a text classification problem with 2 distinct technologies.

“Traditional techniques” refer to any ML algorithms which don’t rely on neural network theory (eg: Word2Vec is excluded): to quote some of them, Hidden Markov Model, XGBoost, SVM, logistic regression and PCA are eligible.

Conversely, DL option should rely uniquely on neural network but it can as well benefit from “neutral” text preprocessing (feature enrichment with external source, stemming, stopWords, …) for fairness sake.

***ML Workbench Environment***

All experiments have been written in Python in the popular Jupyter environment: the notebooks are accessible publicly as a github project whose details are provided in the annex section. I had made use of many python packages to satisfy various requirements:

* data manipulation and visualization: pandas, numpy and matplotlib
* text processing (stemming, stopWords, …): NLTK, standard regex and spellChecker (built from github)
* ML algorithms (XGBoost, PCA, t-SNE): sklearn and XGBoost
* DL framework: Keras + Tensorflow

Most of packages have been installed as is, except for XGBoost I recompiled locally from its github source code to get the GPU accelerated version which is not shipped officially.

In addition to the above runtime packages, this project also takes advantage of public resource or pretrained models (NLTK corpus, FastText word embedding model, ….).

ML jobs had been initially executed with an old MacBook Pro whose chipset was damaged by the heating caused by the overnight DL train, then with a many CPU core/low-end GPU PC workstation.

I finally paid a GPU boosted Amazon instance: even if the GPU/CPU resource is not utilized, the data storage is charged permanently making the overall cost very high (5$/h) during 2 full days.

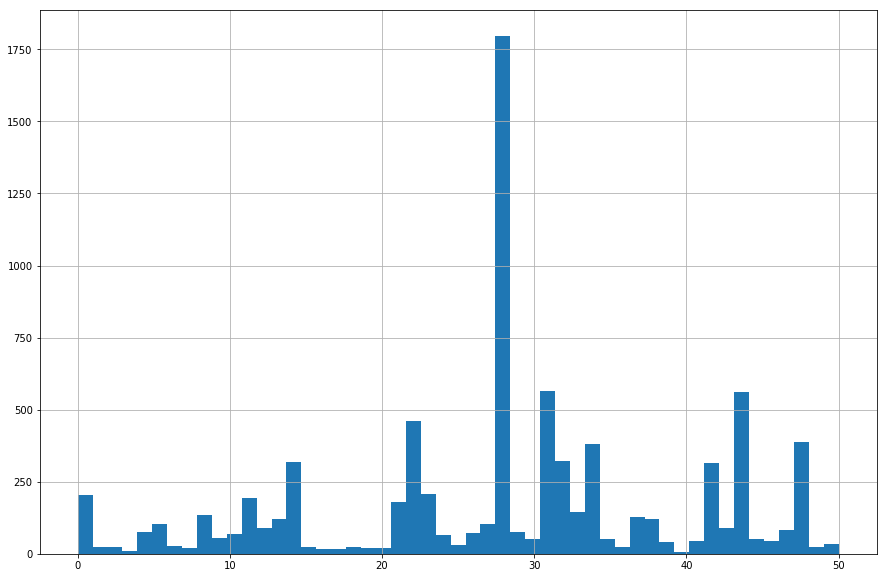
At the very end, the best money saving option was to buy a PC gamer machine with mid-range Nvidia GPU card to carry out the computing workload: I noticed a huge speed improvement at training time.

***Data Exploration***

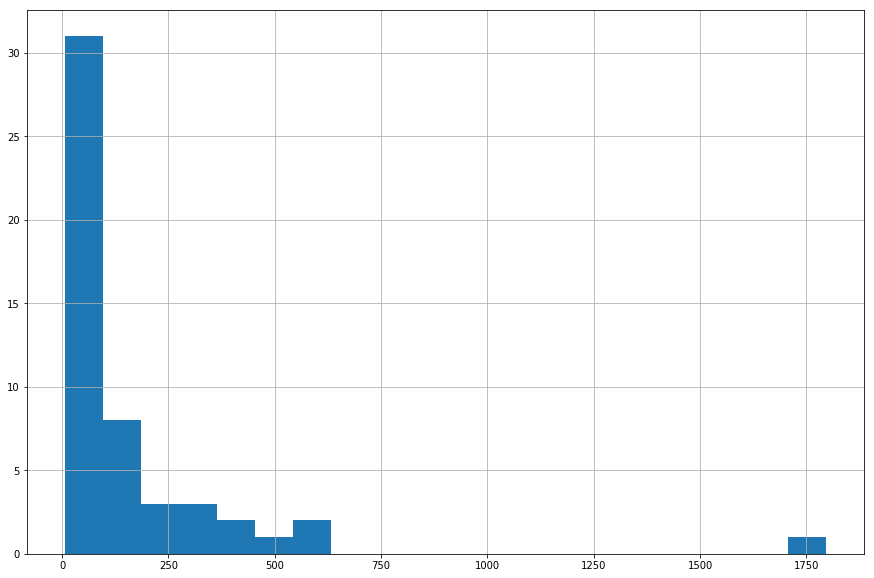
**Data Distribution**

Here’s the target distribution on training dataset revealing that it’s pretty imbalanced with a peak at intention=28.

Target distribution (number of classes = 51)



Most of labels are associated to pretty small number of samples: half of classes have less than 100 rows.



Regardless of the ML algorithms and smart feature engineering, achieving a good enough classification rate is really tricky considering the small training size and imbalanced distribution.

Let’s try to guess intuitively the hidden semantic of the most frequent labels:

* **first mode**: intention=28

It looks like to be related to questions on drug side effects and somehow the contraindications



* **second mode**: intention=31

it concerns questions on symptom-drug adequacy/efficiency



* **multi-topic class**: intention=39

the commonality across text samples seems to the presence of multiple question mark tokens (counting it may be a good option to predict mult-topic label)



The painpoint of the text classification problem is to find a way to estimate the intent similarity between 2 questions written in a natural language.

Let’s focus on 2 questions having the same label (symptom-drug adequacy):

* “épilepsie et havlane?”
* “mon medecin me soigne pour une rhino pharingite et m'a prescrit du amoxicilline comme anti biotique. Est-ce vraiment pour cette indication?”

The writing style differ significantly: on one hand, a very concise expression putting the symptom entity and the drug entity in an adversarial fashion, on the other, the question is more detailed with one sentence to set up the context/fact and the second one to raise the concrete question of adequacy between the 2 entities mentioned previously.

This typically illustrates the stylish complexity and diversity of the human language to convey an idea and more particularly a high level topic!

Moreover, notice that certain questions are lexically and syntactically incorrect: words are misspelled especially when dealing with drug product names which are unfamiliar for most of non-professional persons.

To have a glimpse on the classification difficulty, it’s common to perform some visualizations of the feature space distribution.

The multi-sentence question text is basically converted into of bag of words which is vectorized with TF-IDF transformation. Each question is then represented as a data point within the global vocabulary space. To make such data points humanly observable, a dimension reduction of these features is needed at the cost of some approximations: we will use both linear/fast PCA dimension reduction technique and non-linear/slow t-SNE. The data point color is determined by the associate target class.

**PCA-reduced feature space**



**t-SNE-reduced feature space**



Both shapes are very different but as suspected, we can observe intuitively from both visualizations that the decision boundaries are unclear in the original feature space, just by considering the occurrence of words. To get a chance to obtain a better classification performance, it’s obvious that the raw text needs to be encoded more smartly into a suitable space where the semantic proximity between documents is prevailing.

**Text Anatomy Analysis**

Todo

Vocab / misspelling

Statistics on text structure (sentence, …)

Context then question

Interesting entities: drug name, quantity, time, what/when/…

***General NLP Architecture***

Text classification is a common but non-trivial NLP topic going through the following main steps:



This is the general NLP text classifier framework/guidance but for practical reasons, some processing steps are skipped or significantly simplified to fit the project timeframe but also because of the lack of French language support.

In fact, here’s the concrete pipeline I built per technical scenario:



Each processing unit will be described more precisely in the next sections.

The overall modeling procedure should capture the sequential nature of the text to exploit efficiently the contextual information: typically, the feature representation should preserve the word/symbol order and the classification process should be based on **sequence modeling**.

***Text Preprocessing*** *(common trunk)*



It’s all about operations on the raw text to make it more reliable/workable in order to extract relevant characteristics. It falls into 3 categories:

* tokenization which breaks down the sentence into a sequence of atomic word
* spelling correction which fixes as far as possible words which are misspelled
* lexical and grammar tagging which basically decorates the text tokens with metadata
* text cleansing and normalization which simplify the sentence composition

**Tokenization**

This operation is very common and it doesn’t raise significant issue: I simply use the python string split() function. I tokenized the whole document regardless of the sentence split with punctuation like “.”,”:”,”;”, “!” and “?”.

**Spelling correction** (Syntactical processing)

I assumed as misspelled all words which don’t belong to any trustworthy vocabularies, also known as **OOV** (Out Of Vocabulary) word.

I retained 3 reference vocabularies in the following priority order:

* vocabulary from the word embedding model used downstream in the processing pipeline
  + indeed, it’s very important to avoid random vectorization on OOV words
* Custom vocabulary to capture the specific drug domain, typically on drug product and active ingredient entities where misspelling is frequent. It has been built from the public RCP (Résumé des Caractéristiques du Produit) repository supplied by ANSM.
* Predefined general purpose vocabulary from the github python project pysspellchecker <https://github.com/barrust/pyspellchecker>

Spelling correction basis is to find from a set of reliable vocabularies, the closer word candidate from **Levenhstein** distance standpoint: this distance measures the minimum number of character level operations (change, remove, add) required to map 2 words.

I defined an arbitrary threshold to accept the closest word as a fix on the misspelled word: the ratio between the number of atomic operations and the total number of characters should be under 25%.

I applied this correction method with the last 2 vocabularies: the first vocabulary layer only filters out the recognized words, the unfixed words at the second layer are then passed to the third layer.

Here’s the python output showing the fix on more than 400 drug product names with a reasonable error rate (~ 15%):



For the active ingredient, only 25 fixes have been detected.



The general vocabulary fixes up 430 words with relative high error rate (~25%): as accent encoding is badly handled by pyspellchecker module, I fixed it manually afterwards.



At the very end, it remains 583 unfixed words over an intial2108 unknown words (25%): that corresponds to hard cases where the word is unexpectedly a concatenation of multiple word or transcribed phonetically.



Below diagram illustrates the different python notebooks (parallelogram in yellow) necessary to fix word misspelling: the blue folder represents the file consumed or produced by the python processing unit.

****

**Lexical and Grammar Tagging**

**NER** is a NLP process to tag text token with predefined semantical categories (location, person, quantity, …), enabling to count entities as explanatory feature. Typically, distinct drug product counting may be a discriminating feature to predict labels corresponding to question on drug interaction.



**POS (Part Of Speech)** tagging is a process to markup text tokens with lexical categories (noun, adjective, verb, …), enabling typically to compute tag frequency distribution as explanatory feature.



**Dependency parsing** is a more sophisticated process than POS tagging to discover the grammatical dependencies between words within a sentence. It produces an annotated dependency tree which reveals the nature of the interaction between the words.



I finally didn’t employ these advanced tagging methods because French language is not well supported by most of NLP packages (Spacy, Standorf NLP or NLTK). The only basic NER I put in place is to locate the drug name or active ingredient entities based on the list of words extracted from the RCP repository: such entities are central and their identification within the sentence will be used later on to create additional features.

**Text Cleansing and Normalization**

I implemented some ad-hoc cleansing/normalization rules based on regular expression to tackle special characters, repetitive number, punctuation characters or usual acronyms.



In a second time, for regular words, stopWords eliminates semantically irrelevant and frequent tokens whereas stemming/lemmatization reduces the morphological variants into their etymological root.

For this job, I utilized the NLTK package: this process simplifies gracefully the phrase structure but at the expense of its lexical and grammar correctness.

The stemming/lemmatization preprocessing is counterproductive to word embedding model learnt from corpora which haven’t been stemmed or lemmatized as prerequisite: they are mutually incompatible. For DL scenario, I made use of word embedding excluding de facto this root normalization.

***Classical Technique***

**Abstract**

**HMM** (Hidden Markov Model) is a probabilistic and transitional graph modeling which is actually suitable to tackle sequential structure like sentence. Typically, it’s capable to learn on text corpus and predict POS tags: some experiments have been conducted to address text categorization problem with quite good result. When looking at HMM python implementation, there’s no strong and active community working on it.

The arguable fallback is to use instead non-parametric statistical inference method like SVM, decision true and so on, with the significant downside to lose the native support of sequence modeling. To compensate slightly such discarding, the feature extraction/enrichment would include some handmade tricks trying to grasp some contextual information from the word sequence.

***Feature Enrichment***

This step adds a-priori extra features which may discriminate the label much more than the original features: they can be calculated from the text or can originate from external sources**.**

I incorporated above basic statistics giving insights on the text structure and composition:

* count of sentences
* count of words
* distinct count of drug name entities
* distinct count of active ingredient entities
* count of question marks (typically to identify specifically multi-intent label)
* individual count of interrogative pronoun entities (one column per pronoun: quand, qui, quoi, ou, comment, pourquoi, combien, quel(s|le,..)
* distinct count of time entities (eg: jours, après midi, soir, année, 12h, mardi, samedi, temps....)
* distinct count of quantity entities (eg: 5mg, 10ml, ...)
* count of association entities (eg: et, avec, ou, ...)
* distance between interrogative pronoun and drug name entities
* distance between active ingredient and drug name entities
* distance between quantity and drug name entities
* distance between time and drug name entities
* distance between question marks and drug name entities

They are either **count-based or distance-based statistics**: distance variant is intended to catch native the word context by measuring the relative distance between key entities. This computation needs to put in place the domain-based (list of distinct values) or custom regular expression NER (Named Entity Recognition) to identify the key entities in consideration.

An extra calculated column is added to the train data frame per statistics as below:



If the text sample has identified drug product entities, it’s valuable to extend the primary feature vector with relevant information related to these drug products.

I illustrate below a specialized **knowledge sub graph** centered around the drug product entities with some interesting relationships to other entities (quantity, human body part , …).



Indeed, such related entities characterize well the drug product and they can improve the detection of the commonality between texts sharing same target label: for instance, a drug product class (eg: antidepressant family) may raise particular questions.

Unfortunately, this knowledge graph is not available publicly and should be built by our own: the ANSM provides online the full description of the drug usage indication in HTML format. Such resource can feed a learning system to extract above salient related entities.

I didn’t implement this information extraction from ANSM source because it’s a huge workload which is incompatible with the project scope.

**Feature Representation**

The document (composed of sentences) should be converted into numerical vector because most of ML classifiers can only cope with numeric values and they don’t care about symbol and semantic conveyed by the text entity.

First basic solution is the **BOW** (Bag Of Words) representation where each word of the vocabulary is defined in column and the text in row: the cell value stores the word frequency.

I didn’t consider **n-gram** document representation because as specified earlier, the classical technique scenario doesn’t employ sequence modeling like HMM which is able to treat n-gram structure.

The problem of the **BOW** representation is that rare term which in general discriminates well the document are under estimated in regards with commonly used but irrelevant terms (eg: generic verb, …).

**TF-IDF** (**T**erm **F**requency **I**nverted **D**ocument **F**requency) overcomes this pitfall by overweighting terms which are identified as rare for a given corpus.

The shortcoming is that such vectorization generates a very high dimensional space depending on the vocabulary size. We fall into the well-known **curse of dimensionality** where data distribution is extremely sparse making classification task inefficient when training size is too short.

The space dimension should be reduced consequently:

* stop words and stemming processes already reduce upfront the vocabulary size
* I applied the **PCA** (Principal Component Analysis) linear dimension reduction which keeps the top eigen vectors capturing the maximum of the data distribution variance: PCA is a process which is totally semantic unaware in contrary to word embedding I will tackle later on

TF-IDF application and PCA reduction produce a low dimensional numerical vector per document as below:



**Classification Modeling**

The classifier takes as input a feature space combining the reduced BOW representation and the handcrafted statistics:



I bet on the **XGBoost** classifier delivering excellent accuracy in a reasonable time (it’s multi-thread friendly): XGBoost is based on boosting ensemble technique combining sequentially weak classifiers (in general decision tree) where at each iteration, the weighting on incorrected classified observations is increased to enforce the next classifier to focus its attention on feature sub space with high error.

XGBoost comes up with many hyper-parameters to tune: an inappropriate selection usually leads to suboptimal model.

I followed the standard methodology and best practices:

* find out the optimal hyper-parameters by testing different selective combinations. I retained the one delivering the best accuracy on unseen dataset (validation) with **cross validation** enable as training is very small
* fit the final model with the above fixed hyper-parameters on the whole training and assess the generalization error on test

Here’s the learning pipeline for the classical technique track:



I focused my attention on the following parameters which are the most instrumental to the final accuracy:

* **max\_depth**
  + this parameter drives the decision tree complexity to partition the feature space
  + a low value usually prevents from overfitting and favor the weak learner synergy
  + I tested empirically 3 values: 4 , 6, 8
* **min\_child\_weight**
  + under the threshold, the learner stops splitting and generates a leaf node
  + it controls as well the tree complexity and consequently the overfitting
  + I tested empirically 3 values: 2, 5, 10
* **n\_estimators**
  + this parameter sets the maximum number of stacked trees
  + I fixed it empirically to 100
* **early stopping**
  + the number of trees defined above can be capped by the early stopping criterion
  + it defines the maximum number of failing attempts to improve the accuracy on validation
  + I fixed it empirically to 10
* **learning\_rate (eta)**
  + it controls an important parameter of the gradient descent optimizer
  + I tested empirically 2 values: 0.05, 0.1
* **cross validation fold**
  + cross validation ensures a more reliable generalization error indicator which is not biased by a particular split (test set). It’s valuable typically in imbalanced label or small dataset situation (it’s the data challenge case)
  + I fixed it empirically to 5

Other parameters settings rely on the XGBoost defaulting to avoid excessive processing time caused by the grid search combinatory explosion: by crossing max\_depth, min\_child\_weight and cross validation fold, it represents 72 (3x3x4x2) learning units to reveal the optimal parameter values.

**Result Analysis**

***Deep Learning Technique***

**Abstract**

DL is considered as a universal estimator able of fulfilling any kind of modeling requirements from the feature representation layer to the final decision layer inside a single (and complex) neural network. The key benefit of this holistic capacity is that the loss function optimization to find out the modeling parameters (weights, …) operates consistently across all functional layers regardless of their respective purpose (embedding, decision making, …). In contrast, with traditional method, feature representation and classification are 2 concerns which are engineered/optimized separately.

Sequence modeling

**Feature Enrichment**

No enrichment

**Feature Representation**

Doc or Sentence Embeddings Mean? (eg: Mean Word Embedding)

The **word embedding** addresses this matter indirectly: this method estimates statistically from a large text corpus the co-occurrence between 2 words in a text context. This vectorization process makes that 2 words which are contextually similar (inter-changeable) are represented as 2 close numerical vectors in the target space.

**GloVe** (Global Vectors for Word Representation) implements the word embedding in a non-DL fashion (conversely to **Word2Vec**): it uses behind the scene matrix factorization method such as LSA (Latent Semantic Analysis).

Another more advanced option is to keep the sequence nature of the text and each word is represented by a k-dimensional numerical vector: a text with n words is shaped as a n x k matrix. k can be the vocabulary size running the risk to fall into the high dimensionality curse as vocabulary size is frequently > 10000 even with the stemming or lemmatization preprocessing.

A popular dimension reduction method namely word embedding enables to overcome this issue: an unsupervised model is learnt from a large text corpora to optimize a lower dimensional vector representation where words which share similar context (within a sentence) are close to each other. The wonder of this technique is that vector proximity is governed by semantic similarity. Word embedding is implemented in a DL flavor (Word2Vec) and in a non-DL way too (GloVe project).

Word embedding implementation is available in a DL flavor with **Word2Vec** (Google) and more recently with **FastText** (Facebook): both of them are issued from Mikolov’s research work.

My first thought was to build the embedding model from POSOS training set which contains domain specific terms (drug name like Xanax, active chemical ingredient, …), but the corpus is really too small and badly trustable (misspelled terms, familiar expression, …).

It’s preferable to take advantage of the embedding models learnt from very large and diverse text corpus, which are publicly available from the Web. I only found French trustable models for FastText which in addition offers the capacity to cope with **OOV** (Out Of Vocabulary) case: indeed the embedding model is trained at character level instead of word level.

I experimented both embedding model sources and the general purpose one provided definitively better classification performance (roughly 20% difference in term of classification rate).

The best of breed solution would be to extend the general embedding model by incorporating the pharmaceutical target corpus constituted by the training set and other reliable sources (ANSM, …). This is another and bigger story requiring huge amount of computational resources to rebuild this super model. Even the learning transfer in lieu of an embedding training from scratch is also complicated and out of reach in the context of this study project.

I finally used the French FastText model encoding words into a 300 dimensions space. Medical terms referred in the question texts are either drug product names, special symptoms or active chemical ingredients: such key words are probably absent from the FastText model’s corpus.

How to solve this predominant OOV (Out Of Vocabulary) matter?

One solution is to merely project such unknown words into its **hypernym** more common term with loss of details penalizing indirectly the final classification performance:

* drug product name (Xanax, Abboticine) is converted into constant term ‘médicament’
* active ingredient (eg: Acabavir) is converted into constant term ‘ingrédient’
* …

This projection into hypernym/entity space is possible only if the terms are correctly tagged in the text preprocessing phase (name entity recognition). It’s likely smarter than the common practice to encode unknown words into a random or fixed vector.

“These word embeddings are now the state-of-the-art in NLP. However, it is less clear how we should best represent a sequence of words. a whole sentence, which has complicated syntactic and semantic relations”

local and long-range dependencies

Variable sized vs fixed sized (padding)

**Architecture Choices**

A possible (but simple) architecture is merely composed of a **Doc2Vec** embedding layer plus a fully connected layer for the classification task.

In practice, **CNN** (Convolutional Neural Network) and **RNN** (Recurrent Neural Network) layouts appear to be predominantly chosen in ML competition for text classification: some practitioners/researchers even recommend combining them as they are complementary.

The hybrid solution mixing up CNN and RNN is not considered here for simplicity sake.

**Doc2Vec based Architecture**

This network has 3 main hidden layers:

* Doc2Vec vectorizes the whole multi-sentence question into a single numerical vector
* A series of fully-connected/dense layer plus dropout layer
  + the activation function would be the sigmoid as we are dealing with shallow network
  + the dropout operations ensure the learning regularization to avoid the overfitting pitfall
* last dense layer with softmax as activation function to produce normalized scores we can interpreted as classification probability

Identically to Word2Vec, it’s highly preferable that the embedding model is trained on a very large and representative corpus but I didn’t find any pre-trained public Doc2Vec models from the DL community. The consequence is that I built my own Doc2Vec model from the pre-processed training and test sets of the data challenge.

The primary drawback of the Doc2Vec approach is that in contrast to Word2Vec, we lost the word sequence after the encoding, so that it’s impossible afterwards to model the document semantic based the context defined by the word sequence.

This proposition sounds very naïve but it can serve as a comparison baseline to illustrate the value proposition of more sophisticated CNN and RNN architectures which are capable to extract the context from the sequence.

**CNN Architecture**

In **C**omputer **V**ision (**CV**), convolution operation is well known to be efficient to extract the high level representation of an image by passing a sliding window filter and computing consequently an average value for each filter position as output. These convoluted values are then activated with usual non-linear function and down-sampled thanks to the pooling layer. This pixel-wise processing is inspired by how the visual cortex analyzes the signal sent by the eye receptors.

Surprisingly, such biological inspiration also works well to capture the high level structure of word sequence in NLP. The convolution operates in a 1-dimensional array (word sequence) instead of 2D (pixel matrix) in CV. The sequential filter enforces the neural network to focus its attention on local context which establishes connection between words.

Some research studies demonstrate that convolution is powerful enough to cover embedding requirements making Word2Vec useless. Nevertheless, I preferred to keep going with an embedding layer relying on a proven algorithm (Word2Vec) associated with a convolution layer specialized to extract the high level representation of the text.

In short, here’s the neural network layout:

* word embedding layer as explained in the Feature Representation section
* a series of 1-dimensional convolutional and pooling layers
  + The best practice recommends building many convolutional layers with different filter sizes and/or strides
  + I used Relu (Rectified linear unit) activation function after the convolution operation and the popular maxpooling
* a series of dense layer plus dropout layer
* last dense layer with softmax as activation function

**RNN**

In a classical neural network, we assume that all inputs of a node are independent to each other but it’s not applicable in NLP because text is composed of an ordered sequence of words (inputs) which are inter-related.

Recurrent neural network architecture tries to leverage this sequential information with a particular layout where each layer at the i-th position is fed with the i-th element of the input sequence and the output of the direct preceding layer: each layer captures somehow the hidden state (memory) of the preceding sub sequence of inputs (words).

RNN can be **bi-directional** (instead of forward only) where the i-th layer also depends on the computational output of the direct successor (i+1 th position).



*(source: Wikipedia)*

In practice, the simple layer computational unit (“h” blue box in above diagram) exhibits inability to catch long term dependencies to distant input x(t-n) where n is significant high, due to the vanishing gradient problem.

The **LSTM** (**L**ong **S**hort **T**erm **M**emory) cell unit has been invented to bypass this long-tailed sequential dependency we typically observe in multi-sentence text analysis where the context may be specified upfront far from the concerned word.



*(source: Wikipedia)*

This processing unit adds a secondary flow (upper stream) to update gradually the memory (cell state denoted as C(t)) with contributions controlled by several input gates (shown at the bottom).

The final architecture looks like below:

* word embedding layer as explained in the Feature Representation section
* bi-directional LSTM layer with dropout (for learning regularization)
* dense layer with softmax as activation function

**Implementation and Execution**

I coded all DL experiments with **Keras** as a frontline API on top of Tensorflow engine: Keras provides a very comprehensive and easy-to-use API hiding the Tensforflow verbosity.

Here’s an example of Keras code where the layers are built in a very concise manner thanks to helper object like Convolution2D, Activaction and so forth.

model = Sequential()

model.add(Convolution2D(32, kernel\_size=(3, 3),padding='same',input\_shape=(3 , 100, 100)))

model.add(Activation('relu'))

model.add(Convolution2D(64, (3, 3)))

model.add(Activation('relu'))

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Dropout(0.25))

In general, the model performance depends importantly on the selection of the hyper-parameter values: as the DL training is really slow even with the aid of GPU boosting, I was not able to test a lot of hyper-parameter selections and I proceeded with default or best guess values at the risk to be far from the optimum point.



**Common Setup**

GPU enable config

**Doc2Vec based Implementation**

**CNN Implementation**

**RNN Implementation**

CUDNNLTSM

**Experiment Results**

Nb params to optimize vs nb observation (overfitting) => show error estimation vs validation evolution

***Comparison Study***

Graph to summarize candidate (XGBoost / RNN / CNN / Custom embedding

**Implementation/Engineering**

**Model Accuracy**

Focus on dominant class error

***Improvement Tracks***

**Text PreProcessing**

To combat the training size shortage impose by the data challenge, one option to enrich the former dataset is to complement it with other medication questions from public forum like the one hosted by doctissimo.fr. These extra observations are unlabeled (the 51 target classes are anonymized) and so cannot be used as regular dataset for the supervised training. Nevertheless, it can be consumed to complement gracefully the text corpus for all unsupervised learnings (eg: Word2Vec) to strengthen the modeling result. Such trick is also applicable for unlabeled test dataset of the challenge.

**Classical Technique**

**Deep Learning Technique**

**Feature Engineering**

Building a **custom** **embedding model** from text corpora which are specific to the data challenge domain is a real plus: indeed, the generic FastText model has been trained on clean corpora writing in an elevated literary style by professionals (Wikipedia, newspaper, …) whereas the POSOS text style is much more familiar as questions are mostly raised by uninitiated people.

As POSOS training size is not sufficient, it can be completed with other unlabeled drug question sources (eg: doctissimo.fr forum, …): just to remind that embedding is an unsupervised learning process requiring no exhaustive manual labeling.

Another improvement possibility is to refine the proposed **imputation** that replaces words unknown from the embedding model standpoint, by its named entity (hypernym). Typically, train our own NER (Named Entity Recognition) on French general/domain-specific corpora combination with NLTK package: the trick here is to take advantage of the English NER model provided by Stanford NLP active community and the English-to-French word translation, in order to get for free the global standard label on French corpora.

ANSM source of information on drug product is poorly exploited: I only extracted vocabulary on drug product names and active ingredients to identify such entities into the text because it’s straight-forward (ANSM public dataset contains a drug product name and active ingredients columns). Other relevant named entities (adverse effect, target body part, drug category, …) should be learnt from the ANSM corpus with manual entity labeling.

For the handcrafted features to characterize the text, thanks to a more complete entity identification (adverse effect, body part, quantity, …) , it would be possible to compute more count-based statistics on entity and even introduce selective entity distance to capture indirectly the word sequential distribution.

**Classification Modeling**

For the classification modeling layer, there’s room for improvement as well. Due to lack of time and computing resource shortage, no extensive hyper-parameters search has been performed mainly in the DL sub project: we know for sure that parameter optimization is crucial to achieve good results.

I intentionally separated the DL and non-DL techniques in order to make an academic comparison study but in practice, the state-of-art in ML competition is to build heterogeneous and complex predictive pipeline involving a variety of algorithms and technologies, by sticking to the ensemble learning principle. The DL architecture may be reinforced with handcrafted features utilized in the non-DL approach. In the same vein, CNN and RNN architectures are tested separately but they can be combined nicely into a single neural network to obtain the best of breed learning system.

***Conclusion***

In this present paper, I describe a walkthrough return of experience on text classification problem in a **DL (D**eep **L**earning) and non-DL fashion (a.k.a. traditional method). The goal of the study is to have a fair idea on the pro and cons between above approaches from a practical standpoint on a real use case.

The non-DL scenario enrolling notably the favorite XGBoost classifier delivers poor modeling performance (X % accuracy in cross validation) mainly due to its inability to leverage natively the sequential context from the text structure: this solution is furthermore penalized by the absence of pre-trained French word embedding model based on non-DL principle (eg: GloVe matrix factorization) and not surprisingly, the handcrafted statistics on text don’t improve at all the final accuracy. Nevertheless, its implementation and execution are pretty straight/simple accommodating with mid-range CPU.

DL option with RNN/LSTM architecture outperforms the non-DL candidate in term of accuracy (X % on validation set), partly thanks to the available pre-trained FastText embedding model and its sequential structure awareness. The counterpart is that DL algorithm is hungry of GPU and badly runs on standard machine configuration: on top of that, elaborating an appropriate architecture (operation composition) and tuning the hyper parameters (activation function, dropout rate, …) demand some strong practical engineering experiences to achieve good enough performance.

In addition to the probable suboptimal ML procedure I had built so far, the low accuracy on validation can also be justified by the learning materials: indeed, the training set is really too small (~8000) with regards to the 51 labels and most of the texts are written in an everyday language style with plenty of abbreviations and misspellings.

Even if the comparison is totally unfair/biased, Deep Learning turns out to be the most efficient ML technique in NLP task solving: it’s presently a very active research topic within the ML community as DL expressiveness/capacity are unbound similarly to the brain plasticity.

This short study with non-conclusive performance result, has the educational benefit to make me practice on a large spectrum of domains that a data science should master: from the data analysis to experimental result debriefing, unsupervised (word embedding) vs supervised (classifier), experiment implementation and execution (GPU activation, python programming, AWS computing infrastructure, …), text pre processing techniques, concept of sequence modeling, Deep learning architecture variety and general best practice on classification task (cross validation, early stopping, …).

**Annex**

**Github project**

The project is available fromthe public github repository at the following URL:

<https://github.com/jhuu32/CES>

It contains Jupyter notebooks allowing to reproduce all learning experiments mentioned in this report: the only missing artifact is the FastText embedding model which is too large to be pushed into github (2Gb), but the README.md gives the necessary information to download it.

Here’s the high level project source tree



**References**

todo