ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Международный институт экономики и финансов

Курсовая работа

по теме:

NATURAL LANGUAGE PROCESSING WITH DEEP LEARNING

Выполнил

Студент 3 курса 1 группы

Исхаков А.А.

Научный руководитель

Демешев Б.Б.

Москва

2017

National Research University
Higher School of Economics

International College of Economics and Finance

Mathematics and Economics

Course paper
— *Natural language processing with deep learning* —

Student
Aydar Iskhacov
GROUP 1

Research adviser
Boris Borisovich
Demeshev

HSE, May 31

mail: iskhacov.aydar@gmail.com

Contents

1 Introduction

1.1 What is AI?

One of the latest themes highly debated in the academic sphere is the recent development of AI [1] types of systems and algorithms which significantly change tools and approaches used to analyse the data. What is actually meant by the mysterious word AI is the machine and deep learning algorithms used to analyse various type of data. In fact most of these algorithms were theoretically developed in late 1990-s, however machine and deep learning algorithms are computationally heavy, which restricted their development. Recent development of computer hardware has blazed a trail for ubiquitous propagation for virtually all spheres of human life.

1.2 Why do we need AI?

This theme is of paramount importance not only due to its scientific applicability but also for its wide practical usage in all spheres of human life. One on the striking examples of deep learning usage is application of neural networks to breast cancer detection with the astonishing results: 100% of women who were the most exposed to this type disease were identified correctly [Menéndez et al., 2010]. Further, machine-learning algorithms are widely used in economics and finance. In fact, as it was demonstrated by Nick Polson et al., one of the professors of Econometrics and Statistics of University of Chicago, "deep learning has the potential to improve – sometimes dramatically – on predictive performance" of financial and economic models[Heaton et al., 2016]. This again proves the importance of the topic and it's practical implication for various tasks.
Among the most widely used practical implications of machine learning algorithms are image, voice and text recognition and analysis systems. Apparently, this might be explained by inapplicability of traditional tools for analysis of these types of information.

1.3 What is NLP?

This course paper, in particular, will explore one of the most interesting and rapidly developing parts of deep learning algorithms application – Natural language processing (NLP). Natural language processing is a field of computer science, which explores algorithms for processing natural language corpora, establishing interaction between computer and humans. The reasons of such interest to the field is the diversity and complexity of elementary units (for instance, words) and the corresponding structures (for instance, sentences or articles) formed by an uncountable combinations of those elements. Further, human interactions involves many different tasks which significantly diversifies the subject, which ranges from some simple assignments as spell checking to complex non-algorithmic problems involving semantic analysis, machine translating, question answering systems.

[1] artificial intelligence

1.4 Main goals of the paper

Having discussed the importance of the topic I would like to set up some goals of this paper work:

- First and foremost goal is, indubitably, to explore the theory behind the main deep learning algorithms
- Second aim is to study machine learning techniques application to the natural language processing subject
- Third and the most important objective of this work is to apply the knowledge to real-life problem

1.5 Basic materials used

To achieve proposed goals this work will use wide range of different materials with the primary accent on Stanford's course: "CS224n: Natural Language Processing with Deep Learning"[Sta, 2017] held by Professor Christopher Manning and Dr. Richard Socher. Further, this work will cite some immensely important research papers, such as S. Hochreiter's and J. Schmidhuber's work "LSTM can solve hard long time lag problems"[Hochreiter and Schmidhuber, 1997]. For XGBoost theory this paper will analyse the article discussing primary differences of this algorithm from other gradient boosting techniqes[Chen and Guestrin, 2016]. Since one of the most crucial goals of this paper is practical application of theoretical implications the second part of the paper will explore an example of real-life usage of deep-learning algorithms. In particular, there will be a case study of one of the Kaggle competitions[Kag, 2017] called: "Quora Question Pairs"[QQP, 2017]. The aim of this competition is identification of duplicate questions, which will be a direct application of theoretical knowledge explored earlier.

2 Natural language at its finest

2.1 What is language?

The word language itself and the phenomena hiding behind this term is rather complex and has many different connotations. For instance, Asif Agha, Professor of Anthropology at the University of Pennsylvania, gives at least three definitions of the word language: "kinds of phenomena to which we ordinarily refer by means of words like French, Arabic, or Tagalog", "phenomena reflecting sign-functions", "grammar" [Agha, 2007]. All of these designations can be fairly attributed to the language with one utterly compelling and unifying characterisation – language in its essence is a complex system of communication. Being the key channel of human interactions the language has evolved to one of the most powerful and complicated structures which is virtually impossible to algorithmize due to several reasons. The key challenges are:

- Depending on the language, each word may take different forms depending on a gender, position in the text etc. The key idea here is that internal meaning does not usually change with change of form, hence it becomes difficult to discriminate between the meanings of one word.
- Large corpus of words and related particles, which form an infinite amount of combinations. The problem becomes even worse since not all combinations form meaningful statements.
- Further, the language corpus is not something that is fixed, on the contrary, it is constantly evolving with new words, or as they called neologisms, added every day.
- Presumably, one of the most challenging things is the existence of several meanings of one word.

Fortunately, recent developments in Natural Language processing systems has allowed solving part of the challenges presented.

2.2 Text cleaning

One of the first steps, which starts almost any NLP problem, is text cleaning. It is vitally important procedure due to the several reasons. First, many human-written everyday texts include many clippings, abbreviations that will be misunderstood by the computer. Further, as it has been discussed earlier one word may have different forms. For instance, the word "be" has seven additional forms: "be, being, been, am, is, are, was, were", which, in fact, have resemblant semantic meaning. This problem is usually approached with two main tools: stemming and lemmatization. The first solution refers to cutting off the end of the word, which often implies removal of "derivational affixes". A common example would be change of word "analysis" to "analys", reduction of word "easily" to "easili" or "eas" depending on the algorithm used. It should be stated here that there is a number of different approaches to reduce the affix. The most classical one is "The Porter Stemming Algorithm" [Porter, 1980]. This paper will not discuss the mechanics of the algorithm in details, however the general idea of "Porter's stemmer" [2] is to find some similarities between stems of the words and possible suffixes[3] and basing in those rules change the longest suffix S1 on S2. As an example, for the word "Caresses" the longest S1 found "sses" will be changed on S2 "ss", with the word changing from "caresses" to "caress"[Porter, 1980].
Depending on the set of rules applies there different variations of the algorithm. The illustration of the differences can be seen from the following figure.

[2]In relation to the English
[3]For instance, the stem "ends on word S" or the stem "ends with double consonant"

Sample text: Such an analysis can reveal features that are not easily visible from the variations in the individual genes and can lead to a picture of expression that is more biologically transparent and accessible to interpretation

Lovins stemmer: such an analys can reve featur that ar not eas vis from th vari in th individu gen and can lead to a pictur of expres that is mor biolog transpar and acces to interpres

Porter stemmer: such an analysi can reveal featur that ar not easili visibl from the variat in the individu gene and can lead to a pictur of express that is more biolog transpar and access to interpret

Paice stemmer: such an analys can rev feat that are not easy vis from the vary in the individ gen and can lead to a pict of express that is mor biolog transp and access to interpret

Figure 1: A comparison of three stemming algorithms on a sample of text[ste, 2017].

Despite the relative easiness and clearness of the algorithm, it has some serious drawbacks. In fact, the most critical shortcoming of this family of stemmers is the absence of the link between language corpus and the algorithm itself. This in turn, will lead to the common mistakes like mapping word "argument" to "argum" or inability to map word forms of word "be" to itself. The solution to this problem is lemmatization, which is widely used nowadays, especially for languages "with much more morphology (such as Spanish, German, and Finnish) [ste, 2017]".
Lemmatization is a bit more complex problem and there are a number of different approaches to it. One of common approaches to use so-called Ripple Down Rule [Plisson et al., 2004]. First thing to state here, that it is usually implemented as a machine learning problem with a new rule added iteratively to define what the suffix is. The approach used to divide the word and the possible set of affixes is the following: calculate the length of the word and take all sequences from the end to (n-1) with the sequence of length representing the word itself. For instance, if one takes "working" as an example the possible set of words would be "g", "ng", "ing", "king", "rking", "orking". Then the word "working" would be assigned a label representing a class of transformation of type "change suffix S1 to S2", which in our case would be a simple label "change "ing" to nothing". The possible problem here is that one rule may contradict to the other[4], that is the place where ripple down approach (RDA) solutions helps to resolve the problem. The essence of RDA is to create a rule in such a way that it will not affect other rules. This result is attained through creation of the structure:
If *(condition - 1)* then *(A)*
Except If *(condition - 2)* then *(B)*
Else if *(condition - 3)* then *(C)*
As a result of the following approach, one gets data structure which is similar to decision tree with each nodes has exactly successors, namely "Else" and "Except".
A word of caution must be added here: there is a mixed empirical evidence on the applicability of the following text cleaning approaches[ste, 2017]:

1. As it has been already stated these approaches are especially useful for languages with more

[4]For instance, the suffix "ment" may be a part of the lemma, such as for word "argument" or might be an affix which have to be deleted such as the word "excitement"

variation in morphology, such as Russian or German.

2. This approach is useful for feature creation and semantic analysis as a forms of the word usually have the same meaning, hence lemmatization is vitally important to get an overall meaning of the sentence.
3. There is a loss of precision in some cases, for instance, consider an example described in Stanford tutorials[ste, 2017]
   - *operative* and dentistry
   - *operational* and research
   - *operating* and system

   Stemming the word *operating* would lead to the loss of some information, hence one should be careful with usage of those instruments.

2.3 Semantic analysis

One of the most fascinating and impressive fields of the Natural Language processing field is the semantic analysis, which is by definition "a process of relating the syntactic structures to their language independent internal meaning". The reason for that is the ability of the computers to capture complex structures and peculiarities of the language from rather simple and lucid concepts. Indubitably, it is a little exaggerative to argue that machines are able to gain internal meaning of the terms, however an ability to capture similarity of words from simply reading text is absolutely astonishing[Landauer and Dumais, 1997].
The essence of computer semantic analysis is the translation of the words into a set of number, represented by a vector. There were two main approaches for vectoring:

- Global matrix factorisation methods, such as Hyperspace Analogue to Language (HAL), Latent Semantic Analysis (LSA), COALS etc.
- Local context windows models, represented by Word2Vec model, which in fact includes two main algorithms – continuous bag-of-word (CBOW) and skip-gram models

2.4 Global matrix factorisation models

The basis of the global matrix factorisation approach is a whole corpus statistics of word co-occurrences, which practically means that this method exploits global statistical regularities. The latter is a crucial advantage as this approach exploits the language wide structures and repeats.
The practical realisation of the model usually includes several steps:

1. A global matrix, which depending on the type of model may either be of "Word-document" [5] or "Word-Word" [6]. The idea is to calculate how many times a certain word - i appears either in context of a document j or another word j depending on the type of the model used. For concreteness, consider "Word-Word" type of model with only three "documents" in the corpus:
   (a) I like deep learning.
   (b) I am writing a course-paper.
   (c) I like chocolate.

[5] For instance, Latent Semantic Analysis (LSA) [Deerwester et al., 1990]
[6] For instance, for Latent Semantic Analysis (LSA) [Deerwester et al., 1990]

Then taking for simplicity the "windows"[7] one will get the following matrix:

	I	**like**	**deep**	**learning**	**am**	**writing**	**a**	**course-paper**	**chocolate**	**.**
I	0	2	0	0	1	0	0	0	0	0
like	2	0	1	0	0	0	0	0	1	0
deep	0	1	0	1	0	0	0	0	0	0
learning	0	0	1	0	0	0	0	0	0	1
am	1	0	0	0	0	1	0	0	0	0
writing	0	0	0	0	1	0	1	0	0	0
a	0	0	0	0	0	1	0	1	0	0
course-paper	0	0	0	0	0	0	1	0	0	1
chocolate	0	1	0	0	0	0	0	0	0	1
.	0	0	0	1	0	0	0	1	1	0

Table 1: Global co-occurrence matrix for the sample corpus

2. Then, using the Singular Value Decomposition (SVD), this matrix, say X[8], should be decomposed to the form $X = U * S * V$:

$$\overset{|V|}{\underset{|V|}{\begin{bmatrix} \\ X \\ \\ \end{bmatrix}}} = \overset{|V|}{\underset{|V|}{\begin{bmatrix} | & | & \\ u_1 & u_2 & \cdots \\ | & | & \end{bmatrix}}} \overset{|V|}{\underset{|V|}{\begin{bmatrix} \sigma_1 & 0 & \cdots \\ 0 & \sigma_2 & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}}} \overset{|V|}{\underset{|V|}{\begin{bmatrix} - & v_1 & - \\ - & v_2 & - \\ & \vdots & \end{bmatrix}}}$$

Where S is a singular matrix in the form with σ entries across the diagonal. Depending in the desired percentage of variation captured $\frac{\sum_{i=1}^{k} \sigma_i}{\sum_{i=1}^{|V|} \sigma_i}$ one should cut matrix S at some index k, getting $k \times k$ singular matrix. As a result of this transformation we will have to take sub-matrices: $U_{1:|V|,1:k}$ and $V_{1:k,1:|V|}$ with U representing so-called word-embedding matrix.

$$\overset{|V|}{\underset{|V|}{\begin{bmatrix} \\ \hat{X} \\ \\ \end{bmatrix}}} = \overset{k}{\underset{|V|}{\begin{bmatrix} | & | & \\ u_1 & u_2 & \cdots \\ | & | & \end{bmatrix}}} \overset{k}{\underset{k}{\begin{bmatrix} \sigma_1 & 0 & \cdots \\ 0 & \sigma_2 & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}}} \overset{|V|}{\underset{k}{\begin{bmatrix} - & v_1 & - \\ - & v_2 & - \\ & \vdots & \end{bmatrix}}}$$

3. As a result of the following procedure one gets all corpus $|V|$ by rows and corresponding vectors for each word each of the k dimension.

In practice, however there are some serious drawbacks associated with the following model. First of all, given relatively large corpus, which will include more than a million of entries, one will get at least $10^6 \times 10^6$ matrix. Furthermore, the matrix tends to be rather sparse and biased as while most of the word infrequently co-occur, such words and particles as "he, she, it, a, the" will get to

[7]The radius within which one will calculate co-occurrences

[8]The matrix of $|V| \times |V|$ size, where $|V|$ is the power of corpus

much weight. This disbalance gives no useful information and decreases weight of more important structures which one strives to encounter.
Some of the following drawbacks were partially resolved with various transformation. For example, COALS method[Rohde et al., 2006] uses correlation based transformation first to avoid too frequent occurrence of some words. Other works, namely a work of Bullinaria [Bullinaria and Levy, 2007] and Hellinger PCA (HPCA)[Lebret and Collobert, 2013], use positive point-wise mutual information (PPMI) and square root types of transformations respectively to address the problem of imbalances.

2.5 Local Windows based methods

Another school of though proposes a method based on local context windows, which tries to use the local statistics rather than global co-occurrence numbers. The idea of this approach is:

1. Select some radius that will ensure encapsulation of the local context without inferring too much unnecessary information located far from the centre word[9];
2. Minimizing some loss function by updating the entries of the centre and context vectors;
3. Go through the whole corpus one-by-one selecting each word as "centre".

There are a number of models of the type, for instance, Bengio's model [Bengio et al., 2003] based on some rudimentary NN or a more recent work of Collobert with the similar idea[Collobert et al., 2011]. The more advanced approach developed recently by a number of Google specialists is "Word2Vec" model [Mikolov et al., 2013]. In fact this model includes two algorithms:

- Skip-gram model is used for prediction context words given a centre word, which is a quite familiar algorithm which is widely used almost in all modern smart-phones as a part of the typing assistant suite.
- Continuous bag of words (CBOW) model performs an opposite to the skip-gram action - it tries to predict a word given a context of words.

Consider, for example, a mechanics of the Continuous bag of words (CBOW) model.

1. For concreteness, lets consider some centre cut of abstract sentence in the form of:

$$v_{c-m}, v_{c-m+1}, \ldots, v_{c-2}, v_{c-1}, v_c, v_{c+1}, v_{c+2}, \ldots, v_{c+m},$$

   where c - centre word, m - is the size of our windows[10], v - stands for a vector representation of the word;
2. Known parameters are context word, x^c, represented by one-hot vectors[11] and output vector - centre word, y^c;
3. Input matrix, say $\mathcal{V} \in \mathbb{R}^{n \times |V|}$, where n size of the vector used[12], where each column represents input words;
4. Output matrix, say $\mathcal{U} \in \mathbb{R}^{|V| \times n}$, where each row represents n-dimensional word vector.

As a result we get the following structure:

[9] Usually around $5 - 8$ words

[10] Note, that CBOW usually uses symmetric context window

[11] One-hot vector - vector of size $\mathbb{R}^{|V| \times 1}$ with all zeros and one 1 at the place of representing the word from the corpus $|V|$

[12] Hyper-parameter, which must be fine-tuned to get the best result

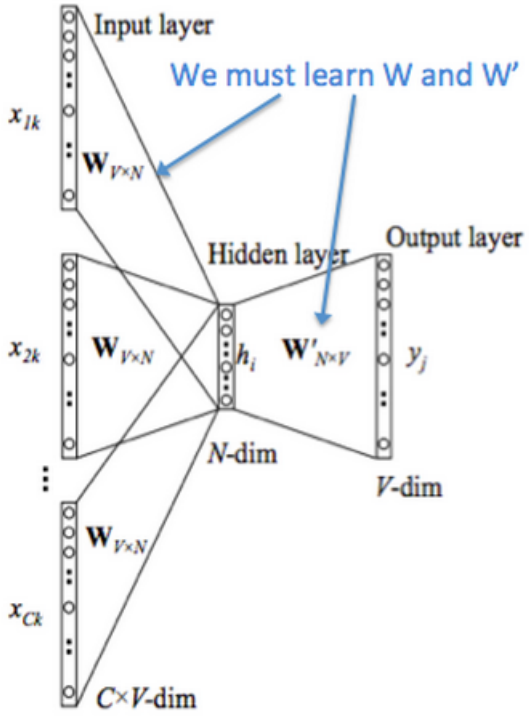


Figure 2: Model representation of the CBOW NN[Sta, 2017].

Practical implementation of continuous bag of words model (CBOW) requires introduction of several ideas:

1. **Idea 1**: take each vector, say v_{c-i}, then get a dot product of the vector with output matrix $\mathcal{U}$, $\hat{z}_{c-i} = \mathcal{U} * v_{c-i}$ and as a result one gets a vector $\hat{z}_{c-i} \in \mathbb{R}^{|V|}$. Using softmax function[13], one may get a probability, which ideally should be maximised to one for the $c-i-th$ entry of one-hot vector x^{c-i}, indicating that transformation matrix is correct.
2. **Idea 2**: **Idea 1** leads to the **Idea 2** that one needs to minimise the distance between two $y, \hat{y}$, for which a common choice is a cross-entropy measure $H(y, \hat{y})$, which calculates the distance between two probability distributions. For a discrete case for one vector loss-function is usually formulated as
$$H(\hat{y}, y) = -\sum_{j=1}^{|V|} y_j \log(\hat{y}_j),$$
for which in case of the excellent prediction for v_{c-i} the $c-i-th$ entry should be one, whereas all other entries should be zero, hence the the overall sum would be zero. In case of the worse prediction and say zero components for all other than $c-i-th$ entry the sum will be $-1 * ln(a \neq 1) > 0$, hence our target is to minimise cross-entropy error.
3. **Idea 3**: One way to model the meaning of the sentence is to find probability of a set of words in a way that meaningful sentences would get high probability, whereas meaningless sentences will be assigned low probability. Hence, one would consider $P(\omega_1 * \omega_2 * \cdots * \omega_n) = \prod_{i=1}^{n} P(\omega_i)$. However, a more natural assumption would be to consider conditional probabilities, as the

[13]Softmax is a function which transforms $i-th$ component of the vector to $\frac{e^{\hat{y}_i}}{\sum_{k=1}^{|V|} e^{\hat{y}_k}}$, which is nothing but positive transformation to get a probability out of the data

probability of the next word usually depends on the previous word, hence it is more natural to consider $P(\omega_1 * \omega_2 * \cdots * \omega_n) = \prod_{i=2}^{n} P(\omega_i|\omega_{i-1})$. The latter model is usually referred to as unigram. One may further encounter even more words getting bigrams, triple-grams etc.

Combining those ideas one may finally formulate loss function, which is:

$$J = -logP(w_c|w_{c-m}, w_{c-m+1}, \ldots, w_{c-2}, w_{c-1}, w_{c+1}, w_{c+2}, \ldots, w_{c+m}),$$

To get some information of the context we may take an average of all the vectors

$$\bar{w} = \frac{v_{c-m} + v_{c-m+1} + \cdots + v_{c-2} + v_{c-1} + v_{c+1} + v_{c+2} + \cdots + v_{c+m}}{2m}.$$

Further, as the goal is to find centre vector, what one needs to minimise is the cross-entropy between target u_c - centre vector and overall context represented by $\bar{w}$, thus $J = -P(u_c|\bar{w})$. The overall problem would then be:

$$\begin{aligned}
\text{minimize } J &= -\log P(w_c|w_{c-m}, w_{c-m+1}, \ldots, w_{c-2}, w_{c-1}, w_{c+1}, w_{c+2}, \ldots, w_{c+m}) \\
&= -\log P(u_c|\hat{w}) \\
&= -\log \frac{\exp(u_c^T \hat{w})}{\sum_{j=1}^{|V|} \exp(u_j^T \hat{w})} \\
&= -u_c^T \hat{w} + \log \sum_{j=1}^{|V|} \exp(u_j^T \hat{w})
\end{aligned}$$

The latter could be resolved with any appropriate method optimisation, for instance, by stochastic gradient descent.

2.6 Global Vectors

Clearly, both of those models are rather good at vectorising words, however they both have mutual drawbacks. Although global matrix factorisation encounters whole corpus statistics it fails to represent correctly important information due to the problem with frequent words. On the other side, Word2Vec family of models does not suffer from the imbalance problem, however it fails to account for the global corpus wide statistics and repeating patterns, which might be crucial in some cases.

A rather neoteric state-of-art solution is the so called Global Vectors approach (GloVe), which encapsulates positive sides of both models[Pennington et al., 2014]. To show how impressive the results of the model are consider the following figure

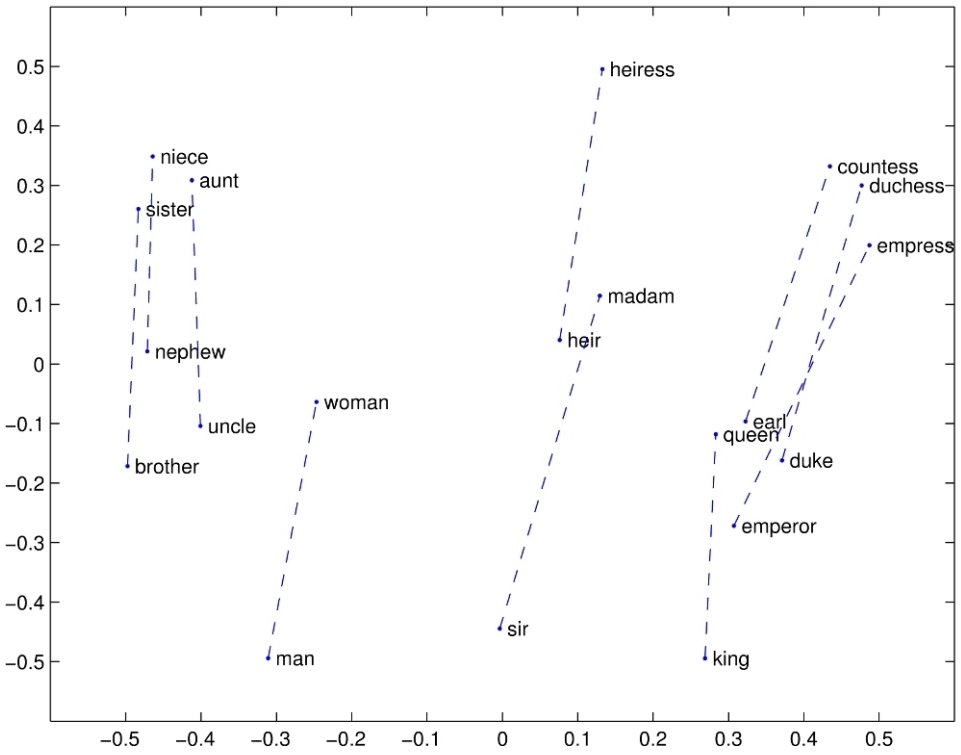


Figure 3: The GloVe model is able to capture the gender difference for a set of words[Pennington et al., 2014].

It vividly shows that the model have been able to account for gender difference as the difference between vector representation of $man - woman$ are approximately the same to the other words with only one difference in gender.
The idea behind the model is very similar to that studied before.

1. We first start by accounting for the global co-occurrence statistics in the same fashion as we did in **"Global matrix factorisation"** paragraph. Taking an original example from GloVe paper [Pennington et al., 2014] consider words: "ice", "steam" and related words "solid", "gas", "water", "fashion". Logically, one would like the model to predict stronger relation of "ice" to "solid", than that of the "steam" to "solid". The vice versa relationship should be true for the word "gas". Last, both "ice" and "steam" have approximately the same relation to "water" and have no relation to fashion, hence one would like this relationship to be represented as well. Observing calculated statistics:

Probability and Ratio	$k = solid$	$k = gas$	$k = water$	$k = fashion$
$P(k\|ice)$	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
$P(k\|steam)$	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
$P(k\|ice)/P(k\|steam)$	8.9	8.5×10^{-2}	1.36	0.96

Figure 4: Global co-occurrence statistics calculated for 6 mln tokens figure is taken from GloVe paper[Pennington et al., 2014].

it becomes clear that in fact, what we are looking for is the ratio of conditional probabilities, as it allows to cancel out some noise which is inevasible due to the reasons discussed earlier.

2. At this point we established a function, say F, which will map three vectors to some scalar, which in turn shows relative distance of the two vectors from the one selected. At this point for the sake of easing the process it is proposed to take the difference of two vectors, say of "ice" and "steam" and take a dot product with the third, compared vector, to match the dimensionality of input and output. As a result one gets the following construction

$$F((w_i - w_j)^T * \tilde{w_k}) = \frac{P_{ik}}{P_{jk}},$$

where the $P_{ik} = P(i|k)$ - probability word i appears in the context of word k and $P_{jk} = P(j|k)$ - probability word j appears in the context of word k.

3. Taking into account the affinity of the word-word co-occurrence matrix (see **??**) one would require invariant property of the F function[14] to match properties of the matrix. This, in turn, can be attained if require two properties:
   (a) First, we require F to be a homomorphism between two groups $(\mathbb{R}, +)$ and $(\mathbb{R}_{>0}, \times)$[15] i.e.,

$$F((w_i - w_j)^T * \tilde{w_k}) = \frac{P_{ik}}{P_{jk}} = \frac{F(w_i^T * \tilde{w_k})}{F(w_j^T * \tilde{w_k})} \Rightarrow P_{ik} = F(w_i^T * \tilde{w_k}) = \frac{X_{ik}}{\sum\limits_k X_{ik}}$$

   Note that X_{ik} corresponds to the entries of global co-occurrence matrix indicating how many times word k appears in the context of word i. This gives us the only possible solution, which is $F = exp$, as it fully corresponds to the property one required[16].
   (b) Second, note that the solution to the equation: $P_{ik} = F(w_i^T * \tilde{w_k}) = \frac{X_{ik}}{\sum\limits_k X_{ik}}$ is $log(P_{ik}) = w_i^T * \tilde{w_k} = log(X_{ik}) - log(X_i)$[17], which does not satisfy an exchange property due to the term $X_i = \sum\limits_k X_{ik}$, hence since this term is independent of k (as we basically sum over all $k - s$) we may add this term into bias of word i, so adding another bias term for word k one gets simplificated version of the initial equation in the form of:

$$w_i^T * \tilde{w_k} + b_i + \tilde{b_k} = log(X_{ik})$$

4. The Global Vector model is partially ready, except for the two main drawback:
   (a) First, the model will have no solution (or better say logarithm will diverge for $X_{ik} = 0$, which will actually happen for certain as from the 2.4 mln corpus there would be at least one word which does not appear in the context window of the other. Luckily, this issue can be resolved by taking not logarithm, but $log(1 + X_{ik})$.
   (b) Second issue is a more complex one: current model gives an equal weights irrespectively the frequency of the word appearance, i.e. $w_i^T * \tilde{w_k} - log(1 + X_{ik}) + b_i + \tilde{b_k}$, which is our basic structure, will have an equal weights irrespectively of the X_{ik} term.
   A solution, proposed by Pennington, Socher, and Manning [2014] is to include weighting term depending on the frequency of the word appearance - $f(X_{ik})$ - for the standard least

[14] The reason is that one may freely exchange reference word (i.e. w_k) with the examined words

[15] The reason is that we may open up the brackets $(w_i - w_j)^T * \tilde{w_k} = w_i^T * \tilde{w_k} - w_j^T * \tilde{w_k}$, which must be consistent with interchangebility $\times$ sign

[16] One can easily check that, as $exp[(w_i - w_j)^T * \tilde{w_k}] = exp[(w_i - w_j)^T]^{\tilde{w_k}} = \frac{exp[w_i^T * \tilde{w_k}]}{exp[w_j^T * \tilde{w_k}]} = \frac{exp[w_i^T]^{\tilde{w_k}}}{exp[w_j^T]^{\tilde{w_k}}} \Rightarrow$ taking both sides to the power $\frac{1}{\tilde{w_k}}$ we will get the required proof of the statement.

[17] Where $X_i = \sum\limits_k X_{ik}$

squares loss function. In fact, this is a well-known weighted least squares approach widely used by econometricians all other the world to cope with heteroscedasticity problem. Consequently, our loss function will be given by:

$$J = f(X_{ik}) * (w_i^T * \tilde{w_k} - log(1 + X_{ik}) + b_i + \tilde{b_k})$$

(c) Indubitably this weighting function must have some desired properties, in particular:

i. It should have zero weight if two words never appear in the context of each other, so our target vectors does not include that information.
ii. It should not overweight too infrequent appearances, i.e. it should be increasing (at least partially)
iii. Finally, this function should not give to much weight for frequently appearing words, as it will bring unnecessary noise to the vectors

Given those prerequisites authors propose piece-wise given function in the form:

$$f(x) = \begin{cases} (x/x_{max})^{\alpha}, \text{ if } x < x_{max} \\ 1, \text{ otherwise} \end{cases}$$

with the hyper-parameters α and x_{max} giving the best result for $\alpha = \frac{3}{4}$ and $x_{max} = 100$. As a result, weighting function gets the following form:

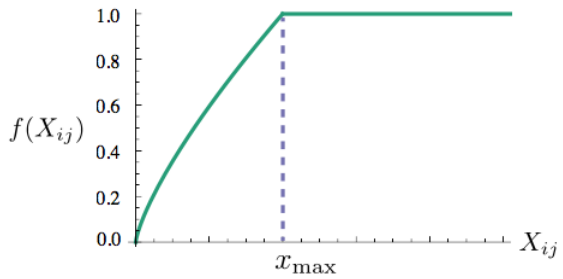


Figure 5: Proposed by Pennington, Socher, and Manning [2014] weighting function, figure is taken from GloVe paper[Pennington et al., 2014].

Having briefly reviewed the mechanics of the GloVe model it is time to compare it with other word-embedding algorithms(see paragraphs: 2.4 and 2.5). This paper will again use the results of Pennington, Socher, and Manning [2014] due to the complexity of the required calculations.
As expected, due to the usage of positive sides of both models, i.e. usage of the global corpus statistics within a certain windows size with a normalisation and weighting Global Vectors algorithms achieves the highest accuracy almost for all tasks "given the same corpus, vocabulary, window size, and training time"[Sta, 2017].

Model	Size	WS353	MC	RG	SCWS	RW
SVD	6B	35.3	35.1	42.5	38.3	25.6
SVD-S	6B	56.5	71.5	71.0	53.6	34.7
SVD-L	6B	65.7	72.7	75.1	56.5	37.0
CBOW†	6B	57.2	65.6	68.2	57.0	32.5
SG†	6B	62.8	65.2	69.7	58.1	37.2
GloVe	6B	65.8	72.7	77.8	53.9	38.1
SVD-L	42B	74.0	76.4	74.1	58.3	39.9
GloVe	42B	**75.9**	**83.6**	**82.9**	**59.6**	**47.8**
CBOW*	100B	68.4	79.6	75.4	59.4	45.5

Figure 6: Word vectorising model comparison on the basis of word similarity task, figure is taken from GloVe paper[Pennington et al., 2014].

Clearly, figure **??**, shows the supremacy of GloVe model on word analogy task. The latter is one of the intrinsic evaluation techniques, which performed through calculation of the correlation between the results of human judgment and machine outputs. In particular, a group of person were asked to evaluate level of similarity for different word on 1-10 scale. Global Vectors model outperforms all other model, even with twice a lower corpus, used for training(for the case of CBOW trained on 100B of tokens[18]).
Furthermore, another type of intrinsic valuation shows the similar results:

Input	Result Produced
bad : worst : : big	biggest
bad : worst : : bright	brightest
bad : worst : : cold	coldest
bad : worst : : cool	coolest
bad : worst : : dark	darkest
bad : worst : : easy	easiest
bad : worst : : fast	fastest
bad : worst : : good	best
bad : worst : : great	greatest

Input	Result Produced
dancing : danced : : decreasing	decreased
dancing : danced : : describing	described
dancing : danced : : enhancing	enhanced
dancing : danced : : falling	fell
dancing : danced : : feeding	fed
dancing : danced : : flying	flew
dancing : danced : : generating	generated
dancing : danced : : going	went
dancing : danced : : hiding	hid
dancing : danced : : hitting	hit

Figure 7: Results of word-vector analogy task, figures are taken from [Sta, 2017].

Word vector analogy task[19] depicted on figure **??** confirms the results, as Global Vectors model was able to correctly identify comparative and superlative relations for adjectives and capture forms similarity for verbs.

[18]In fact, that is a rather interesting outcome, which can be explained by the corpus used for training. Obviously, rule: the larger the corpus the more precise the model works to some extent, however there is a diminishing returns to scale. Furthermore, the 100B corpus used to train CBOW contains outdated information, which could be one of the reasons for such poor performance.

[19]Word vector analogy task requires maximisation of cosine similarity between vectors, i.e. one takes the difference of two word vectors, say "bad" - "worst", then takes word vector "big" and tries to find word-vector, such that "big" - "vector we are trying to find" is closest to the difference found before. Mathematically, given three vectors a,b,c one tries to find vector d, such that $d = argmax_i \frac{(a-b+c)^T * x_i}{||a-b+c||}$

Last practical note to mention here is the pre-trained Global Vector models available to use. Stanford university kindly provides pre-trained models, which were trained on the corpuses of various size starting from the smallest 1B Wikipedia2014 dataset and ending with the largest dataset available - Common Crawl[Com, 2017] with 42B tokens in it. As expected, the larger the corpus, the better the data rule holds:

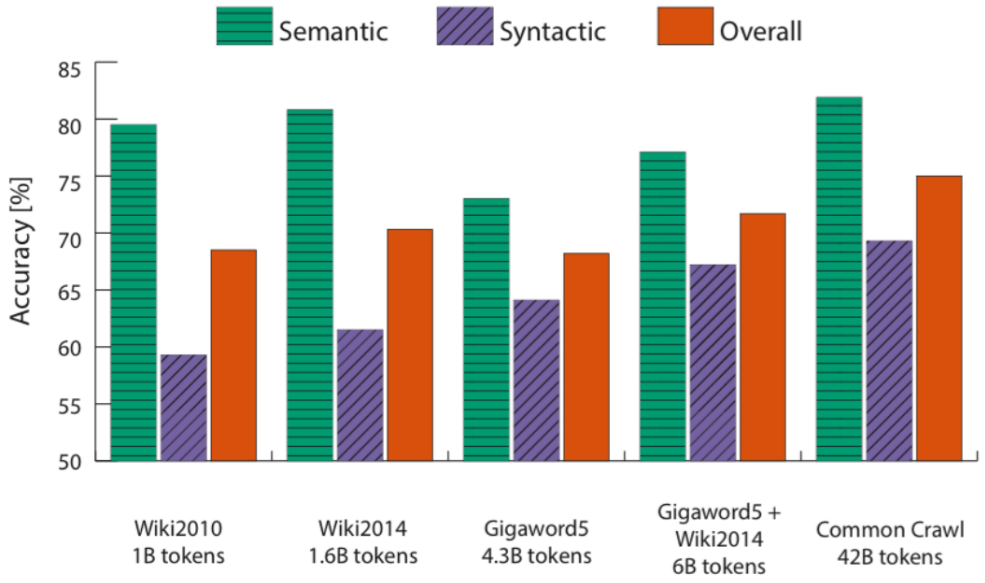


Figure 8: GloVe performance comparison depending on the corpus used, figure is taken from [Sta, 2017].

Obviously, given enough computational resources the Global Vector model trained on 42B Common Crawl[Com, 2017] corpus should be used due to it's superiority in any aspect.

3 Deep learning techniques

3.1 XGBoost

XGBoost - machine learning algorithm appearing on virtually all Kaggle[Kag, 2017] competitions. It is widely used for it's combined setup simpleness and high predictive power.
XGBoost - is one of the strongset representatives of the gradient boosting prediction models, which in a nutshell construct a prediction function of the form[Chen and Guestrin, 2016] :

$$\hat{y_i} = \phi(x_i) = \sum_{k=1}^{K} f_k(x_i), f_k \in (F)$$

by optimising a certain objective function, which is usually regularised, to avoid overfitting problem:

$$L(\phi) = \sum_i l(\hat{y_i}, y_i) + \sum_k \Omega(f_k)$$

where l - is a twice differentiable convex loss function, for instance, least squares objective.
At each step a new model is added to improve the previous imperfect model, which might be initialised as an average of values. The fitted model is logically is $h(x) = y - F_m$, where y - is the true output, F_m - previous imperfect model. Hence, each further model F_{m+1} learns from the previous one F_m.
An important part of the XGBoost and gradient boosting models is the regularising term, which is usually expressed as

$$\Omega(f) = \gamma * T + \frac{1}{2} * \lambda * \sum_{j=1}^{T} w_j^2,$$

Where T is the number of leafs, w - vector of scores on leafs and γ, λ - hyper-parameters.
These are the main blocks for Gradient boosting model constructing. The question is, however, why is XGBoost is so popular in particular? The answer is in it's ability to effectively handle data through by transforming data into block format which reduces computational difficulty from O(n*log(n)) to O(n)[Chen and Guestrin, 2016].

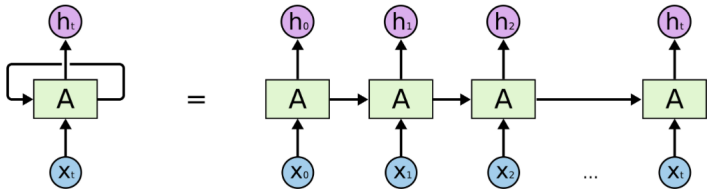
3.2 Long short term memory RNN

Artificial neural networks is one of the most debatable topics of the past two years, as their development has been so significant in virtually all spheres of human life. The problems which were seen as non-algorithmic has been successfully resolved by implementation of neural network. One of such themes is the ancient game "Go" invented more than 2500 years ago in China. It was believed that this game is non-algorithmic until the recent development of AlphaGo algorithm by Google's DeepMind division. The results of using neural nets and reinforcement learning has made AlphaGo virtually unbeatable[Shaban, 2017].
This paper will discuss one of the most advanced neural networks - Long-short term memory recurrent NN[Hochreiter and Schmidhuber, 1997]. Fundamentals of LSTM have been developed in the previous century, however this type of RNN gained it's popularity only recently due it's computational difficulty. Currently LSTM is one of the most popular NN for NLP tasks and is used by almost all top high tech companies. It is widely used by Apple, Google, Amazon, Microsoft and many other

companies as a parts of speech recognition, next word prediction, smart assistant and many other types of systems.
The logical question is of course the roots of such popularity. First thing to notice that LSTM belongs to the Recurrent neural network family, which is in contrast to feedforward networks, have some type of memory. This memory lies in the fact that RNN additionally to the new information receives the previous data, so called "feedback loop". This peculiarity in construction have a drastic effect as the NN basically analyses it's own outputs while receiving new inputs. Mathematically, past information circulating in the hidden state can be described with function:

$$h_t = \phi(W_h * x_t + U_h * h_{t-1} + b_h),$$

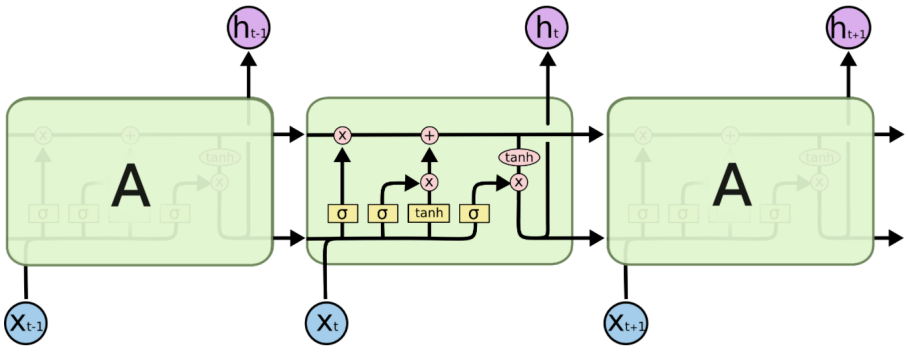
where h_t - hidden at iteration t layer, ϕ - activation function, W_h - weight (parameter) matrix, U_h - matrix of weight from hidden state to hidden state matrix.



An unrolled recurrent neural network.

Figure 9: Recurrent neural network unrolled, picture is taken from[lst, 2017].

However, vanishing gradient problem have become a major hindrance for RNN in late 1990-s. Furthermore, as it turned out RNN can not cope with learning long run dependencies and the longer the gap the more difficult for NN to capture the relevant information ([Bengio et al., 1994], [Hochreiter, 1991] (German)). One of the solutions to the problem is the LSTM neural nets proposed by Hochreiter and Schmidhuber [1997].
The solution, to the problem, as someone elegantly called it "learning how to forget". Let's discuss the statement in context of the LSTM structure.



The repeating module in an LSTM contains four interacting layers.

Figure 10: LSTM structure, picture is taken from[lst, 2017].

First thing to notice is that as any other recurrent neural network LSTM consist of repeating parts, however it's internal structure is a bit different, hence lets examine it in further detail.

There are basically two key things to bear in mind when studying LSTM. First, is the cell-state which is an informational flow which passes through each cell and affects each output node h_t.

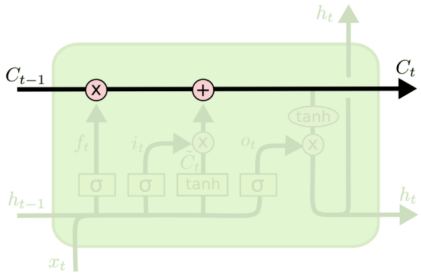
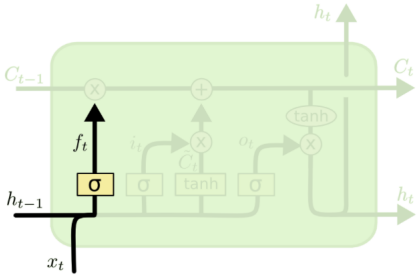


Figure 11: Cell state flow from $t-1$ cell to t, picture is taken from[lst, 2017].

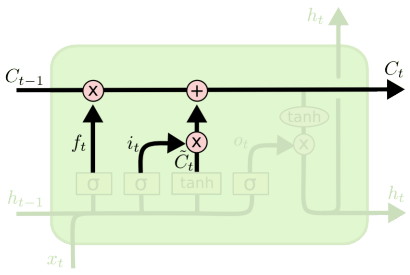
Second, is the "forget gate layer", which is sigmoid a layer deciding how much information from previous output to pass to the new output. As it can be seen form the figure **??** the new information x_t and information from the previous gate to include h_{t-1} multiplied by the weight matrix W_f are passing through sigmoid function with the output ranging from 0 to 1. In case of the 0 state, multiplication by C_{t-1} will give zero output meaning that no past information is going forward (forget state). One, on the contrary, remembers whole information.



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Figure 12: Forget gate layer structure, picture is taken from[lst, 2017].

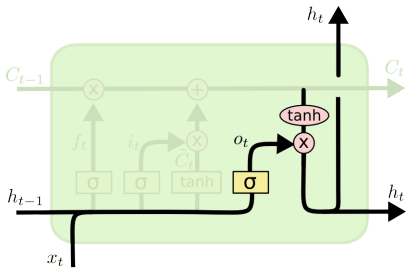
Third stop is the "input gate", which decides, how much information we will update. The construction is similar with the same sigmoid ranging from 0 to 1 :$i_t = \sigma(W_i * [x_t, h_{t-1}] + b_i)$ and the new information combined with previous output which passes through activation function. $\tilde{C}_t = tanh(W_C * [x_t, h_{t-1}] + b_C)$
All of these information is combined into one flow which passes to the next cell. Mathematically, $C_t = \tilde{C}_t * i_t + f_t * C_{t-1}$



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Figure 13: Final informational "cell" flow, picture is taken from[lst, 2017].

Finally, current cell flow (C_t) passed through activation function and multiplied by the third output of sigmoid ($o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$) (to find how much information should pass to the next layer from the hidden layer) the final output is $h_t = o_t * tanh(C_t)$



$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Figure 14: Final output of the current layer, picture is taken from[lst, 2017].

4 Practical application of Natural Language processing and deep learning techniques

"The father of simplex algorithm and linear programming", George Bernard Dantzig, ones famously quoted: "The final test of a theory is its capacity to solve the problems which originated it."[Dantzig, 1963]. This wonderful thought should appear at every article, as one of the crucial goals of any theory is in it's practical application and usefulness to the humanity. Bearing in mind this extremely important idea this paper intends to use theoretical approaches discussed previously in the paper to solve real-life problems.

4.1 Quora Question pairs

First of all, let me introduce to the problem selected and all the aspects connected to it.
This paper will deal with the competition published on the Kaggle.com[Kag, 2017] web site, which is a home for a machine and deep learning competitions. One of the recent competitions, closed at the beginning of June, is the Quora Question pairs competition([QQP, 2017]), which deals with identification of duplicates.
In fact, Quora inc. is one of the largest "question answering web sites", where anyone may ask a question and get a thorough reply on it. However, as it often happens, public tends to be lazy and instead of performing wide Google research people simply add a new question and waiting for the answer. Indubitably, it is extremely difficult if not impossible to provide high quality answers to all question, hence it is vitally important to be able to match similar questions to one pool to provide high quality experience for users.
The problem raised is important not only in context of post-mortem duplicates detection, but for proactive detection and suggesting systems including intelligent assistant, web-search queries and to the needs of Quora company itself, which could suggest similar questions in advance leading to a more efficient resource usage. All of these make this problem crucially important and potentially extendable for a larger set of tasks.
The problem becomes even more interesting as one has to beat an existing decision-tree based existing Quora solution. As a training dataset the company has posted a csv file with just under half a million of questions (404,290 to be precise). This file has a pair of questions and manually binary labelled data - suggesting whether the questions are duplicated or not. Further, there is a test dataset with other 2 mln. of questions available, however, labels for it are unavailable, hence, to evaluate effectiveness of algorithm proposed train dataset will be randomly divided into two subsets in such a way, that distributions of duplicates would approximately match. The weights assigned to the generated datasets would be 90% and 10% for train and test questions respectively. Luckily, there is enough data for both datasets.
Using "scikit-learn"[sci, 2017] train_test_split command we divide our dataset on two:

1. Train dataset: df_train with 363,861 questions and the share of duplicates of 36.92756%
2. Test dataset: df_test with 40,429 questions and the share of duplicates of 36.8498%[20]

[20]Apologies for including code as a picture. This done to avoid possible problems with anti-plagiarism systems, which detects any Python code as a plagiarism

```
start = datetime.now()
#Divide the set
#First lets divide our dataframe into two frames, train and test ones
#I will divide the sets here and fix this division, as it is necessary for further LSTM feature creation
df_train, df_test= train_test_split(df, test_size=train_test_split_ratio, random_state=4242)
end = datetime.now()
print 'sets are divided, time taken:',end-start
```

```
sets are divided, time taken: -1 day, 23:54:50.417761
```

```
#Check the distribution of labels in train and test samples, which is quite good!
a=(df_train["is_duplicate"]==1).sum(axis=0)
b=(df_train["is_duplicate"]==0).sum(axis=0)
print a/float(a+b)
```

```
0.36927562998
```

```
#Check the distribution of labels in train and test samples, which is quite good!
a=(df_test["is_duplicate"]==1).sum(axis=0)
b=(df_test["is_duplicate"]==0).sum(axis=0)
print a/float(a+b)
```

```
0.368497860447
```

Figure 15: Dataset division with distribution of duplicate questions in each.

4.2 Simple exploration of data

Diving deeper into the data reveals that the share of duplicates in the train data is approximately 36.9199%, which corresponds to that in the datasets divided.

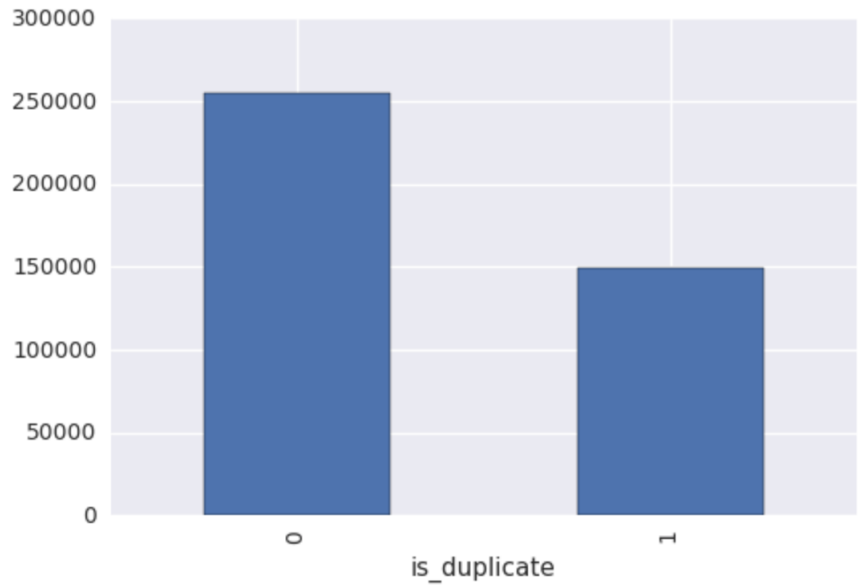


Figure 16: Distribution of duplicates, where 1 - corresponds to the duplicate questions, 0 - non-duplicate ones.

Further, some more beauty might be added by exploring the most frequently appearing words:

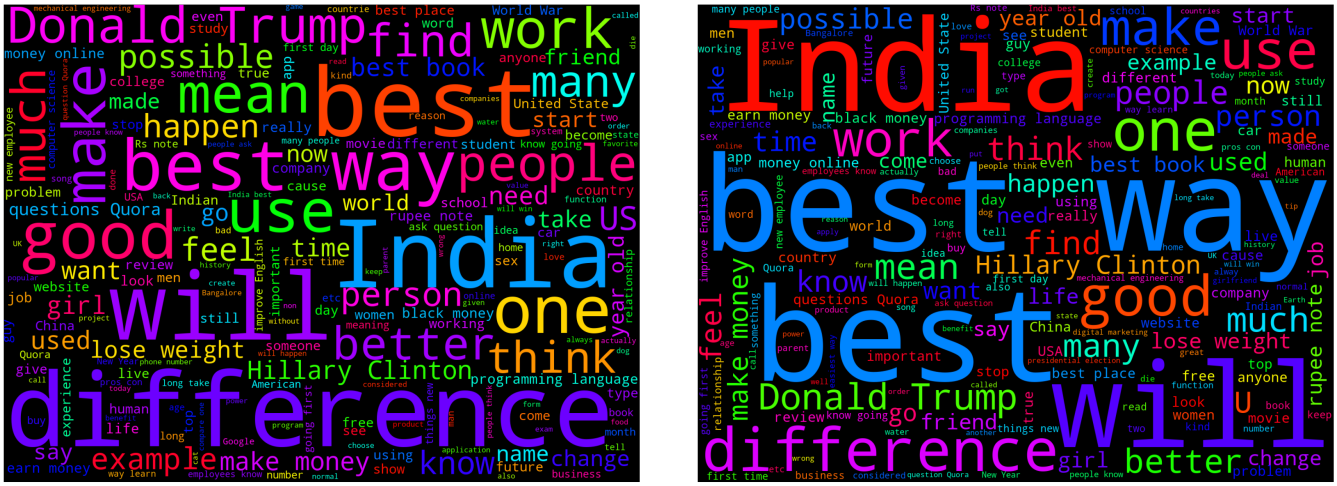


Figure 17: The most frequently appearing words.

One may also observe the distribution of the characters across the train dataset, which peaks at about 50 characters per question and F-type of the distribution. One more thing to be added here is that there is very low number of questions with more than 150 characters, which could be a good feature for long questions.

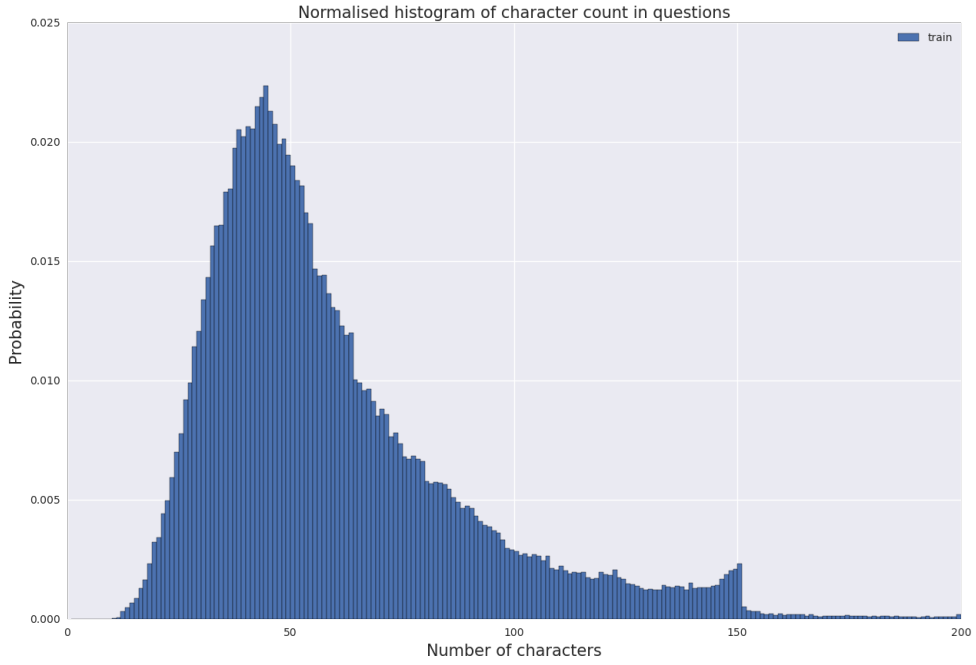


Figure 18: Distribution of characters count across the questions.

As expected word count distribution looks very similar to that of the character count:

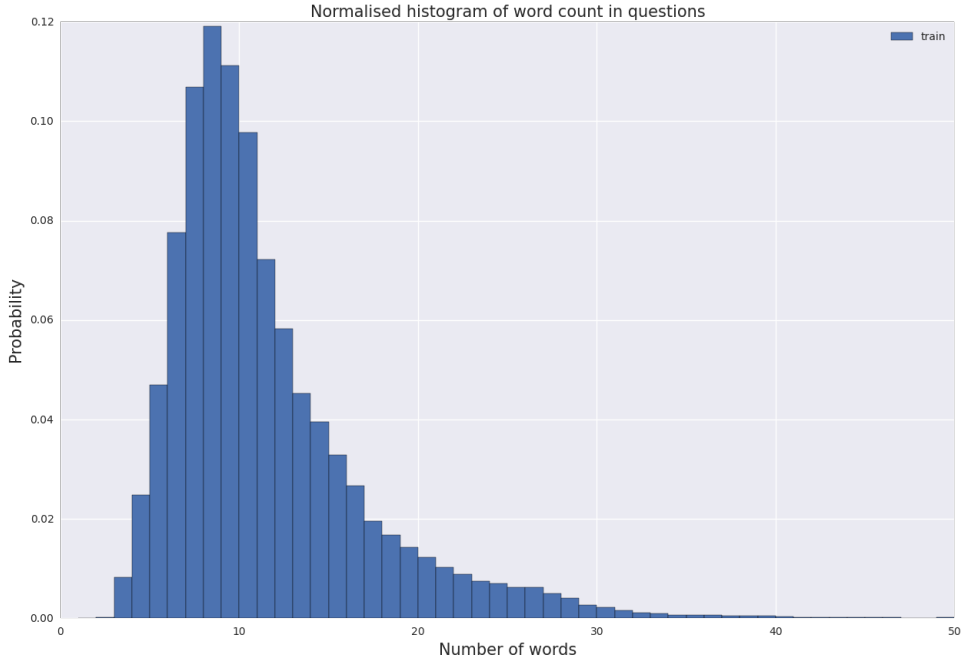


Figure 19: Distribution of word count across the questions.

One more interesting statistics to infer is the usage of some common punctuation and acronyms. In fact, nearly 100% of questions have question signs, which will not give any useful information and will deter correct word spelling, as punctuation sign usually go after the word. Furthermore, nearly 5% of the questions have "is" → "'s" reduction, which is again undesirable feature.

```
from string import punctuation
qmarks = np.mean(train_qs.apply(lambda x: '?' in x))
math = np.mean(train_qs.apply(lambda x: '[math]' in x))
fullstop = np.mean(train_qs.apply(lambda x: '.' in x))
scount = np.mean(train_qs.apply(lambda x: r"'s" in x))
recount = np.mean(train_qs.apply(lambda x: r"'re" in x))
dcount = np.mean(train_qs.apply(lambda x: r"'d" in x))
llcount = np.mean(train_qs.apply(lambda x: r"'ll" in x))
print('Questions with question marks: {:.2f}%'.format(qmarks * 100))
print('Questions with [math] tags: {:.2f}%'.format(math * 100))
print('Questions with full stops: {:.2f}%'.format(fullstop * 100))
print('Questions with is acronym: {:.2f}%'.format(scount * 100))
print('Questions with are acronym: {:.2f}%'.format(recount * 100))
print('Questions with would acronym: {:.2f}%'.format(dcount * 100))
print('Questions with will acronym: {:.2f}%'.format(llcount * 100))
```

```
Questions with question marks: 99.87%
Questions with [math] tags: 0.12%
Questions with full stops: 6.31%
Questions with is acronym: 4.65%
Questions with are acronym: 0.23%
Questions with would acronym: 0.04%
Questions with will acronym: 0.04%
```

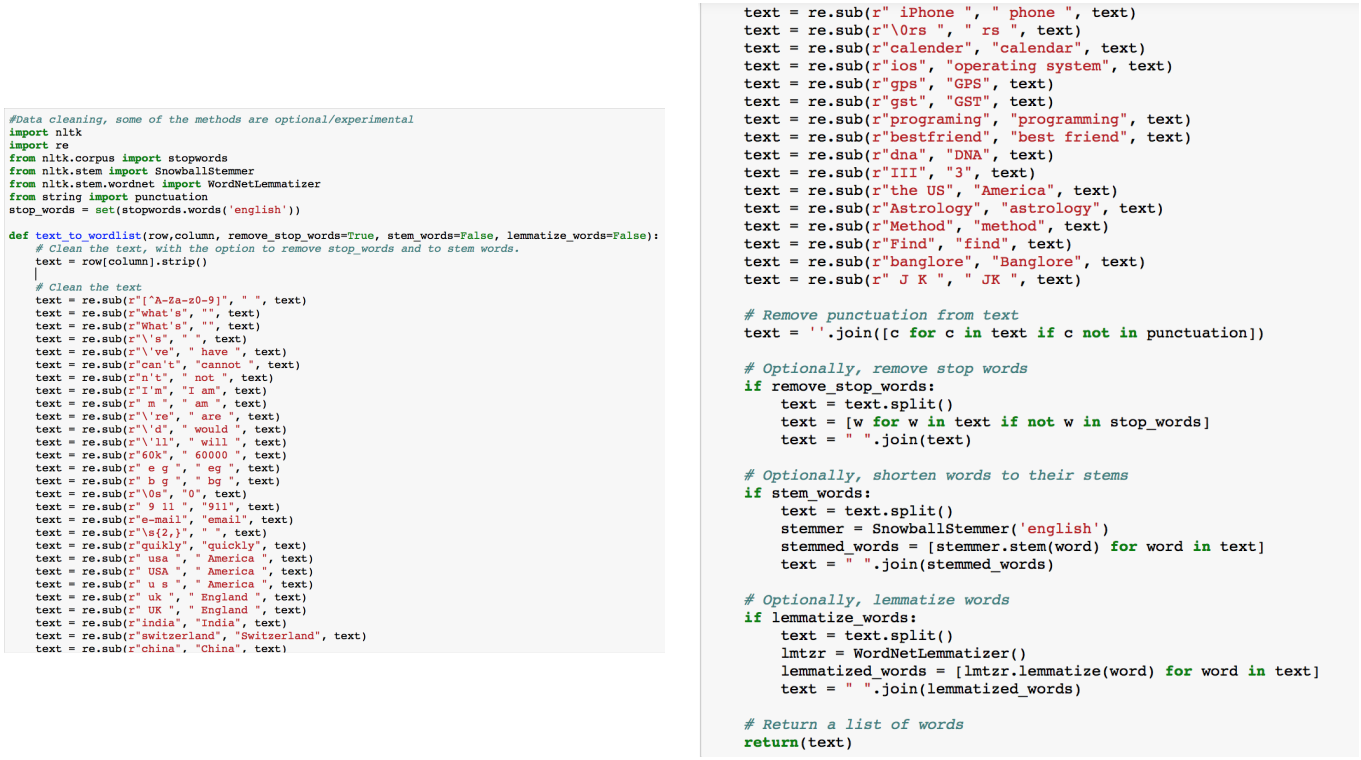
Figure 20: Some of the punctuation and acronyms statistics.

4.3 Text cleaning in practice

Having tasted the dataset we may proceed with the actual algorithms. As it was noted at 2.2 paragraph with some empirical proofs derived in paragraph 4.2 on figure **??** text cleaning problem becomes vitally important. It might be divided into three parts:

1. First and the easiest is to delete all of the punctuation
2. Second, is to replace some common acronyms and abbreviations, to get a clearer language
3. Third, is to perform lemmatization discussed in paragraph 2.2

The code for this part is rather straightforward, with WordNetLemmatizer usage from NLTK package[Bird, 2006].



```
#Data cleaning, some of the methods are optional/experimental
import nltk
import re
from nltk.corpus import stopwords
from nltk.stem import SnowballStemmer
from nltk.stem.wordnet import WordNetLemmatizer
from string import punctuation
stop_words = set(stopwords.words('english'))

def text_to_wordlist(row,column, remove_stop_words=True, stem_words=False, lemmatize_words=False):
    # Clean the text, with the option to remove stop_words and to stem words.
    text = row[column].strip()

    # Clean the text
    text = re.sub(r"[^A-Za-z0-9]", " ", text)
    text = re.sub(r"what's", "", text)
    text = re.sub(r"What's", "", text)
    text = re.sub(r"\'s", " ", text)
    text = re.sub(r"\'ve", " have ", text)
    text = re.sub(r"can't", "cannot ", text)
    text = re.sub(r"n't", " not ", text)
    text = re.sub(r"I'm", "I am", text)
    text = re.sub(r" m ", " am ", text)
    text = re.sub(r"\'re", " are ", text)
    text = re.sub(r"\'d", " would ", text)
    text = re.sub(r"\'ll", " will ", text)
    text = re.sub(r"60k", " 60000 ", text)
    text = re.sub(r" e g ", " eg ", text)
    text = re.sub(r" b g ", " bg ", text)
    text = re.sub(r"\0s", "0", text)
    text = re.sub(r" 9 11 ", "911", text)
    text = re.sub(r"e-mail", "email", text)
    text = re.sub(r"\s{2,}", " ", text)
    text = re.sub(r"quikly", "quickly", text)
    text = re.sub(r" usa ", " America ", text)
    text = re.sub(r" USA ", " America ", text)
    text = re.sub(r" u s ", " America ", text)
    text = re.sub(r" uk ", " England ", text)
    text = re.sub(r" UK ", " England ", text)
    text = re.sub(r"india", "India", text)
    text = re.sub(r"switzerland", "Switzerland", text)
    text = re.sub(r"china", "China", text)
    text = re.sub(r" iPhone ", " phone ", text)
    text = re.sub(r"\0rs ", " rs ", text)
    text = re.sub(r"calender", "calendar", text)
    text = re.sub(r"ios", "operating system", text)
    text = re.sub(r"gps", "GPS", text)
    text = re.sub(r"gst", "GST", text)
    text = re.sub(r"programing", "programming", text)
    text = re.sub(r"bestfriend", "best friend", text)
    text = re.sub(r"dna", "DNA", text)
    text = re.sub(r"III", "3", text)
    text = re.sub(r"the US", "America", text)
    text = re.sub(r"Astrology", "astrology", text)
    text = re.sub(r"Method", "method", text)
    text = re.sub(r"Find", "find", text)
    text = re.sub(r"banglore", "Banglore", text)
    text = re.sub(r" J K ", " JK ", text)

    # Remove punctuation from text
    text = ''.join([c for c in text if c not in punctuation])

    # Optionally, remove stop words
    if remove_stop_words:
        text = text.split()
        text = [w for w in text if not w in stop_words]
        text = " ".join(text)

    # Optionally, shorten words to their stems
    if stem_words:
        text = text.split()
        stemmer = SnowballStemmer('english')
        stemmed_words = [stemmer.stem(word) for word in text]
        text = " ".join(stemmed_words)

    # Optionally, lemmatize words
    if lemmatize_words:
        text = text.split()
        lmtzr = WordNetLemmatizer()
        lemmatized_words = [lmtzr.lemmatize(word) for word in text]
        text = " ".join(lemmatized_words)

    # Return a list of words
    return(text)
```

Figure 21: Text cleaning code extract.

As expected, the shape distribution of character count statistics does not drastically changes, however the distribution becomes more dense with a parallel decrease of the of the peak to 30-35 characters per question.

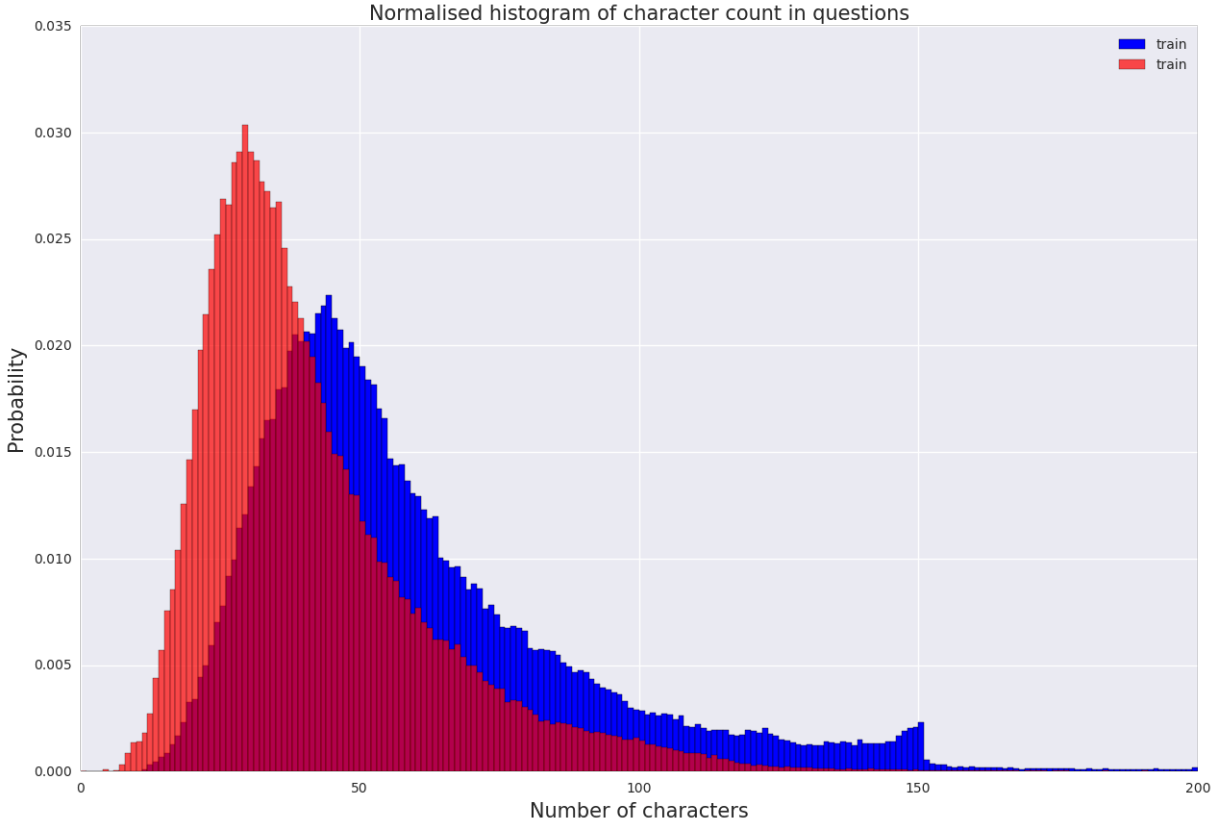


Figure 22: Change of character count distribution after text cleaning.

4.4 Basic features creation

There are some basic set of features, which will help in identification of duplicates. These are:

1. Character length of the sentence, which is useful in capturing non-duplicates for questions longer than 100 characters length and smaller questions due to the higher probabilities on these areas. See graph for more details;

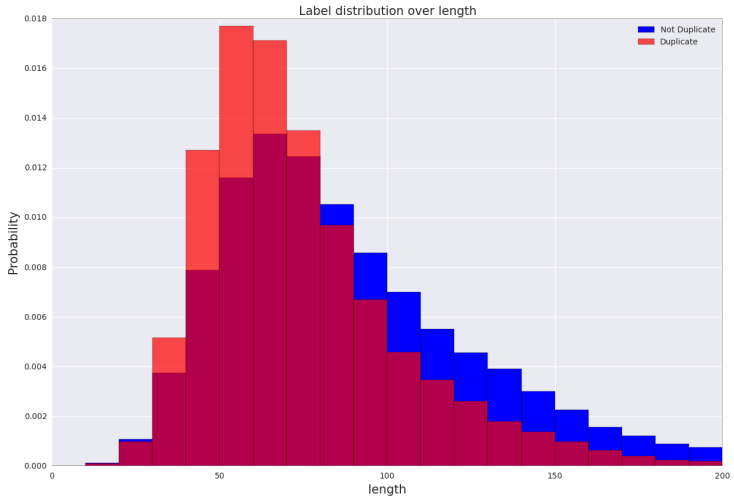


Figure 23: Character count distribution depending on whether the question is duplicate or not. Note that maximal length is constrained with 200, as questions lower than threshold are duplicates.

2. Difference of character length. Obviously, this feature is useful for capturing identical questions;
3. Number of words in each question, this metric is quite similar to the first one, however it might add some additional signal;
4. Normalised word-share between questions, this feature is a bit more complex and requires creation of a set from both questions. After creating two sets one simply finds it's intersection and divides by overall number of elements in two sets. This feature is rather useful, which can be clearly observed from the following graph;

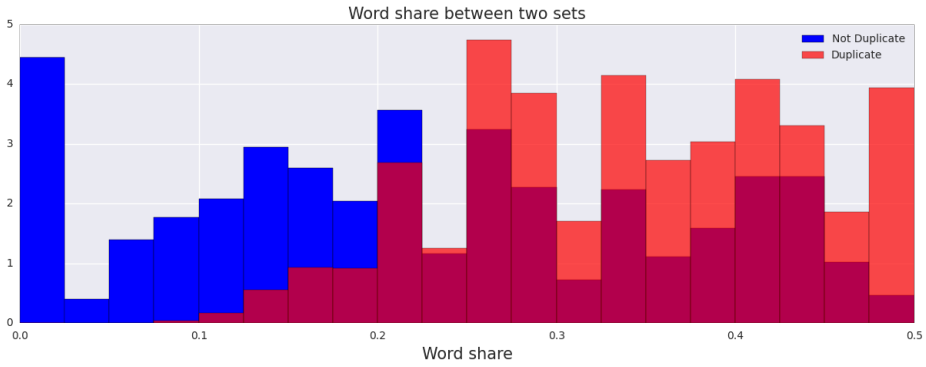


Figure 24: Clearly, word share feature is especially useful in identifying non-duplicates when word share is relatively low. The opposite is true for close to 50% word share questions.

5. Jaccard distance between two sets of words. Jaccard distance is a metric on a finite set showing level of dissimilarity between two sets. The formula is $d_J(A,B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|}$. It must be stated that Jaccard distance is similar to word share calculated, however it may provide some additional signals, as it is a measure of dissimilarity. Further, it's formula is a bit different, compared to the word share calculation;
6. Sørensen distance between two sets of words, which is again quite similar to the of Jaccard distance, however it is more persistent to the outliers and retains sensitivity in heterogeneous data. Formula is $d_S(A,B) = 1 - \frac{2*|A \cap B|}{|A| + |B|}$;
7. Fuzzy ratios are based on the FuzzyWuzzy library[Fuz, 2017] and represent one more way of calculating difference between two strings. However, the approach is drastically different due to the metric and overall philosophy implied. In fact fuzzy ratios are calculated basing on the Levenshtein distance, introduced by Soviet scientist Vladimir Levenshtein[Levenshtein, 1966]. The Levenshtein metric [21] calculates the number of characters required to make the questions identical. Depending on what we call similar strings FuzzWuzzy[Fuz, 2017] calculates the appropriate metric. For instance, for simple fuzzy ratio strings "New York Yankees" and "Yankees" are quite different with a score 60 out of 100. However, if we consider those strings as a name of one baseball team, then this strings are identical, hence fuzzy partial ratio will give 100% match. Further, a package also offers two more metrics, token sort ratio, which will

21

$$\mathrm{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \mathrm{lev}_{a,b}(i-1,j) + 1 \\ \mathrm{lev}_{a,b}(i,j-1) + 1 \\ \mathrm{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

Where a,b two strings; first i or j - characters of string a or b; $1_{a_i \neq b_j}$ - indicator function taking 0, when first i and j characters of strings are equal, 1 otherwise.

give an exact match score for "New York" and "York New" strings and token set ratio, which is fully based on the sets and will not count repeats.

4.5 More features - better predictions

Having tried those features on sample XGBoost classifier one got decent result of 0.42 log-loss metrics for over a half hour of training time. Hence, we will proceed with even more features. Consider a piece of code:

```
def try_divide(x, y, val=0.0):
    if y != 0.0:
        val = float(x) / y
    return val

def get_the_position_param(str1, str2):
    q1 = str(str1).lower().split(' ')
    q2 = str(str2).lower().split(' ')
    pos_of_obs_in_target = [0]
    if len(q2) != 0:
        pos_of_obs_in_target = [j for j,w in enumerate(q2, start=1) if w in q1]
        if len(pos_of_obs_in_target) == 0:
            pos_of_obs_in_target = [0]
    return [pos_of_obs_in_target]

def get_the_position_param_normalized(str1, str2):
    q1 = str(str1).lower().split(' ')
    q2 = str(str1).lower().split(' ')
    pos_of_obs_in_target = [0]
    if len(q2) != 0:
        pos_of_obs_in_target = [j for j,w in enumerate(q2, start=1) if w in q1]
        if len(pos_of_obs_in_target) == 0:
            pos_of_obs_in_target = [0]
    return [try_divide(min(pos_of_obs_in_target),len(q1)),
                       try_divide(max(pos_of_obs_in_target),len(q1)),
                       try_divide(np.mean(pos_of_obs_in_target),len(q1)),
                       try_divide(np.median(pos_of_obs_in_target),len(q1)),
                       try_divide(np.std(pos_of_obs_in_target),len(q1))]
```

```
str1="What will happen if Queen Elizabeth II dies"
str2="What will happen when the Queen dies"
get_the_position_param(str1, str2)
```

```
[[1, 2, 3, 6, 7]]
```

Figure 25: Feature creation based on repeating word positions. An example taken from one of the question pairs.

A set of standard statistical features, such as minimum, maximum, mean, median and standard deviation was created from the positions of similar words in the questions. The logic behind this set of number is that the more often the common words appear in the text the more likely they are going to be a duplicates. One further consideration is that similarity of the questions may also depend on the position of the word in the text, which is explicitly included in that feature. For a more thorough results these numbers were calculated for question 1 in question 2 and vice versa with additional normalisation to account for the length difference of the questions.

4.6 Power of Word embedding and LSTM

One of key themes of this paper work is GloVe word embedding technique, however it is rather difficult to make sense of this powerful set of data. Our goal is to get an overall meaning of the sentence, yet the dimensionality of the question is 300×"length of the question"[22], whereas we need a number to compare questions.
One of the possible solutions is to use "Long Short term memory" (LSTM) recurrent neural network

[22]The Global Vectors pre-trained data used is Common Crawl 42 B. dataset with 300 dimension for each vector

of Hochreiter and Schmidhuber [1997] discussed at paragraph 3.2. To structure explanation of neural network used let me first start with the inputs used, then proceed to the internal structure of the network and finish with results and overall comments.

1. As one of the main accomplishments of the LSTM RNN[23] is natural language processing, the main inputs of the network will be questions themselves preliminary passed through word embeddings techniques. Main problems associated with passing questions is the choice of how to pass those words to computer and how to control significant question length variability. The solution to the first issue, it to tokenise all corpus, by creating a dictionary:

$$word - number$$

Luckily, there is a wonderful tool for the task - Keras package[ker, 2017], hence we may proceed with creating a library with 81,066 uniques tokens found.

```
#Tokenizing texts
start = datetime.now()

#Train tokenizer
tokenizer = Tokenizer(num_words=MAX_NB_WORDS)
#Create a dictionary number - word
tokenizer.fit_on_texts(texts_1 + texts_2+test_texts_1+test_texts_2)

#Translate texts according the dictionary created a step before
sequences_1 = tokenizer.texts_to_sequences(texts_1)
sequences_2 = tokenizer.texts_to_sequences(texts_2)
test_sequences_1 = tokenizer.texts_to_sequences(test_texts_1)
test_sequences_2 = tokenizer.texts_to_sequences(test_texts_2)

#print tokenizer.word_index
word_index = tokenizer.word_index
print('Found %s unique tokens' % len(word_index))

data_1 = pad_sequences(sequences_1, maxlen=MAX_SEQUENCE_LENGTH)
data_2 = pad_sequences(sequences_2, maxlen=MAX_SEQUENCE_LENGTH)
labels = np.array(labels)
print('Shape of data tensor:', data_1.shape)
print('Shape of label tensor:', labels.shape)

end = datetime.now()
print 'finished tokenizing and forming tenzors for train dataset, in:', end-start

test_data_1 = pad_sequences(test_sequences_1, maxlen=MAX_SEQUENCE_LENGTH)
test_data_2 = pad_sequences(test_sequences_2, maxlen=MAX_SEQUENCE_LENGTH)
test_labels = np.array(test_labels)
test_ids = np.array(test_ids)

end = datetime.now()
print 'finished tokenizing and forming tenzors, in:', end-start
```

```
Found 81066 unique tokens
('Shape of data tensor:', (363861, 30))
('Shape of label tensor:', (363861,))
finished tokenizing and forming tenzors for train dataset, in: 0:00:19.633515
finished tokenizing and forming tenzors, in: 0:00:20.025953
```

Figure 26: Code for tokenising the corpus.

Next step is to translate our questions into numbers using freshly created word-index. The problem is that lengths of the questions vary and the possible options is to set length on either $Max[question1.tolist() + question2.tolist()]$ or cut of the questions at some point. According to the paragraph 4.2 figure **??** the vast majority of uncleaned words have less than a 30 words,

[23]Recurrent neural network

hence for a cleaned (see 4.3) subset a maximum of 30 words will be enough.
Basically, the main steps for input data creation are made with few technicals steps left:

(a) Index a pre-trained GloVe[Pennington et al., 2014] file:

```
#Indexing pre-trained glove vectors. We use the largest corpus available, which is availible here:
#https://nlp.stanford.edu/projects/glove/
#the structure of the document is word - 300d vector/word - 300d vector/etc
start = datetime.now()
print('Indexing word vectors')

embeddings_index = {}
f = open(pth_in+'glove.840B.300d.txt')
count = 0
for line in f:
    if count%100000==0:
        end = datetime.now()
        print 'finished indexing:',count, ' in: ', end-start
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
    count+=1
f.close()

end = datetime.now()
print 'GloVe library indexed, time taken:',end-start
print('Found %d word vectors of glove.' % len(embeddings_index))
```

Figure 27: Code for indexing GloVe file.

(b) Add a vector representation for each token found in figure **??** or as it technically called - embedding matrix creation.

```
start = datetime.now()
#Introduce GloVe embeddings, which were read previously
print('Preparing embedding matrix')
not_found_words=[]
nb_words = min(MAX_NB_WORDS, len(word_index)) + 1

embedding_matrix = np.zeros((nb_words, EMBEDDING_DIM))
for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector
    else:
        not_found_words.append(word)
print('Null word embeddings: %d' % np.sum(np.sum(embedding_matrix, axis=1) == 0))
end = datetime.now()
print 'finished embedding, in:', end-start
```

```
Preparing embedding matrix
Null word embeddings: 18564
finished embedding, in: 0:00:01.565141
```

Figure 28: Creating an embedding matrix.

(c) "Train dataset" division into train itself and validation datasets. Note we create a train dataset by stacking question 1 (aka data_1) and question 2 (aka data_2) one under another for the purposes on RNN, which will be explained later.

```
perm = np.random.permutation(len(data_1))
idx_train = perm[:int(len(data_1) * (1 - validation_split))]
idx_val = perm[int(len(data_1) * (1 - validation_split)):]

data_1_train = np.vstack((data_1[idx_train], data_2[idx_train]))
data_2_train = np.vstack((data_2[idx_train], data_1[idx_train]))
leaks_train = np.vstack((leaks[idx_train], leaks[idx_train]))
labels_train = np.concatenate((labels[idx_train], labels[idx_train]))

data_1_val = np.vstack((data_1[idx_val], data_2[idx_val]))
data_2_val = np.vstack((data_2[idx_val], data_1[idx_val]))
leaks_val = np.vstack((leaks[idx_val], leaks[idx_val]))
labels_val = np.concatenate((labels[idx_val], labels[idx_val]))
```

Figure 29: Taking part of the train dataset for validation purposes.

Lets summarise what do we have so far:

(a) We have created a token representation for all questions and put one question after another getting the data of the following shape.

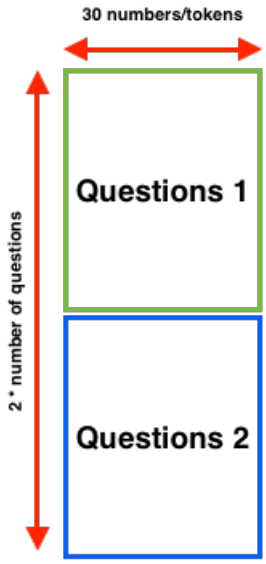


Figure 30: Shape of the input data before embedding layer.

(b) Further, to capture the meaning of the words we pass our 2 dimensional data tensor through the embedding layer to get the 3 dimensional input tensor for our LSTM RNN. The same procedure will be done for data_2_train which has exactly the same structure with the only difference that questions 2 are put above questions one.

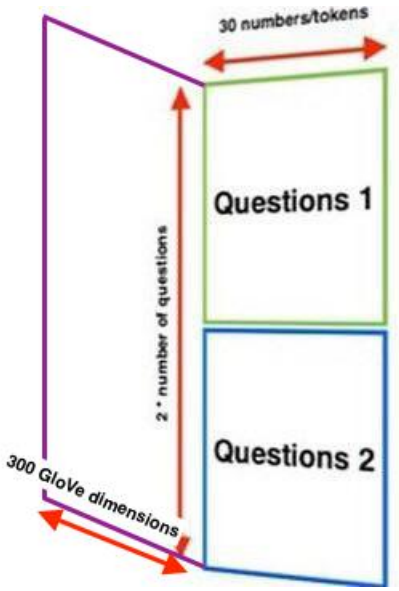


Figure 31: Shape of the input data after embedding layer.

Now, last thing to discuss here is usage of additional features on the second stage of our LSTM RNN to enhance the results. We proceed by creating so-called "leaky-feature" by creating a feature based on the whole corpus of vocabulary, including test data. We create three features:

(a) Feature one, number of repeats of this question in column "question 1";
(b) Feature two, number of repeats of this question in column "question 2";
(c) Feature three, number of repeats of this question in both columns, i.e. number of intersections of two sets, consisting of questions 1 (set 1) and question 2 (set 2).

The logic behind this feature is that frequency of word appearance may give us some information about the question, which may either be duplicate with high probability or that certain important information is lost on the step of text cleaning, which made questions exactly the same, whereas they are not duplicates. Further, the usage of the train dataset corpus is also legitimate, as this data will be available to the company before assessment and it does not leak the labelling of the question.

```
start = datetime.now()
#Generate leaky features. Well, actually they are not that leaky, they just explore the whole
#which is feasible, for Quora, for instance
ques = pd.concat([df_train[['question1', 'question2']],
                  df_test[['question1', 'question2']]], axis=0).reset_index(drop='index')

end = datetime.now()
print 'finished conatenating questions, in:', end-start

#Creating dictionary of type quiestion 1 - question 2
q_dict = defaultdict(set)
for i in range(ques.shape[0]):
    q_dict[ques.question1[i]].add(ques.question2[i])
    q_dict[ques.question2[i]].add(ques.question1[i])

end = datetime.now()
print 'finished creating a dictionary, in:', end-start


def q1_freq(row):
    return (len(q_dict[row['question1']]))


def q2_freq(row):
    return (len(q_dict[row['question2']]))


def q1_q2_intersect(row):
    return (len(set(q_dict[row['question1']]).intersection(set(q_dict[row['question2']])))))


df_train['q1_q2_intersect'] = df_train.apply(q1_q2_intersect, axis=1, raw=True)
df_train['q1_freq'] = df_train.apply(q1_freq, axis=1, raw=True)
df_train['q2_freq'] = df_train.apply(q2_freq, axis=1, raw=True)

end = datetime.now()
print 'finished creating leaky features for train dataset, in:', end-start

df_test['q1_q2_intersect'] = df_test.apply(q1_q2_intersect, axis=1, raw=True)
df_test['q1_freq'] = df_test.apply(q1_freq, axis=1, raw=True)
df_test['q2_freq'] = df_test.apply(q2_freq, axis=1, raw=True)
```

Figure 32: Code for leaky feature creation.

2. Next step is one of shortest cells with the code, however it took more than 4 hours to properly setup the LSTM model.

```
#Define the model structure
embedding_layer = Embedding(nb_words,
                            EMBEDDING_DIM,
                            weights=[embedding_matrix],
                            input_length=MAX_SEQUENCE_LENGTH,
                            trainable=False)
lstm_layer = LSTM(num_lstm, dropout=rate_drop_lstm, recurrent_dropout=rate_drop_lstm)

sequence_1_input = Input(shape=(MAX_SEQUENCE_LENGTH,), dtype='int32')
embedded_sequences_1 = embedding_layer(sequence_1_input)
x1 = lstm_layer(embedded_sequences_1)

sequence_2_input = Input(shape=(MAX_SEQUENCE_LENGTH,), dtype='int32')
embedded_sequences_2 = embedding_layer(sequence_2_input)
y1 = lstm_layer(embedded_sequences_2)

leaks_input = Input(shape=(leaks.shape[1],))
leaks_dense = Dense(num_dense / 2, activation=act)(leaks_input)

merged = concatenate([x1, y1, leaks_dense])
merged = Dropout(rate_drop_dense)(merged)
merged = BatchNormalization()(merged)

merged = Dense(num_dense, activation=act)(merged)
merged = Dropout(rate_drop_dense)(merged)
merged = BatchNormalization()(merged)

preds = Dense(1, activation=act)(merged)
```

Figure 33: Model setup.

We start by setting up two layers:

(a) First is a well known embedding layer, which will add a third 300 size dimension to our questions (As discussed here: figure **??**).

(b) Second is the LSTM layer itself, which will have 30 boxes as discussed in section 3.2. Each box, will correspond to the input word from the question.
Further, LSTM takes three hyper-parameters:

i. num_lstm - dimensionality of the output word-vector, which will initialise randomly on each epoch. The possible range of values is set at [175, 275] range. The reasoning for this particular numbers is the following: there would be some most important and informative information stores first, however, some other information might not be that useful and will take computational powers without, hence the upper bound is set a bit lower than 300. The lower bound is set from the hypothesis that at least half of the input values should be informative.

ii. Values of rate_drop_lstm, rate_drop_dense are set to default with a little randomness on each iteration. This parameters will ensure the solution to the overfitting problem, hence the value of the parameters may be close to the standard values.

Overall it is important to state that such randomness will allow intrinsic fine-tuning as the fitting is set in a way to ensure the best possible results until there is no improvements in three epoch.

Next step in model setup is to announce the input values, which in our case would be two-dimensional matrices, data_1_train and data_2_train as sequence_1_input and sequence_2_input. It is important to mention that Tensorflow backend Keras requires explicit announcement of dimensions and datatype, which in our case maximum 30 by Oy[24] axis with integer numbers as tokens.

Again, embedded_sequences_1 and 2 are nothing but input arrays passed through the embedding layer (As discussed here: figure **??**).

Finally, x1 and y1 are the output of the LSTM RNN layer. The process is the same as discussed in section 3.2 with an important remark, that by default return sequence parameter of LSTM RNN is set to false, which basically means that neural network compresses two out of three dimensions, returning 2-D array with the following configuration.

[24]By some magic API for Keras and Tensorflow changes very often, so do not get confused with shape=(MAX_SEQUENCE_LENGTH,) it means that we set a bound for Oy axis and no bound for Ox axis

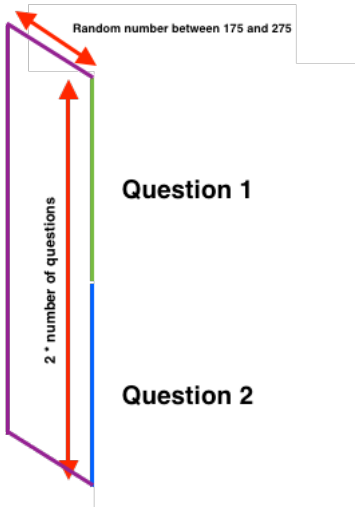


Figure 34: LSTM RNN return tensor.

The best log-loss statistics has been achieved for 204 dimensionality.
One of the latest steps is to merge x1 and y1 LSTM output tensors to put them through other Densely-connected neural network to reduce dimensionality and proceed out leaky features. At this stage it becomes clear that the necessity for two data_train inputs with stacking is explained by the requirement for symmetric input to the second layer and increase in the data to proceed.
Again, the Dense layer has a space dimensionality parameters which are further randomly choses as a way to fine-tune the parameters. The same is true about dropout parameters which are set for a standard values with a slight randomness.

3. Final step a LSTM RNN training took about 18 hours with each epoch taking up to 1.5 hours to calculate. This can be explained by the calculations on CPU rather than GPU processors for which Tensorflow has been designed. That is also the main reason of such a random fine-tuning of the parameters instead of the more classical grid search approach.
Results of the LSTM are astonishing, which allows to identify even rather difficult in terms of the "mechanical" features questions, such as for instance:

	B	C	D	E
	id	question1	question2	is_duplica
	358105	Is there life on other planets and are they more advanced than us?	Is it possible that there is life in other planets?	1
	281845	How do I can boost my self confidence?	How do I improve confidence?	1
	227026	What is most important thing in life? Is it money or relations or status?	What is the most important thing in life? And why?	1
	233141	How will the ban of 500 and 1000 rs notes will affect land prices?	How will abolishing Rs. 500 and Rs. 1000 notes affect the real estate businesses in India?	1

Figure 35: Sample from LSTM results identified as duplicates with high probability.

Indubitable, there is a lot of further space for improvement, however it requires more computational resources and time

4.7 XGBoost time

One of the main developments and growth of the machine learning to deep learning is the ability to combine the results of different models, or as it called stacking, to get a synergy from usage the best

of various model. Hence, using XGBoost is a form of stacking to produce a better results.
Before proceeding with the XGBoost itself one needs to merge the LSTM RNN outputs with the features created before.
The XGBoost setup is much easier and requires few parameters, such as:

1. Objective function, which in our case is a logistic function as we are trying to predict probability of questions to be a duplicates;
2. Evaluation metrics is also classical for probability predictions type of model, which is a log-loss metrics;
3. Hyper-parameter one - "eta" - learning rate has been fine tuned using scikit-learn[sci, 2017] grid search at the rate 0.1
4. Hyper-parameter two - "max_depth" - maximum depth of the tree has also been fine tuned using scikit-learn[sci, 2017] grid search at the value 5

Last step is to fit the model and voila 0.29 log-loss on the leader board!

5 Conclusion

5.1 Achieved results

Summing this up, we achieved the proposed goals, namely we explored a number of Natural Language processing techniques from the most simple ones such word share or sentence length calculations to the advanced state-of-art solutions, which proved an ability to capture complex structures underlying human language. In particular, we explored Global Vectors model developed by Pennington, Socher, and Manning [2014], which significantly improved results of our model and proved an ability to capture the semantic meaning of the sentences.
Further, we explored some of the Deep learning techniques, which showed extremely good results. In particular, this paper explored a classical XGBoost solution, which is relatively easy and not time consuming to configure and fit and one of the most advanced neural network solution - LSTM recurrent neural network, which showed brilliant results with Global Vector embedding matrix.

5.2 Further steps to improve the model

Of course, there is a much space to develop and practice in. I would suggest three main steps to achieve higher results, which for this competition reached an astonishingly low log-loss results of 0.11580 (team: "DL guys"):

1. There is a much space to develop further features and develop previous ones through adding word2vec or tf-idf techniques or, for instance, sentence embedding techniques, such as Doc2Vec, Sent2Vec. Further, our features were only developed for the so called one gram representation of language, other n-grams, i.e. seeing language as a combination of n words, could produce even more useful information;
2. Given enough computational resources our LSTM RNN could be further improved by appropriate fine tuning techniques;
3. The most promising fields of development is usage of several layers of stacking with more models involved. For instance, first place solution used 4 stacking layers: around 300 classification algorithms on the first layer (KNN, XGB, LGBM, etc.), around 150 models on second layer; 2 linear models on third layer with blending on the fourth layer.

References

The common crawl corpus, June 2017. URL commoncrawl.org.

Fuzzy string matching in python, June 2017. URL pypi.python.org/pypi/fuzzywuzzy.

Your home for data science, June 2017. URL www.kaggle.com.

Quora question pairs, June 2017. URL www.kaggle.com/c/quora-question-pairs.

Cs224d: Deep learning for natural language processing, June 2017. URL cs224d.stanford.edu.

Keras: The python deep learning library, June 2017. URL keras.io.

Understanding lstm networks, June 2017. URL colah.github.io/posts/2015-08-Understanding-LSTMs/.

scikit-learn, machine learning in python, June 2017. URL scikit-learn.org/stable/index.html.

Stemming and lemmatization, June 2017. URL nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html.

Asif Agha. Language and social relations. Number 24. Cambridge University Press, 2007.

Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. IEEE transactions on neural networks, 5(2):157–166, 1994.

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. Journal of machine learning research, 3(Feb):1137–1155, 2003.

Steven Bird. Nltk: the natural language toolkit. In Proceedings of the COLING/ACL on Interactive presentation sessions, pages 69–72. Association for Computational Linguistics, 2006.

John A Bullinaria and Joseph P Levy. Extracting semantic representations from word co-occurrence statistics: A computational study. Behavior research methods, 39(3):510–526, 2007.

Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining, pages 785–794. ACM, 2016.

Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. Journal of Machine Learning Research, 12(Aug):2493–2537, 2011.

GB Dantzig. Linear programming and extensions, princeton univ. Press, Princeton, NJ, page vii, 1963.

Scott Deerwester, Susan T Dumais, George W Furnas, Thomas K Landauer, and Richard Harshman. Indexing by latent semantic analysis. Journal of the American society for information science, 41 (6):391, 1990.

JB Heaton, NG Polson, and JH Witte. Deep learning in finance. arXiv preprint arXiv:1602.06561, 2016.

S Hochreiter. Untersuchungen zu dynamischen neuronalen netzen [in german] diploma thesis. TU Münich, 1991.

Sepp Hochreiter and Jürgen Schmidhuber. Lstm can solve hard long time lag problems. In Advances in neural information processing systems, pages 473–479, 1997.

Thomas K Landauer and Susan T Dumais. A solution to plato's problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge. Psychological review, 104(2):211, 1997.

Rémi Lebret and Ronan Collobert. Word emdeddings through hellinger pca. arXiv preprint arXiv:1312.5542, 2013.

Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In Soviet physics doklady, volume 10, pages 707–710, 1966.

L Álvarez Menéndez, Francisco Javier de Cos Juez, F Sánchez Lasheras, and JA Álvarez Riesgo. Artificial neural networks applied to cancer detection in a breast screening programme. Mathematical and Computer Modelling, 52(7):983–991, 2010.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In Advances in neural information processing systems, pages 3111–3119, 2013.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In Empirical Methods in Natural Language Processing (EMNLP), pages 1532–1543, 2014. URL http://www.aclweb.org/anthology/D14-1162.

Joël Plisson, Nada Lavrac, Dr Mladenić, et al. A rule based approach to word lemmatization. 2004.

Martin F Porter. An algorithm for suffix stripping. Program, 14(3):130–137, 1980.

Douglas LT Rohde, Laura M Gonnerman, and David C Plaut. An improved model of semantic similarity based on lexical co-occurrence. Communications of the ACM, 8:627–633, 2006.

Hamza Shaban. Google's alphago beats the world's best go player — again. The Washington Post, May 2017. URL www.washingtonpost.com/news/innovations/wp/2017/05/26/googles-alphago-beats-the-worlds-best-go-player-again/.