Technische Universität Wien

Natural Language Processing

Noah Weber

Vienna, 6. June 2018

Table of content

1 Introduction

Goal of this paper is to give Introduction to Natural Language Processing (NLP for short), to elaborate different techniques that propose themselves as solutions to different problems in context of text processing, to explain potential pitfalls and drawbacks and finally to present some real-world python code that should give us an practitioners overview of the situation. We are going to rely heavily on [1] as well as some other references. Regarding python code we are going to implement word2vec For that we are going to consult GenSim package and its creator Radim Řehůřek and [7,8]. And finally we are going to discuss different neural network architectures that have evolved over the years specifically to tackle text problems, like classifying whether the question was sincere or not.

2 Natural Language Processing

So what is Natural Language Processing (NLP)? Firstly natural language represents a way we, humans, communicate with each other. Specifically speech and text. Text is all around us, telephones computers, books. This is what we are going to mainly work on. Speech is even more present if we think about it. It is easier to speak than to write, and in aggregate we do it a lot more. Both of them are data. Given the presence and importance of this data we ought to have methods to understand and reason about natural language using technology. To avoid rigid definitions, we can freely describe NLP as group of algorithms and methods that are created to analyse natural language and to give informative answers about this large dataset. But not only computer knowledge is sufficient to answer these problems. Just think about problems when we face when learning a new language. Therefore we must turn to something that will supplement our methodology, namely to linguistics. We can divide our usage of language in distinct levels. Linguistic levels (see [2, ch. 7.1]). Beginning of the levels is phonetics, which are the sounds of a language. Phonology describes the organization of those sounds into words. Next linguistic level is morphology and studies how words are different from their lemma (root) or headword depending on their placement in a sentence. The syntax deals with the structure of the sentence and the grammar. Semantics then studies the meaning of sentence by logical analysis, e.g. it can differentiate nonsensical sentences: "Dead dog was alive", since dead and alive are contradictive. Pragmatics level is there to put the meaning of the sentence into context, e.g. I am out of the woods. Did I solve my problem or have I had a picknick with my family? We can already object that with only these rules it gets pretty hard to encode them and use them sufficiently. What about all the other rules that we do not know or we forget. Natural language is highly ambiguous. e.g. I played with her toys or I played with her heart. What can we do about it? Find a way to parse natural language and use machine learning algorithms on train data so they can learn and test them on test data. There are already was that achieve sufficiently good results. One interesting challenge would be to classify article based on their subject. Well it turns out that through visual representation of the statistically relevant occurrences in the text which leads to the choice of the answer of the problem, we can explain thought process better than thinking about it ourselves (see Bag of Words down below). Another interesting implementation can be found in [3, Sec.16.1.2], where prediction of the singularity or plurality of the subject and the verb in the sentence, which must coincide for the grammar to be correct, is deduced. We already said that NLP can not function alone as a field in computer science, we ought to use linguistics to reinforce our predictions. For example when we design a specific algorithm we can use linguistics to guide us in what direction should we move. Another situation where linguistics could come in handy is when we do not have enough training data. We could simulate some data using linguistic rules when certain conditions are met.

3 Statistical Methods/Algorithms

3.1 N-gram

Let us say we want to distinguish documents that are written in English or German (see [1]). What would be insightful metric? As it turns out, frequencies of letters together with frequencies of bi-grams, which are two consecutive letters, are able to predict documents. Bi-grams are just taken from a more general method called N-grams. N-gram can be defined as a sequence of number of items (numbers,digits, words, letter etc.) from a corpus, which is a large collection of text. Usually we speak of words that represent that sequence but we will see letters can be informative also. To introduce us to N-grams we shall use [3]. Let us start by trying to compute probability $\mathbb{P}$(w|h). In other words, we are interested in probability of a word w occurring given some history h. For example:

$$\mathbb{P}(\text{to}|\text{I would like})$$

We could start naively and try to estimate the probability based on relative frequency. Let us take a very large corpus, count the number of times we see I would like and count the number of times we see I would like to. Using basic probability equation we are able to answer a question, given that we have observed history h, what can we infer about w?

$$\mathbb{P}(\text{to}|\text{I would like}) = \frac{\mathbb{P}(\text{I would like to})}{\mathbb{P}(\text{I would like})}$$

So lets say that our corpus is the whole web, than we could easily just count the occurrences and estimate the probability. This solution is plausible if our corpus and problem that we are trying to solve has met some conditions. For example determining whether a document is English or German encounters following problems. Firstly, language is creative. Even if we assume we can calculate it, we are bound to find some misrepresentations of occurrences, or even not calculate the ones that were not used yet. Secondly what about computational power to calculate that? Essentially we are calculating number of occurrences of I would like in a set of ALL possible 4 word sentences. It is a lot. Let us solve first German-English challenge and then we will talk about another way of estimating probabilities if conditions are not met (see subsection 3.2). As it was already alluded we are going to use bi-grams as a sequence of consecutive letters. Let us take 28 letters (a-z) and a symbol for special character, e.g. numbers. Let us represent our document as $x \in \mathbb{R}^{784} \cong \mathbb{R}^{28\times 28}$, where each vector entry $x_{[i]}$ is actually number of occurrences of bi-gram $\alpha\beta$ in our entire document D.

$$x_{[i]} = x_{\alpha\beta} = \frac{\#_{\alpha\beta}}{|D|}.$$

So we computed this vector and we decide to plot number of occurrences:

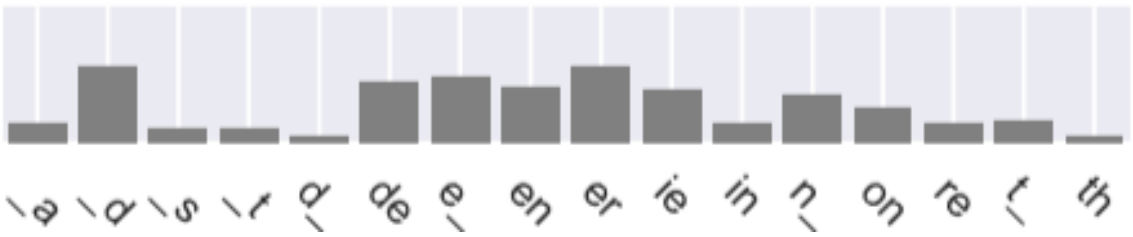


Fig. 1 – Frequencies of bi-gram from one document (from: [1])

Just by looking at it we would be inclined to say that it is a German document. Number of de, er and so on is just not so common in English language. But we must have quantitative way to decide. We said in beginning that we want to differentiate between English and German documents. That is so called classification problem. Even though it is not in scope of this paper let us tackle it briefly just to get the motivation of a quantitative method used to decide.

3.1.1 Classification

Firstly we ought do differentiate between binary and multi-class classification. Lets say we want to classify where apartment is located based on price and size

Fig. 2 – Data from craigslist concerning price and size (from: [1])

The blue circles are Dupont Circle and the green crosses are in Fairfax. Each data point is actually an apartment and we see that we they are linearly separable i.e. we can use a straight line to differentiate between these two groups. If we use an approach:

$$f(x) = x \cdot w + b$$

where x is a vector or data, w are weights, $\cdot$ is dot-product and b is a scalar, we get that range of this linear function is $[-\infty, \infty]$. Now in order to use it as a binary classifier it is common to use sign function in order to get -1 and 1 for two distinct classes. In other words we get following linear model that we are going to use in our document example:

$$\hat{y} = sign(x \cdot w + b)$$

That was an example of a binary, two case classification. But in reality we are going to have a multi-class classification problem. We need to assign an example to a k different classes. Analogous example (see [1]) would be again to differentiate between languages, but now between English, French, German, Italian, Spanish, Other. With that we get six weight vectors w^{En}, w^{Gr}, $\cdots$, and biases. One for each possible language. With that we would predict a language that results in highest possible score, e.a.

$$\hat{y} = f(x) = \underset{M \in \{En,Fr,Gr,It,S,O\}}{\operatorname{argmax}} x \cdot w^M + b^M$$

We can see why that makes sense. Some of the entries of x will be negative, close to 0 or large, it all depends on the weight w that we trained. So when we give our new data vector x_{new} to our model, we ought to get the biggest value for the projected language.

Now that we know what the procedure is, we proceede to train the model on training set, and to test its accuracy on the testing set. Training is effectively finding out best possible w in order to achieve highest accuracy for our model. Once we have a reliable model we can use it on new data points to classify them. We also need to remind that sometimes it is not even possible to separate points with a straight line (or a hyper plane in a case we have more than two features). That is subject of its own, namely non-linear classifiers and is out of scope of this paper.

3.1.2 Bi-gram

Now that we have a quantitative method to separate the groups, let us turn to our problem of document classification. Is it German or English? As we already noted, grams that we are going to use are character bi-grams. Sequence of 2 characters. We will predict document as English if $f(x) = 1$ and German otherwise. With that we get following results:

Fig. 3 – Character bi-gram plot (from: [1])

Documents in English are blue, and German are green. Values that are low on both cases represent pairs that are either rare or common in both documents. We should also point out, to avoid confusion, that we had here much more features (predictors) and only two classes to predict. Therefore we can not have a visual representation as in Fig.2. Now we are in higher dimensional space and logical representation is impossible. Therefore we ought to think of these high dimensional models not as lines, but rather as assignment of weights w to its features.

3.2 Alternative with N-grams

As we said we could potentially encounter some difficulties when using frequentistic approach to estimate probabilities. We need a smarter way to address this issue. Before we go into solution let us just solidify our notation. (As in [3] Chapter 4, Page 3) When we say $\mathbb{P}$(to) what we actually mean is the probability of the random variable X_i taking on the value to i.e. $\mathbb{P}(X_i = \text{to})$ Sequence of n words we are going to notate as w_1,w_2,$\cdots$,w_n or as w_1^n. When we talk about joint probability of a sequence of N words, i.e. $\mathbb{P}(X_1 = w_1, X_2 = w_2, \cdots, X_n = w_n)$ than we are going to use notation $\mathbb{P}(w_1, w_2, \cdots, w_n)$. With that out of the way, how can we compute the probability of $\mathbb{P}(w_1, w_2, \cdots, w_n)$? Remember we need it as a part of formula $\mathbb{P}(\text{to}|\text{I would like}) = \frac{\mathbb{P}(\text{I would like to})}{\mathbb{P}(\text{I would like})}$. Well one solution is to use chain rule of probability

$$\mathbb{P}(X_1, \cdots, X_n) = \mathbb{P}(X_1)\mathbb{P}(X_2|X_1)\mathbb{P}(X_3, |X_1^2)\cdots\mathbb{P}(X_n|X_1^{n-1}) = \prod_{k=1}^{n}\mathbb{P}(X_k|X_1^{k-1}) \tag{1}$$

if we apply it to words we get eventually:

$$\mathbb{P}(w_1^n) = \mathbb{P}(w_1)\mathbb{P}(w_2|w_1)\mathbb{P}(w_3, |w_1^2)\cdots\mathbb{P}(w_n|w_1^{n-1}) = \prod_{k=1}^{n}\mathbb{P}(w_k|w_1^{k-1}) \tag{2}$$

We could indirectly see from (2) equation that probability can be calculated from a multiplication of a number of conditional probabilities. But that is not very helpful since we get back to same issues.
To resolve this problem we are not going to look at the entire history but rather we can approximate the history by just looking at a few last words. This is actually the essence of N-gram model. Lets take a look what would that imply for bi-gram model, even tough same intuition is behind N-gram. What we wish to achieve is approximate the probability of a word given all the previous words $\mathbb{P}(X_n|X_1^{n-1})$ by using only the previous word, i.e. $\mathbb{P}(X_n|X_{n-1})$. To put it in context let us demonstrate it on a simple sentence, we will not compute the probability of

$$\mathbb{P}(\text{home}|\text{Yesterday I was on my way})$$

what we rather do is approximate it with:

$$\mathbb{P}(w_n|w_1^{n-1}) \approx \mathbb{P}(w_n|w_{n-1}) \tag{3}$$

Yes, this is already familiar Markov assumption that states we can predict probability of future event without looking too far in the past. As we see in the case of bi-gram we are looking only at the last word, if we move on to tri-gram that would implicate looking at the 2 last words and more general with **N**-gram we would be looking at N-1 words in the past. More formally

$$\mathbb{P}(w_n|w_1^{n-1}) \approx \mathbb{P}(w_n|w_{n-N+1}^{n-1})$$

Now we can also ask ourselves what would a probability of a sequence of words look like. For that let us substitute (3) in (2) and we get:

$$\mathbb{P}(w_1^n) \approx \prod_{k=1}^{n}\mathbb{P}(w_k|w_{k-1}) \tag{4}$$

Now how do we estimate (4) and (2), that is how do we estimate bi-gram and N-gram probabilities? We are going to use maximum likelihood estimate or MLE. Lets say we want to use it to compute the estimate of a bi-gram (man can argue that would suffice since N-gram is just a

multiplication of bi-grams). in that case MLE states that to compute the probability of a word x, given previous word y we ought to compute probability of bi-gram $\mathbb{P}(xy)$ and then normalize by the sum of all possible bi-grams in that corpus that have word x in them. With that we make sure that values are between 0 and 1 what we need if we are talking about probability.

$$\mathbb{P}(w_n|w_1^{n-1}) = \frac{\mathbb{P}(w_n w_{n-1})}{\sum_w \mathbb{P}(w_n w_{n-1})}$$

If we think for a moment divisor is actually only $\mathbb{P}(w_{n-1})$ meaning that we get finally:

$$\mathbb{P}(w_n|w_1^{n-1}) = \frac{\mathbb{P}(w_n w_{n-1})}{\mathbb{P}(w_{n-1})}$$

Let us look at an simple example (see [3]): We need to augment beginning of the sentences with <s> in order two preserve bi-gram properties of the first word of the sentence and also add special symbol </s> at the end of sentences in order to respect probabilistic properties.

- <s> I am Sam </s>
- <s> Sam I am </s>
- <s> I do not like green eggs and ham </s>

With this simple corpus we can calculate some bigram probabilities

$$\mathbb{P}(\text{I}| < s >) = 0.67 \; \mathbb{P}(\text{do}|\text{I}) = 0.33$$

And finally let us look at whatdo we get for general N-gram case when applying MLE

$$\mathbb{P}(w_n|w_{n-N+1}^{n-1}) = \frac{\mathbb{P}(w_n w_{n-N+1}^{n-1})}{\mathbb{P}(w_{n-N+1}^{n-1})}$$

3.3 Bag of Words

In its essence Bag of Words-(BoW) is just a way of extracting words from a document in order to use machine learning techniques on them. We need to quantitatively represent these words from a corpus, since machine learning algorithms work only with numbers. Therefore we need to create vector of numbers. Extraction tool is exactly bag of words. We ought to know two things when applying BoW. Vocabulary of known words in text and a way of measuring presence of known words. The name bag comes from the fact that we do not care about structure of the words, we are interested only in occurrences of these words in corpus. We do it with a premise that the more we see certain words the more we are able to conclude. But that can lead to its own drawbacks. We will remedy it with Term Frequency-Inverse Document Frequency (TF-IDF), but more on that later on. So what does it look like on an example? Let us borrow an example from [4]

- It was the best of times
- it was the worst of times,
- it was the age of wisdom,
- it was the age of foolishness,

We will look at each bullet point as an document on its own and all of them represent corpus. Next step is to make our vocabulary, we are going to exclude commas and repetitive words.

- “it”
- “was”
- “the”
- “best”
- “of”
- “times "
- “worst”
- “age”
- “wisdom”
- “foolishness”

So now we know vocabulary has 10 words. We can encode it in a vector. 1 from being present in a document (here sentence) 0 for being apscent. It is also known as 1-hot encoding. For the first document it would imply:

- “it”=1
- “was”=1
- “the”=1
- “best”=1

- “of”=1
- “times-1
- “worst”=0
- “age”=0
- “wisdom”=0
- “foolishness”=0

Written as a vector other 3 would look like:

```
"it was the worst of times" = [1, 1, 1, 0, 1, 1, 1, 0, 0, 0]
"it was the age of wisdom" = [1, 1, 1, 0, 1, 0, 0, 1, 1, 0]
"it was the age of foolishness" = [1, 1, 1, 0, 1, 0, 0, 1, 0, 1]
```

Fig. 4 – Vector encoding (From [4])

So now we have a trivial example that we can easily extend to other sentences and eventually we would get thousands of books in our corpus. That implies we would have large dimensions, and as we can see these vectors are pretty sparse. They require more memory and computational power, and not all algorithms can handle efficiently these dimensions. We have to take some actions to reduce sparsity. As first we can use some text cleaning techniques. Ignoring commas, punctuation and common words. We can also reduce words to their lemma (root) i.e. Lazily->lazy. Another tool that we can use is with tokenization. We said that when we build a vocabulary we separate only words. But what if we take bi-grams, or trigrams instead? That reduces dimension significantly. Example of reducing the standard problems of BoW is StanfordCoreNLP. This algorithm that is based on Gibbs sampling (which is a Markov Chain Monte Carlo algorithm). We wont go into much details here we can only reference it [1],[6]. What we are interested is the pipeline-design behind StanfordCoreNLP. It is built following way:

$$\text{Tokenizer} \rightarrow \text{Ssplit} \rightarrow \text{POS} \rightarrow \text{NER} \rightarrow \text{Dcoref.}$$

To showcase why we used it we need to explain only what does POS step do. ”POS” assigns to each couple of words its part-of speech-tag (POS-tag), e.g.:

"[NP the girl] [PP with] [NP the toy] [VP opened]
[NP the elevator] [PP without] [NP a key]",

where "NPïs noun phrase, "PP"prepositional phrase and "VP"verb phrase. Let us look now at dense embedding vector (see [1] page 85)

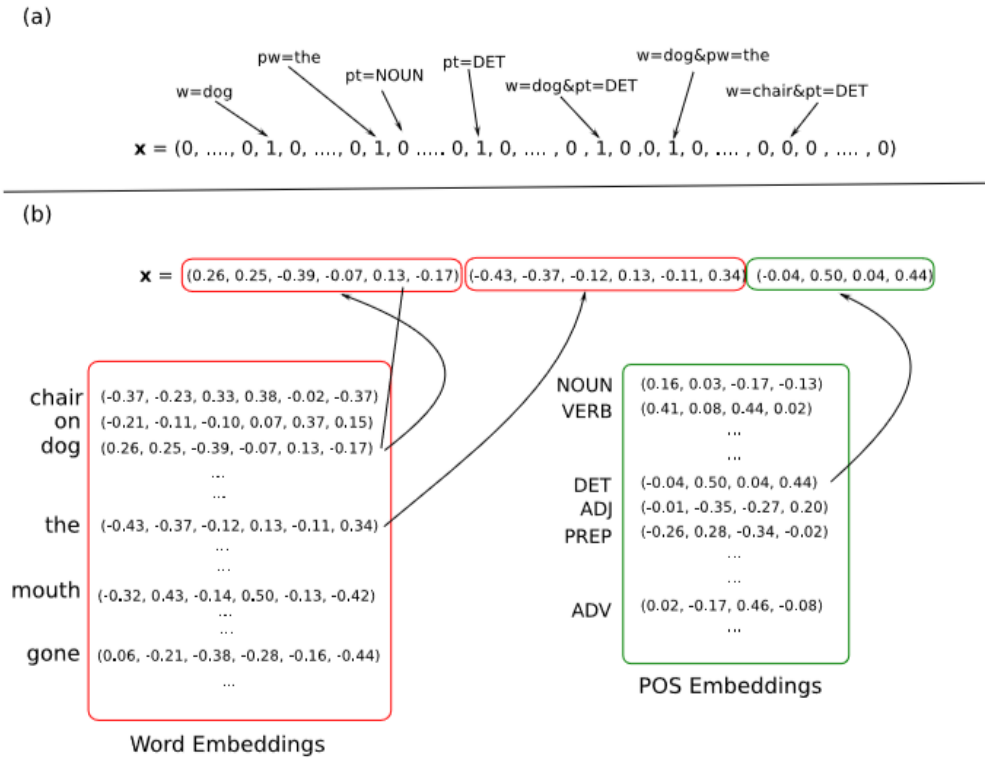


Fig. 5 – Dense Representation-example (From [1])

Now we have reduced dimensions, but we also need to take into account weighting of the words. For example, if we read a corpus about mountains we want more frequent words, words that we already know play important role, weighted down less than less famous words. More generally we know that äör "the"will always have it presence but they do not hold any useful information. We need to weight them down. Lets make it more formal. We already made 1-hot encoding, meaning that every vector (representing one document in corpus) has 1 or 0. Now in order to weight it down we multiply it with following weight, Term Frequency-Inverse Document Frequency (TF-IDF) weight.

$$\frac{\#_m(w)}{\sum_{w' \in m} \#_m(w')} \times \log\left(\frac{|M|}{|\{m' \in M : w \in m'\}|}\right)$$

"Term Frequency"means firstly we modify the 1-hot approach so that in a vector we get a frequency of the occurrence of that word in a document. Ïnverse Document Frequency"we are interested in how rare this word is. w is our word, m is one document in corpus (collection of documents) M. Lets think about it for a moment. If the word is frequent than the term in logarithm function will converge to 1 meaning that the whole equation will go to 0 indicating that we are not interested in words that appear all the time everywhere. Even tough Bag of Words approach is relatively easy there a couple of pitfalls. Firstly how do we proceed when creating a vocabulary, single words, bi-grams tri-grams? We need to be careful about sparsity. When our corpus is large enough that could impose serious computational problems. And lastly we disregard any context or order. It is just important that we throw the words in the bag and count, whether it is (this is easy) or (is this easy) we do not care. Which limits our predictive possibilities.

3.3.1 Continuous Bag of Words-CBOW

CBOW is basically a text processing technique that is integrated with neural network. It offers an approach to solving an interesting issue. It is a in a way a modification of Bag of Words in a sense that here we look at the words from both side of a desired prediction word. We will start with our basic 1-hot encoding and use a simple neural network to train the model and finally predict a probability using softmax as output function. Lets say we have following problem (see [5]) "The recently introduced continuous Skip-gram model is an efficient method for learning high-quality distributed vector representations that capture a large number of precises syntatic and semantic word relationships." Lets say we go over a text and we can not decipher whether learning is really learning or some other word. We can picture it as scanner that goes through every word trying to confirm it fits there with looking at previous and up and coming words i.e.

Fig. 6 – Continuous bag of words (From [5])

We have our context vocabulary that we are going to encode with 1-hot principle. If the size of the vocabulary is V than we get the following situation (see [5]):

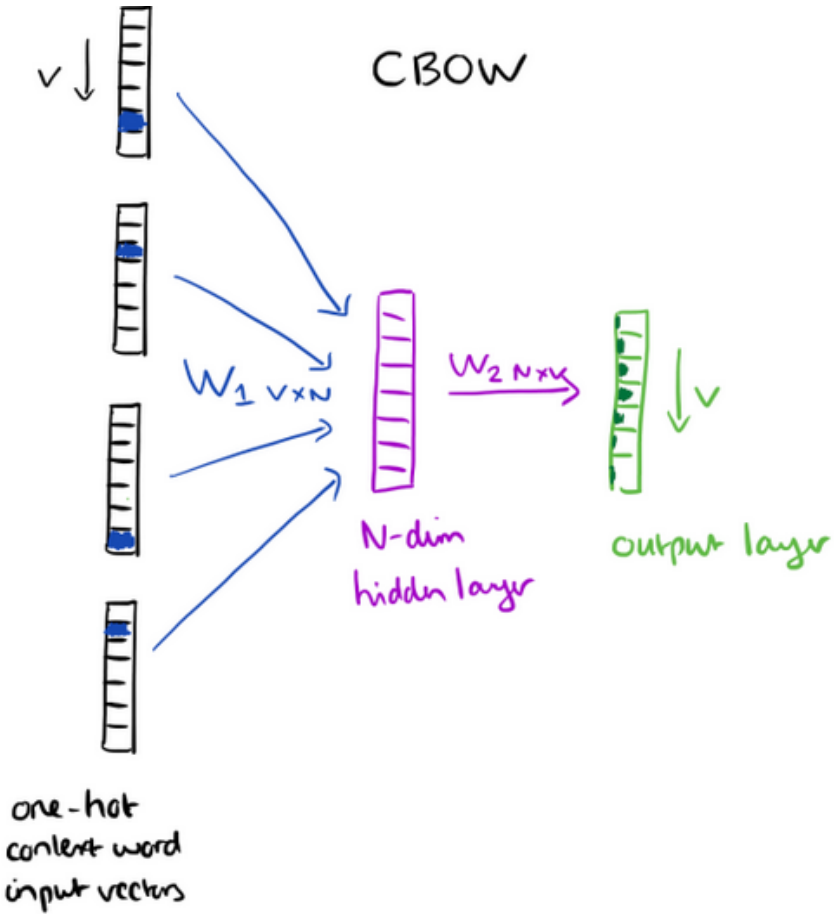


Fig. 7 – Continuous bag of words (From [5])

Now since vocabulary are vectors with only one 1, effectively we just select i-th row when multiplying with first weight Matrix W_1

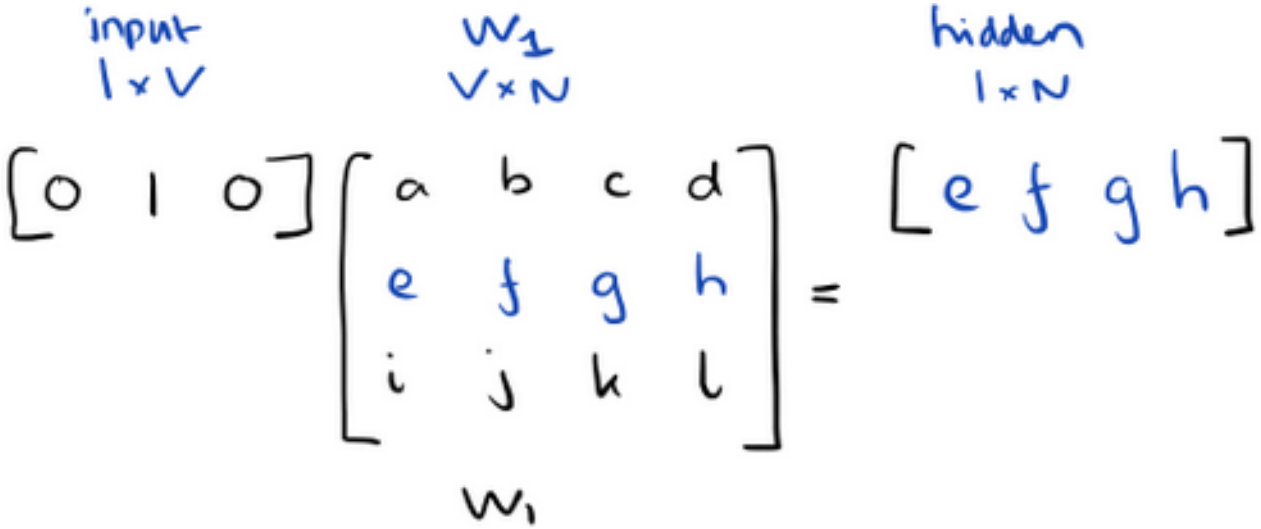


Fig. 8 – Multiplication (From [5])

In order to get the purple vector we ought to apply the activation function for the hidden layer.

In this case lets say we have H hot vectors than we sum e, f, g, h i.e. the 1-hot row and divide by H. With that we get the purple vector which is a node in our network. From a hidden layer to the output layer we use W_2 matrix which can be used to compute the scores for each word in the vocabulary (we ought to know that we train these weights on a training set - see [1]). Finally we get output layer-green vector on which we can apply the softmax function. Softmax function is just generalization of the logit function. We want to have values between 0 and 1 in order to interpret the results as probability. Remember that objective of training this neural network is maximizing the probability of getting the focus word (learning) given the input context words (an,efficient, etc.). Softmax function is simply (see [1] page 24)

$$\text{softmax}(x_{[i]}) = \frac{e^{x_{[i]}}}{\sum_j e^{x_{[j]}}} \text{ for j=1,..., V}$$

Now lets say our focus word is i-th in the Vocabulary, than we just apply softmax on the green vector to calculate the probability. On the other hand lets say that now we have exactly the opposite problem. We have our focus word, but we would like to know what are the context words, i.e.

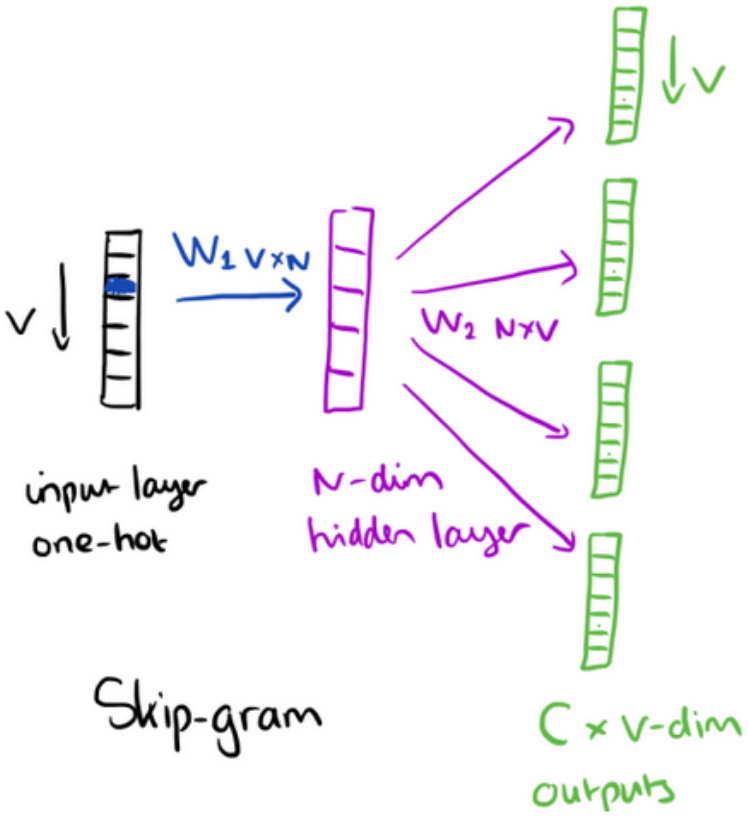


Fig. 9 – Opposite Problem (From [5])

This is so called skip-gram method. Input layer is our focus word encoded in one vector, we proceed backwards meaning that now activation function of the hidden layer is basically just a row from Fig.7. But now instead of one output vector (eventually one probability) we get C multinomial distributions. As with almost every model our goal is to minimize summed prediction error of every context word in our vocabulary. In other words we hope to get for our focus word learning as output layer “an”, “efficient”, “method”, “for”, “high”, “quality”, “distributed”, “vector”.

3.4 Word2Vec

We can not characterize Word2Vec as a single algorithm. Rather it is a method in a software package (see Python below) implementing two different text processing techniques namely CBOW and Skip-Gram which we already touched up on and two different optimization objectives, Negative-Sampling and Hierarchical Softmax. Until now we have analyzed count-based methods that we fed to our neural networks. Another approach would be distributed representation of word meanings. Here each word is described via vector where the importance of the word with respect to some corpus is captured in the different dimensions of the vector, as well as in the dimensions of other words.

Fig. 10 – Distributed representation (From [5])

Relationships of different words in a text with each other are shown expressively. We can forward these vectors again to neural network. This is a much better and intuitive way to represent text. Reason beeing is that if we wanted to add another item to our "memory"than we do not need to increase sparsity of our representation matrix we can already fit it in our 4 vectors since there are infinitely many number between 1 and -1. Now we might confuse some concepts that are similar (i.e. have similar values) but that is exactly what happens with our brain. If we assume that one neuron represents one word than one concept is represented by more than one neuron

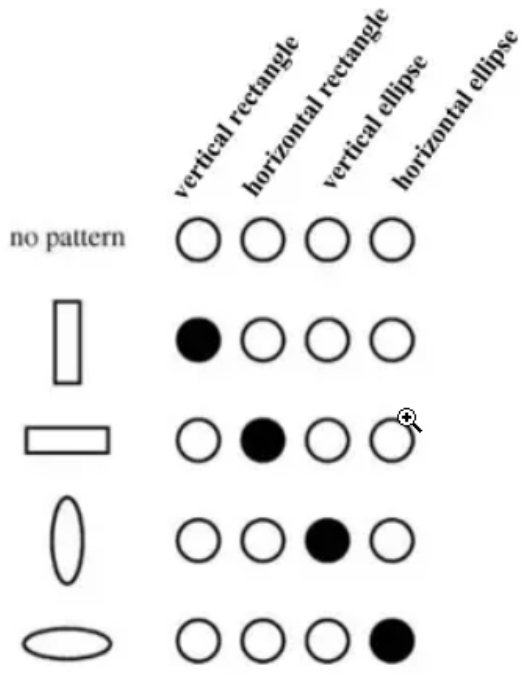


Fig. 11 – Sparse-one hot representation (From [9])

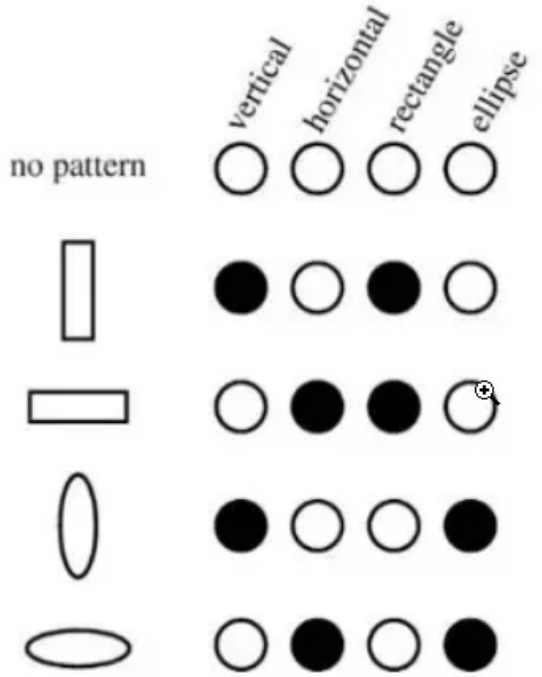


Fig. 12 – Distributed representation (From [9])

Moving on to the second optimization objective. We are going to work with Negative-Sampling variant of Word2Vec as demonstrated in [1 page 117]. Negative Sampling (NS) stands for the fact that we sample from negative set i.e. set that has incorrect predictions or pairing of our words. We have our input vectors for neural network. Goal is now to train the network so that it can differentiate between good and bad pairing. We have a set of good correct word-context pairs D and set of bad correct word-context pairs $\tilde{D}$. $\tilde{D}$ is generated by taking each correct

word-context pair $(w, c) \in D$, sampling k words $w_{1:k}$ and adding each (w_i, c) as a negative example to $\tilde{D}$. Finally we get that $\tilde{D}$ is k times larger than D The goal of the algorithm is to estimate the probability, that the word-context pair (w, c) is in the set of correct pairs D.

$$P(D = 1|\mathbf{w}, \mathbf{c}) = \frac{1}{1 + e^{-s(\mathbf{w}\cdot\mathbf{c})}} \tag{5}$$

If a pair (w, c) is from D than this probability should be close to one and if $(w, c) \in \tilde{D}$ this probability should be close to zero. As we can see probability (5) is modeled as already famous function sigmoid over the score function $s(w, c)$ that compresses our values between 0 and 1 so that we can interpret it as probability.
The corpus-wide objective of the algorithm is then to maximize the log-likelihood of the data $D \cup \tilde{D}$, which is modeled as

$$\mathcal{L}(\Theta; D, \tilde{D}) = \sum_{(\mathbf{w},\mathbf{c})\in D} logP(D = 1|\mathbf{w}, \mathbf{c}) + \sum_{(\mathbf{w},\mathbf{c})\in \tilde{D}} logP(D = 0|\mathbf{w}, \mathbf{c}).$$

Please note that parameter of our model is actually k, number of negative samples. Now to discuss how CBOW and skip-gram comes into play. We are going to formalize what we already discussed in previous subsection about these two methods. Essentially depending on how we define our word context scoring function s. For a multi word context $c_{1:k}$, CBOW variant in Word2vec defines that context vector $\mathbf{c}$ can be written as following sum $\mathbf{c} = \sum_{i=1}^{k} \mathbf{c}_i$. It then defines score function s to be simply $s(w, c) = \mathbf{c} \cdot \mathbf{w}$, which results in our probability

$$P(D = 1|w, c_{1:k}) = \frac{1}{1 + e^{-(\mathbf{w}\cdot\mathbf{c}_1+\mathbf{w}\cdot\mathbf{c}_2+,....,+\mathbf{w}\cdot\mathbf{c}_k)}}$$

Let us think for a second why this works. w and c are vectors (If we have more c than we add them together as noted) So we have a 2 vector a we perform a dot product. IF they are similar their product will result in a larger value because they will coincide on the same spots in the vector. Hence the probability will in total go to 1. Remember values in these vector w and c came from CBOW for example and they are in essence, when translated, probabilities that these two (or more if more c-words) appear together in text. We pre-trained a neural network on this. Opposite approach was skip-gram method. Skip gram method assumes for multi word context $c_{1:k}$ that c_i contexts are independent of each other. We will observe them as k different contexts: $(w, c_1), (w, c_2), ..., (w, c_k)$ Scoring function is similar to CBOW approach only that now we use independence:

$$P(D = 1|w, c_i) = \frac{1}{1 + e^{-(\mathbf{w}\cdot\mathbf{c}_i)}}$$

$$P(D = 1|w, c_{1:k}) = \prod_{i=1}^{k} P(D = 1|w, c_i) = \prod_{i=1}^{k} \frac{1}{1 + e^{-(\mathbf{w}\cdot\mathbf{c}_i)}}$$

Important Negative sampling is the optimization approach on the whole corpus but methods to actually get the distributed representations from Fig.10 are in essence NN (or a extensions of) that learn what the representation vector should look like in regard of the entire corpus and their relationships to each other. Very interesting property of Word2Vec is when we look at it geometrically. Namely we can measure similarity between two vectors u,v cosine measure, i.e.:

$$sim_{cos}(u, v) = \frac{u \cdot v}{||u||_2||v||_2}$$

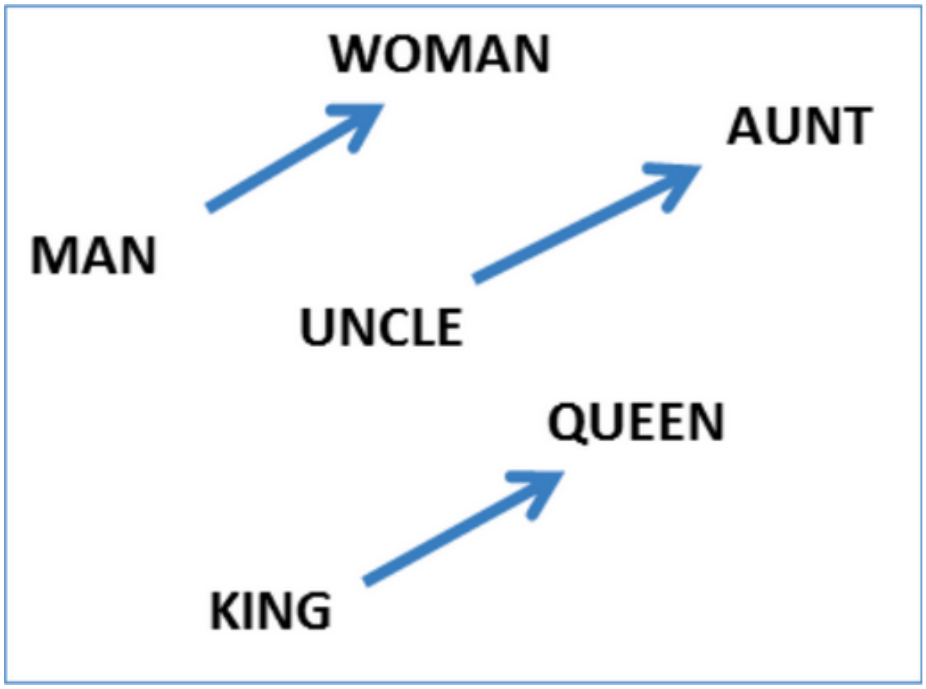


Fig. 13 – Three vector pair illustrating gender relation (From [5])

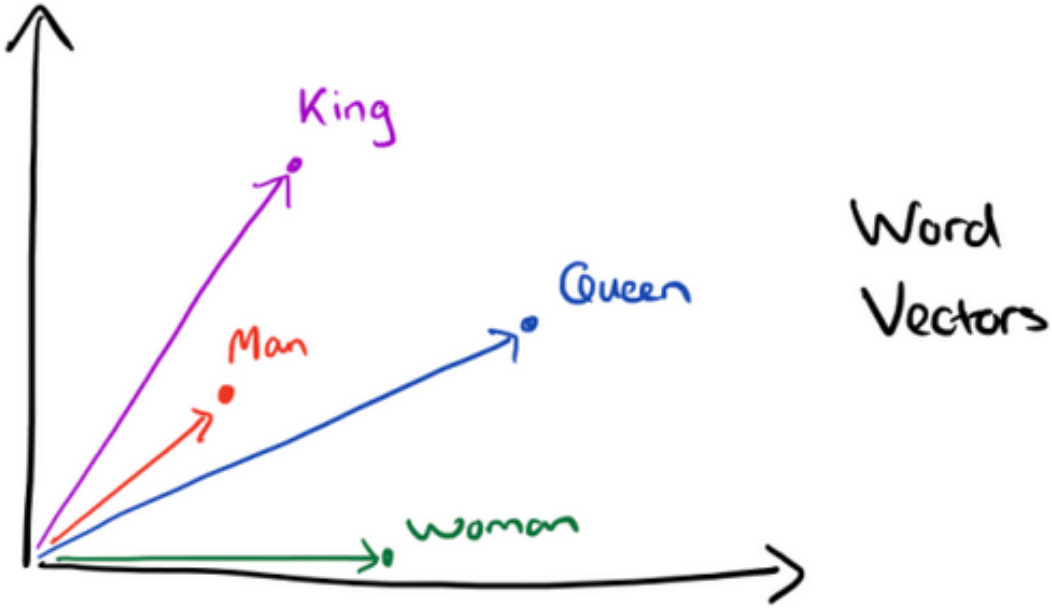


Fig. 14 – Vectors for King, Man, Queen and Woman (From [5])

Cosine similarity measure describes the cosine of the angle between the vectors u and v. Whats the aim of this? Words that are similar will be mapped closely to each other on a plane. Now that we can project vectors on a plane we can answer analogy questions of the form a is to b as c is to ? We answer such question in which perform basic algebra on vectors, i.e. As we can see from Fig. 15 we can answer King – Man + Woman=? (Queen) just by performing some basic älgebra". More specifically what happens here is we calculate it with following function

Fig. 15 – Vector composition of King,Man,Queen and Women (From [5])

$$analogy(a : b \rightarrow c :?) = \underset{\mathbf{v} \in V \setminus \{\mathbf{a},\mathbf{b},\mathbf{c}\}}{\operatorname{argmax}} \cos(\mathbf{v}, \mathbf{c}) - \cos(\mathbf{v}, \mathbf{a}) + \cos(\mathbf{v}, \mathbf{b}).$$

Now that we know how they look like in space (Fig13-15) we can answer some interesting questions. Like what are the most similar words or we can find out the relationships between different cities and countries (Fig 17.)

Relationship	Example 1	Example 2	Example 3
France - Paris	Italy: Rome	Japan: Tokyo	Florida: Tallahassee
big - bigger	small: larger	cold: colder	quick: quicker
Miami - Florida	Baltimore: Maryland	Dallas: Texas	Kona: Hawaii
Einstein - scientist	Messi: midfielder	Mozart: violinist	Picasso: painter
Sarkozy - France	Berlusconi: Italy	Merkel: Germany	Koizumi: Japan
copper - Cu	zinc: Zn	gold: Au	uranium: plutonium
Berlusconi - Silvio	Sarkozy: Nicolas	Putin: Medvedev	Obama: Barack
Microsoft - Windows	Google: Android	IBM: Linux	Apple: iPhone
Microsoft - Ballmer	Google: Yahoo	IBM: McNealy	Apple: Jobs
Japan - sushi	Germany: bratwurst	France: tapas	USA: pizza

Fig. 16 – Skip-gram method trained using word2vec (From [5])

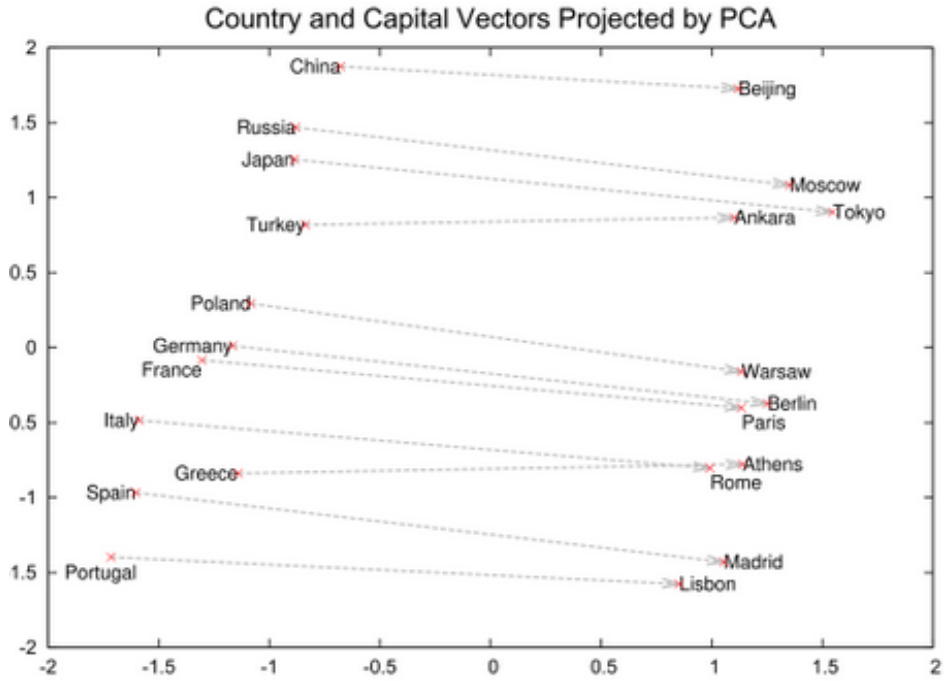


Fig. 17 – Country-capital city relationship with 2-Dim PCA (From [5])

3.5 Python implementation

As we mentioned in introduction we are going to rely on [7,8]. Radim made already beautiful UI that demonstrates what kind of interesting questions we can answer with GenSim and Word2Vec. Making this whole App (Fig. 18) is a combination of GenSim and Bokeh libraries

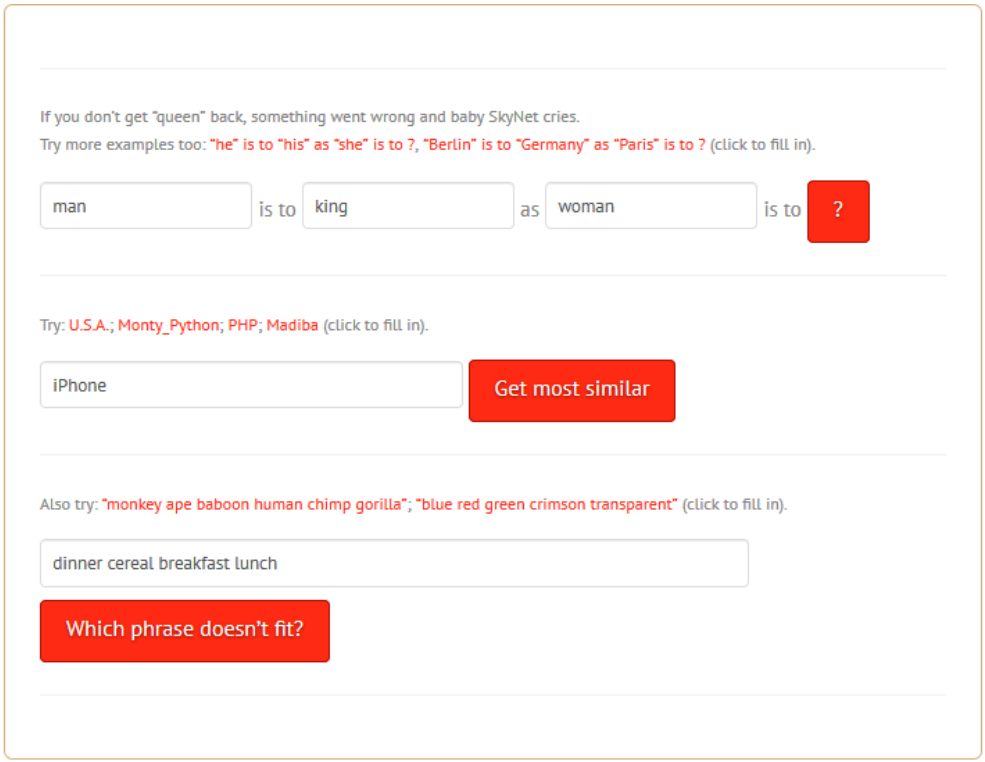


Fig. 18 – UI with GenSim and Bokeh (From [7])

of Python. We are only going to reproduce the code that works behind the scenes, i.e. implementation of functionalities from GenSim. [see 7,8] First thing we are going to need is corpus (text) that we are going to train our model on. App above uses 100 Billion words to train on. We do not have that computational power so we just need to some general purpose text that is reasonable size we can train on. For that we found text8 from `mattmahoney.net/dc/text8.zip`.

```
import gensim
import logging

logging.basicConfig(format='%(asctime)s : %(levelname)s : %(message)
    s', level=logging.INFO)

sentences = word2vec.Text8Corpus('text8')
model = word2vec.Word2Vec(sentences, size=400)

v_monarch=model.wv['monarch']
print(v_monarch)
print(v_monarch.shape)



model.wv.doesnt_match("breakfast cereal dinner lunch".split())

model.score(["The fox jumped over a lazy dog".split()])

model.most_similar(positive=['woman', 'king'], negative=['man'],
    topn=2)

model.most_similar_cosmul(positive=['woman', 'king'], negative=['man
    '], topn=2)

model.wv.similarity('woman', 'man')


model.most_similar_cosmul(positive=['China', 'Russia'], negative=['
    Bejing'], topn=2)

model.most_similar_cosmul(positive=['Japan', 'Turkey'], negative=['
    Tokio'], topn=2)

model.most_similar_cosmul(positive=['Poland', 'Germany'], negative=[
    'Warsaw'], topn=2)

model.most_similar_cosmul(positive=['Spain', 'Portugal'], negative=[
    'Madrid'], topn=2)


automation= ["good bad better", "paris france berlin", "tall
    basketball short"]
for auto in automation:
    a, b, x = auto.split()
    predicted = model.most_similar([x, b], [a])[0][0]
    print( "'%s' is to '%s' as '%s' is to '%s'" % (a, b, x,
        predicted))
```

LOGGING just allows us to have better overview hen executing the code. In line 7 we trained the model on neural network with 400 layers. Than we encoded the word "monarchïn line 9 and explored the shape. Later on we explored some of the functionalities of word2vec model. For example in line 15 we asked ourselves what does not match in sentence "breakfast cereal dinner lunchänd we got cereal as answer. We would like to know what is the probability of a sentence showing up given a text we trained on? No problem -> line 17. Want to implement Fig.12 ? We did on two ways line 19 and 21. Want to express similarity of man women in probabilistic terms? We did that on line 23. Than lets say we want to replicate Fig.16 or Fig.17 We could just start as in line 26. Or we could automate it with a for loop (line 35). I would like to point out that we WILL get different results as [7]. We used different corpus with different size of neural network. Also in order to predict accurately Fig.16 we ought to find some text that contains information about countries and train on that and not some general text as we did here (text8).

4 Different Neural Network Architectures for NLP

That was a basic introduction into NLP, but how would we procede to answer some interesting question. For example given an email how can we classify it as spam or not spam, or given a question how can we classify it assincere or not sincere (*https://www.kaggle.com/zikazika/kernels*). Essentially these are all (multi) classification problems that we can tackle with standard algorithms, SVM, RF and so on but in the recent years modifications of neural networks were specifically designed to tackle these problems. Why do we need to modify NN and cNN? Well first of all we want to capture the context of different words and sentences, not encode them in isolation. In other words we want so some sort of memory. If we want to conclude that E-Mail was spam it would be beneficial to "remember"that in the beginning of the email there was a sentence "dont miss this oportunity". Now one can argue that we already have memory in form of weights that are updated via backpropagation algorithm. But this is static, it does not address the problem directly. There is a solution (in a minute). Another problem is the vanishing gradient issue. It is looked at as a major drawback of neural networks. In short what happens when we derive the usual activation functions (take tanh for example) we get values between 0 and 1. Now IF we have a lot of layers (what is a necessity in text problems) we are multiplying a lot of values between 0 and 1, if we have (and we must have) a lot of epochs, gradient will slowly vanish and we will stop to converge towards the minimum, i.e. we wont be updating the weight anymore. On the other hand if we have values bigger the 1, then opposite happens. Gradient explodes eventually. We can tackle this problem when using LSTM and GRU (modifications of NN).

4.1 Recurrent Neural Networks

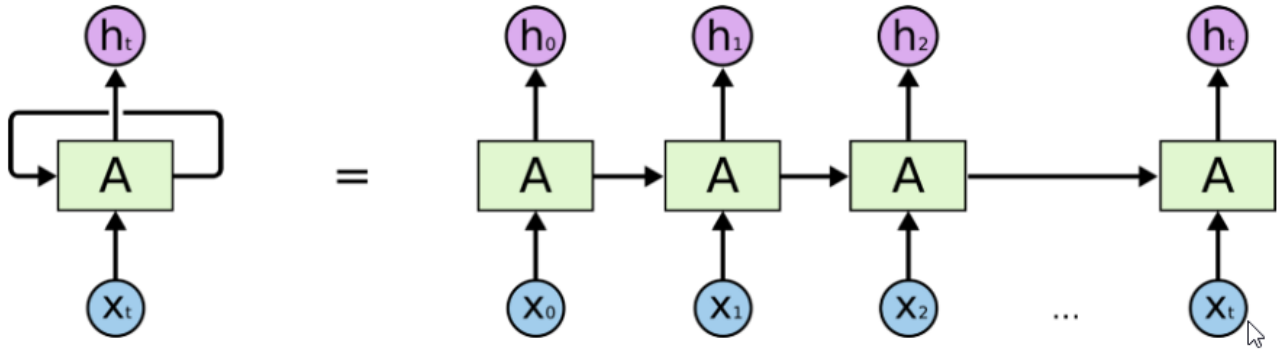


Fig. 19 – rNN (From [10])

It is best explained with the Fig.19. At each time step t we take input (usually word) in form of vector x_t and another input from previous layer a_{t-1}. We perform computation to produce h_t which will again go to the next node and so on. Pattern is clear here, we get the words but also everything that happend before this word, that is why we can speak from a dynamic memory of a sort. And that would be the basic Idea behind rNN, rest we can proceede as a normal NN (Backpropagation, Network architecture, etc...) Now how do we calculate h_t more formally.

Let a_t represent the ouput from the previous node

$$a_t = f(h_{t-1}, x_t)$$

$$g(x) = \tanh x$$

$$a_t = g(W_{hh} \cdot h_{t-1} + W_{xh} \cdot x_t)$$

$$a_t = \tanh W_{hh} \cdot h_{t-1} + W_{xh} \cdot x_t$$

$$h_t = W_{hy} \cdot a_t$$

Fig. 20 – Calculating output of rNN (From [10])

One can notice that we can not actually control how far does rNN remember. We can assume that it goes a couple of words back. Long short term memory can control that-**LSTM**:

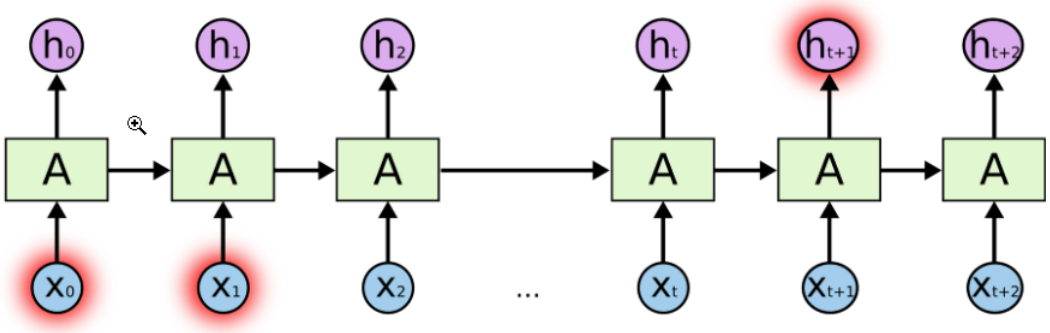


Fig. 21 – rNN setback (From [10])

In other words we might need to, in order to understand to concept fully, look at the beginning of text, i.e. at time step $t = 0$ when calculating $t + 1$ step.

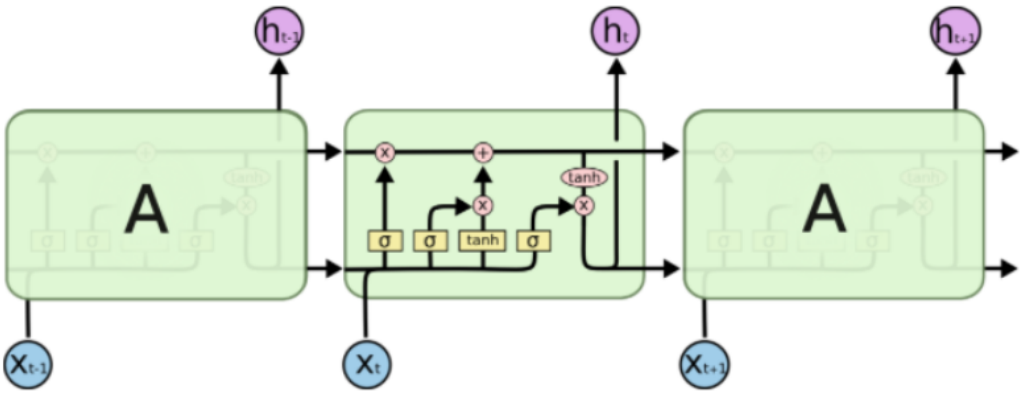
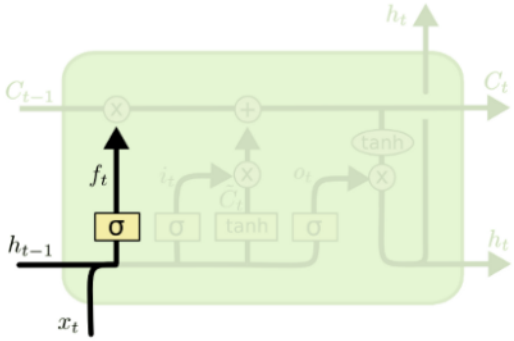


Fig. 22 – LSTM (From [10])

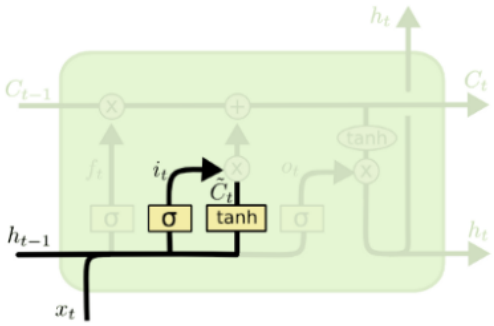
We can see from Fig.22 that the structure is the same as with rNN it is just that the working layer (time step t here) is different, more complex. We can break down each operation in the working layer into smaller pieces to see the big picture. Solo steps only make sence when we glue them back together in the end.



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

Fig. 23 – Forget gate operation (From [10])

In Forget gate operation we are effectively trying just that, to forget or not to forget memory (C_{t-1}) from the previous step. We get the current input x_t and the input from the previous layer h_{t-1} than after concatinating and, multiplying with Weights (will be updated eventually) and adding the bias and on top of that applying the sigmoid function we get values in between 0 and 1. The closer the values to 1 the more of the memory from the previous layer we keep and vice verse. Reason is the multiplication between the two terms.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Fig. 24 – Update gate operation (From [10])

Now in the update gate operation/step we have basically two shallow NN. First one where we apply the sigmoid activation function is called the forget "valve"(in analogy to drain pipes in real life) usually we also have C_{t-1} concataneted with $[h_{t-1}, x_t]$ because we are trying to model how much memory will be kept from the previous time step. This values will passed as an addition to the C_{t-1}. Other shallow NN with tanh as an activation function models current memory. It goes further into calculation and helps us decide how much current memory will be going into C_t.

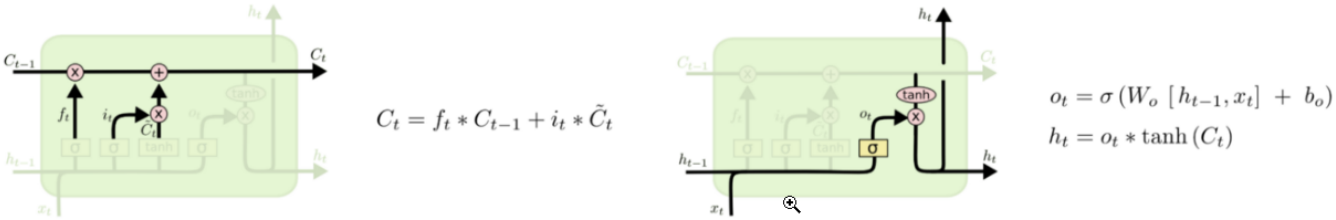


Fig. 25 – Output gate operation (From [10])

And finally two outputs that we pass along. First of all new memory C_t and new output h_t. And we proceede as a normal NN, updating the weights with backpropagation and improving our loss function.
Problem is training time. As we can see there a lot of operations that should be done in only one step. If we increase the number of training samples the run time will explode. That is GRU-Gated recurrent network was introduced. It can still remember things far from the past, but in a lesser time.

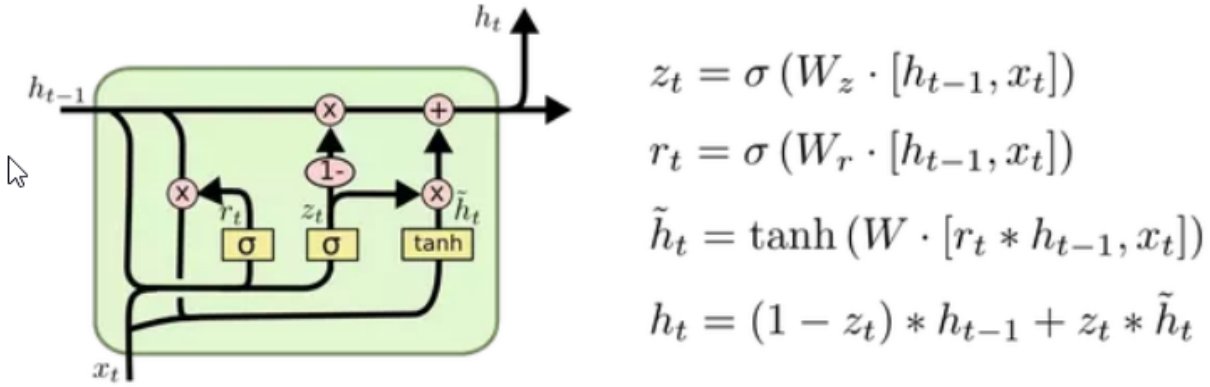


Fig. 26 – GRU-Gated recurrent network (From [10])

Now in GRU we have forgone the concept of memory in form of C_t and have introduced two new gates. Update gate z_t and reset gate r_t. Update gate (again a shallow NN) helps us to model how much information will be passed along from the previous steps. On the contrary reset gate tells us how much information to forget from the previous step. They are in essence the same just the training values (and bias) are different, since we learn different things. Furthermore $\tilde{h}_t$ represents current memory content. Input in this shallow NN is no surprise. We feed it with how much info we learned to forget from the previous steps along with the current input in the time step x_t. To understand it a bit better let us say that we have want to give some text (pre-processed) to the whole NN structure. NN learns that everything in the beginning of the text is useless and it ought to forget it. In that case r values are going to be 0 or close to zero, and vice-versa if it finds it important. And finally we have final memory unit h_t. It is no surprise that we have affine combination of the previous elements, because if we look at the equation and z_t is close to one (sigmoid squashes it between 0 and 1) that tells us that the network learned the fact that information from the past is not so valuable, in that case we are going to value current memory status, and vice-versa. Finally in this kernel where I tried to classify questions (*https://www.kaggle.com/zikazika/quora-nlp-classification/edit*) I also used concept of bidirectional NN. Concept is relatively simple. Let us say that when predicting words we also

want to take into account not only words from the past in the current step t but also future words. (we already tackled past words-memory)

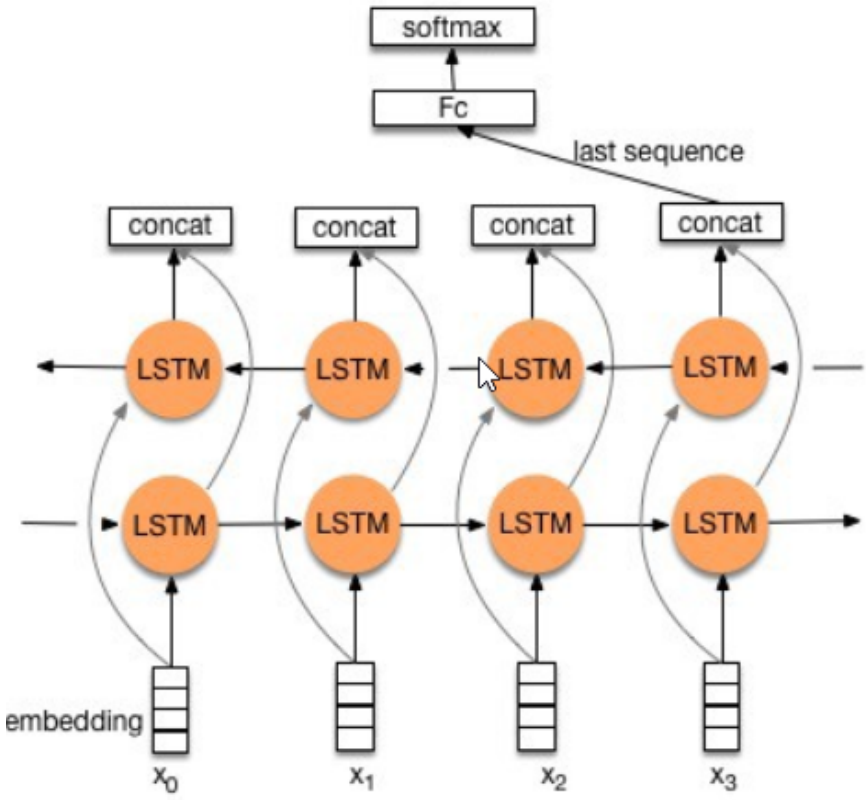


Fig. 27 – Bi-directional architecture (From [10])

Approach is relatively straight forward. We take our word embeddings and we feed the network (rNN, LSTM,GRU) one time from the from the beginning to the end, and one time opposite. We concatenate the results and pass it along to final layer to make a prediction. And finally we said that these new NN architectures tackle vanishing gradient problem. Let us assume that network learns to pass along memory from the previous steps, for simplicity let us say step t=1. Now we now that than the activation function is in essence constant (equal to 1). We are also familiar with the fact that backpropagation is in essence a chain rule that (let us say in step $t = 100$) depends on the previous steps ($t = 1$) so if we multiply it with one, it wont diminish or decrease. And that is how remedy gradient problem (not solve entirely).

5 Reference

[1] Yoav Goldberg, *Neural Network Methods for Natural Language Processing*, Synthesis Lectures on Human Language Technologies, Vol. 10, No. 1, pp. 1-309 (2017)

[2] Nils J. Nilsson, *The Quest for Artificial Intelligence*, 1st Edition, Cambridge University Press (2009)

[3] Daniel Jurafsky,James H. Martin. *Speech and Language Processing. Free-online (2014)*

[4] *https://machinelearningmastery.com/gentle-introduction-bag-words-model Access on: 11.12.2018*

[5] *https://blog.acolyer.org/2016/04/21/the-amazing-power-of-word-vectors Access on: 11.12.2018*

[6] *https://stanfordnlp.github.io/CoreNLP/*

[7] *https://radimrehurek.com/gensim/models/word2vec.html Access on: 11.12.2018*

[8] *https://rare-technologies.com/word2vec-tutorial/ Access on: 11.12.2018*

[9] *https://www.quora.com/Deep-Learning-What-is-meant-by-a-distributed-representation l/ Access on: 11.12.2018*

[10] *https://towardsdatascience.com/introduction-to-sequence-models-rnn-bidirectional-rnn-lstm-gru-73927ec9df15*