Information Retrieval 1: Homework 2 Report

Gabriele Bani, 11640758
University of Amsterdam
gabriele.bani@student.uva.nl

Bence Keresztury, 11797967
University of Amsterdam
kbence16@gmail.com

Andrii Skliar, 11636785
University of Amsterdam
andrii.skliar@student.uva.nl

1 TASK 1

In the first part, we will discuss implemented retrieval models, such as a Vector Space model (TF-IDF) and Probabilistic models (BM25, Jelinek-Mercer Smoothing, Dirichlet Prior Smoothing, Absolute Discounting Smoothing, Positional Language Models). While TF-IDF and BM25 are very simple models, only defined for the query terms, that are present in the document, they are quite powerful and have been empirically proven to give very high performance despite their simplicity. To adjust Maximum Likelihood estimate to avoid zero probability and therefore give us a framework for working with the documents, which don't contain query term, smoothing can be introduced. Smoothing allows us to incorporate our prior belief of what a distribution characterizing the topic of a document should look like. This gives the probability of an unseen word probability, which is proportional to the probability of a word given a reference/background language model. There are different approaches for estimating background probability and they will be discussed later in the section.

1.1 Implementation Details

1.1.1 TF-IDF. TF-IDF is a vector space model, which uses term frequencies, i.e. total number of occurrences of a term in a document, and document frequencies, i.e. total number of documents containing the term, to calculate document-query score for every document-query pair.

To calculate this score given document (d) and term (t), we first calculate term frequencies: $tf(t,d) = count(t,d)$, where $count(t,d)$ - number of occurrences of term t in document d. Term frequency shows how relevant is the document for the given word. However, due to the fact that relevance does not increase proportionally with term frequency, we apply logarithm to make function more plausible in a following manner: $tf = log(1 + tf(t,d))$.

After that, inverse document frequency (idf) is calculated using document frequency (df): $idf(t) = \log \frac{n}{df(t)} = \log n - \log df(t)$, where n is the number of documents in a corpus. Inverse document frequency shows how discriminative the word is. The idea is that word, which is contained in many documents can't give us much information about the importance of the document for the current query and therefore it's weight should be adjusted to that.

One of the last steps is to calculate document-term score by multiplying tf and idf in a following manner: $score(t,d) = tf(t,d) \cdot idf(t)$. Using this approach, we have that to have higher score, words should appear with higher frequency in the document and with lower frequency in the rest of the documents. It's important to notice, that when word occurs in all the documents, the score is the lowest.

Final step is to calculate document-query score:

$$score(q,d) = \sum_{t \in q} tf(t,d) \cdot idf(t)$$

Though being very simple, TF-IDF is a very good scoring method, which is quite difficult to beat.

1.1.2 BM25. BM25 is similar to TF-IDF due to the fact that it also uses term frequency and inverse document-frequency for scoring document-query pairs. However, the formula for calculating the score is completely different and will be explained in a more detail further. BM25 was loosely inspired by probabilistic models and therefore can't be considered proper probabilistic model. Because BM25 is a name for the family of scoring functions, there are several options for which interpretation to use, however, we have used the one provided during the lecture, which only goes over unique words in a query.

$$score(q,d) = \sum_{unique\ t \in q} \frac{(k_1+1)tf(d,t)}{k_1((1-b)+b(\frac{l_d}{l_{avg}}))+tf(d,t)} \cdot idf(t)$$

As we can see, BM25 is based both on tf and idf, but has tuning parameters for scaling those properly. The variable k_1 is a tuning parameter for term frequency, corresponding to model without term frequency, when having low value and to using raw term frequency, when the value gets bigger.

BM25 also uses length normalization in the denominator ($\frac{l_d}{l_{avg}}$), which allows to normalize term frequency by document length. This might not always be useful, however, it is especially important in following case: imagine you have two documents with the same term frequency of word 'apple', however, one document is quite short and tells a lot about apples, while the second one is very long and talks about fruits in general, however, it contains the same amount of word 'apple' due to its length. In that case, it would be reasonable to choose the first document, because it is more relevant, but tf-idf would give the same preference to both of the documents. To optimize length normalization, variable $b \in (0,1)$ has been introduced as another tuning parameter, which is used for scaling by document length: $b = 1$ corresponds to full length normalization, while $b = 0$ corresponds to no length normalization.

Normally, parameters k_1 and b should be optimized, however, we were using the suggested parameter values of $k_1 = 1.2, b = 0.75$.

1.1.3 Jelinek-Mercer Smoothing. As mentioned already, Jelinek-Mercer is similar to any other smoothing as it was introduced to work with query terms, which are not present in the document. To estimate the probability of an unseen word, it considers, that unseen words contribute equally, so:

$$p(t|C) = \frac{tf(t;C)}{|C|},$$

where C - entire document collection,
$|C|$ - size of the entire document collection,
$tf(t;C)$ - number of occurrences of term t in the entire collection

Using this background language model and term frequency with length normalization ($\frac{tf(t,d)}{|d|}$), we get following formula for Jelinek-Mercer smoothing for term-document pair:

$$\hat{P}_\lambda(t|d) = \lambda \frac{tf(t,d)}{|d|} + (1-\lambda)\frac{tf(t;C)}{|C|},$$

where λ is an interpolation coefficient.

This allows us to calculate log-probability of a query given a document using geneal smoothing scheme as following:

$$logp(q|d) = \sum_{t \in V} tf(t;q) \log p(t|d),$$

where V is the vocabulary,
$tf(t;q)$ is number of occurrences of term t in query q

We can go from calculating probability of a query given a document to calculating probability of a document, given query, assuming that we have uniform prior, as following:

$$P(d|a) = \frac{P(q|d)P(d)}{P(q)}$$
$$P(d|q) \sim P(q|d)P(d) \quad \text{Dropping document-independent values}$$
$$P(d|a) \sim P(q|d) \quad \text{Assuming uniform prior}$$

So, $\log p(d|q) = \log p(q|d) = \sum_{t \in V} tf(t;q) \log p(t|d)$.

1.1.4 Absolute Discounting Smoothing. Absolute Discounting Smoothing also considers, that unseen words contribute equally, but, unlike, Jelinek-Mercer Smoothing, it lowers the probability of seens words by subtracting a constant from their counts and adding this mass to unseen words.

This gives the following equation:

$$p_\delta(t|\hat{\theta}_d) = \frac{max(tf(t;d) - \delta, 0)}{|d|} + \frac{\delta |d|_u}{|d|} p(t|C),$$

where $|d|$ is the length of document d,
$|d|_u$ is the number of the unique words in d

Applying general smoothing scheme, we can calculate log-probability of the document, given a query.

1.1.5 Dirichlet Prior Smoothing. If we assume, that document d has Multinomial Language Model, so $p(d|\theta_d) = \prod_{t \in d} p(t|\theta_d)$ and use Maximum A Posteriori (MAP) estimation with Dirichlet Distribution as conjugate prior, we get Multinomial language model with Dirichlet prior smoothing:

$$p(t|\hat{\theta}_d) = \frac{tf(t;d) + \mu p(t|C)}{|d| + \mu}$$

Applying general smoothing scheme, we can again calculate log-probability of the document, given a query.

1.1.6 Positional Language Models. Although many variants of language models have been proposed for information retrieval, there are two related retrieval heuristics remaining "external" to the language modeling approach [7]:

- proximity heuristic which rewards a document where the matched query terms occur close to each other;
- passage retrieval which scores a document mainly based on the best matching passage.

To deal with these two issues, Positional Language Models (PLMs) were proposed. Positional Language Models offer an indirect way of capturing the proximity in the language modelling framework. The other option for doing this would be using n-grams, but this can quickly make the size of vocabulary blow up and makes such a model very impractical.

The idea of PLMs is to propagate word occurrence to all the positions in the current document. The most basic idea is that if word occurred in position i, it would also give a discounted count to other positions in the document. Let's formalize this approach, as originaly suggested in [7]:

- $D = (t_1, t_2, \dots t_N)$ - document D
- $c(t,i)$: the count of term t at position i in document D. If t occurs at position i, it is 1, otherwise 0
- $k(i,j)$: the propagated count to position i from a term t at position j (i.e., t_j)
- $c'(t,i)$: the total propagated count of term t at position i from the occurrences of t in all the positions. That is, $c'(t,i) = \sum_{j=1}^{N} c(t,j) \cdot k(i,j)$.

Using these notations, we can form term frequency vector $(c'(t_1,i), c'(t_2,i), \dots c'(t_N,i)$ for each position i, which can logically be summarized to language models in a following way:
$p(t|D,i) = \frac{c'(t,i)}{\sum_{t' \in V} c'(t',i)}$, where V is the vocabulary of the query. Using KL-divergence retrieval model, we get following equation for scoring the document:

$$S(Q,D,i) = -\sum_{t \in V} p(t|Q) \log \frac{p(t|Q)}{p(t|D,i)},$$

where $p(t|Q)$ is an estimated query language model.

To estimate $p(t|Q)$, we have been using MLE approach:
$p(t|Q) = \frac{count(t,Q)}{|Q|}$, where $count(t,Q)$ is the number of occurrences of term t in query Q and $|Q|$ is the length of the query.

However, just as in other language models, to deal with documents, which don't contain specific query term, we need to introduce smoothing. To do so, we have been using Dirichlet Prior smoothing, which looks as following for the PLMs:

$$p_\mu(t|D,i) = \frac{c'(t,i) + \mu p(t|C)}{Z_i + \mu},$$

where μ is smoothing parameter,
$Z_i = \sum_{t \in V} c'(t,i)$ - length of the virtual document, so, document length normalization heuristic.

The main issue at that point is choosing kernels to use for propagating the proximity. We have used following kernels:

(1) Gaussian kernel: $k(i,j) = \exp[\frac{-(i-j)^2}{2\sigma^2}]$

(2) Triangle kernel: $k(i,j) = \begin{cases} 1 - \frac{|i-j|}{\sigma} & if |i-j| \le \sigma \\ 0 & otherwise \end{cases}$

(3) Cosine kernel: $k(i,j) = \begin{cases} \frac{1}{2}[1 + \cos(\frac{|i-j| \cdot \pi}{\sigma})] & if |i-j| \le \sigma \\ 0 & otherwise \end{cases}$

(4) Circle kernel: $k(i,j) = \begin{cases} \sqrt{1 - (\frac{|i-j|}{\sigma})^2} & if |i-j| \le \sigma \\ 0 & otherwise \end{cases}$

(5) Passage kernel: $k(i,j) = \begin{cases} 1 & if |i-j| \le \sigma \\ 0 & otherwise \end{cases}$

To efficiently calculate the score between query and a document, we have made several engineering decisions in order to speed up the calculations.

First of all, we have preprocessed all the documents in order to throw out stopwords and while calculating the score, we only deal with preprocessed documents. Also, it would be very computationally expensive to search for a term in the document for every term, therefore, to solve this problem, we have created inverted index of positions, which show at which position in the document, query term appears.

The last issue is, probably, the most important - calculating document length normalization heuristic is very expensive if you do it naively, because that would mean, that for every position of the document we have to iterate over the whole document, which makes it impossible to calculate PLM in a reasonable amount of time (it was taking us a minute for query term - document pair). To solve this issue, as proven in [7], we can use the fact that $\sum_{t \in V} c'(t,i) = \sum_{j=1}^{N} k(i,j) = \sum_{j=1}^{N} k(i,j) = \sum_{j=1}^{N} k(i,j)$. This allows us to calculate document length normalization as an integral.

To simplify calculations and, as will be explained further, speed up the computation, we have decided to use the fact that all the kernels are symmetric and are based on the distance from current position to the beginning and end of the document in order to calculate integrals. We do following steps in order to calculate each integral: we split it into two parts - left and right, which correspond to the part of the document on the left of current position (i for position i) and to the right of the current position ($N-i$ for position i and document of length N). After that, we sum up the left and right partial integrals and get the document length normalization. This allows us to base our calculations only on the distances to the left and to the right. We got the following integral for each of the kernels for the distance d (notice, that distance is always positive):

(1) Gaussian kernel: $I_d = \sqrt{2\pi\sigma^2} \cdot (\Phi(\frac{d}{\sigma}) - \Phi(0))$

(2) Triangle kernel: $I_d = \begin{cases} d - \frac{d^2}{2\sigma} & if d \leq \sigma \\ \frac{\sigma}{2} & otherwise \end{cases}$

(3) Cosine kernel: $I_d = \begin{cases} \frac{\sigma}{2} + \frac{\sigma}{2\pi} \sin\left(\frac{\pi d}{\sigma}\right) & if d \leq \sigma \\ \frac{\sigma}{2} & otherwise \end{cases}$

(4) Circle kernel: $I_d = \begin{cases} \frac{\sigma}{2}(\arcsin\frac{d}{\sigma} + \frac{1}{2}\sin(2\arcsin(\frac{d}{\sigma}))) & if d \leq \sigma \\ \frac{\sigma\pi}{4} & otherwise \end{cases}$

(5) Passage kernel: $I_d = \begin{cases} d & if d \leq \sigma \\ \sigma & otherwise \end{cases}$

Using this notation, for document d of length N, document length normalization at position i would be calculated as followig:
$Z_i = I_{i-0} + I_{N-i} = I_i + I_{N-i}$.

Because kernels are also only based on the distance ($|i-j|$ is always used except for Gaussian kernel, where distance between i and j is brought to the power of two, so sign doesn't matter as well), calculating integrals and kernels using distance allowed us to speed up the computations by precomputing the kernels and integrals for all possible values of distance in all the documents. The length of the longest document doesn't exceed 2500 symbols, therefore, we have created table of values of kernels and their integrals for distances from 0 to 2500, which we retrieve during the runtime. In practice, it has changed made our calculations 8x faster for Gaussian kernel and 2x faster for all other kernels.

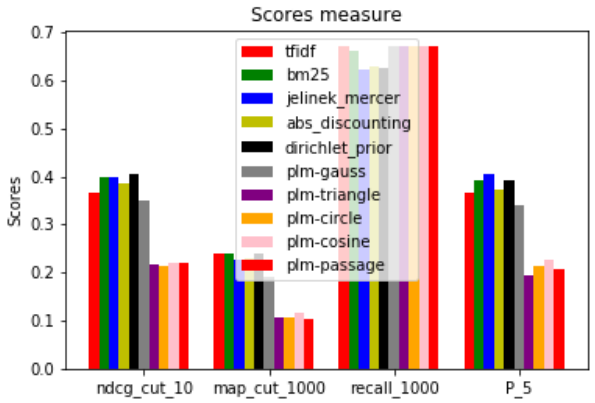

Figure 1: comparison of different scoring functions on different evaluation metrics

1.2 Results

We ran several experiments in order to test whether different scoring methods perform differently on the given data. For every method, we set the parameters as show in the plots at the end of the section. We choose the parameter value according to the nDCG@10 on the validation set of the method with that parameter. We then compare the different methods for every evaluation measure, which are nDCG@10, precision@5, recall@1000 and map@1000. In figure 1 add figure we can see the comparison of the different measures using different scoring functions. One evident result is that positional language models behave very differently than other models. From the plots, the actual difference between the performance of the methods cannot be correctly evaluated; tfidf, bm25, dirichlet, JM and absolute discounting seem to perform similarly for all four evaluation metrics. Thus, we run t-tests to quantitatively observe the difference between the results produced by different methods. For every pair of methods we run a t-test on the rankings produced by the different measures. The test is two tailed, so the resulting p value can only tell whether one method is different from the other. We used Bonferroni correction to tackle the multiple comparison problem. To check which method is significantly better than the other, we checked the mean values. The p values for different methods are plotted in 2.

2 TASK 2

In the second task we experimented with different distributional semantic methods.

2.1 Semantic Matching

All the methods implemented in the first exercise rely on the lexical matching of words in documents with words in queries. That means in order to find relevant documents, the words in it have

	abs_discounting	bm25	dirichlet_prior	jelinek_mercer	plm-circle	plm-cosine	plm-gauss	plm-passage	plm-triangle	tfidf
abs_discounting	-	0.281471	0.151008	0.013114	3.01318e-08	2.35804e-09	0.000249777	1.07983e-08	1.59539e-09	0.0718474
bm25	0.281471	-	1	9.68998e-06	3.92307e-10	2.30534e-11	7.81745e-07	3.13296e-11	1.59102e-11	0.386952
dirichlet_prior	0.151008	1	-	7.80753e-05	1.12744e-09	2.93274e-11	1.94395e-06	1.93979e-10	3.30155e-11	0.459503
jelinek_mercer	0.013114	9.68998e-06	7.80753e-05	-	5.13965e-06	2.41127e-07	0.0155715	2.94366e-07	1.93023e-07	2.20626e-05
plm-circle	3.01318e-08	3.92307e-10	1.12744e-09	5.13965e-06	-	0.11347	7.80753e-05	0.515214	0.053165	3.73572e-11
plm-cosine	2.35804e-09	2.30534e-11	2.93274e-11	2.41127e-07	0.11347	-	2.98811e-06	0.360295	1	3.22379e-12
plm-gauss	0.000249777	7.81745e-07	1.94395e-06	0.0155715	7.80753e-05	2.98811e-06	-	5.98617e-06	2.23129e-06	1.65077e-07
plm-passage	1.07983e-08	3.13296e-11	1.93979e-10	2.94366e-07	0.515214	0.360295	5.98617e-06	-	0.294617	5.06867e-12
plm-triangle	1.59539e-09	1.59102e-11	3.30155e-11	1.93023e-07	0.053165	1	2.23129e-06	0.294617	-	1.69235e-12
tfidf	0.0718474	0.386952	0.459503	2.20626e-05	3.73572e-11	3.22379e-12	1.65077e-07	5.06867e-12	1.69235e-12	-

Figure 2: p values for different methods on ndcg@10 rankings

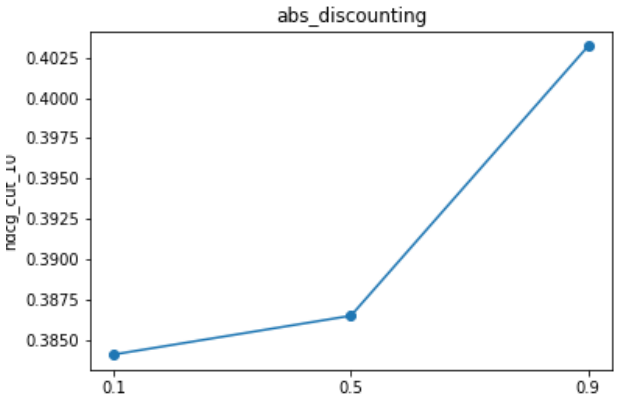

Figure 3: Absolute discounting

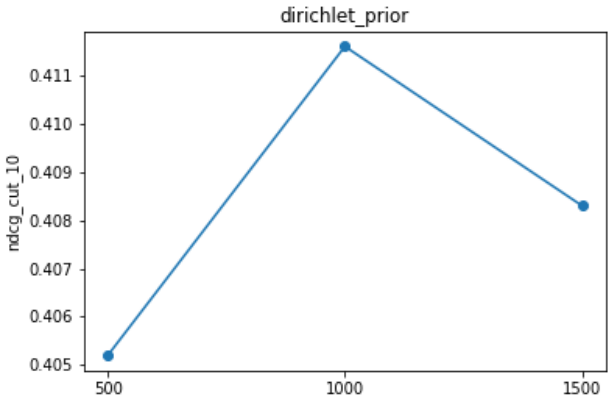

Figure 4: Dirichlet prior

to be exactly in the same form as in the queries. There are multiple problems with that approach. One of the main issues is that synonyms are not taken into account. For example if I search for "rock concerts near me", the search engine will not recognize that a document talking about a "music festival in Amsterdam" could be a potential match, because there is no common words appearing therefore the algorithm will rank this document very low. To tackle

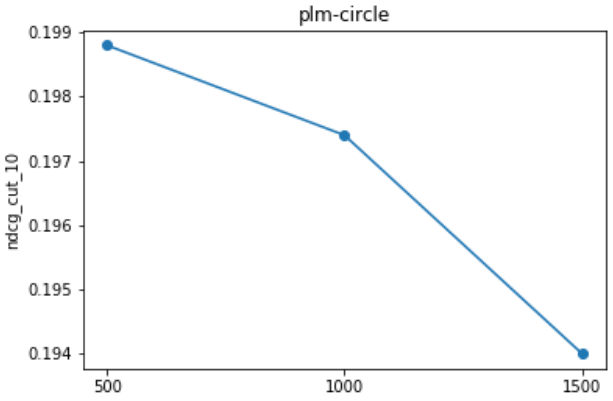

Figure 5: Plm with circle kernel

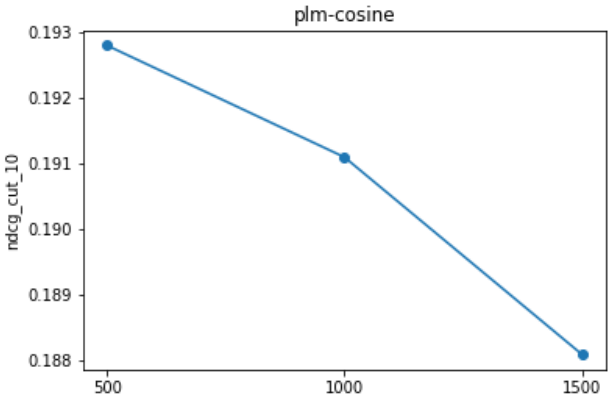

Figure 6: Plm with cosine kernel

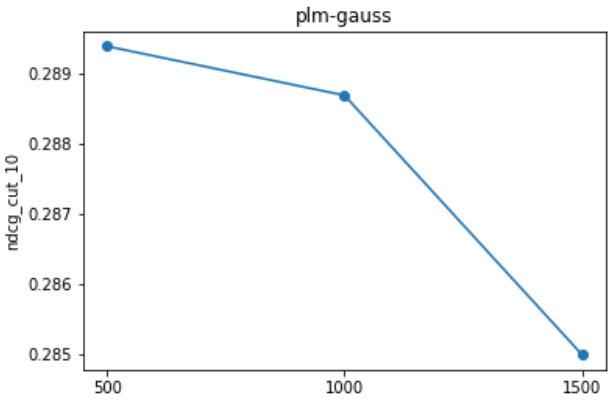

Figure 7: Plm with Gauss kernel

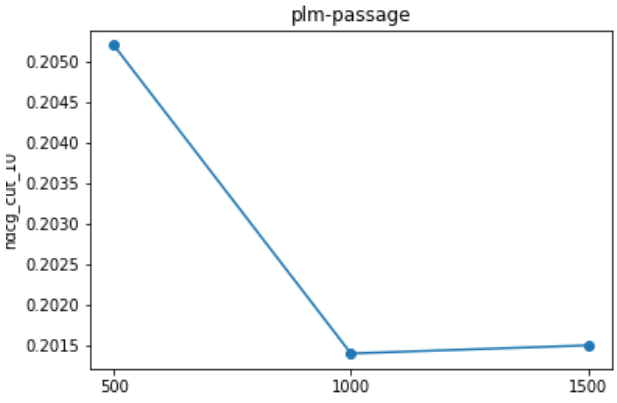

Figure 8: Plm with passage kernel

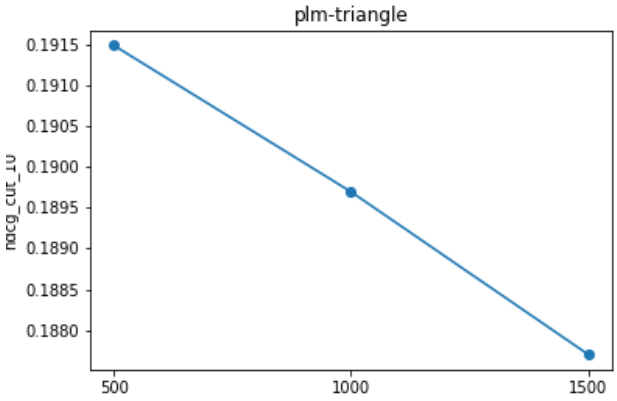

Figure 9: Plm with triangle kernel

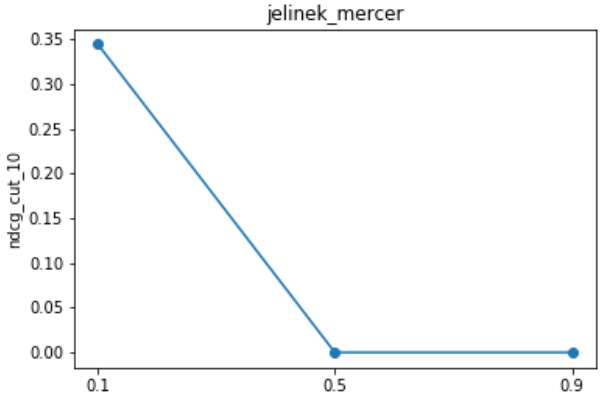

Figure 10: Plm with Jelinek-Mercel kernel

this problem, retrieval models based on semantic matching try to represent the meaning of the words and find documents that are close to the query in that semantic space. This representation is either based on multi-dimensional geometry (vector space models) or on probability theory (probabilistic models). In this exercise we experimented with the method called Latent Semantic Indexing (LSI), which is based on the vector space model and with Latent Dirichlet Allocation (LDA), which is a probabilistic method.

2.2 Latent Semantic Indexing

The naive way to represent documents in a high-dimensional space is to create a vector space with size of the whole vocabulary. In this space every axis corresponds to a word and a document is represented as a sparse vector where there is a zero if the word does not appear in the document and there is a c if the word appears c times. If we concatenate the vectors for different documents we get a matrix where every row corresponds to a term and every column corresponds to a vector representation of a document. The problem with this naive approach is that the dimension of the vectors is too big. Moreover, this way are not able to solve the problem of synonyms, since synonyms still do not contribute to the similarity, because the two words are represented in different dimensions. A better way is to reduce the dimension of the term-by-document matrix to obtain a dense representation of the documents. We want to find to find a k dimensional manifold (where $k << |V|$) that corresponds to the intrinsic dimensionality of the data. That means we hope to find an optimal k which is big enough to capture all the real structure and meaning in the data, but small enough to disregard unimportant details like specific word choice. Latent Semantic Indexing tries to do exactly this task. It uses a mathematical method call Singular Value Decomposition (SVD) to find the important dimensions (topics) in the data. SVD factorizes the original matrix M into a product of three matrices:

$$M = U\Sigma V,$$

where Σ is a diagonal matrix with the singular values of M. Bigger values correspond to more important dimensions, so we can set lower values to zero. This way we reduced the dimension of the data. The dimensions can be thought as semantic topics (eg. politics, sport, ...), therefore similar documents will most likely end up close to each other in this new dense vector space.

2.3 Latent Dirichlet Allocation

Latent Dirichlet Allocation is a generative probabilistic model that assigns topic distributions to documents. That means that every document will be represented as vector of probabilities where each topics has a difference chance of generating the next word. The different topics are in itself represented as a distribution over words and can generate different words with different probabilities. The LDA treats this distributions as variables which also have adjustable parameters. In our homework however, we did not experimented with these parameters due to the limited computing capacities. The full generative nature of the LDA model can be seen in Figure 11,

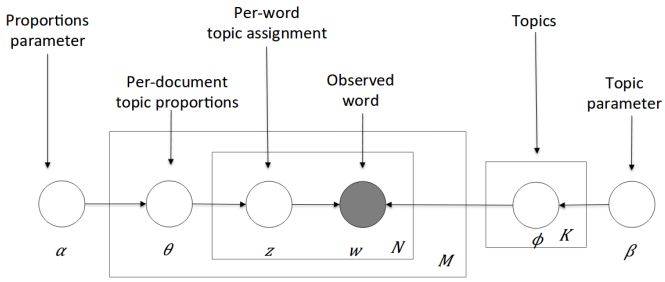

Figure 11: Graphical model for LDA

where the only observed variables of the model are the omitted words.

2.4 Implementation details

We used a similar training procedure for both the LSI and LDA algorithms. We first represented every document as a Bag of Words, where the order of the words did not matter, only the number of appearances in the document. The whole corpus consisted of a list with the BoWs as elements. Finding the right number of topics (dimension of representation) is not an easy task in either of the algorithms, therefore we experimented with different k-values (10, 25, 50). The choice of dimensions was rather arbitrary. Using the gensim library we trained three LDA and three LSI model with the three different parameters each. We used these trained models to create the LDA and LSI representation of every document in our corpus. In a separate for-loop we calculated the LDA and LSI representation for every query and calculated its similarity to the top 1000 documents (based on tf-idf scores) using cosine similarity metric. We used pandas library to save out trained model and the representation for every document in the data set (for every parameter). This way our results are more easily reproducible and faster to load.

2.5 Results

We plotted the results for Lda and Lsi using different evaluation metrics. For the Lsi the best parameter was k=10, while in case of Lda the best one was k=50. As we can see on Figure 12 the results are similar based on all the metrics. To check whether we can say anything about the difference of the two metrics we ran two-tailed t-test (with Bonferroni correction for tackling the multiple comparisons problem), but we did not find any significant difference ($t = 0.7983, p = 0.4361$).

3 TASK 3

Continuing the idea of semantic matching, described in task 2, we would like to use Neural Language Models in order to satisfy three IR regularities [4]:

- semantic matching: words occurring in each others vicinity are learned to have similar representations,
- the clustering hypothesis: documents that contain similar language will have nearby representations in latent space, and

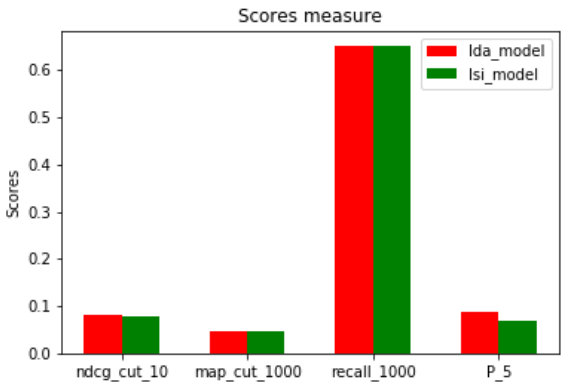

Figure 12: Different evaluation metrics for LDA and LSI

- term specificity: words associated with many documents will be neglected, due to low predictive power.

To satisfy mentioned regularities, a lot of approaches were suggested, with word2vec [8] being the most used one. However, though very useful for NLP tasks, such as semantical matching of separate words and sentences, pure word2vec is not very suitable for IR tasks due to the fact that it doesn't offer good framework for representing documents and queries and matching them together. However, to fight this problem, similar approaches were suggested, such as Doc2Vec [6], Doc2VecC [1]. One of the latest approaches suggested is Neural Vector Space Model (NVSM) [4], which uses neural networks in order to train both document and words represantations from scratch. We decided to use modified version of it, which will be described later in the section.

3.1 Implementation Details

Training procedure described is practically the same as described in [4]. Given set of documents $D = d_1, \dots d_{|D|}$, set of queries $Q = q_1, \dots q_{|Q|}$, we first form the vocabulary containing all the terms from all the documents and all the queries $V = t_1, \dots t_|V|$. We create two matrices: document matrix R_D of dimensionality $k_d x|D|$, where k_d is the dimensionality of documents embeddings, vocabulary matrix R_V of dimensionality $k_t x|V|$, where k_t is the dimensionality of term embeddings and projection matrix W of dimensionality $k_d x k_t$. Document matrix and projection matrix are initialized randomly using Xavier Initialization, as suggested in [2]. Vocabulary matrix is initialized differently, which is also the main modification compared to the original paper - instead of using random initialization, we initialize Vocabulary matrix with word2vec embeddings, which are learnt beforehand using Skip-gram approach. During training procedure, we keep Vocabulary matrix fixed.

While training, we construct mini-batch B as suggested:

(1) Stochastically sample document $d \in D$ according to $P(D)$, which we assume to be uniform. Document is represented by its embedding from Document matrix.

(2) Sample phrase of n contiguous terms $t_i, \dots t_{i+n}$ from document d. Each term is represented by its embedding from Vocabulary matrix.
(3) Add the phrase-document pair $(w_i, \dots w_{i+n}; d)$ to the batch.
(4) Repeat until the batch is full.

Given a batch B, we then transform it in order to calculate loss. For every sample in a minibatch (denote it as $B_i = (B_i^t, B_i^d)$, where B_i^t is phrase and B_i^d is a document), we do following operations:

- Compose a representation of a phrase B_i^t using averaging (remember, every term is represented as embedding from vocabulary matrix, so, here $t_i = R_V^{t_i}$): $g_i = g(t_1, t_n) = \frac{1}{n}\sum_{i=1}^{n} t_i$.
- Do L2-normalization of the representation we get: $norm_i = norm(g_i) = \frac{g_i}{\|g_i\|}$
- Porject normalized phrase representation into document space: $f_i = f(norm_i) = W \cdot norm_i$
- Do featurewise standardization for projection of phrase/document pair B_i using hard-tanh [3]: $T_i = T(f_i) = hard{-}tanh(\frac{f_i - E(f_i)}{\sqrt{V(f_i)}} + \beta)$, where $E(f_i)$ is per-feature mean, $V(f_i)$ is per-feature variance, β is bias vector of dimensionality k_d. β is inialized randomly and learnt during training.
- Apply an adjusted-for-bias variant of negative sampling (remember, every document is represented as embedding from document matrix, so, here $d_i = R_D^{d_i}$): $P(S|d, B_i^t) = \sigma(d_i \cdot T_i)$, where $\sigma(t) = \frac{1}{1+exp(-t)}$ and S is the indicator, showing similarity between document d and projection of a phrase B_i^t.
- Approximate probability of a document B_i^d given phrase B_i^t using uniform sampling of z contrastive examples: $\log \hat{P}(B_i^d|B_i^t) = \frac{z+1}{2z}(z \log P(S|d, B_i^t) + \sum_{k=1, d_k \sim U(D)}^{z} \log(1.0 - P(S|d_k, B_i^t)))$, where $U(D)$ stands for the uniform distribution over documents D and is used for obtaining egative examples.
- Calculate loss function: $L(R_V, R_D, W, \beta|B) = \frac{1}{m}\sum_{i=1}^{m} \log \hat{P}(B_i^d|B_i^t) + \frac{\lambda}{2m}(\sum_{i,j} R_{V_{i,j}}^2 + \sum_{i,j} R_{D_{i,j}}^2 + \sum_{i,j} W_{i,j}^2)$, where m is a batch size, λ is a weight regularization hyper-parameter.

We have used Adam optimization [5] with standard parameters ($lr = 0.001, \beta = (0.9, 0.999), \epsilon = 1e-08$) to optimize our parameters. We have used most of the hyperparameters as suggested in the original paper: $k_t = 300$, $k_d = 256$, $z = 10$, $\lambda = 0.01$, $n = 10$ (n-gram size). However, though paper suggests to use batch size of 51200, it was infeasible for us due to the lack of computational power. Therefore, we have used batch size of 10000 samples, because this was the maximum amount that would fit into our GPU memory. Following origin paper suggestion, we would need to have $\frac{1}{m}\sum_{d \in D}(|d| - n + 1)$, which for us would turn into following equation (total length of train and test documents is 13692482 and total number of train and test documents is 45310): $\frac{1}{10000}(13692482 - 453100 + 45310) = \frac{13284692}{10000} = 1328.4692 \approx 1328$. However, it would take us at least 3 hours to perform single epoch, which was also too much, therefore, we decided to have 500 batches per epoch. We have run training for 10 epochs before evaluating the results.

It would be a good idea in future to run as proposed in the paper and evaluate the results.

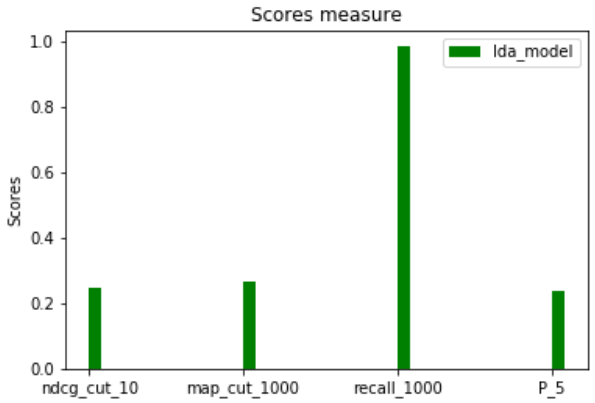

Figure 13: Results for NVSM on training set

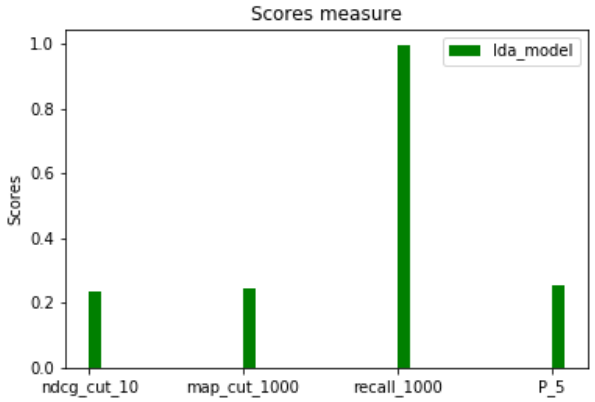

Figure 14: Results for NVSM on validation set

Our model is not yet suitable for information retrieval due to the fact that we haven't defined how to calculate score between given document and a query. Given document d and query q, we calculate score as cosine similarity between query averaged and projected into document space and document as following:

$$score(q, d) = \frac{(f \circ g)(q) \cdot R_D^d}{\|(f \circ g)(q)\| \|R_D^d\|},$$

where functions $f(x) = Wx$ and $g(t_1, \dots t_n) = \frac{1}{n}\sum_{i=1}^{n} R_V^{t_i}$ are the same as in training procedure

The main issue with the given model is, of course, the fact, that it has to train for all the documents beforehand and it is not possible to get embeddings for the documents at runtime, but for our purposes it was suitable due to the fact that the collection of documents is fixed.

3.2 Results

As we can see, NVSM didn't perform too good on both train 13 and test set 14. We consider, that this might be caused by the fact, that model is quite complex and needs more time to properly train. Also, because our computational power was quite limited, we were unable to train model with the parameters, which were suggested in the paper. We consider these to be the reason for such poor results. Therefore, it would be interesting to have more experiments with the model, especially with different hyperparameters, because these might be very important during the learning process. We also assume, that the model was intended for learning from scratch, therefore, it was not able to properly finetune vocabulary matrix with pre-computed word2vec embeddings. All of these things are worth trying and can be used for future work.

However, due to the fact that these features didn't perform well, we decided not to use them for Learning To Rank, although, they can possibly serve as good measures for learning to rank.

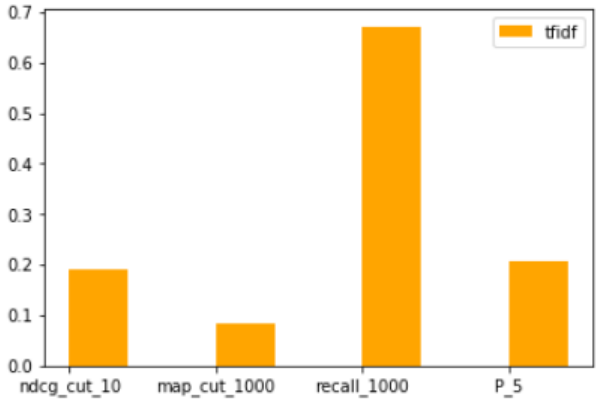

Figure 15: Results for RankNet on validation set queries

4 TASK 4 LEARNING TO RANK

4.1 RankNet

We implemented RankNet, which uses the following pairwise loss function:

$$\mathcal{L} = \sum_{d_i > d_j} \phi(f(x_i) - f(x_j))$$

Where f is any scoring function, $\phi(x) = \log(1 + \exp(-\gamma x))$ and γ is set to 1 for simplicity. Since the RankNet loss is defined for all documents of a single query, during the training, we calculate the loss separately for every query. The losses are summed up and the gradient are calculated after all the queries are processed.

We try two models for predicting the scores: first we use a In our case, we chose to use a one hidden layer neural network with 500 hidden layers. The network uses RELU activation function and applies a sigmoid to its output, so that the score predictions are between 0 and 1. This was done to avoid some minor numerical instabilities. We used SGD as optimizer, with momentum 0.5 and weight decay 0.001. We use 10 fold cross validation on the test set as requested and evaluate on the validation set. As requested in the task, we evaluate on the validation queries only the top 1000 documents ranked by tfidf.

4.1.1 Batched implementation. Naively iterating over all the document pairs to calculate the loss is very inefficient, so we used a trick for calculating the loss with much less computational effort. For one single query, we can say that our goal is to obtain a matrix $D \in \mathcal{R}^{n \times n}$ whose element $d_{ij} = \phi(f(x_i) - f(x_j))$, and n is the number of documents we consider for that query. To compute D we create a matrix $A \in \mathcal{R}^{n \times n}$ for which $a_{ij} = f(x_i)$. In other words, the columns of A are all the equal, and every column has element $a_i = f(x_i)$. It is easy to see then that $D = A - A^\top$. To correctly sum only over the documents that have higher relevance than other documents as indicated in the loss function, we also use the same trick so that we do not have to iterate over all document pairs. We create a matrix R in the same way we created D, but instead of using $f(x_i)$ as starting vector, we use the vector r of relevances of the documents. In this way, we get a matrix whose values are greater than zero when $r_i > r_j$, lower than zero if $r_i < r_j$ and zero otherwise. We threshold the values of the matrix so that negative numbers become zeros, and from the resulting matrix we can get a masking matrix which we can multiply to $\phi(D)$ to obtain only the score values required to compute the loss function. These operations can be done efficiently for reasonably high number of documents n, especially when using GPUs, and in our case using these computations alowed us to run the training more than 100 times faster.

4.2 Input features

As input features, we choose to use the scores of tfidf, bm25 and all the language models of task 1, together with document length and query length. In our analysis we are interested in finding out whether combinations of those features can allow a better scoring than if we only use the features as inputs. So, we experiment using a network with one hidden layer in order to make the network capable of learning more complex feature combinations. Also, we facilitate this process by producing new features for every of the 10 scores of the language models. We do so by producing the first 10 powers of those for every document query pair. This is because polynomials are often used as features (such as in linear regression) because they can potentially approximate any function. Using these would then help finding out whether very complex functions of the scoring methods can be used for obtaining optimal rankings.

4.3 Results

After training the model, we test it on the top 1000 documents ranked by tfidf for every query in the validation set, and obtain the results in figure 15. The results are clearly inferior to the ones obtained in previous sections, and this is most likely caused by an improper training process. As we discovered, the network weights changed very slowly during the training, and the loss was also decreasing very slowly. Since the method uses features from other ranking methods, we can expect that after a proper training procedure the method can lead to at least comparable results to the original methods.

REFERENCES

[1] Minmin Chen. 2017. Efficient Vector Representation for Documents through Corruption. *CoRR* abs/1707.02377 (2017). arXiv:1707.02377 http://arxiv.org/abs/1707.02377

[2] Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATSâĂŹ10). Society for Artificial Intelligence and Statistics*.

[3] Çaglar Gülçehre, Marcin Moczulski, Misha Denil, and Yoshua Bengio. 2016. Noisy Activation Functions. *CoRR* abs/1603.00391 (2016). arXiv:1603.00391 http://arxiv.org/abs/1603.00391

[4] Christophe Van Gysel, Maarten de Rijke, and Evangelos Kanoulas. 2017. Neural Vector Spaces for Unsupervised Information Retrieval. *CoRR* abs/1708.02702 (2017). arXiv:1708.02702 http://arxiv.org/abs/1708.02702

[5] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *CoRR* abs/1412.6980 (2014). arXiv:1412.6980 http://arxiv.org/abs/1412.6980

[6] Quoc V. Le and Tomas Mikolov. 2014. Distributed Representations of Sentences and Documents. *CoRR* abs/1405.4053 (2014). arXiv:1405.4053 http://arxiv.org/abs/1405.4053

[7] Yuanhua Lv and ChengXiang Zhai. 2009. Positional Language Models for Information Retrieval. In *Proceedings of the 32Nd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '09)*. ACM, New York, NY, USA, 299–306. https://doi.org/10.1145/1571941.1571994

[8] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. *CoRR* abs/1301.3781 (2013). arXiv:1301.3781 http://arxiv.org/abs/1301.3781