COMP0084 Information Retrieval and Data Mining (2024/25) Coursework 2 - Report

Abstract

This coursework explores a multi-stage information retrieval pipeline using learning-to-rank techniques. It begins with traditional IR methods like BM25 to generate initial candidate rankings, followed by feature-based models including Logistic Regression and LambdaMART for reranking. Word2Vec embeddings are used to represent queries and passages, with cosine similarity as an additional feature. Finally, a neural re-ranking model based on DSSM is implemented to further improve relevance estimation. Evaluation is conducted using standard IR metrics such as Mean Average Precision (mAP) and Mean Normalized Discounted Cumulative Gain (mNDCG) across all stages.

1 Introduction

1.1 Data

This experiment utilizes a total of four datasets:

- **validation_data.tsv**
- **train_data.tsv**
- **candidate_passages_top1000.tsv**
- **test-queries.tsv**

Among them, **validation_data.tsv** and **train_data.tsv** contain 4,364,339 rows and 1,103,039 rows instances respectively, each formatted as '<qid pid query passage>'. These two datasets are employed for model training and evaluation.

The **candidate_passages_top1000.tsv** file is used to construct input features for the test set and to generate test instances. The **test-queries.tsv** file defines the format and scope of the final submission, ensuring that ranking results are generated only for the queries in the test set.

1.2 Data Processing

1.2.1 Data preprocessing: All query and passage texts were standardized through a series of preprocessing steps, including lowercasing, punctuation removal, tokenization, and the elimination of English stopwords. These operations were performed using regular expressions in conjunction with the stopword list provided by NLTK. his part is realized by function `data_pre_processing`.

1.2.2 Subsampling for training set: Upon inspecting the training dataset, it was observed that there exists a significant imbalance between relevant and irrelevant query-passage pairs. Specifically, there are 4,359,542 irrelevant records compared to only 4,797 relevant ones. To address this issue and enhance the efficiency of model training, this study employed a sub-sampling strategy. This approach retains all relevant records while randomly selecting 10 irrelevant records for each query. As a result, a smaller and more balanced training set was constructed, comprising a total of 50,548 records. After generating word embeddings, this processed dataset was saved as **train_new.txt** and subsequently used for training all models after Task 1 in this study.

2 Task 1 - Evaluation Metrics and BM25

2.1 Average Precision

In information retrieval, it is not only important whether a model successfully identifies relevant documents, but also whether it ranks them at higher positions in the result list. To evaluate the quality of ranking in retrieval results, Average Precision (AP) and mAP are widely adopted as standard evaluation metrics.

For a given query q, suppose the system returns a ranked list of documents $D_1, D_2, \ldots, D_N$, and the set of relevant documents is denoted as R_q. The Average Precision (AP) for query q is calculated as:

$$AP(q) = \frac{1}{|R_q|} \sum_{k=1}^{N} P(k) \cdot rel(k). \tag{1}$$

In Equation 1, $|R_q|$ is the total number of relevant documents for query q (i.e., the ground truth). $P(k)$ represents the precision at rank k, which is defined as the proportion of relevant documents among the top k retrieved results. $rel(k)$ is a binary indicator function that equals 1 if the document at rank k is relevant to the query, and 0 otherwise.

Instead, mAP was used as an assessment criterion in this experiment. Given a set of queries $Q = \{q_1, q_2, \ldots, q_m\}$, the mAP is defined as the mean of the AP scores across all queries:

$$mAP = \frac{1}{|Q|} \sum_{i=1}^{|Q|} AP(q_i). \tag{2}$$

In Equation 2, $|Q|$ is the total number of queries, and $AP(q_i)$ is the Average Precision for the i-th query.

2.2 Mean Normalized Discounted Cumulative Gain

In addition to precision-based metrics, Normalized Discounted Cumulative Gain (NDCG) is widely used to evaluate the ranking quality in information retrieval systems. NDCG takes into account the position of relevant documents in the ranked list, giving higher weights to documents appearing at higher ranks.

For a single query, the Discounted Cumulative Gain (DCG) at rank position k is defined as:

$$DCG@k = \sum_{i=1}^{k} \frac{2^{rel_i} - 1}{\log_2(i+1)}, \tag{3}$$

where rel_i denotes the relevance score of the document at position i. The Ideal DCG (IDCG) is the maximum possible DCG obtained by sorting documents by their ground-truth relevance in descending order:

$$IDCG@k = \sum_{i=1}^{k} \frac{2^{rel_i^*} - 1}{\log_2(i+1)} \tag{4}$$

In Equation 4, rel_i^* denotes the relevance score at rank i in the ideally sorted list. The normalized version of DCG is computed as:

$$NDCG@k = \frac{DCG@k}{IDCG@k}. \tag{5}$$

To evaluate performance over a set of queries $Q = \{q_1, q_2, \ldots, q_M\}$, the mNDCG is computed as the average NDCG across all queries:

$$\text{mNDCG} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} NDCG(q_i), \tag{6}$$

where $NDCG(q_i)$ denotes the NDCG score for the i-th query.

2.3 BM25 Evaluation Results

In this task, the evaluation results based on the mAP and mNDCG are presented in Table 1.

Table 1: Evaluation Results for BM25

mAP	mNDCG
0.2286	0.3693

3 Task 2 - Logistic Regression (LR)

In Task 2, we implemented a logistic regression model based on the average word embeddings generated by Word2Vec, which is used to predict the relevance between queries and passages. We evaluated the effect of different learning rates on the training loss and assessed the model performance on the validation set using mAP and mNDCG. Finally, the model was applied to the test set to generate the output file **LR.txt**.

3.1 Word Embedding - Word2Vec

After subsampling, in order to convert queries and passages into numerical feature vectors suitable for machine learning models, this study employs the Word2Vec technique for text vectorization. The entire process consists of five steps.

First, the data preprocessing function `data_pre_processing`, as defined in Task 1, is applied to the queries and passages in the training, validation, and test sets to normalize the text.

Second, the list of tokens for each query and passage is concatenated into sentences using whitespace and saved into separate text files (e.g., **queries_train.txt**, **passage_train.txt**). Each line in these files corresponds to a single sample, and the format is compatible with the `gensim.models.word2vec.LineSentence` interface, making it suitable for subsequent Word2Vec training.

Third, the `Word2Vec` tool provided by Gensim is used to independently train word embedding models for queries and passages. The training parameters are summarized in Table 2.

Table 2: Parameter Set for Word2vec

sg	vector_size	window	mini_count	negative	workers
1	100	5	1	5	4

The key parameters used in the Word2Vec training process are explained as follows.

The **vector_size** parameter specifies the dimensionality of the word embeddings, i.e., the number of dimensions each word is encoded into. A larger embedding size enables the model to capture richer semantic information but also increases computational complexity. Due to hardware constraints, **vector_size** is set to 100 in this study.

The **window** parameter determines the size of the context window, which defines the number of neighboring words (to the left and right) considered for each target word during training. A commonly used value for **window** is 5[4].

The **min_count** parameter defines the minimum frequency threshold for a word to be included in training. Words that appear less frequently than this threshold are ignored, which helps reduce noise from rare terms and improves training efficiency. In this work, **min_count** is set to 1.

After preprocessing, for each of the training, validation, and test sets, two separate Word2Vec models are trained: one for queries and one for passages.

In the fourth step, for every query and passage, the average of all word vectors (obtained from the corresponding Word2Vec model) is computed to obtain a single query vector and a single passage vector. These two vectors are then concatenated <**query** $\oplus$ **passage**> to form the final feature vector for each sample. For training and validation samples, the associated label (0 or 1) is also recorded.

Finally, each sample's query ID and passage ID (**qid**, **pid**), the concatenated feature vector, and its label (if available) are saved into text files named **train_new.txt**, **val_new.txt**, and **test_new.txt**, which are used in the subsequent training, evaluation, and prediction (re-ranking) stages for future three models.

3.2 Logistic Regression Model

Given an input feature vector

$$\mathbf{x} = [x_1, x_2, \ldots, x_d] \in \mathbb{R}^d,$$

and model parameters:

- Weight vector: $\mathbf{w} = [w_1, w_2, \ldots, w_d] \in \mathbb{R}^d$
- Bias term: $b \in \mathbb{R}$

The predicted relevance score $\hat{y}$ is computed using the sigmoid function:

$$\hat{y} = \sigma(\mathbf{w}^\top \mathbf{x} + b), \tag{7}$$

where the sigmoid activation is defined as:

$$\sigma(z) = \frac{1}{1 + e^{-z}}. \tag{8}$$

To optimize the model, this experiment use the binary cross-entropy loss function. For a single training instance with true label $y \in \{0, 1\}$ and predicted score $\hat{y}$, the loss is:

$$\mathcal{L}(y, \hat{y}) = -\left[y \cdot \log(\hat{y}) + (1 - y) \cdot \log(1 - \hat{y})\right]. \tag{9}$$

The overall average loss over n training samples is:

$$\mathcal{L}_{\text{avg}} = -\frac{1}{n} \sum_{i=1}^{n} \left[y^{(i)} \cdot \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \cdot \log(1 - \hat{y}^{(i)})\right]. \tag{10}$$

The gradient of the loss with respect to the weights and bias is given by:

$$\nabla_{\mathbf{w}} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}^{(i)} - y^{(i)}) \cdot \mathbf{x}^{(i)}, \quad \nabla_b = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}^{(i)} - y^{(i)}). \quad (11)$$

The model parameters are updated via batch gradient descent using a learning rate η:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \cdot \nabla_{\mathbf{w}}, \quad b \leftarrow b - \eta \cdot \nabla_b.$$

In our implementation, we use full-batch gradient descent instead of mini-batch or stochastic variants, as the dataset size is relatively small and computationally manageable.

3.3 Training and Evaluation for LR Model

We trained the logistic regression model using the previously generated **train_new.txt** file. In this experiment, two rounds of model training were conducted. In the first round, the maximum number of iterations was set to 500, and the learning rate list was set to 0.01, 0.005, 0.001, 0.0005, 0.0001. In the second round, the maximum number of iterations was increased to 5000, while keeping the same set of learning rates.

In addition, a tolerance parameter **tol** was introduced as a convergence criterion during training. During training, we monitor the L2 norm of the gradient and stop the optimization process early if it falls below a small threshold **tol**, i.e.,

$$\|\nabla_{\mathbf{w}}\|_2 < \mathbf{tol}.$$

This serves as a convergence criterion to prevent unnecessary computation after the model has essentially stabilized. In both training experiments, the tolerance parameter was set to $\mathbf{tol} = 10^{-4}$, which served as a reasonable stopping criterion for convergence.

Subsequently, the loss convergence curves under different learning rates were plotted for both training experiments, as shown in Figure 1 and Figure 2.

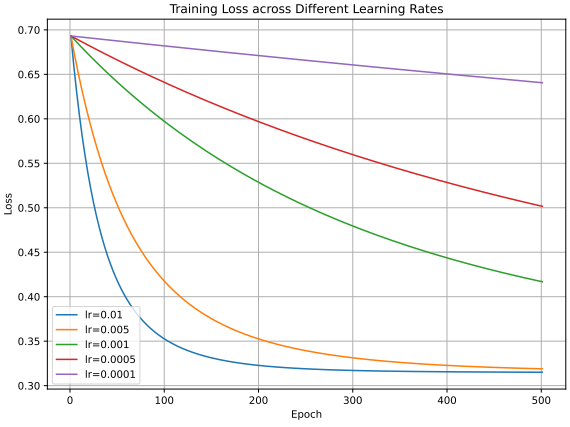

Figure 1: Loss Change for 500 Iterations

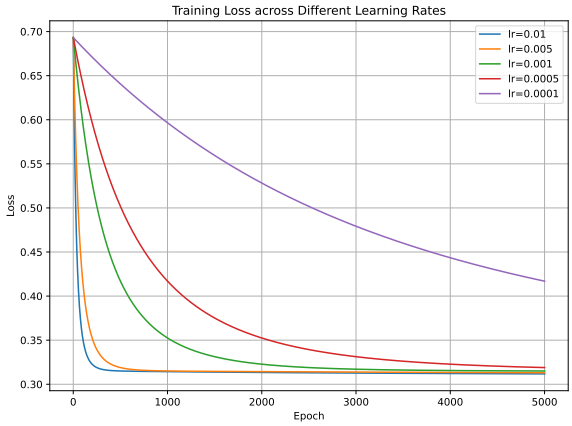

Figure 2: Loss Change for 5000 Iterations

By examining Figure 1 and Figure 2, it can be observed that higher learning rates lead to faster convergence of the loss. For instance, when the learning rate is set to 0.01, the model converges rapidly and nearly stabilizes within just 500 iterations. When the number of iterations is increased to 5000, learning rates such as 0.01 and 0.005 achieve convergence to a loss value of approximately 0.322. In contrast, lower learning rates, such as 0.0001, fail to reach the same level of convergence within the same number of iterations.

Furthermore, the validation set **val_new.txt** is loaded and evaluated using the pre-trained logistic regression model. Relevance scores are computed for each query-passage pair and sorted in descending order. The evaluation metrics, mAP and mNDCG, are computed using the same methodology as in Task 1. The final results are summarized in Table 3.

Table 3: Evaluation for Logistic Regression Model

Model Name	mAP	mNDCG
LR	0.0137	0.1329

As shown in Table 3, the model's performance is not satisfactory. There are several possible reasons for this suboptimal result. First, the limited number of relevant samples in the dataset may lead to a severe class imbalance problem. Although subsampling has been performed, the majority of training instances still are labeled as irrelevant. This imbalance can cause the model to overfit the dominant class and predict most pairs as non-relevant.

Second, the poor performance may be attributed to the limited quality of the word embeddings. Word2Vec typically requires a large-scale corpus, such as Wikipedia or news data, to learn meaningful representations. However, in this experiment, the input texts consist mostly of short queries and passages, which may result in insufficient word co-occurrence statistics. Consequently, many important words—especially low-frequency ones—may have poorly trained embeddings.

Lastly, the use of average word vectors for representing queries and passages may dilute crucial semantic information, such as key terms or negation words. This averaging process may fail to capture

fine-grained alignment or matching signals between the query and passage, making it difficult for the model to accurately distinguish relevant pairs even when trained correctly[3].

4 Task 3 - LambdaMART (LM)

In this Task, a learning-to-rank model based on LambdaMART was implemented using XGBoost. The same feature vectors from Task 2 were used, with an additional cosine similarity value appended to each query-passage pair. The model was trained using pairwise ranking objectives, and a grid search over multiple hyperparameters was conducted. The best-performing model was selected based on validation performance using mAP and mNDCG, and the final predictions for the test set were saved to **LM.txt**.

4.1 Feature Preparation

In addition to the horizontal concatenation of the query and passage vectors, an additional feature—the cosine similarity between the two vectors—was included. As a result, the input dimensionality increased from 200 to 201. The final input format is represented as:

$$\textbf{query\_vec} \oplus \textbf{passage\_vec} \oplus \textbf{cosine\_similarity}$$

The cosine similarity is computed using the following formula:

$$\text{cosine\_similarity}(\mathbf{q\_v}, \mathbf{p\_v}) = \frac{\mathbf{q\_v} \cdot \mathbf{p\_v}}{\|\mathbf{q\_v}\| \cdot \|\mathbf{p\_v}\|}. \tag{12}$$

4.2 Data Grouping

To train the ranking model, query-passage pairs were first grouped by their qid, with each group containing all candidate passages corresponding to the same query. For each group, the pairs were sorted in descending order according to their relevance labels to match the input format required by **XGBRanker**. The group sizes were recorded and passed to the model during training to inform it of the ranking structure within the dataset.

4.3 Parameter Setting

The learning-to-rank model was implemented using **XGBRanker** from the **xgboost** library. The booster type was set to **gbtree**, and the objective function was set to **rank:ndcg**, which instructs the model to optimize for the NDCG ranking metric. Although **XGBRanker** does not explicitly name itself as LambdaMART, this configuration is functionally equivalent to the LambdaMART algorithm, as it trains gradient-boosted decision trees using pairwise ranking objectives aligned with NDCG.

4.4 Hyper-parameter Tuning

The hyperparameters were set as shown in Table 4.

Table 4: Hyperparameter Setting

Hyperparameter	Value
Learning Rate	0.01, 0.1
Maximum Tree Depth	5, 10
Subsample Ratio	0.8, 1
Column Subsample Ratio	0.8, 1
Number of Trees	100, 200

The following provides explanations for the four hyperparameters excluding the learning rate:

- Maximum Tree Depth (**depth**): This parameter limits the maximum depth of each decision tree. A deeper tree allows the model to capture more complex interactions but also increases the risk of overfitting. In this experiment, values of 5 and 10 were tested.
- Subsample Ratio (**subsample**): This parameter denotes the fraction of the training data used to grow each tree. Subsampling helps reduce overfitting and improves model robustness. Values of 0.8 and 1.0 were explored.
- Column Subsample Ratio (**colsample**): This controls the fraction of features (columns) randomly sampled for constructing each tree. Using a smaller fraction can help introduce feature-level regularization and reduce correlation among trees. Values of 0.8 and 1.0 were explored.
- Number of Trees (**trees**): This determines the number of boosting rounds or decision trees in the ensemble. A higher number of trees typically allows better fitting but increases computational cost. In this experiment, 100 and 200 trees were used across all configurations.

Validation and Testing

After enumerating all possible combinations of the selected hyperparameter values, a total of 32 models were trained. Upon completion of training, each model was evaluated on the **val_new.txt** dataset using mAP and mNDCG as evaluation metrics. The results are summarized in Table 5.

Table 5: Evaluation Results for LM Models

Model	mAP	mNDCG
1	0.0113	0.1302
2	0.0122	0.1312
3	0.0112	0.1311
...	...	...
9	0.0119	0.1327
10	0.0121	0.1322
11	**0.0151**	**0.1348**
...	...	...
32	0.0144	0.1341

Due to space limitations, only a subset of the results is presented here. As shown, Model 11 achieved the best performance, with both the highest mAP and mNDCG scores. The corresponding hyperparameter configuration is highlighted in red in Table 3. Although the improvement is not substantial, the best-performing LambdaMART model outperforms the LR model by a small margin.

Finally, the best-performing model selected during the validation phase was used to predict relevance scores on the **test_new.txt** dataset. For each query, the top 100 passages with the highest scores were retained according to the order of queries in **test-queries.tsv**. The re-ranked results were saved to the **LM.txt** file, containing a total of 19,290 records.

5 Task 4 - Neural Network (NN)

In Task 4, a neural ranking model was implemented using a feed-forward Deep Structured Semantic Model (DSSM) architecture in PyTorch. The model reused the same feature vectors from Task 3, which consist of the concatenation of query and passage embeddings along with an additional precomputed cosine similarity value. The network comprises two parallel subnetworks that project the query and passage into a shared semantic space via multi-layer perceptrons (MLP). A cosine similarity is computed between the semantic representations, and this value is concatenated with the raw cosine similarity from the input. The combined similarity features are then passed through a final classification layer to predict relevance. The model was trained using the cross-entropy loss and optimized via the Adam optimizer. Evaluation on the validation set was conducted using mAP and mNDCG. Finally, the trained model was applied to the test set to generate the output file **NN.txt**.

5.1 Neural Network Structure

We implemented a neural model inspired by the DSSM architecture proposed by Huang et al. (2013)[2]. The network follows a twin-tower design, where both the query and the passage vectors are passed through independent but structurally identical MLP. Each MLP contains a linear layer, followed by a Tanh activation and dropout for regularization. The architecture of the model is illustrated in Figure 3.

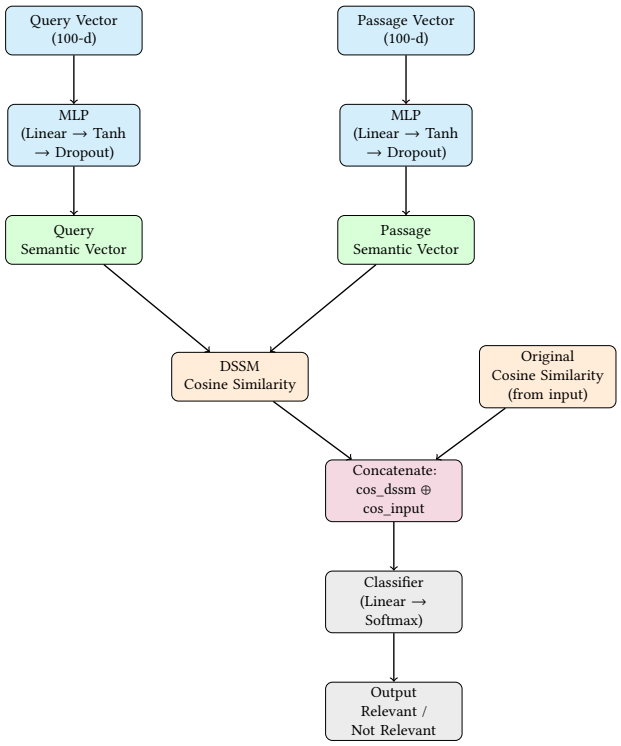

Figure 3: Architecture of the DSSM-based Neural Ranking Model

Specifically, the architecture proceeds as follows:

- **Input Representation**: The input vector is same as Task 3, a concatenation of the query embedding (100-d), passage embedding (100-d), and their raw cosine similarity (1-d), forming a 201-dimensional vector.
- **Semantic Projection**: The query and passage embeddings are separately passed through MLPs, projecting them into a semantic space (hidden_dim = 128).
- **Similarity Calculation**: A cosine similarity is computed between the projected query and passage vectors.
- **Feature Augmentation**: We concatenate this learned similarity with the raw cosine similarity from the input. This explicitly incorporates both raw and learned similarity signals, serving as an additional handcrafted feature.
- **Classification Layer**: The two similarity values are fed into a final linear layer with a softmax output, predicting the binary relevance label.

5.2 Why Feature Augmentation

This design choice for features was motivated by the need to enhance the discriminative power of the DSSM by augmenting it with handcrafted similarity features. Simply put, the original cosine similarity is computed in the "unprojected" semantic space, whereas the cosine similarity produced by DSSM is calculated in the transformed semantic space after nonlinear projection through neural networks. The original DSSM relies solely on the learned similarity, but we hypothesize that incorporating the raw cosine similarity can bring additional value, especially when the learned semantic space is imperfect or the training data is limited. In the implementation, this corresponds to concatenating the two cosine similarity values using `torch.cat`, which are then fed into the final classifier.

5.3 Why DSSM

There are two main reasons for adopting the DSSM architecture in this part of the experiment.

First, the input feature design aligns well with the DSSM structure. Since Task 4 reuses the composite input features constructed in Task 3—**<query_vec ⊕ passage_vec ⊕ cosine_similarity>**—the DSSM model encodes the query and passage vectors separately using the `self.query_proj` and `self.passage_proj` modules, each consisting of a `Linear → Tanh → Dropout` sequence. This non-linear transformation maps the original average word embeddings into a new, learnable semantic space. The similarity between the query and passage representations is then computed via `F.cosine_similarity()`, which is concatenated with the original cosine similarity feature and passed into the final classification layer for relevance prediction. Although average word embeddings are used, the model structure is naturally compatible with the input format **<q_vec ⊕ p_vec ⊕ cosine>**, making DSSM a suitable choice for this task.

Second, from an architectural perspective, DSSM is essentially a Siamese (dual-tower) network. Such architectures are widely adopted in large-scale industrial retrieval systems, including Bing and YouTube recommendation models [1]. Their key advantage lies in modeling query and document representations separately and enabling efficient matching through a shared semantic space

or similarity function. This design allows the model to capture semantic relationships even when the query and document share little or no lexical overlap, making it highly suitable for query-document matching in information retrieval tasks.

5.4 Validation and Testing

During the evaluation phase, the **val_new.txt** dataset was first loaded. Following the same procedure as in training, the cosine similarity between each query-passage vector pair was computed and concatenated with the embedded vectors, resulting in a 201-dimensional input for each instance. The trained DSSM model was then reloaded from the **DSSMmodel.pkl**, which saved in training process, using to generate predictions on the validation set in evaluation mode. For each query-passage pair, the model produced a relevance probability through a softmax output, and the score for the "relevant" class was used for ranking.

The model performance was evaluated using mAP and mNDCG, and the results are presented in Table 6.

Table 6: Evaluation for DSSM Model

Model Name	mAP	mNDCG
NN (DSSM)	0.0108	0.1297

In the final testing stage, the **test_new.txt** dataset was fed into the trained model to obtain relevance scores. The output results were directly based on the scores produced by the DSSM model, resulting in a total of 19,290 records. Following the order of queries in **test-queries.tsv**, the top 100 re-ranked passages for each query were saved in the file **NN.txt**.

References

[1] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*. 191–198.

[2] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. 2013. Learning deep structured semantic models for web search using clickthrough data. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*. 2333–2338.

[3] Zhouhan Lin, Minwei Feng, Cicero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, and Yoshua Bengio. 2017. A structured self-attentive sentence embedding. *arXiv preprint arXiv:1703.03130* (2017).

[4] Xin Rong. 2014. word2vec parameter learning explained. *arXiv preprint arXiv:1411.2738* (2014).