COMP0084 Coursework 2

Runyang You
runyang.you.22@ucl.ac.uk
University College London
London, UK

ACM Reference Format:
Runyang You. . COMP0084 Coursework 2. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 6 pages.

1 DATA PROCESSING

1.1 Data Description

The dataset was pre-divided into training set and validation set, for training the model and for validate the model performance respectively, each line within the tsv file is a query-passage pair, who provides the following information:

Name	Data Type	Description
qid	int	(Discontinuous) query identifier
pid	int	(Discontinuous) passage identifier
query	string	Query content
passage	string	Passage content
relevancy	int	Binary relevancy

Table 1: data description

Given the statistic from Table 2, it is worth-noticing that this a heavily unbalanced dataset, where the target-background ratio is 0.0011.

Relevancy	train dataset	validation dataset	percentage
1	4797	1208	0.11%
0	4,359,542	1101831	99.8%

Table 2: overall data statistic: the number of pairs and the percentages of different relevancy in the training set and validation set

Among the 4590 queries within the training set, 94% have exactly 1000 candidate passages, while the majority of those of the left 288 queries lie in $[0, 50]$.

1.2 Data Preprossessing

For ease of calculation and further processing, the following steps were taken during the data preprocessing stage:

(1) The tsv file was read using pandas library and re-stored in parquet format, which is a more efficient and highly-compressed storage format that is optimized for reading large datasets.
(2) The 2 datasets are sorted separately by the qid, relavency and pid sequentially.
(3) 2 columns are added for both datasets, namely q_idx(query index), which can be seen as the continuous query identifier starts from 0; and p_idx(passage index), which is the index of the passage within the candidates for a given query that also starts from zero.

1.3 Embedding

1.3.1 Choice of Word Embedding Method. Table 3 shows a detailed comparison of different pre-trined word embedding models. Although it is generally fair to say that the state-of-the-art model produces better results in diverse natural language processing(NLP) tasks, considering the limited computation of my laptop, the embedding method for this coursework is set to be FastText.

FastText is a library developed by Facebook's AI Research (FAIR) lab for learning word embeddings and performing text classification [4]. Compared to ELMo [12] and GPT-2 [13], FastText produces smaller result vectors, which reduces the memory and computation requirements.

One of the key advantages of FastText over Word2Vec [9] and GloVe [11] is its ability to handle out-of-vocabulary (OOV) words by utilizing subword information. This enables FastText to generate meaningful representations for morphologically complex words, including inflected forms, without requiring stemming or lemmatization. In fact, some studies have suggested that lemmatization may cause the loss of important subword information and reduce the quality of the generated embeddings [5].

According to the above analysis, the passages and queries are tokenized and then embedded based on FastText pretrained model directly, without stemming or lemmatization. This can be a one-phase procedure using *Flair* [1] library.

1.3.2 Averaging Word Embeddings. There are various methods of averaging word embeddings in a document, such as the very classic simple averaging, weighted averaging [2], and more advanced methods based on a specific word embedding solely [8]. However, many handcrafted weighting methods, apart from some neural network-based algorithms, require a well-constructed corpus and rely on certain metrics, such as the probability of occurrence [6] or Shannon's Mutual Information (MI) among words [3]. These methods heavily depend on a large, high-quality corpus, which can be difficult to find online for free, or may result in lower performance due to suboptimal weight optimization [15].

To preserve the crucial benefits of FastText, among which is the robustness against unseen tokens and fast inference without preprocessing tokens, the document (a passage or a query)'s embedding was computed by averaging all the word embeddings within and normalizing the result by the document's length, as instructed in the coursework.

Model	Training Algorithm	Context Type	Vector Size
Word2Vec	Continuous bag-of-words (CBOW) or skip-gram	Word-level	300
GloVe	Count-based method	Word-level	25-300
FastText	Continuous bag-of-words (CBOW) or skip-gram	Subword-level	300
ELMo	Deep bidirectional language model	Character-level	1024 or 2048
GPT-2	Transformer	Word-level and subword-level	768-2048

Table 3: Comparison between popular pre-trained embedding model [4, 9, 11–13]

1.4 Negative Sampling

Negative sampling is a technique used in natural language processing (NLP) and specifically in word embeddings to enhance the efficiency and accuracy of training models. In word2vec [9], for example, only a small number of negative samples (usually 5-20) are used to compute the probability of words in the vocabulary, resulting in faster and more efficient training while maintaining accuracy.

Inspired by this idea, I adopted negative sampling by only feeding a fixed number (e.g., 10) of irrelevant passages, denoted as $|P|$, into the model for weight updating. During each epoch, if a query has less than $|P|$ candidates, all candidates are fed into the model. If a query has more than $|P|$ candidates, relevant candidates are selected while irrelevant candidates are randomly chosen to make up the difference to $|P|$.

This approach guarantees that all positive samples are included in the training process while utilizing the original negative sample pool. Thus, the model learns unique features from different negative samples in each epoch, leading to better generalization performance. Additionally, the number of negative samples can be adjusted based on the specific characteristics of the model to further improve the training process.

2 EVALUATION METRIC

Mean average precision and NDCG (normalized discounted cumulative gain) metrics are used for the evaluation of all retrieving methods within this coursework.

2.1 Evaluate Query

To evaluate the retrieving result of a given query, the *eval_per_query* function should be called to calculates the average precision and NDCG scores by taking in 3 parameters:

- `relev_rank`: array-like object that contains rankings of the relevant passages in the retrieved results.
- `at`: a list of integers at which precision and NDCG are to be computed.
- `log`: an optional callable object that defaults to the numpy logarithm.

The function sorts in ascending order, and then computes the cumulative sum of the inverse of logarithm of 1 plus the `relev_rank` parameter elements, which is the standard DCG calculated using the following equation:

$$DCG_p = \sum_{i=1}^{p} \frac{rel_i}{\log(i+1)} = \sum_{r \in R_p} \frac{1}{\log(r+1)} \tag{1}$$

$$R = \{i \in \mathbb{N} \mid \text{rel}_i = 1\}, \quad R_p = \{i \in R \mid i \le p\}$$

Here p denotes the current position, R is the set representation of *relev_rank*, $\mathbb{N}$ is a set of indices of retrieved passages.

For each position p in at , the function will:

(1) filters out the ranks in *relev_rank* behind the current position and obtain R_p;
(2) computes the average precision value as a normalized sum of individual precision scores

$$AP = \frac{1}{|R_p|} \sum_{i=1}^{|R_p|} \frac{i}{r_i} \tag{2}$$

where r_i is the i_{th} rank value in R, $|R_p|$ denotes the size of R_p (amount of relevant passages retrieved at p)
(3) computes the ideal DCG supposing the retrieved relevant passages all rank at the top:

$$IDCG_p = \sum_{r=1}^{min(N,|R|)} \frac{1}{\log(r+1)} \tag{3}$$

(4) computes the NDCG value by dividing DCG value by the ideal DCG at this position:

$$nDCG_p = \frac{DCG_p}{IDCG_p} \tag{4}$$

2.2 Evaluate Model

For each model discussed in the following sections, the evaluation process involves retrieving the results of every query within the validation set and calculating their NDCG and AP values using the aforementioned function. The resulting lists, each with a length equal to the number of queries, are used to determine the mean values of both AP and NDCG, referred to as *mAP* (mean average precision) and *mNDCG* (mean NDCG), respectively. These two scores serve as the final evaluation metrics for assessing the performance of each model.

Metric, At	3	10	100
mAP	0.196	0.231	0.242
mNDCG	0.214	0.288	0.358

Table 4: Evaluation result of BM25

The evaluation result of BM25 as an Example is shown in Tab. 4

3 LOGISTIC REGRESSION

Logistic regression is a statistical method that uses a sigmoid function to convert the output of a linear regression model into the probability of the dependent variable belonging to a certain class. This probability indicates the likelihood of an outcome occurring based on the input variables. A threshold of 0.5 is commonly used to make a decision based on the probability. In this work the threshold is set to be 0.5, as a common practice.

3.1 Data Preparation

As outlined in Section 1.3, both passages and queries are represented as 300-dimensional vectors. For logistic regression, each training sample is a concatenation of the query and passage vectors, resulting in a 600-dimensional vector.

Due to the imbalanced nature of the dataset and the way logistic regression minimizes the error rate over all observations, the algorithm may prioritize minimizing the error rate for the larger class - the '0' relevancy, leading to poor ranking performance. Therefore, negative sampling (explained in Section 1.4) is used with a setting of $|P| = 20$ to reduce computational load while retaining informative and important samples.

3.2 Training

Algorithm 1 Batch Logistic Regression

```
Require: $X$ of shape $(N, d)$, $y$ of shape $(N)$, learning rate $lr$, number of iterations $n\_iters$, batch size $N_{batch}$
1:  $b \leftarrow 0$
2:  $w \leftarrow \mathbf{0}_d$
3:  for $t \leftarrow 1$ to $n\_iters$ do
4:      for $i \leftarrow 1$ to $N$ with step $N_{batch}$ do
5:          $n\_batch \leftarrow \min(N - i, N_{batch})$
6:          $X_b \leftarrow X[i : n\_batch]$
7:          $y_b \leftarrow y[i : n\_batch]$
8:          $\alpha \leftarrow \frac{lr}{n\_batch}$
9:          Compute the predicted probabilities
10:         $p = \sigma(X_b \cdot w + b)$
11:         Compute the gradients
12:         $dw \leftarrow {X_b}^T \cdot (p - y_b)$
13:         $db \leftarrow \sum(p_j - {y_b}_j)$
14:         Update parameters
15:         $w \leftarrow w - \alpha \cdot dw$
16:         $b \leftarrow b - \alpha \cdot db$
17:         $L \leftarrow -\frac{1}{n} \sum_{i=1}^{n} \left[ {y_b}_i \log(p_i) + (1 - {y_b}_i) \log(1 - p_i) \right]$
18:     end for
19: end for
20: return $w, b$
```

The logistic regression model uses gradient descent to optimize its weights and bias, which is shown in Algorithm 1. The objective of gradient descent is to adjust the weights and bias in a way that minimizes the BCE(binary cross-entropy) loss, which measures the difference between the predicted probabilities and the actual labels. To speed up the optimization process and avoid memory overflow, mini-batch gradient descent is employed, which updates the parameters based on a small subset of the training data at a time. The model was trained for 200 epochs with different learning rates, the epoch-loss curvy is illustrated in Fig. 1.

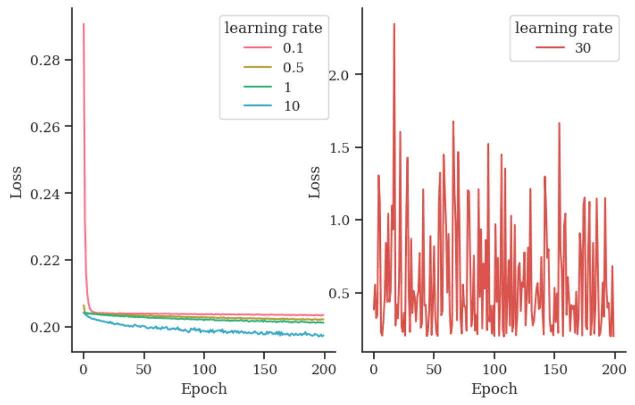

Figure 1: Training Epoch-Loss curve for different learning rates

Learning rate is a hyperparameter that affects how quickly or slowly your model learns from the data. When the learning rate is set unreasonably high (e.g., 30), the model not only oscillates dramatically but also fails to converge, resulting in poor training loss and model instability. This happens because the aggressive parameter updates cause the model to overshoot the optimal solution and fluctuate erratically. Conversely, if the learning rate is too low, the model may converge slowly and struggle to escape a local minimum, thereby converges to a relatively high loss, leading to suboptimal performance.

Therefore, it is crucial to select an appropriate learning rate through experimentation. To yield the best result I started experiment with a high learning rate, and gradually decreases it until the loss stops improving or oscillates around the optimal value. This can help the model to converge quickly while avoiding overshooting the optimal solution. Finally I choose the model trained for 200 epochs with a learning rate of 1.

3.3 Result and Discussion

Compared to the basic BM25 model in Tab. 4, the logistic regression model performs worse in terms of both mAP and NDCG, especially at larger positions, as shown in Tab. 5. This suggests that the logistic regression model has difficulty in accurately ranking and retrieving relevant documents for a given query.

The difference in performance between the two models can be attributed to both the level of complexity and the assumptions made by each algorithm. While BM25 is a more complex and sophisticated algorithm that considers multiple factors to determine document relevance, logistic regression is a simpler algorithm that may struggle to capture the complex relationships between the features and the relevancy of documents. Additionally, the poor performance of logistic regression could be due to underfitting, which highlights the importance of selecting a model with sufficient complexity to capture the underlying patterns in the data. Overall, the results suggest that the BM25 method is better suited for information retrieval tasks in this experiment, as evidenced by its superior performance in both mAP and mNDCG.

Metric, At	3	10	100
mAP	0.0092	0.0135	0.0183
NDCG	0.0102	0.0200	0.0324

Table 5: Performance of Logistic Regression Model

4 LAMBDAMART

LambdaMART is a ranking algorithm used in information retrieval that works by learning a function that maps queries and documents to their respective relevance scores. The LambdaMART algorithm is a variant of the MART (Multiple Additive Regression Trees) algorithm that uses gradient boosting to optimize the ranking function.

The `XGBRanker` from *XGBoost* library was used to train the LambdaMART model. `XGBRanker` is a class provided by the *XGBoost* library for learning-to-rank problems. It is an implementation of the Gradient Boosting framework, where each tree built during the training process is optimized to minimize a ranking loss function.

4.1 Data preparation

The data used for this model is the same as the logistic regression model, each sample is a 600-dimensional vector. Negative sampling (explained in Section 1.4) is used with a setting of $|P| = 20$. However, due to the inflexibility of the training interface provided by `XGBRanker`, the training samples are pre-defined and the chosen negative samples are fixed for every epoch, differ from that of Logistic Regression.

4.2 Training

4.2.1 XGBRanker Setting. As mentioned above, `XGBRanker` inherits from `XGBModel`, but it differs from the typical classification or regression models in that it takes an additional input `qid` to group the data regarding the query. The objective is set to `rank:ndcg`, which means that the model is trained to optimize the NDCG ranking metric.

4.2.2 Cross Validation. The hyper-parameter tuning process involved employing cross-validation, utilizing the `GroupKFold` method from *scikit-learn*. This approach was suitable for this task since LambdaMART uses list-wise loss, meaning that the order of the samples matters, and cross-validation must ensure that samples from the same query are kept together in the same fold.

The cross-validation was performed with n-splits using the `GroupKFold` method. Here the n was set as 5, same as the default choice of *scikit-learn*. For each split, a LambdaMART model was trained using the training set, and the performance was evaluated on the validation set. The hyper-parameters considered for tuning were the maximum depth of the trees, the number of estimators, the learning rate, the gamma value and the type of booster (gbtree or dart). These hyper-parameters were searched using a list of different combination of the might-be-best hyperparameters values using the `ParameterGrid` method from *scikit-learn*.

For each hyper-parameter combination, the LambdaMART model was trained and evaluated $n = 5$ times using cross-validation. Performance metrics were calculated using NDCG@100, since, due to the limited size of the dataset used to train, NDCG values with smaller k may contain more randomness and fail to reflect slow and small learning outcomes.

The combination of hyper-parameters that yielded the highest average NDCG@100 score over the five splits was selected as the optimal hyper-parameters, resulting in {'gamma': 2, 'max_depth': 15, 'n_estimators': 15}. These hyper-parameters were then utilized to train the final LambdaMART model on the entire negative-sampled training set.

4.3 Result and Discussion

Metric, At	3	10	100
mAP	0.0072	0.0114	0.0187
NDCG	0.0083	0.0151	0.0393

Table 6: Performance of LambdaMART Model

Based on the provided performance metrics, the LambdaMART model outperforms the logistic regression model sightly at higher position, this can be attributed to the cross-validation process. While the LambdaMART model was able to achieve better performance compared to the logistic regression model, it was still not able to outperform the BM25 baseline. This is likely due to a combination of factors, including the simplicity of the embedding method used and the small number of training samples.

5 NEURAL NETWORK

5.1 Data Preparation

This model uses a different data loading approach than previous methods. It takes in the query embedding $\mathbf{q}$ of size $N \times 1 \times 300$ and the passages embeddings $\mathbf{p}$ of size $N \times |P| \times 300$ separately. Negative sampling is also used, but for passage collections with a size smaller than $|P|$, relevant passages from other queries are used as extra negative samples to make up for the difference, ensuring that the input tensors have a fixed size. This method shortcuts the complexity of the model architecture design, which is detailed in the following section 5.2.

5.2 Model Design

5.2.1 Model Architecture. The architecture of the retrieval system is illustrated in Fig. 2, which can be roughly seperated into 3 parts:

Query Encoder The query encoder consists of 3 linear layer each followed by an leaky relu activation, with output shape of 128, 256, 512 respectively. This encoder aims to find a high-dimensional representation of the query that preserves the key information that might have been lost in the process of averaging embeddings. The successive linear layers enable the encoder to gradually learn more complex and abstract features from the input, while the leaky

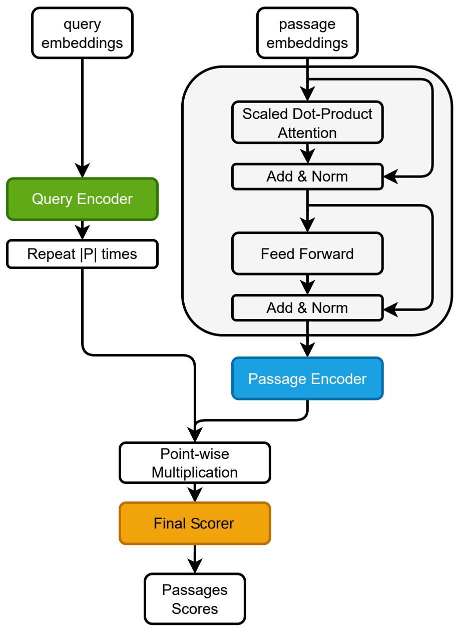

Figure 2: Model Architecture

relu activation helps to introduce nonlinearity and prevent the vanishing gradient problem during training. The final output shape of 512 is chosen to strike a balance between the expressiveness of the representation and the efficiency of the model, and can be further used in downstream tasks.

Passage Encoder The passage encoder is built on a CNN architecture that includes 3 convolutional layers, as illustrated in Fig. 3. The filters in these layers progressively increase from 16 to 32 and ultimately to 64. Each convolutional layer is followed by batch normalization and LeakyReLU activations to enable effective extraction of complex hierarchical features from the input embeddings. To reduce overfitting and mitigate the dimensionality of feature maps, the first 2 convolutional layers apply max pooling with kernel size 2. In addition, the final convolutional layer's output is flattened and passed through a dense layer that holds 512 units, uses a Sigmoid activation function, and incorporates dropout regularization at 0.1. The CNN model architecture of the passage encoder is optimized for efficiently processing embeddings whilst minimizing the risk of overfitting.

Final Scorer The scorer is constructed using two simple linear layers, each followed by a activation: LeakyReLU and sigmoid, respectively. The reason for this simple design is based on the assumption that the query and passage encoders have already learned sufficient amount of features, and are effective at detecting relevant content. Therefore, the scorer needs only to effectively weight and combine the features provided by the encoders to produce score.

5.2.2 *Design Choice.*

Separate Query and Passage Encoder. Using separate encoders for queries and passages is a common practice in information retrieval because queries and passages have different characteristics [7, 10]. Queries are often shorter and more straightforward, containing precise information and requiring shallower network for understanding. Furthermore, since queries and passages are subsequently combined via multiplication after encoding, the query encoder primarily functions to locate the relevant features while assigning values to prioritize relevant and de-emphasize irrelevant feature. This method also provides additional flexibility in adjusting the architecture and hyperparameters of each encoder independently to optimize the model's performance.

Transformer Implementation. The Transformer architecture is effective for pre-processing embeddings [14] due to its self-attention mechanism, which enables embeddings to capture contextual information and relative importance of other embeddings, avoiding local dependency issues and allowing the model to consider global structure. This helps the model focus on important parts of the input and integrate multi-faceted information from different passages. This, in turn, leads to improved representations of the passages, making them more informative and easier to process.This is also a temp to circumvent the unbalanced dataset problem, since the negative samples can now provide extra information for the passages collection as a whole.

Overfitting Prevention. Multiple approaches have been adopted to prevent overfittng:

Dropout regularization: The dense layer in the passage encoder incorporates dropout regularization at a rate of 0.1, meaning that 10% of the connections between neurons in that layer are randomly dropped out. This helps to prevent the model from relying too heavily on any one particular feature.

Batch normalization: Each convolutional layer in the passage encoder is followed by batch normalization, which helps to stabilize and standardize the output of each layer. This can prevent the model from becoming too sensitive to small variations in the input data.

Parameter regularization: The model uses NAdam optimizer incorporated with a L2 regularization to penalize overly complex or large weight values.

Early stopping: During training, the model's performance is monitored on a validation set every few epochs according to the adjust input parameter `val_freq`, and training is stopped early if the validation loss stops improving. This can additionally save computational resource by identifying underperforming model architecture candidates.

5.3 Loss

The loss is defined in Eq. 5. In the above expression, $|Q|$ represents the number of queries, and $|P|$ represents the number of candidate passages per query. y_i is the true label (relevancy) of the i^{th} passage, and $\hat{y}_i$ is the predicted score of the i^{th} passage.

$$L_{rank} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \sum_{\substack{j=1 \\ y_j=0}}^{|P|} \sum_{\substack{k=1 \\ y_k=1}}^{|P|} [j \neq k] \cdot \max(0, \hat{y}_j - \hat{y}_k) \tag{5}$$

This loss function represents a migration from the pair-wise margin ranking loss provided by *pytorch* to a list-wise ranking approach. It was efficient to compute owning to the unbalance and binary nature of the dataset. The ranking loss offers a more relevant alternative to the binary cross-entropy (BCE) loss used in logistic

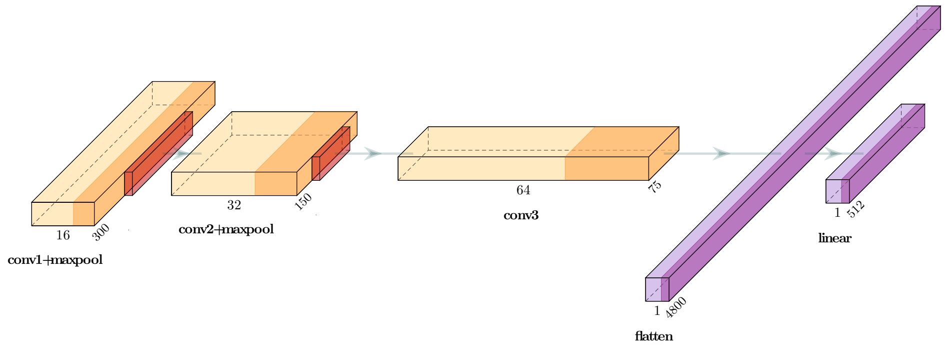

Figure 3: Passage Encoder Architecture

regression model for its key emphasis that lies in prioritizing the elevation of relevant samples to higher ranks rather than pushing all negative sample scores towards zero.

5.4 Result

Metric, At	3	10	100
mAP	0.00740	0.0108	0.0148
NDCG	0.00795	0.0151	0.0420

Table 7: Performance of Neural Network Model

REFERENCES

[1] Alan Akbik, Duncan Blythe, and Roland Vollgraf. 2018. Contextual String Embeddings for Sequence Labeling. In *COLING 2018, 27th International Conference on Computational Linguistics*. 1638–1649.
[2] Sanjeev Arora, Yingyu Liang, and Tengyu Ma. 2017. A Simple but Tough-to-Beat Baseline for Sentence Embeddings. In *International Conference on Learning Representations*. https://openreview.net/forum?id=SyK00v5xx
[3] Ignacio Arroyo-Fernández, Carlos-Francisco Méndez-Cruz, Gerardo Sierra, Juan-Manuel Torres-Moreno, and Grigori Sidorov. 2019. Unsupervised sentence representations as word information series: Revisiting TF–IDF. *Computer Speech Language* 56 (2019), 107–129. https://doi.org/10.1016/j.csl.2019.01.005
[4] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching Word Vectors with Subword Information. arXiv:1607.04606 [cs.CL]
[5] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching Word Vectors with Subword Information. arXiv:1607.04606 [cs.CL]
[6] Kawin Ethayarajh. 2018. Unsupervised Random Walk Sentence Embeddings: A Strong but Simple Baseline. In *Proceedings of the Third Workshop on Representation Learning for NLP*. Association for Computational Linguistics, Melbourne, Australia, 91–100. https://doi.org/10.18653/v1/W18-3012
[7] Jiafeng Guo, Yixing Fan, Qingyao Ai, and W. Bruce Croft. 2016. A Deep Relevance Matching Model for Ad-hoc Retrieval. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. ACM. https://doi.org/10.1145/2983323.2983769
[8] Bohan Li, Hao Zhou, Junxian He, Mingxuan Wang, Yiming Yang, and Lei Li. 2020. On the Sentence Embeddings from Pre-trained Language Models. arXiv:2011.05864 [cs.CL]
[9] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. arXiv:1301.3781 [cs.CL]
[10] Rodrigo Nogueira and Kyunghyun Cho. 2020. Passage Re-ranking with BERT. arXiv:1901.04085 [cs.IR]
[11] Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. GloVe: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Doha, Qatar, 1532–1543. https://doi.org/10.3115/v1/D14-1162
[12] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. arXiv:1802.05365 [cs.CL]
[13] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).
[14] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. arXiv:1706.03762 [cs.CL]
[15] Congcong Wang, Paul Nulty, and David Lillis. 2021. A Comparative Study on Word Embeddings in Deep Learning for Text Classification. In *Proceedings of the 4th International Conference on Natural Language Processing and Information Retrieval* (Seoul, Republic of Korea) *(NLPIR 2020)*. Association for Computing Machinery, New York, NY, USA, 37–46. https://doi.org/10.1145/3443279.3443304