Information Retrieval and Data Mining Coursework 2

ABSTRACT

This project aims to develop and improve the information retrieval model in CW1. Task 1 implements the average precision and NDCG metrics functions for retrieval quality evaluation. Task 2 computes queries and passages embeddings by word vectors and then uses the logistic regression model to assess relevance of a passage to a given query. To assess different model performances, Task 3 tries the LambdaMART model and Task 4 tries Neural Network model.

1 INTRODUCTION

1.1 Data

There are two datasets used in this project, which are *train_data.tsv* and *validation_data.tsv* respectively with a format of <qid pid queries passage relevancy>. *train_data.tsv* has 4364339 rows and *validation_data.tsv* has 1103039 rows.

1.2 Text processing

The text processing method used in this project is the same as the method used in coursework 1. First, all the letters in text will be converted to lowercase letters when reading the text. At the same time, long blanks and all non-English letters(e.g. numbers, Greek letters and punctuation) will be removed. Then the input text will be cut into tokens using the 'Regexp-Tokenizer' function and stop words will be deleted. Lastly, the lemmatisation step gives the base form of a word and 'SnowballStemmer' algorithm will be used to convert the derived word to a word stem.

2 TASKS

2.1 Task 1 - Evaluating Retrieval Quality

Implement methods to compute the average precision and NDCG metrics. Compute the performance of using BM25 as the retrieval model using these metrics.

BM25 Model
BM25 model(with parameters $k1 = 1.2$, $k2 = 100$, and $b = 0.75$) applied on the test dataset *validation_data.tsv* in this task is the same as the model used in coursework 1. After text processing, this task retrieved respectively at most 3, 10, 100 passages from within the 1000 candidate passages for each test query. The results are stored in dictionaries *BM25_score_*3(cutoff = 3), *BM25_score_*10(cutoff = 10) and *BM25_score_*100(cutoff = 100) with a format of $\{qid : \{pid : score\}\}$ for 1148 qids.

Average Precision (AP)
AP is the average of precisions at relevant documents measuring the accuracy of the retrieval model

$$AP = \frac{\sum_{k=1}^{n} P(k) \times rel(k)}{N}$$

where k is the rank in the retrieved passages, $P(k)$ is the precision at cutoff and N is the total number of retrieved relevant passages for this query. If a relevant document is not retrieved by the model, precision is consider as 0.

Mean Average Precision (mAP)
mAP is the average AP values over all queries in this system

$$mAP = \frac{\sum_{q=1}^{N_q} AP_q}{N_q}$$

where N_q is the total number of queries, which is 1148 in this task.

Discounted Cumulative Gain (DCG)
DCG is another metric to accumulate graded relevancy. Industrial DCG is defined as

$$DCG_q = \sum_{i=1}^{q} \frac{2^{rel_i} - 1}{log_2(i+1)}$$

where q is the total ranking length, rel_i is the relevancy for the passage ranking ith.

Normalized Discounted Cumulative Gain (NDCG)
NDCG is DCG against the optimal DCG (the perfect ranking) for one query, whose value is between 0 and 1.

$$NDCG_q = \frac{DCG_q}{optDCG_q}$$

Mean Normalized Discounted Cumulative Gain(mNDCG)
mNDCG is the average NDCG values over all queries in this system

$$mAP = \frac{\sum_{q=1}^{N_q} NDCG_q}{N_q}$$

where N_q is the total number of queries, which is 1148 in this task.

In this task, mAP and mNDCG will be used to evaluate the retrieval qualities for the BM25 systems with cutoff 3, 10, 100. The results is shown in Table 1 below.

Table 1: Metrics Evaluating BM25 Model

BM25	Cutoff @	mAP	mNDCG
BM25_score_3	3	0.1767	0.1979
BM25_score_10	10	0.2167	0.2768
BM25_score_100	100	0.2329	0.3441

2.2 Task 2 - Logistic Regression (LR)

Represent passages and query based on a word embedding method. Compute query (/passage) embeddings by averaging embeddings of all the words in that query (/passage). With these query and passage embeddings as input, implement a logistic regression model to assess relevance of a passage to a given query. Using the metrics you have implemented in the previous part, report the performance of your model based on the validation data. Analyze the effect of the learning rate on the model training loss.

Sub-sampling
I used negative down sampling to generate a sample of the training data, which helps speed up training and mitigate the influence of the class imbalance. As we see, the dataset is extremely unbalanced

with around 0.1% relevant passages and 99.9% irrelevant passages for most queries. He et al. [2] concluded negative down sampling helps to solve the class imbalance problem and the model achieves the best performance when the sampling rate is set to 0.025. Hence, 0.025 will be used as negative down sampling rate in this task. I do the negative down sampling by following steps:

(1) for each query, keep all of its relevant passages
(2) for each query, keep its irrelevant with a rate 0f 0.025

As the passages retrieved for a small percentage of queries were much smaller than 1000, to simplify this process I will randomly choose 25 irrelevant passages for the query who has more than 25 irrelevant passages since most queries have around 1000 corresponding passages. For the query who has less than 25 irrelevant passages, I will keep all of its irrelevant passages.
After sub-sampling *train_data.tsv*, I got a new training dataset *train_subdata* containing 118434 rows. Then the same text processing will be done on the queries and passages in *train_subdata*.

Word Embedding

I applied *Word2Vec* embedding with skip-gram algorithm from *gensim* on passages and queries for training data and test data. First, passages and queries are preprocessed and saved line by line in a txt file. Then *LineSentence* is applied to transform the original corpus into an iterator of sentences. Finally, *Word2Vec* is used to generate word vectors. The more dimensions the word vector has, the better the word representation is. Considering the computing power of the laptop, the length of word vector is set to 100. The window size is 5, the size of negative sampling is 5, minimal word count is 1. To get average embedding for passages and queries, I averaged word vectors of all tokens in each query and passage.

Logistic regression

The Logistic (sigmoid) function is

$$\sigma_w(x_i) = (1 + e^{-w^T x_i})^{-1}$$

and the binary cross-entropy loss function for n observations is

$$Loss(w) = -\frac{1}{n}\sum_{i=1}^{n}[y_i ln(\sigma_w(x_i)) + (1 - y_i) ln(1 - \sigma_w(x_i))]$$

where w is weight for regression, x_i is the ith input value and y_i is the ith input label. Then the gradient becomes

$$\frac{\partial Loss(w)}{\partial w_j} = -\frac{1}{n}\sum_{i=1}^{n}[x_{i,j}(y_i - \sigma_w(x_i)]$$

I used mini-batch stochastic gradient descent in this task and stop training until the gradient convergences. The stopping criterion is

$$\|gradient\| < tolerance$$

The inputs of the logistic regression in this task are the average embedding vectors for each query (q_{vec}) and its corresponding passages (p_{vec}) in *train_subdata*. Then the formulation for this logistic regression is

$$\begin{pmatrix} w_0 & w_1 & w_2 \end{pmatrix} \begin{pmatrix} 1 \\ q_{vec} \\ p_v ec \end{pmatrix} = w_0 + w_1 \times q_{vec} + w_2 \times p_{vec}$$

where q_vec and p_vec have the same length 100. Then 1, q_{vec} and p_{vec} are stacked to a longer vector V with length 201 and w_0, w_1 and w_2 are stacked to a longer vector W with the same length 201. Then the input x_i is the stacked word vector V for ith row in *train_subdata* and the label y_i is the relevancy for ith row in *train_subdata*. The initial input weight W chosen is a zero vector with length 201 and the max iteration is 10000. Then the model is trained using mini-batch stochastic gradient descent with a batch size of 10, a learning rate of 0.01 and a tolerance of 0.01. Then the weight W calculated will be tested on *validation_data.tsv* and the model achieves the top 100 passages with highest predicted scores for each query, storing in *LR.txt* with a format of <qid A2 pid ranking score LR> for each line. The retrieval quality of this LR system will be evaluated by mAP and mNDCG. As shown in Table 2, this LR system gives inaccurate results on validation dataset. There are few possible reasons leading to this bad performance of LR and LM models in task 2 and 3:

(1) Iterations are not enough in LR model. The gradient did not converge within 10000 iterations
(2) Word embeddings I used are not accurate. I did not use any pre-trained model when embedding. The corpus in queries are not large enough since lots of queries are repeated
(3) The dimension of word vectors is not large enough. The word vector with length 300 might achieve better performance
(4) The window size for embedding is not suitable. Since queries after processing are converted to very short sentence containing three or four tokens, the window size of 5 is not reasonable.

Table 2: Metrics Evaluating LR Model

LR	Cutoff @	mAP	mNDCG
LR	100	0.0040	0.0206

Training Loss and Learning Rate

I trained the same model with different learning rates to investigate the effect of learning rates on the model training loss. I tries learning rates of 0.01, 0.005, 0.001, 0.0005, 0.0001, and plotted their losses during each training process within 500 and 5000 iterations respectively in Figure 1 and Figure 2 below.

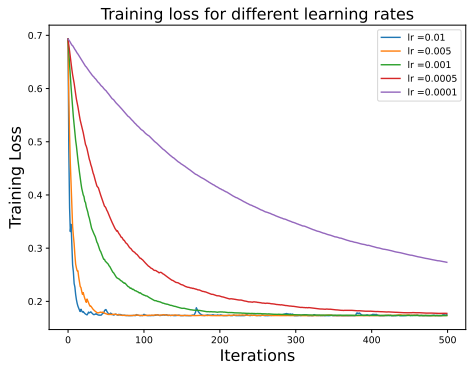

Figure 1: Training Loss within 500 Iterations with various learning rates

Figure 1 shows a bigger learning rate leads to a quicker convergence of training loss. The training loss decrease most rapidly with a learning rate of 0.01 in these learning rates.

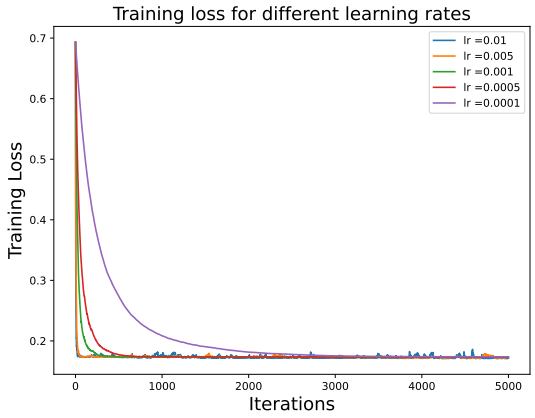

Figure 2: Training Loss within 5000 Iterations with various learning rates

Figure 2 shows that the training loss with all learning rates decrease rapidly and converge eventually with enough iterations though they have different convergence speeds.

2.3 Task 3 - LambdaMART Model (LM)

Use the LambdaMART learning to rank algorithm (a variant of LambdaRank we have learned in the class) from XGBoost gradient boosting library to learn a model that can re-rank passages. You can command XGBoost to use LambdaMART algorithm for ranking by setting the appropriate value to the objective parameter as described in the documentation. You are expected to carry out hyper-parameter tuning in this task and describe the methodology used in deriving the best performing model. Using the metrics you have implemented in the first part, report the performance of your model on the validation data. Describe how you perform input processing, as well the representation/features used as input.

LambdaMART

LambdaMART is a combination of MART and LambdaRank. MART represents Multiple Additive Regression Tree, which is also known as Gradient Boosting Decision Tree (GBDT). MART performs gradient descent using regression trees, which is a boosting algorithm. LambdaRank specifies the desired gradients directly instead of calculating them from a cost. LambdaMART, a match of MART and LambdaRank, achieves great performance in ranking problems in practical applications [1].

Training

The training input is still the stacked embedding of the query and its passage, which has a length of 200. The training label is the relevancy 0 or 1. Then I grouped the training dataset by qids and used *XGBRanker* function from *xgboost* library.

Hyper-parameter Tuning

booster decides the type of learner used. I used the the default gbtree (i.e. CART decision tree) in this task. *objective* specifies the learning task and I chose objective $'rank : pairwise'$, which uses LambdaMART to perform pairwise ranking by minimizing the pairwise loss. *eta* is the step size shrinkage used in update the feature weights, also known as learning rate. I evaluated models with *eta* of 0.01, 0.005 and 0.001. *n_estimators* is related to the complexity of the XGBoost model, representing the number of weak learners in the decision tree. I evaluated models with *n_estimators* of 100, 200 and 300. *max_depth* limits the depth of trees. I took *max_depth* value from 5, 6 and 7. Then I used grid search to find the optimal model and its hyper-parameters. Finally, I got results for all models in Table 3.

Table 3: Metrics Evaluating LM Models with Cutoff 100

Model	eta	n_est	depth	mAP	mNDCG
1	0.01	100	5	0.0043	0.0192
2	0.01	100	6	0.0054	0.0217
3	0.01	100	7	0.0067	0.0240
4	0.01	200	5	0.0045	0.0187
5	0.01	200	6	0.0044	0.0202
6	0.01	200	7	0.0042	0.0210
7	0.01	300	5	0.0048	0.0185
8	0.01	300	6	0.0040	0.0198
9	0.01	300	7	0.0037	0.0196
10	0.005	100	5	0.0051	0.0202
11	0.005	100	6	0.0082	0.0240
12	0.005	100	7	0.0083	0.0251
13	0.005	200	5	0.0039	0.0184
14	0.005	200	6	0.0058	0.0229
15	0.005	200	7	0.0067	0.0240
16	0.005	300	5	0.0035	0.0165
17	0.005	300	6	0.0047	0.0225
18	0.005	300	7	0.0040	0.0207
19	0.001	100	5	0.0039	0.0176
20	0.001	100	6	0.0062	0.0215
21	0.001	100	7	0.0056	0.0234
22	0.001	200	5	0.0045	0.0197
23	0.001	200	6	0.0070	0.0232
24	0.001	200	7	0.0067	0.0247
25	0.001	300	5	0.0051	0.0208
26	0.001	300	6	0.0076	0.0234
27	0.001	300	7	0.0074	0.0250

eta represents the learning rate, *n_est* represents number of estimators and *depth* represents the maximum depth of a tree.

In Table 3, model 12 with learning rate of 0.005, 100 estimators and a maximum tree depth of 7 achieves the best mAP and mNDCG. Hence, model 12 is the best performing model though all these models did a bad job in this task.

2.4 Task 4 - Neural Network Model (NN)

Using the same training data representation from the previous question, build a neural network based model that can re-rank passages. Justify your choice by describing why you chose a particular architecture and how it fits to our problem. Using the metrics you have implemented in the first part, report the performance of your model

on the validation data. Describe how you perform input processing, as well as the representation/features used.

RNN

A recurrent neural networks (RNN) uses sequential or time-series data and feeds the output from previous step as input to the current stage. Recurrent networks learn from training input, but unlike convolutional neural network (CNN), RNN has 'memory', which influences current input and output by using prior information [5].

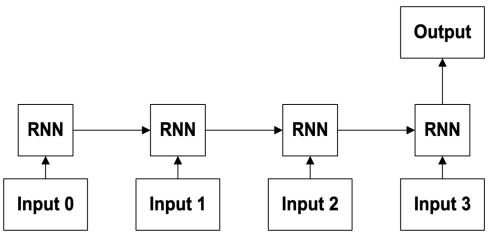

Figure 3: RNN architecture for sentiment analysis

As shown in Figure 3, each calculation will consider the state of the previous step. RNN calls the previous 'memory' over to analyse the new 'memory', and if analysing more data, it would accumulate all the previous memories[5]. RNNs can be used as long as the data is sequential, such as human speech, telephone numbers, image pixels or text. Hence, RNN is suitable for this task with the given datasets.

LSTM-RNN

Standard RNNs have the problem of vanishing gradients, making learning long data sequences challenging. Long Short-Term Memory (LSTM) is introduced to deal with this issue. Even with noisy, incompressible input sequences, LSTM learns to bridge temporal spans in excess of 1000 steps without sacrificing small time lag capabilities [3]. LSTMs use a gating mechanism that controls the memorizing process. There are three different gates in an LSTM cell: a forget gate, an input gate, and an output gate (shown in Figure 4) [3].

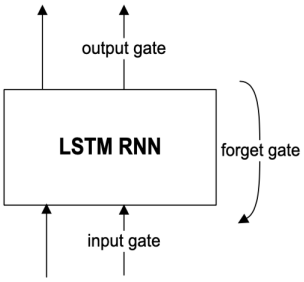

Figure 4: LSTM cell architecture

As discussed above, RNN-LSTM is a suitable architecture for this task. Compared with CNN, RNN, as a time-series related network structure, captures relationships between tokens in each query and passage. LSTM captures long term memory and achieves increasing information as going through a sentence, which is significantly useful for document retrieval tasks[4, 6].

Input processing

For LSTM-RNN, the input data should be processed in a different way from the method in Task 2 and 3. Since I trained my own word vectors among quries and passages from *validation_data.tsv* and *train_data.tsv*, word vectors I calculated are not consistent in different datasets. It would be easier to have the same word vectors for each word in different datasets. So in this task I used a pre-trained word embeddings, Stanford's GloVe 100d word embeddings [1], to create a consistent look-up table. This table contains two dictionaries, respectively *token_ind_dict* and *ind_vec_dict*. *token_ind_dict* gives each word in the corpus of our datasets a unique index and *ind_vec_dict* connect the token's index to the token's word vector. After data possessing, the size of corpus for our datasets is 150119, the max length of passage is 137 and the length of each word vector is 100. Then I transferred tokens in queries and passages to its corresponding index, which are training inputs and test inputs. For the sentence with length smaller than the max length, I add zeros at the remaining space. The training labels and test labels are still the relevancy column in original datasets.

[1] download from https://www.kaggle.com/datasets/danielwillgeorge/glove6b100dtxt

REFERENCES

[1] Christopher JC Burges. 2010. From ranknet to lambdarank to lambdamart: An overview. *Learning* 11, 23-581 (2010), 81.

[2] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, et al. 2014. Practical lessons from predicting clicks on ads at facebook. In *Proceedings of the eighth international workshop on data mining for online advertising*. 1–9.

[3] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[4] Hamid Palangi, Li Deng, Yelong Shen, Jianfeng Gao, Xiaodong He, Jianshu Chen, Xinying Song, and Rabab Ward. 2016. Deep sentence embedding using long short-term memory networks: Analysis and application to information retrieval. *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 24, 4 (2016), 694–707.

[5] Iqbal H Sarker. 2021. Deep learning: a comprehensive overview on techniques, taxonomy, applications and research directions. *SN Computer Science* 2, 6 (2021), 1–20.

[6] Mohamed Trabelsi, Zhiyu Chen, Brian D Davison, and Jeff Heflin. 2021. Neural ranking models for document retrieval. *Information Retrieval Journal* 24, 6 (2021), 400–444.