IRDM Course Project Part II Report

Ziwei Lin (19035341)

ABSTRACT

After the Implementation of Inverted Index and re-rank the passages using the Vector Space and BM25 models in part I, this part II assignment realises the evaluation of ranking results using average precision and NDCG, and applies LTR (Learning to Rank) models to re-rank. With the development of Logistic Regression Model, LambdaMART Model and Neural Network Model, train them on a very large "train_data.tsv", test them by re-rank "validation_data.tsv" which contains 1148 queries with about 1000 passages each and evaluate the results.

- File structure:

```
SubmitFolder
   ├── IRDM Course Project Part 2 Report.pdf
   ├── Code
   |     ├── part1_code_for_BM25
   |     |           ├── Passages (For the output of extracted passages)
   |     |           ├── ExtractPassages.py
   |     |           ├── generateBM25.py
   |     |           └──BM25.txt
   |     ├── all_queries.tsv
   |     ├── Metrics.py
   |     ├── ModelsEvaluation.py
   |     ├── train_data_line_extract.py
   |     ├── train_data_line_1.txt
   |     ├── validList.txt (Number of valid(1.0) items per 10000 lines)
   |     ├── Embed.py
   |     ├── embed10000.tsv (Embedding of 2310000 - 2320000 train_data)
   |     ├── LR
   |     |    ├── LogisticRegression.py
   |     |    ├── generateLR.py
   |     |    ├── LRWeight_10000.txt
   |     |    ├── LR001.txt
   |     |    ├── LR002.txt (LR.txt)
   |     |    └── LR003.txt
   |     ├── LM
   |     |    ├── LambdaMART.py
   |     |    ├── generateLM.py
   |     |    ├── xgb1.json
   |     |    ├── xgb2.json
   |     |    ├── xgb3json
   |     |    ├── xgb4json
   |     |    ├── LM-1.txt
   |     |    ├── LM-2.txt (LM.txt)
   |     |    ├── LM-3.txt
   |     |    └── LM-4.txt
   |     └── NN
   |          ├── NNmodel.py
   |          ├── genNN.py
   |          ├── nn.pkl
   |          └── NN.txt
   ├── part2 (Too large to be submitted, so it's empty)
   |     ├── train_data.tsv
   |     └── validation_data.tsv
   ├── Test Results
   |          ├── LR.txt
   |          ├── LM.txt
   |          └── NN.txt
   └── EvaluationResults.txt
```

- Note1: Since the dataset files in folder "part2" is too large to be uploaded together to the Moodle (160MB at most), they are removed. Please move the two files into "part2" folder before the running of all the code.
- Note2: The implementation code in this Project Part II assignment is written in Python 3.7. Please make sure you have installed the following libraries before running the code: PyTorch, XGBoost, "spacy", "matplotlib.pyplot", "pandas", "numpy" and "operator.itemgetter".

1 GOAL AND PROCESS

This section describes the goal of the part2 and the general idea of the implementation process.

In subtask one, the goal is to implement the evaluation of the ranking using average precision and NDCG as two metrics, and use an existing (BM25) model ranking to test it. The process is first define the metrics calculation for each query in Metrics.py, then define the calculation for a whole ranking by iterating through all queries and obtain the mean value in ModelsEvaluation.py, last use part I method to generate BM25 ranking on the new dataset, and test the result in ModelsEvaluation.py.

For subtasks two to four, the goal is to generate three Learn to Rank Models and use the trained model to calculate the relevancy of a given query - passage pair so that the passage can be re-ranked, and the ranking result can be evaluated. The process is first create the corresponding model, train them on a subset of "train_data.tsv" (the working machine's memory does not allow the whole dataset) in [SOMEMODEL.py] file by inputting a 600 x data length matrix and analysing the loss on the data length x 1 relevancy matrix, then generate the model's ranking [SOMEMODEL.txt] on the "validation_data.tsv" by using query-passage embedding vector (1 x 600) as input, obtain the output as relevancy score, sort pid by this score and write the ranking to the final file. The ranking results are all evaluated in ModelsEvaluation.py.

2 RETRIEVAL QUALITY EVALUATION

2.1 Preparation

Since the "validation_data.tsv" is different from the "candidate_passages_top1000.tsv" used in part I, the BM25 ranking has to be re-generated for the new dataset. The code in **"part1_code_for_BM25"** folder slightly improves the implementation in part I (not writes the inverted index to files). By first running **ExtractPassages.py**, the passages for each query will be extracted to the "Passages" folder and the **"all_queries.tsv"** (queries are sorted by qid in ascending order) from the validation_data is also output to the parent folder to benefit further process. Then, **generateBM25.py** combines inverted index creation, BM25 score calculation and **"BM25.txt"** output together without redundant operations.

The code for implementing the two metrics - Average Precision and NDCG is inside **/Code/Metrics.py**. It defines a class **EvaluateMetrics** which contains five fields:

model - the DataFrame [qid, A1, pid, rank, score, algoName] read from the ranking file "SomeModel.txt". This is given during initialisation;
qid - the qid of current query whose metrics are calculated. This is also given during initialisation;
ModelRank - a list of all the pid of the passages under the current query. This is extracted by selecting from the query_candidate DataFrame;
query_candidate - all data from "validation_data.tsv";
bestRank - a list of some ordered pids that has the largest DCG (IDCG). This is obtained by merge ModelRank and the relevancy in query_ candidate together, and sort the pids in ModelRank by relevancy in ascending order.

2.2 Average Precision

The **get_avePrecisionByQid(self)** function obtains the average precision of the current query's passage ranking. The whole ranking is scanned through with a variable "sum" kept, and once an entry whose relevancy is equal to 1.0 is encountered, the **precision = current number of relevant passages / current number of retrieved passages** is added to the "sum". The final average precision of the current query is **sum / total number of retrieved passages**.

2.3 NDCG

The normalised discounted cumulative gain - NDCG is range from 0 to 1, which is useful to measure how a ranking result can place relevant passages on the top of the ranking, say, make the relevant passage useful. NDCG can be calculated by:

$$NDCG@k = \frac{DCG@k}{IDCG@k} \tag{1}$$

where the DCG is:

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

IDCG is the ideal DCG, which is the DCG when all the most relevant passages are ranked at the top, as the **bestRank** has done.
Since the relevant passages are few in the dataset, and calculate all enries of the ranking does not help too much in reality, k = 100 is applied in this assignment. **get_dcgByQid(self, ranking)** implements DCG@100 calculation by iterating through the ranking list until rank 100, and adding the DCG of the entries whose relevancy is 1.0. While **get_ndcg(self)** becomes get_dcgByQid(ModelRank) / get_dcgByQid(bestRank).

2.4 Evaluation of BM25

With the import of EvaluateMetrics, **ModelsEvaluation.py** handles the evaluation of all the model's ranking results. The core function **getMetrics(rank)** first loads queries in "all_queries.tsv", prepares a list of average precision of each query: (qid, average precision) - **avePrecisionList**, the sum of average precision - **avePreSum**, a list of NDCG: (qid, NDCG@100) - **ndcgList**, the sum of NDCG - **ndcgSum**. The parameter rank is the DataFrame reads from any model rank file in [qid, A1, pid, rank, score, algoName] format. A EvaluationMetrics instance will be created each time the function iterated an item in the all_queries. The average precision list and the NDCG list will be printed out and the mean average precision and mean NDCG will be returned.
Outside the function definition above, for BM25.txt evaluation, the file is first loaded and set as the parameter of the getMetrics function, and the two mean values will be printed separately as the below result shows:

```
BM25:
Mean average precision for BM25 = 0.055
Mean NDCG@100 for BM25 = 0.117
```

3 LOGISTIC REGRESSION (LR)

The "train_data.tsv" has 4364340 lines of data (printed by "train_data _line_extract.py"), which is far too large for the working machine, as MemoryError will pop up when tried to use 100,000 lines of data. With this consideration and the time consumed, 10,000 lines of train data is decide to use for all the later model training. To be specific, line 2310000 to line 2320000 is chosen as it contains relatively more valid data - where relevancy is 1.0, so it could be good for model training. "train_data _line_extract.py" also generates a list of (endLine, number of valid entries) and writes it to the file validList.txt, the endLine is increasing by 10000 each entry to represent the number of valid entries per 10000 lines of train_data. This is useful to select subdata for training, and 2310000 to 2320000 has 87 rows of valid data.

3.1 Preparation - Word Embedding

Spacy is a powerful Python library for NLP, its pretrained model for word embedding is "en_core_web_md", which contains 20k 300-dimensional word vectors with GloVe trained on Common Crawl[1], is used in this part.
The query's and passage's word embedding expression is represented as the mean word embedding (300 feature values each), and since the result obtained from the Learning to Rank models are scores that measures the relevancy of each query-passage pair, the input should be a concatenation of the two 1 x 300 vectors. Therefore, a 1 x 600 vector is used to represent a query-passage pair as one entry of data, and the validation data becomes a 1103040 x 600 size matrix, and the used training data is a 10000 x 600 size matrix. At the beginning, the word embedding is defined in each model creation file, and the embedding is done every time a model creation code is running; however, it will cost too much time when different model is generated and tested for many times. Later, the 10000 lines train data word embedding is extracted in **Embed.py**, and written to **"embed10000.tsv"**, and the training data input can be directly load from the "TSV" file later.

3.2 Model Implementation and Training

The creation and training of the Logistic Regression Model is inside **"LogisticRegression.py"** file. It defines the model in **lrModel(X, Y, learning_rate)** function by taking in the training data X - a 10000 x 600 matrix containing 10000 pieces of embedding query-passage data, Y - a 10000 x 1 matrix containing 10000 relevancy scores, learning_rate - the learning rate when train the model (this makes the setting of different learning rates easily for latter comparison).

Inside the function, it sets the iteration number to 100 (this number is decided after several times of trying), initialises the weight matrix Theta to a 600 x 1 matrix of zeros, and prepares a Series J to store the average loss of the whole dataset each iteration for plotting the loss function image. During each iteration, as the Logistic Regression method defined, it uses the Sigmoid function:

$$h(X \cdot Theta) = \frac{1}{1 + e^{-(X \cdot Theta)}} \tag{3}$$

to calculate the predicted Y, denoted as h, and use the cost function:

$$J[i] = \Sigma \frac{(-Yln(h) - (1 - Y)ln(1 - h))}{length(data)} \tag{4}$$

to calculate the mean loss of the i_{th} iteration. Then, the gradient will be:

$$gradient = X.T \cdot (h - Y) \tag{5}$$

and the next Theta is updated to:

$$Theta = Theta - learning\_rate * gradient \tag{6}$$

and continue the next iteration to gradually adjust the weight Theta to reduce the loss and fit the prediction to the actual result Y. The training of the model is by loading X data from the previous "embed10000.tsv" file and Y data from the validation_data, set the learning rate (the best learning rate is 0.002 here), the above function will returns the Series J and weight Theta for future process (the weight is written and saved to **"LRWeight_10000.txt"**).

3.3 Ranking by the Trained Model

After the finish of the model training, the **"generateLR.py"** file handles the prediction of relevancy score on the validation data and the output of **"LR.txt"** ranking result. It first reloads the 600 x 1 weight matrix from "LRWeight_10000.txt" as a parameter used in **getLRScore** function, then iterates through all_queries, extracts the passages of the current query out of validation_data, iterates through these candidate passages and generates the query-passage embedding vector as the input of getLRScore function to obtain the relevancy score and store the (qid, pid, score) in a list. Next, outside the passage iteration, it sorts the score list by the score, and appends the ranking to "LR.txt" per query until the end of the all_queries iteration.

3.4 Ranking Quality Evaluation & Comparisons among Models with Different Learning Rates

The LR.txt generated above is then evaluated using the two metrics defined in the subtask one. Inside the ModelEvaluation.py a getLRMetrics(lr) function is defined, where lr is the DataFrame loaded from this LR.txt, and the results are obtained by calling getMetrics(lr).

Back to the LogisticRegression.py file, the training of the model will result in a Series J containing the average loss of each training iteration. This is useful for plotting the image of the cost function to see how the loss is change through training steps. The lrModel() function also leaves a learning rate to be set during model initialisation is also for the comparison of the effect on the model when different learning rate is applied. Five learning rates - 0.005, 0.003, 0.002, 0.001, 0.0005 - are tried for analysing the learning rate's effect on the cost function. The result is shown in figure 1 with the following findings:

- The training of the LR Model can be considered finished after about 57 iterations.
- The cost function value keeps decreasing when learning rate equals to or smaller than 0.001.
- The cost function value first jumps to a high point p and then decreases linearly in a slope s. The height of p and the slope s is has positive correlation with the learning rate, when the learning rate is bigger than 0.001.
- After the "finish of training", the cost function still fluctuates violently when learning rate is 0.005, and fluctuates slightly when learning rate is 0.003, but keeps stable for learning rate 0.002, 0.001 and 0.0005.

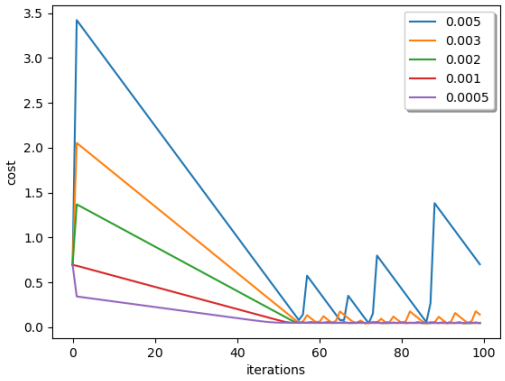

Figure 1: The Training Loss for LR Model with Different Learning Rates

The generation of the ranking results using different models (this means models with different learning rate) with different weight matrix is very slow, as the embedding of the query-passage needs to be recreated each time.**[Note: the embedding of validation data is not written to a separate file, because the validation data is too large and writing extra file takes out space and seems not help too much every time reload it.]** Only three models with learning rate 0.001, 0.002 and 0.003 is chosen, and LR001.txt, LR002.txt, LR003.txt are generated for evaluation. The results are listed below:

```
LR:
learning rate: 0.001
Mean average precision for LR = 0.014
Mean NDCG@100 for LR = 0.041

learning rate: 0.002
Mean average precision for LR = 0.015
Mean NDCG@100 for LR = 0.043

learning rate: 0.003
Mean average precision for LR = 0.014
Mean NDCG@100 for LR = 0.04
```

It can be judged that the model with learning rate = 0.002 has the best perform among this three learning rates. The perform of the LR model here is not good generally, even worse than the BM25 model. The reasons could be that the 10000 lines of training data is not enough for LR model to best obtain the features of query and passages; or during the training period, the prediction should be convert to 0/1 but not the actual float calculated by the Sigmoid function. The first problem could be improved using more rows of data inside the working machines permission, or change the 10000 lines to line 2790000 to 2800000 which has 110 entries of valid data. The second problem could be solved by further developing in the future.

4 LAMBDAMART MODEL (LM)

This section describes the development of the LambdaMART Model for page ranking using the XGBoost gradient boosting library. This can be install by running "pip3 install xgboost" [4], since no GPU is needed in this project, the binary wheel can be ignored. The support of xgboost in the code requires the import of xgboost (as xgb).

4.1 Model Implementation and Training

The **LambdaMART.py** defines and trains the LambdaMART Model. It prepares training data same as what LogisticRegression.py has done - by reload the "embed10000.tsv" and "train_data.tsv" for obtaining the X matrix and Y matrix respectively. Then xgboost use DMatrix to combine the X_train and Y_train together, as one train_data to be taken in for xgboost training. Next, the core work is to set the parameter for the model. Since LambdaMART model is required, just set the 'objective' value to 'rank:ndcg' will inform XGBoost to use LambdaMART listwise ranking by maximising NDCG[5]. To investigate the effects of parameter tuning, four models with different parameters are generated for test. The details of parameters setting will be shown in the 4.3 Ranking Quality Evaluation section. As the final LM.txt is based on xgb2, it is used as the example for the process description. The parameter for xgb2 is same as the default setting given by XGBoost, with only eta = 0.03 is different. This eta is trying to make the model learn with a smaller learning rate to see if the model can be better. Then, xgb2 can be defined and trained as xgb2.train(parameters, train_data, num_boost_round = 60), and xgb2.save_model("fileName") will save the model in json format.

4.2 Ranking by the Trained Model

The file **generateLM.py** implements using the model (xgb2 here) to predict the score of each query - passage pair and rank them each query and write the result to "LM.txt". Same as testing the LR model, generateLM.py also loads data from validation_data.tsv and iterates through queries in all_queries.tsv, generates the query-passage embedding vectors and combines all vectors into one input matrix X_test with the size of (number of passages in current query) x 600, and converts the X_test to a DMatrix. Next, it loads the model xgb2 by xgb.Booster().load_model("xgb2.json"), predict the score by xgb2.predict(X_test) which returns a one dimensional list of score. Then, after adding the (qid, pid, prediction_score) into another list and sort it, the final ranking result can be write to the LM.txt file.

4.3 Ranking Quality Evaluation

The LM.txt generated above is then evaluated using the average precision and NDCG@100 inside the ModelEvaluation.py by a getLMMetrics(lm) function, where lm is the DataFrame loaded from this LM.txt, and the results are obtained by calling getMetrics(lm).
To seek improvement of the performance of the model, four models xgb1.json, xgb2.json, xgb3.json, xgb4.json trained with different parameter tuning and result in different ranking LM-1.txt, LM-2.txt, LM-3.txt, LM-4.txt The output of the evaluation result is shown below:

```
[xgb1.json, LM-1.txt]
params1 = {'max_depth': 6, 'min_child_
weight': 1, 'gamma': 0, 'eta': 0.3,
'objective':'rank:ndcg'}
xgb.train(params1, train_data,
num_boost_round=6)
Mean average precision for LM = 0.012
Mean NDCG@100 for LM = 0.026
-----------------------------------------
[xgb2.json, LM-2.txt]
params2 = {'max_depth': 6, 'min_child_
weight': 1, 'gamma': 0, 'eta': 0.03,
'objective':'rank:ndcg'}
xgb.train(params2, train_data,
num_boost_round=60)
Mean average precision for LM = 0.013
Mean NDCG@100 for LM = 0.031
-----------------------------------------
[xgb3.json, LM-3.txt]
params3 = {'max_depth': 6, 'min_child_
weight': 1, 'gamma': 0, 'eta': 0.003,
'objective':'rank:ndcg'}
xgb.train(params3, train_data,
num_boost_round=600)
Mean average precision for LM = 0.013
Mean NDCG@100 for LM = 0.03
-----------------------------------------
[xgb4.json, LM-4.txt]
params4 = {'max_depth': 6, 'min_child_
weight':1, 'gamma': 0, 'eta': 0.3,
'subsample': 0.7, 'colsample_bytree': 0.7,
'objective':'rank:ndcg'}
xgb.train(params3, train_data,
num_boost_round=10)
Mean average precision for LM = 0.01
Mean NDCG@100 for LM = 0.026
```

Among this four rankings, the best one is LM-2.txt which use the default max_depth = 6, min_child_weight = 1, gamma = 0, and the eta, which is the learning rate, changes from 0.3 to 0.03 has the mean average presicion 0.013 and mean NDCG@100 0.031. The general perform is even worse than the LR model. It is suspicious that the reason could be the training data is far more different from the validation data, so more training data and more powerful working machine is needed.

5 NEURAL NETWORK MODEL (NN)

This section describes the development of a Neural Network Model for page ranking using the PyTorch machine learning framework. The installation command can be found in PyTorch's website as Figure 2 shows: With the installation success, just import torch will

Figure 2: The Installation of Stable PyTorch for Python on Windows OS using pip and without GPU [2]

enable all the functions supported by PyTorch in the file.

5.1 Model Implementation and Training

The **"NNmodel.py"** implements and trains the Neural Network Model. It first defines the Neural Network Model **NN** that extended from the torch.nn.Module class, in the __init__(self, input, hidden1, hidden2, output) function, the NN defines two hidden layer. Since the input will be the number of vector features which is 600, and the output is the number of classification types, which is 2 (0/1) in this case, NN should derive 2 neurons from 600 neurons. Therefore, the design is use **nn.Linear** function, the first layer converts 600 neurons to hidden1 neurons, and the second layer converts hidden1 neurons to hidden2 neurons, and the last output layer converts hidden2 neurons to 2 neurons. Next, the forword() function should also be overridden[3], the input x is processed by each layer and is activated using the ReLU function that can prevent the gradient vanishing problem every time it goes through a layer.

After the defining of the NN model, the training data is prepared as before (load X from "embed10000.tsv" and Y from "validation_data.tsv"), this time, X should be transferred to FloatTensor and Y becomes LongTensor. They are combined as a TensorDataset, which can be load by the DataLoader which further assigned batch_size attribute to 1 as the training data size is not too large, the model will train with full data directly. Next, the model is instantiated as **model = NN(600, 256, 64, 2)**, whose inner structure is shown in figure 3:

```
NN(
  (hidden1): Linear(in_features=600, out_features=256, bias=True)
  (hidden2): Linear(in_features=256, out_features=64, bias=True)
  (out): Linear(in_features=64, out_features=2, bias=True)
)
```

Figure 3: The inner structure of NN

The loss function criterion is also defined using torch.nn. CrossEntropyLoss() function.

Here comes the training phase: 1) Set the training epoch to 15 (after some trying), keep a Series J to store the average loss of each epoch, and define the optimizer using torch.optim.Adam() function, set the learning rate lr = 0.001. 2) Begin the epoch looping: prepare a list to store the current loss - Y_train_loss; start iterating through each row of data in the DataLoader. 3) Inside DataLoader enumeration, calculate the output of NN model.forward(X) - output, calcualte the current loss which is the criterion(output, Y), and append the current loss to Y_train_loss. Then, clear the last gradient, calculate the gradient using loss.backward() provided by PyTorch, and update the weight using optimizer.step(). 4) Outside the DataLoader enumeration, at the last of one training epoch, the average of Y_train_loss is assigned to J[epoch] and output this average loss of the current epoch as figure 4 shows. 5) After the 15 epochs training, the average loss is only 0.0176 decrease a lot from the original 0.066. The whole model is saved to file **nn.pkl**.

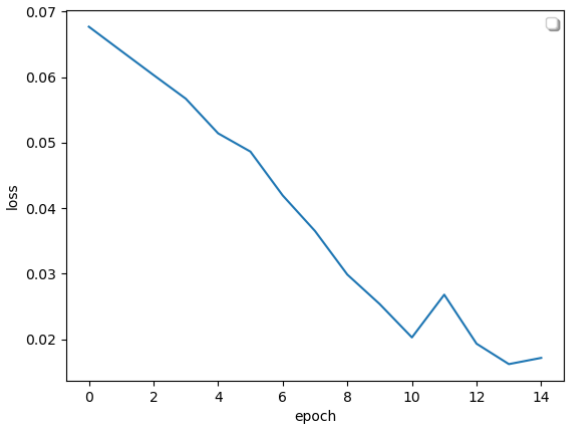

Figure 4: The Average Loss change by epoch

5.2 Ranking by the Trained Model

File **"genNN.py"** generates the ranking of the validation data using the NN model trained before and output the **"NN.txt"** ranking result. Since the whole NN model is saved to nn.pkl, to reload the model, the original definition of NN needs to be copy from NNmodel.py (otherwise, NN attributes cannot be found error will pop up), and use torch.load("nn.pkl") to finish reload.

Next, the preparation of test data is similar as before, by iterating through all_queries, generate the query-passage embedding vector, however, this time, it constructs the entire input matrix X for the current query, make the prediction by calculating model(X), which is 0/1 in this case, and insert the passages with score 1 to the top of the rank list and append the passages with score 0 to the end, and write to the file NN.txt.

5.3 Ranking Quality Evaluation

The NN.txt generated above is then evaluated using the average precision and NDCG@100 inside the ModelEvaluation.py by a getNNMetrics(nn) function, where nn is the DataFrame loaded from this NN.txt, and the results are obtained by calling getMetrics(nn). The output of the evaluation result is shown below:

```
Mean average precision for NN = 0.009
Mean NDCG@100 for NN = 0.025
```

The performance is even worse than the performance of the LM model. Although this time, the score can be output as 0 or 1, most of the scores are all 0, and these evaluation metrics result can then becomes random, since the final rank highly depends on the passages' original position, and the ranking can switch to anywhere with score 0. This is even not reliable.

6 CONCLUSION

In conclusion, tasks in this assignment part II provides good practice for using average precision and NDCG to evaluate the ranking result, develop Logistic Regression Model, LambdaMART Model (with XGBoost), Neural Network Model (with PyTorch) to calculate the relevancy score and re-rank the passages. For a student who is new to Python language and machine learning concepts, great efforts should be make to understand the concepts, learn to operate the dataset, try various learning parameters and accomplish these tasks.

To Emphasis, most code especially the code relevant to model test on the validation_data and write the TXT ranking files takes 4 to 5 hours and maybe longer to run. Therefore, models with different parameters cannot be tried much. Since the embedding of the validation data is generated together with testing and file writing. This drawback could be improved by writing the whole **1103040** items of query-passage embedding vectors to a file and read it when use them.

The general performance of the three models are not good, which may result from both the data aspect and the model aspect. Due to the limitation of the working machine's power, the selection of training data is essential, if the selected data is not good enough, the model will not learn the features need to be learnt. While the model design is complicated, to understand the parameters' meaning and find the best values for each parameter needs deep understanding of the model and rich experience, especially when using XGBoost and PyTorch.

REFERENCES

[1] Explosion AI - Spacy. 2020. Core Models English. https://spacy.io/models/en Accessed: 2020-4-14.

[2] Facebook Open Source. 2020. Quick Start Locally. https://pytorch.org/ Accessed: 2020-4-14.

[3] Facebook Open Source. 2020. torch.nn.Module. https://pytorch.org/docs/stable/nn.html?highlight=module#torch.nn.Module Accessed: 2020-4-14.

[4] xgboost developers. 2020. Installation Guide. https://xgboost.readthedocs.io/en/latest/build.html Accessed: 2020-4-14.

[5] xgboost developers. 2020. Parameter. https://xgboost.readthedocs.io/en/latest/parameter.html Accessed: 2020-4-14.