Passage Retrieval Report

ABSTRACT

In this Report, my implementation for the information Retrieval and Data Mining Coursework II (COMP0084) will be explained and the results obtained will be showcased.

During this report, I will highlight every subtask and explain how I tackled the subtask and give my justification.

Text files submitted were generates using CW1's test data not the validation data. However, the performance reported for the mentioned models were generated using the validation data.

I Subtask 1

1.1 The Question

Subtask 1 was related to the evaluation metrics of a model. It was asked to compute the average precision and the nDCG metrics on the validation data provided using CW1's BM25 algorithm. Since there is more than one query in the validation set. What will be calculated for this step is mean average precision (MAP) and the mean nDCG (MnDCG).

1.1.1Summary for tasks

- Implement and compute MAP@k=100.
- Implement and compute MnDCG.

1.2 Implementation

Before starting this subtask. We first take our validation data and apply all the pre-processing techniques used in CW1 (removing punctuation, removing stop words, and lemmatizing words). Then we re-rank the query/document pairs in the validations set using our BM25. For this part the **whole validation dataset** was used.

1.2.1 Implementing MAP

Since we have our re-ranked pairs. MAP was computed using the following two steps:

First, we take a query with its top **100** ranked documents, and iterate from top to bottom. Since the validation data has a new extra column which judges whether that document is relevant to that query or not (**1** or **0**). We use that to calculate the precision using the following equation:

$$precision = \frac{the\ rth\ relevant\ document\ to\ be\ found}{ranking\ in\ the\ re-rank\ list}$$

Meaning that, for example, if, for a given query, the top ranked document is relevant, then the precision is $\frac{1}{1} = 100\%$. However, if then the second top ranked document is not judged as relevant but the third one is relevant, the precision for that is then $\frac{2}{3} = 66\%$.

If we do that for all the top ranked documents for a given query and take an average, we then would have computed the average precision.

Second, after finding the average precision for all the validation queries, we take the average of these values to find the Mean Average Precision at k=100.

Result: MAP = **0.1488 = 14.88%.** Which could be considered as a good result considering there are very few "relevant" query/document pairs. (average 1.05 per query.

1.2.2 Implementing MnDCG

To find MnDCG, we first need to compute DCG, which can be calculated using the following equation:

$$\text{DCG@100} = \frac{\sum_{i=1}^{100} 2^{relevance_i} - 1}{log_2(i+1)}$$
$$where\ i\ is\ the\ ranked\ document$$

Meaning that for every document that **is relevant** for a given query we add its cumulative gain, where i here is also the ranking.

After finding DCG@100 for a given query, normalising needs to be done to get nDCG. One way to normalise is by using the following equation:

$$\text{nDCG@100} = \frac{DCG}{iDCG}$$

where iDCG is the ideal DCG. It was calculated in python in such a way where for each query, the number of positive relevant documents for that query are counted (n for example) and then calculating the DCG as if all these documents were ranked as the top n positions in the ranking.

After finding nDCG for each given query, we then average over all the queries to get MnDCG @ k=100.

Result: **MnDCG= 0.1961 = 19.61%.**

II Subtask 2

2.1 The Question

In this subtask, we were asked to represent the passages and the query based on a word embedding method. Then using these query and passage embeddings as input, we would implement a logistic regression model to assess the relevance of a document from 0 to 1.

2.1.1 Summary for tasks

- Apply word embedding method

- Implement Logistic Regression and analyze learning rate
- Report performance

2.2 Implementation

Since the training dataset was too large, it was noticed that using the whole training dataset on my computer was not time efficient, therefore I implemented a **negative sampling** method where that for each 10 negative samples (relevance is 0), I collect one positive sample (relevance is 1), which should take away from the hypothetical best accuracy of the model as there is less data (information) now. However, one positive out of the negative sampling method is that it reduces the chances for our model to be pulled into predicting everything as zero because that would result in the lowest training loss.

2.2.1 Apply word embedding method

The Word2Vec word embedding method was used for embedding our training data. The word2vec algorithm makes uses of a family of neural networks and optimization method to draw out associations of words, resulting in vectors representing each word available in the model's corpus. One of the algorithm's strong points is that the learned vectors are normally similar to synonym words, or directly related words. Meaning that for example the words "car" and "vehicle" would have similar embedding values (high cosine similarity), which is an asset in information retrieval.

The Word2Vec word embedding method from the library **genism** was used, the model chosen was the 'word2vec-google-news-300' model, which makes use of the google news web articles and results in learned vectors for words of size **300**.

The word embedding method was applied on the negative-sampled training data set for each query/document pair. As instructed by the assignment, all the embeddings for queries/documents were averaged out to give out a single **300** element vector for each query/document.

One important note to add here is that we use the **same** pre-processing techniques that was used in the first CW before finding the averaged-out query and document vector.
A version of these query/document vectors were used where the only pre-processing executed on the words was removing the punctuation but that resulted in worse results than the ones found in this report. As using stop words in that context would result in a lot of documents/queries being similar even though they are not.

2.2.2 Implement Logistic Regression and analyze learning rate

2.2.2.1 *Input used*

Using the word2vec embedding algorithm, the vector values for each word in a query/ or a document was averaged out to produce one vector for each query and one vector for each document. What was then done was that the absolute difference between each query-document vector pair was taken to result in one **300x1** vector for each query-document pair, which was then used as the **input** to our model.

2.2.2.2 *Logistic Regression*

As said, the input used for our logistic regression model was then a **52,615x300** matrix representing the **training features** (52,218 training samples containing 300 features), and **52,615x1** vector for the **training observations** (relevance either **0** or **1**).

The logistic regression model works in a simple way. First, we assume that our raw output can be written as the following linear:

$$g(x) = \theta^T x$$
$$where\ \theta \text{ is our weights for each feature (size 300)}$$

However, since we want our output only to be within the range of 0 and 1. We will use the sigmoid function for our raw output into generating an output "prediction" between 0 and 1, as the following:

$$Y\ Pred = h_\theta(x) = \frac{1}{1 + \exp\{-\theta^T x\}}$$

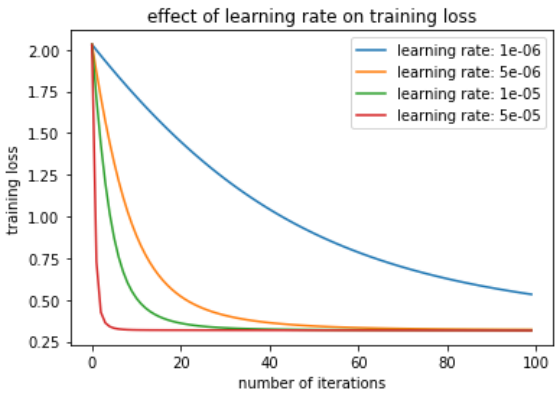
We then define a cost function, which measures how far our current predictions from the actual true observed output (relevance) as the following:

$$C(\theta) = \frac{-1}{m}[\sum_{i=1}^{m} y_i \log\big(h_\theta(x_i)\big) + (1 - y_i)\log(1 - h_\theta(x_i)]$$

Since the main objective here is to minimum cost function (gradient is 0) with respect to our chosen weights θ. The weights should then be updated with regards to the gradient of the cost function. Therefore, the weights should be updated the opposite direction of where the gradient of the cost function is pointing at. This is done in the hopes of finding the gradient with the least change, i.e. minimum. This is known as gradient descent, which is done in the implementation of the logistic regression. The summarized steps for the logistic regression can then be said to be as following:

- Calculate cost function for the current given weights.
- Calculate gradient of that cost function
- Update weights in the opposite direction of the sign of the gradient.
- Repeat until no significant change in weight (or a fixed number of iterations).

The change in the learning rate (the factor which determines for how much are the weights updating per gradient descent step) was analyzed by running 4 logistic regression model against different learning rates. The resulting plot of the training loss for a fixed number of iterations (100) using different learning rate is as follows:



As expected, it can be seen that using a relatively larger learning rate results in a faster convergence of the training loss, as "bigger" steps are taken per iteration.

The reason the learning rate range was chosen to be small is because our features are small values.

2.2.1.2 Report Performance

We then using the mentioned steps in **Subtask 1,** test our generated weights using the validation set and find the evaluation metrics:

Result: **MAP= 0.0095= 0.95%.**
Result: **MnDCG= 0.0306 = 3.06%.**

This is of course, not a good result. However, the model sometimes does in fact rank the true relevant document at the top (e.g. query number: 86 for this run of code).

One reason to explain these results is that the logistic regression model can be thought of to be a "simple" model, as it offers interpretability instead of accuracy. Another reason could be the small dataset used, but that is because using the whole training set on my computer was not time-ideal.

III Subtask 3

3.1 The Question

In subtask 3, it was asked to implement the LambdaMART model, which is similar to the LambdaRank algorithm learned in class, to infer the relevance of a query-document pair. It was asked to implement the model using the XGBoost library and tune the hyperparameters into finding the best performing model, then report back the performance metrics.

3.1.1 Summary for tasks

- Implementing LambdaMART
- Hyperparameter tuning
- Report Performance

3.2 Implementation

3.2.1 Implementing LambdaMART

Before training the LambdaMART algorithm in the XGBoost library, our training data had to be converted to a "DMatrix" which is a special XGB matrix that takes the training features and

training observations into the matrix to be input to the model whilst training. The same input and features used for logistic regression was used for this model, which is the absolute difference between both the average query vector and its average document vector pair.

The way LambdaMART can be used in the XGBoost library is to train the model using "XGB.train" whilst using the line "'objective': 'rank: pairwise'" which uses the LambdaMART algorithm to minimize the pairwise loss (features and observations).

3.2.2 Hyperparameter tuning

After looking up the XGBoost documentation, it was found that there are a lot of parameters that can be tuned for improving performance. However, in the interest of time, the 3 main hyper parameters we will be tuning are the stepping size (learning rate), the number of boosting rounds, and the maximum depth of the algorithm's tree.

The tuning method which was used was **cross-validation**. A range of adequate step size, depth size, and number of rounds values were created, and for each instantiation of these hyper parameters, we would train the model using the training set and report performance on the validation set. The parameters with the highest MAP and MnDCG score were then used to train the final model.

The ideal parameters found for the model was:

- Step size: **0.3667**
- Depth size: **6**
- Number of rounds: **9**

3.2.3 Report Performance

Result: **MAP= 0.0208= 2.08%.**
Result: **MnDCG= 0.0468 = 4.68%.**

Though these results are slightly better than the logistic regression model, it is still not ideal. Reasons could still be the use of a relatively small dataset considering the complexity of the problem.

IV Subtask 4

4.1 The Question

For this subtask, it was asked that, using the same training representation input, we should build a neural network-based

model using existing packages, we should then report the performance of it.

4.1.1 Summary for tasks

- Implement Neural Network
- Report Performance

4.2 Implementation

For building the neural network, the library package PyTorch was used as it eases the implementation complexity of the neural network as well as access to its parameters.

4.2.1 Implementing the Neural Net.

4.2.1.1 Layers and activation function

For the neural network, a **feed forward** neural network (FNN) was used initially to observe standard results.

To train the neural network using PyTorch, the training data had to be converted to tensors, which are a representation of vectors and matrices that PyTorch makes use of, equivalent to a NumPy array.

The architecture used for the neural network was as follows:

- Input layer: **300** nodes, **Sigmoid** activation function
- Hidden layer: **200** nodes, **Sigmoid** activation function
- Output layer: **1** Node.

A good rule of thumb for the size of the hidden layer used is to use a size of $\approx \frac{input\ layer + output\ layer}{2}$. However, using a hidden layer of 200 proved to give better results.

Activation functions are what add non-linearity in the neural network, allowing for separation of seemingly non-separable data. Therefore, nonlinear activation functions should be used. And since what our end result objective is a relevance value between 0 and 1. It is convenient to use the Sigmoid activation function.

4.2.2.2 Cost function

This is considered to be a binary classification of problem, where there is only one output class that could be between 0 and 1. Therefore, the recommended cost function to be used in that situation is the **Binary Cross Entropy Loss,** which can be described as the following:

$$l(x, y) = -[y_n . \log(x_n) + (1 - y_n) . \log(1 - x_n)]$$

However, since the Sigmoid activation function is applied to the output layer class before calculating the loss, one can then make use of the **"BCEWithLogitsLoss**" loss function which takes the logit output (raw output) and combines a Sigmoid layer and the BCELoss in a single step, which proves to be more numerically stable. Another strong point for this function is the option to add a weight on the positive samples.

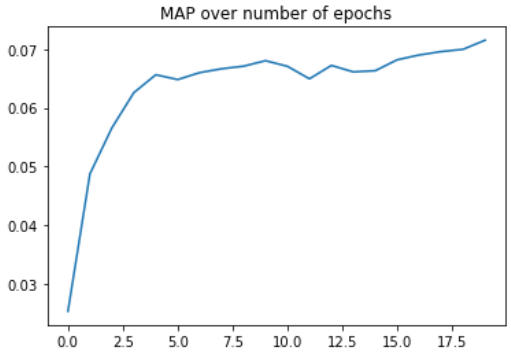
$$l(x, y) = -[p_c y_n . \log \sigma(x_n) + (1 - y_n) . \log(1 - \sigma(x_n))]$$

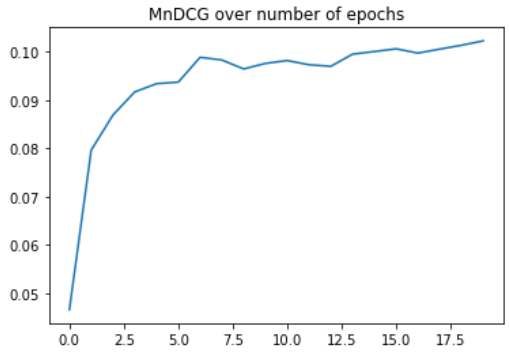
$$where\ \sigma(x_n)\ is\ the\ sigmoid\ function$$
$$and\ p_c\ is\ the\ positive\ samples\ weight$$

One problem we have is that the training data is imbalanced (10 negative samples vs 1 positive sample), this can result in our algorithm to be pulled into biasing towards outputting zero all the time as it would give a lower accumulated cost function. Therefore, one method to combat this problem is to add a weight on the positive samples, a suitable value for the positive weight is to equate it to the ratio between the negative and positive samples, in order to have a virtual balanced dataset. Therefore, a value of **10** was used for the positive weight parameter.

Using our mentioned network architecture, the model is trained with batches of 64 shuffled training samples using back propagation to find the gradient change.

Two plots of our validation metrics were constructed for each epoch (training session) to verify the model is learning (higher metrics), the plots for both MAP and MnDCG is shown below:





4.2.2 Reporting Performance

Result: **MAP= 0.701 = 7.01%**

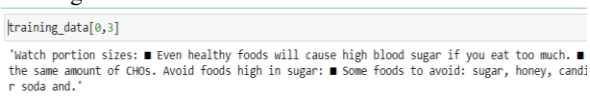
Result: **MnDCG= 0.1014 = 10.14%.**

Interpretation for these results are given below in the next section.

V Issues encountered and summary

5.2 Issues encountered

One issue that was encountered regarding the training dataset was that the first document show had a bug where there was a black box shown in the document which resulted in the program crashing. It was then decided that we remove this row.



Another issue that was encountered was that some queries had misspelled words which resulted in the embedding model to fail, this was solved by catching these exceptions using try-except method on python, and these words were consequently, ignored. However, there were also some queries that "only" contained misspelled words, this was handled by if that case happens, then the embedding vector for the query would be a vector of zeros.

5.2 Comparison and Summary

It is evident that the results from the BM25 model was the highest, which implies that modifications in the ML models are required. One potential reason as to why the ML models had low results as opposed to the BM25 is because it did not contain explicit information, but rather the average of all the words in a given query/document. For example, if a query was to contain two "unrelated" words, which in turn got converted to their equivalent embedding vector, the average of these two unrelated words could result in a completely different word which can have a similar vector. Or, the average of these two unrelated words could be closer to an averaged-out document which does not contain these words than a document that does in fact contain these two words but it also contains other words which forced the average to drift away from the query average.

One potential fix for that problem is to include the **BM25** score for each query/document pair as an **additional feature** in the ML models. Which means the model now takes both the explicit information and the general "context" of the queries/documents into account. Resulting in a more accurate retrieval method.