# Information Retrieval and Data Mining (COMP0084)
## Coursework 2

## General Notes

The code for this coursework is divided into four Jupyter Notebooks, one for each task: “task1.ipynb”, “task2.ipynb”, “task3.ipynb”, and “task4.ipynb”. The four notebooks have clear sections and descriptions for the code which can be helpful for the marker. To save time and computation resources, the data features generated in Task 3 are saved as text files and then imported in Task 4 when they are needed. Specific programming choices are commented in the actual scripts, but the marker should refer to this report for any other clarifications such as general methodology, data frames or text-processing techniques used for each task.

There are two additional notebooks, "BM25_TFIDF_train" and "BM25_TFIDF_val" used to calculate the BM25 scores and cosine similarity scores of TF-IDF vectors for all the data samples in the training and validation sets. These are used for generating the data features for the models implemented in Task 3 and Task 4.

Note: I advise the examiner not to run the hyperparameter tuning code in "task3.ipynb" since it is computationally very expensive and took several hours for me to run. Instead, simply run the final model with the optimal parameters in the next section of the notebook.

## Task 1 - Evaluating Retrieval Quality

This task focuses on implementing metrics to analyse the performance of retrieval models, which we will use throughout this report to check how well each of the different models we create ranks passages for a given query.

### Dataset

We are given a training dataset consisting of a training dataframe of 4,364,339 samples, and a validation dataframe of 1,103,039 samples. Each dataframe sample has 5 columns: a query, its corresponding identifier *qid*, a passage, its corresponding identifier *pid*, and the relevancy score the passage has to the query, i.e. whether the passage is relevant for the query or not. The score is binary, either 0 (non-relevant) or 1 (relevant).

| - | Train. Set | Val. Set |
|---|---|---|
| **unique queries** | 4,590 | 1,148 |
| **relevant passages** | 4,797 | 1,208 |
| **non-relevant passages** | 4,359,542 | 1,101,831 |
| **total (qid,pid) pairs** | 4,364,339 | 1,103,039 |

Table 1: *Dataset summary statistics.*

It is worth nothing the characteristics of the dataset specified in Table 1. We note that in both training and validation set, there is at least 1 relevant passage per query, but in most cases, only 1 relevant passage per query, and at most 2 or 3 in some cases.

### AP and NDCG metrics

We define two rank-aware evaluation metrics: Average Precision (AP) and Normalised Discounted Cumulative Gain (NDCG). These metrics take into account the position relevant items have in a ranking, since more relevant items should rank higher than irrelevant ones.

The Average Precision is a binary relevance metric that evaluates a list of recommended items up to a specific cut-off point specified by parameter k. For example, the Average Precision @ k=10 for a certain ranking will look at the first 10 items in the ranking and assess which of the retrieved items within the cut-off are relevant, giving higher weight to higher ranking items. We can compute the AP for each query ranking, where the "items" correspond to the passages. Once we have the AP@k for all the queries in the dataset, we can compute the mean Average Precision (mAP) by doing the average of all the AP in the dataset. The formulas

for AP and mAP can be expressed as follows:

$$AP_q = \sum_{r=1}^{R} \frac{r}{rank_r} \tag{1}$$

$$mAP = \sum_{q=1}^{Q} AP_q \tag{2}$$

Where *Q* is the total number of unique queries in the dataset, *R* the total number of relevant passages in the specified cut-off of the ranking, and $rank_r$ is the rank of each relevant passage in the ranking.

Normalised Discounted Cumulative Gain is similar to the average precision, but it takes into account the level of relevancy of the items. Now relevancy is not binary 0,1, but it can take different values, i.e. scores from 0 to 5 for example. Therefore, the NDCG assesses if high-relevant items are ranked higher than low-relevant items, which in turn should be ranked higher than non-relevant items.

$$DCG_q = \sum_{i=1}^{p} \frac{2^{rel_i} - 1}{log_2(i+1} \tag{3}$$

$$IDCG_q = \sum_{i=1}^{|REL_p|} \frac{2^{rel_i} - 1}{log_2(i+1} \tag{4}$$

$$NDCG_q = \frac{DCG_p}{IDCG_p} \tag{5}$$

$$mNDCG = \sum_{q=1}^{Q} NDCG_q \tag{6}$$

Where *p* is the number of elements in the cut-off ranking, *rel* is the relevance score of each position, and $|REL_p|$ is the list of relevant documents in the corpus up to position *p*, ordered by their relevance.

It is worth noting that in this coursework, it feels like we are not exploiting the full potential of the NDCG metric, since we deal with binary relevancies. Therefore, the metric scores will simply depend on the position in the ranking of the very few relevant items in the set.

We calculate the metrics for cut-off points of k=3,10,100. By nature, the metrics at k=100 will be higher than at k=10, and in turn higher than k=3, since there will be more probability for the one relevant passage to be included in the cut-off point. However, in the ideal case where the retrieval models worked very well and always ranked the relevant passage in the top 3 positions for every query, the metrics should be the same for all cutoff points.

| **@k=** | 3 | 10 | 100 |
|---|---|---|---|
| **mAP** | 0.192 | 0.229 | 0.241 |
| **mNDCG** | 0.211 | 0.287 | 0.357 |

Table 2: *BM25 summary performance metrics.*

**Text normalisation**

We follow the same steps for pre-processing and tokenising the queries and passages text. I will not go into detail because that was more the focus of the previous coursework, but in summary:
1) Turn all the text to lower-case.
2) Expand contractions.
3) Remove punctuation and replace it for a white space.
4) Tokenise by splitting at whitespaces.
5) Remove stop words.
6) Lemmatise nouns - only in this task. In further tasks, lemmatisation has been ignored, since GloVe contains vector representations for different forms of the same word.

**Performance of BM25**

We implement the BM25 model developed in Coursework 1 on the validation set. Using the BM25 scores, we create a ranking of (at most) the top 100 scoring passages and store them in the file "bm25_cw2.csv". Then we use this ranking to compute the mAP and mNDCG metrics. Note two things when we computed the metrics:
1) In the case where the query had no relevant documents in the cut-off point in the ranking, the AP and NDCG metrics for that query were both given a value of 0.
2) Sometimes a query has less candidate passages than the cut-off point specified. In this case, we stop computing the metric when we reach the last passage. This has a favourable effect on the metric score for that query, since the relevant passage has more odds of being in a higher position than in queries with more candidates even if the retrieval model performs badly (because the relevant passage can only be ranked as low as the number of candidate passages). The number of queries with small number of candidate passages is relatively low in the entire corpus, so this will not affect much the global mAP and mNDCG scores.

The mAP and mNDCG results are summarised

in Table 2. We can deduce that for most queries, the relevant passages are within the top 5 in the BM25 ranking, which indicated that BM25 is a good retrieval model for passage re-ranking.

Note that taking into account that for most queries there is only one relevant passage with score 1, the metrics reduce to:

$$AP_q = \frac{1}{rank_r} \tag{7}$$

$$NDCG_q = \frac{1}{log_2(rank_r + 1)} \tag{8}$$

Therefore, it is expected that mNDCG > mAP, since $log_2(rank_r - 1) < rank_r$, with the difference getting larger for lower rankings, i.e. the badly ranked queries have more impact in reducing the mAP than they do in mNDCG.

## Task 2 - Logistic Regression

In this task we explore passage re-ranking using a Logistic Regression model, which tries to find a hyperplane that divides the data into two sub-spaces, one for each class. In this case, the classes are non-relevant documents (0) and relevant documents (1).

The inputs of the logistic regression model are arranged in an array $X$ of shape (N,D), where N is the number of data samples and D is the number of input features. The outputs are stored in vector $y$ of shape (N,). $X$ is mapped to predictions $y_{pred}$ via a weights vector $w$ or shape (D,). For binary classification, the output value is mapped from $(-\infty, +\infty)$ to (0,1) using a sigmoid function:

$$y_{pred} = \sigma(Xw) \tag{9}$$

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{10}$$

The predictions are evaluated with respect to their expected values using a Binary Cross Entropy loss function:

$$L(y, y_{pred}) = -\frac{1}{N}\Bigg(ylog(y_{pred}) + (1-y)log(1-y_{pred})\Bigg) \tag{11}$$

Note that this notation uses vector multiplication that acts as a summation over data samples. The gradient of the loss function is:

$$\Delta_w L = X^T(y_{pred} - y) \tag{12}$$

Optimisation is done initialising the weights at zero and using gradient descent with learning rate $lr = 0.05$ and the following stopping criteria: loss changes less than a set number $\epsilon$ between two consecutive iterations, weights are updated by less than $\epsilon$, or we reach a maximum number of iterations. These are all hyper-parameters that can be tuned during training and validation if needed.

### Data manipulation

We have set out the framework for the logistic regression model. Now we just need to manipulate the data so that we can feed it to the model.

The queries and passages are lists of strings. These can be vectorised using word embeddings. For this, we use GloVe (Global Vectrs for word representation), a pre-trained algorithm for obtaining vector representation for words. With this we create a dictionary where each word (or token) has its own vector representation of length (50,), and there is also a vector representation for token <unk> which accounts for the words that are not found in the GloVe vocabulary.

We apply GloVe to every query and passage, getting vector representations for each of the tokens in the query/passage, and then doing the average of those vectors to obtain a unique 50-length vector for each query/passage. However, ranking data works with (query,passage) pairs, and therefore, we need to find a way to represent these pairs numerically. This can be done in 2 ways:
1) Compute the average between the query embedding vector and the passage embedding vector, resulting in a vector of length 50. The resulting input data will have size (N,D=50), where N is the number of (query,passage) pairs.
2) Concatenate the query vector embedding and the passage vector embedding, resulting in a (query,passage) vector of length 100. The input data will have size (N,D=100)

The average approach was chosen for its computational benefits and its improved performance over the concatenating approach, although neither of them resulted in good performance of the LR model.

| @k= | 3 | 10 | 100 |
|---|---|---|---|
| **mAP** | 0.003 | 0.005 | 0.008 |
| **mNDCG** | 0.004 | 0.009 | 0.025 |

Table 3: *Logistic Regression summary performance metrics.*

### Performance of LR for passage re-ranking

After the model has been trained with the input training data of the format specified in the previous section, alongside its target relevancies, the model weights $w$ are estimated, and can be used to make predictions on the validation data. These predictions are then used to rank the passages for each query. This ranking is saved in the output file "LR.txt", which contains the top 100 passages (at most) for each query, in order from highest rank (rank 1) to lowest rank (rank 100 if applicable), with their respective logistic regression scores. Here the scores are the probability from 0 to 1 of the passage being relevant for the query.

The ranking from the LR scores is then used alongside the Average Precision and NDCG functions designed in Task 1 to assess the performance of the LR model. The performance metrics results are summarised in Table 3.

The performance metrics have extremely low scores, representing the poor job the LR model does at ranking the passages. This can be attributed to the feature representation used for the data (average of query and passage embedding vectors) not being a good representation of the data for learning (query,passage) relevancy. However, the most likely issue for the poor appropriateness of the LR model is its loss function. We are using cross-entropy loss function which focuses on the accuracy of the predictions for individual data points, it does not assess ranking. Therefore, since the vast majority of the data has a relevancy score of 0 (imbalanced data), the model will minimise its loss function by learning to give a prediction very close to zero, and in doing so, ignoring the relevant documents, which are essentially the most important part of the data.

A workaround could be to have a weighted loss, such that data points with relevancy = 1 have higher weight, so that predicting them wrong would have a higher impact on the loss function.

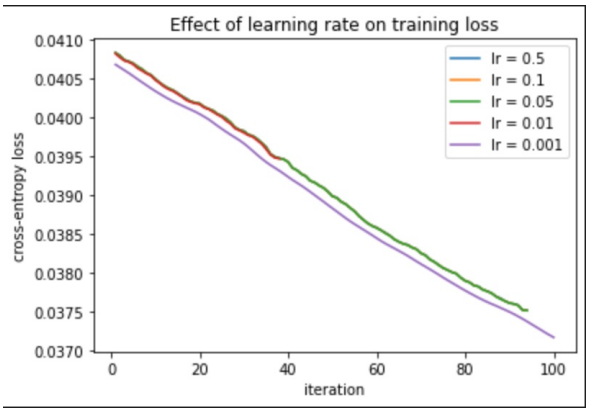


Figure 1: *Training loss behaviour for different learning rates.*

### Effect of the learning rate on the model training loss

We also investigate how the learning rate affects the cross-entropy loss during training. We train the model with learning rate values $[0.5, 0.1, 0.05, 0.01, 0.001]$, and we plot how the training loss behaves at each iteration. This is illustrated in Figure 1.

It can be seen that the loss decreases linearly with the iterations, and takes similar values for all the leaning rates. However, for lower learning rates, we can train further before reaching the stopping criterion, and so we can reach a smaller loss. However, the loss is already so small (due to the fact that most data points have relevancy 0), that training further does not improve the model.

## Task 3 - LambdaMART Model

### Learning2Rank and LambdaMART

LambdaMART is an algorithm from the Learning to Rank techniques that are used to solve ranking problems using supervised machine learning. More in detail, LambdaMART combines Lambda Rank, which uses NDCG in its cost function, and Multiple Additive Regression Trees (MART).

Contrasting with the Logistic Regression model, here we focus on the ranking of the data while training, and hence, it is more appropriate for the type of problem we have. Therefore, we can expect better performance.

### Input processing and feature representation

In this model, we will not use the average embedding vectors of the (query,passage) pairs as the input data to the model, as we did in LR. Instead, we will manipulate the embedding vectors and use

other metrics to build a new set of features for each (query,passage) pair.

We define 5 features for our data:
f0) The cosine similarity between the query embedding vector and the passage embedding vector created with GloVe. The similarity between two vectors is computed using the dot product of the two vectors and their norms, as shown in the following equation:

$$similarity(q,p) = \frac{dot(q,p)}{norm(q) \times norm(p)} \quad (13)$$

f1) The BM25 score of the (query,passage) pair.
f2) The cosine similarity between the TF-IDF vector of the query and the passage, computed in the same way as specified in Coursework 1.
f3) The query length: number of tokens in the query after text normalisation (excluding lemmatisation).
f4) The passage length: number of tokens in the passage after text normalisation (excluding lemmatisation).

These features have been chosen because they can numerically represent the similarity between a query and a passage. The query and passage length can also give some information to whether a passage is relevant to a certain query. For example, it is more probable that a complicated query has considerable length, which is then appropriately answered by a passage that also has considerable length. Therefore, we could expect a proportional relationship between the length of a query and its relevant passages.

These feature representations are stored in order in input data arrays of size (N,5), where N is the number of (query,passage) pairs in the respective dataset, which are used in the LambdaMART model for training and validating purposes.

**Hyper-parameter tuning**

The LambdaMART model by XGBoost has a very large number of parameters that can be fine-tuned to find an optimal model for the dataset we are working with. A way to do this would be with a massive grid search with all the possible parameters, but that would require a lot of time and computing power resources, so we focus on the most important parameters that have the biggest effect on a model. These are differentiated between parameters that control overfitting with model complexity: maximum depth of a tree and minimum sum of instance weight needed in a child; and parameters that control overfitting by adding randomness: subsample ratio of training instances taken into account, subsample ratio of features taken into account, and step size.

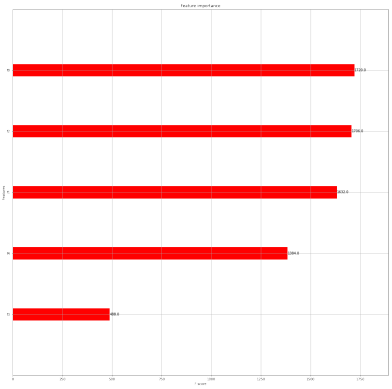

Figure 2: *Importance the LambdaMART model gives to each feature.*

We are not provided test data, so the validation set is the one we use for testing. Therefore, we randomly separate the training set in two parts, 90% is used for training, and 10% is used for hyperparameter tuning.

With this established, we run loops of small grid searches, training the LambdaMART model with different parameter combinations, and testing its performance on the tuning set using the mNDCG metric, as defined in Task 1. The parameter combinations that yield highest mNDCG values are the ones carried over to the next grid searches for other hyperparameters, until we have tuned all 5 parameters. The optimal parameters found after tuning are: max. depth = 11, min. child weight = 7, subsample ratio = 0.7, feature ratio = 0.7, step size = 0.005.

The model is trained once more with the full training set now, and with the optimal hyperparameters. Then we feed it the validation data to make predictions and generate a passage ranking for each query, saved as "LM.txt".

We illustrate in Figure 2 the importance the model gives to the data features, where the highest importance is attributed to f0 (GloVe embeddings similarity) and f2 (BM25 score), and the least to f3 (query length). This could be expected, since the similarity scores are more significant to relevancy than passage or query length.

| @k= | 3 | 10 | 100 |
|---|---|---|---|
| **mAP** | 0.148 | 0.178 | 0.192 |
| **mNDCG** | 0.162 | 0.228 | 0.304 |

Table 4: *LambdaMART summary performance metrics.*

| @k= | 3 | 10 | 100 |
|---|---|---|---|
| **mAP** | 0.007 | 0.010 | 0.013 |
| **mNDCG** | 0.008 | 0.014 | 0.033 |

Table 5: *MLP summary performance metrics.*

### Performance of LambdaMART for passage re-ranking

The passage ranking generated in the previous section is used to compute the performance metrics for the LambdaMART model. These are summarised in Table 4.

The metrics are similar to the BM25 results, meaning that the model has a good performance and manages to rank the relevant passages within the top 10 for the vast majority of queries. This performance could be further improved by generating more meaningful features for the dataset in addition to the five we already presented.

## Task 4 - Neural Network Model

In this final part we attempt passage re-ranking using a Neural Network retrieval model in PyTorch. The feature representation of the data is the same as stated in Task 3, for the same reasons. We simply saved the features and imported them in this task.

### Multi-Layer Perceptron

We implement a feed-forward Multi-Layer Perceptron (MLP), where the initial input has 5 features, as specified in Task 3, and the information goes through 6 hidden layers of 10 neurons each until reaching the output layer containing a single feature, representing the relevancy score of the (query,passage) pair input.

The information in the nodes of each layer is a linear combination of the information on the nodes in the previous layer, determined by the network weights, which are the parameters we are trying to estimate using a loss function and backpropagation. At every layer, the resulting linear combination is then passed through an activation function (ReLU in our case) to add non-linearity, and we apply drop-out that randomly ignores 20% of the nodes as an attempt to regularise the model. In the output layer, we apply the sigmoid function to ensure the prediction is a score ranging from 0 (non-relevancy) to 1 (relevancy). It is important to note that the number of hidden layers adds to the complexity of the model, which contributes to overfitting. That could be parameterised to find the optimal number of hidden layers needed to maximise performance during prediction. We also apply weight decay in the optimiser as a form of L2 regularisation.

I did not find a loss function implemented in PyTorch that could work with rankings (as LambdaMART's loss function does) instead of individual predictions, so the binary cross-entropy loss was used, as in Logistic Regression. This time, to account for the class imbalance, I weighted the loss giving 0.9 importance to relevant (query,passage) pairs, and 0.1 to non-relevant ones, to avoid the case where the model learns to give very low score to all data points to minimise the loss. Note that the weights could also be calculated from the proportion of instances of each class in the dataset.

We train the network for 20 epochs (note that early stopping could be applied if we needed further regularisation), and the resultant models with fitted weights is used on the validation set to make predictions, which are in turn used to make a ranking of passages for each query, saved in the file "NN.txt".

### Performance of Neural Networks for passage re-ranking

Using the scores from "NN.txt", we calculate the perfomance metrics for the Multi-Layer Perceptron model. These are summarised in Table 5.

The scores are very low, of the same order of magnitude as the Logistic Regression model, but slightly improved, probably thanks to the added complexity of the network and the weightings in the loss function. However, it is clear that this is not an appropriate model for passage re-ranking, probably because the cross-entropy loss function focuses on the prediction of individual datapoints and not on their relative ranking, which results in a different objective. If I had time to improve on this, I would look for possible ranking loss functions that were available to implement in PyTorch that used metrics such as mAP or NDCG, in a similar way to LambdaMART.