IRDM Course Project Part II: Ranked Passage Retrieval

Student Number: 16034489

ABSTRACT

Information retrieval models play a vital part in many applications, from search and question answering to recommendation. In *Ad-hoc* passage retrieval a user generates a query and the system evaluates a collection of short texts (passages), returning them in an order of decreasing relevance. In this project, we implement and evaluate a number of different passage-ranking algorithms, such as Logistic Regression, LambdaMART and a Multilayer Perceptron. All of the code for this assignment can be accessed using this link: **Google Colab Code Repository**.

1 EVALUATING RETRIEVAL QUALITY

To assess the quality of a ranking, various evaluation metrics can be used. Here, we implement from scratch Average Precision and Normalised Discounted Cumulative Gain (NDCG) to evaluate the quality of a ranking produced using BM25.

A good ranking system should return the relevant document before the non-relevant ones. In the case where the list is unranked, useful measures of performance are precision and recall. However, it is also possible to apply these to ranked lists by computing these metrics for each top n results, where $n = 1, 2, ...k$. This results in a precision-recall curve. However, rather than the curve itself, it is more useful to compute the area under the curve - this can be done by computing **Average Precision**, which is a piece-wise linear approximation of the area under the precision-recall curve.

To compute average precision for a single query, we first look at the labelled dataset and extract the IDs of relevant and irrelevant documents. Next, we transverse the list ranked by our algorithm, BM25 in this case and record every time one of the documents retrieved by our algorithm is relevant. At each step we compute the cumulative number of relevant documents retrieved so far, divided by the rank of the document and appended to a list. At end, all of the observations in the list are divided by the total number of relevant documents and summed together, giving us the Average Precision. To compute average precision for an entire collection we compute the mean of the average precisions for all individual queries.

Another measure of model performance is Discounted Cumulative Gain. DCG takes into account the reciprocal of a rank, hence it is very sensitive to rank position. It does not consider explicitly the total number of relevant documents, but rather whether at least some are relevant ones are retrieved near the top of the list. While in this case we have simply a binary measure of relevance, this metric can be very useful when the measure of relevance is not binary, incorporating the value of graded relevance into the metric. However, comparing raw DCG values between different models and datasets is impossible, thus normally nDCG is used instead. nDCG normalises DCG against the best possible DCG result, ie. the perfect ranking, thus, the range of nDCG is such that $0 \leq \text{nDCG} \leq 1$, allowing for comparison against different models and making average easier for queries where the number of relevant documents is different.

The discounted cumulative gain (DCG) is computed using:

$$\text{DCG}_\text{p} = \sum_{i=1}^{p} \frac{rel_i}{\log_2(i+1)}$$

While the formula for the 'perfect' ranking takes the form of:

$$\text{IDCG}_\text{p} = \sum_{i=1}^{|REL_p|} \frac{rel_i}{\log_2(i+1)}$$

where Rel_p indexes the list of relevant documents in the corpus, ordered according to their relevance, up to position p.

The normalised DCG is then computed by taking the ratio of the two formulae:

$$\text{nDCG}_\text{p} = \frac{DCG_p}{IDCG_p}$$

It is important to note that to compute the metrics we only use the documents which have been ranked in the top 100 by BM25. There are a number of compelling reasons to do so. Firstly, the entire collection contains 955211 passages, hence there is an enormous time-saving component. And while using only the top 100 documents may lead to a marginal loss in accuracy if a relevant document is retrieved at a position further than 100, however when we consider this from a practical point of view, it does not matter very much to the user whether the relevant document is retrieved at a 150th or 1000th position, as both of these are ranked significantly too low. Hence, by considering only the top 100 documents we essentially treat all the relevant documents retrieved at positions beyond 100 as not retrieved at all, thus contributing 0 to the overall model performance. It is worth noting that in this specific collection each query has only one or two relevant documents associated with it, thus it is of utmost importance that those are retrieved near the top of the list.

Code: The implementations of the metrics are located in the `metrics.py` file, and used as such throughout the entire assignment. The ranking of the validation set using BM25 and the evaluation results are found in the `Question_1.ipynb`.

2 LOGISTIC REGRESSION

2.1 Word Embeddings

A great deal of thought and effort was put into the representation of passages and queries using word embedding methods in this assignment. Experimenting with different methods, I decided to train my own *FastText* embeddings, using the gensim library. For best results, I trained the model using the entire training dataset, which is considerably large. This posed some significant computational challenges, but also proved to be a great opportunity for learning how to make code more efficient. Firstly, I created dictionaries which mapped unique query/passage IDs to their corresponding

strings. The strings were then pre-processed in the same manner as in the last assignment, using my `pre-process` function, located in the `utils` folder. However, in this particular dataset I found that my implementation of pre-processing occasionally returned an empty list of processed tokens - this was because my code originally removed all tokens which where 3 characters long or shorter. I found that amending the code to keep tokens which are 3 tokens long solved this issue. Once the preprocessing was performed, I concatenated the pre-processed queries and passages from the training dataset, shuffled it and trained the FastText model. The model was then saved and used to generate the embeddings for all of the remaining questions in this assignment. The dataset was created as follows: the model was used to generate a representation for each word. Then the representation for each passage/query was obtained by averaging all the words in the given query/passage. Lastly, the representation for each passage-query pair was obtained by concatenating the passage and the query. Initally, I trained a 200-dimensional FastText model, however later found that it was computationally unfeasible to use a 400-dimensional dataset (from concatenating the queries and passages) of this size, hence 100-dimensional embeddings where later trained instead. However, even when using only a 200-dimensional dataset using the full dataset was unfeasible, as the training dataset was 41GB and took nearly two days to generate. Hence, it was essential to reduce the size of the dataset through downsampling. Here, I experimented with a number of different approaches, primarily random downsampling and negative sampling. As the number of positive examples is extremely sparse in this dataset, with each query having only one or two positive examples, in both approaches we ensure that we include all of the positive examples. In the first approach the negative examples are selected randomly, while in the second approach 100 negative examples where selected for each query. I found that the second approach performed better, hence it was used thereafter and it is applied to both, the training and the validation dataset.

Code: Due to the extremely large dataset size, data was saved at each step of pre-processing. These intermediate files can be found in the `data` folder in the Colab repository - they were not included in the submission of local files due to their very large size. The final embeddings are stored in the `data/fast_text` repository, as `train_dataset_150.csv` and `valid_dataset_150.csv`.

2.2 Logistic Regression Implementation

Next, we implemented Logistic Regression from Scratch. The full code can be found in `Question_2.ipynb`. Here, we will apply logistic regression to a pseudo-classification problem - determining whether a given document-query pair is relevant (1) or irrelevant (0). Logistic regression works by first computing a prediction using a linear function, ie. a linear combination of weights and features, and then passing this output via a sigmoid function. The output can thus be formalised as $h_\theta(x)$:

$$h_\theta(x) = g\left(\theta^T x\right) \tag{1}$$

where in this case g is the sigmoid function:

$$g(z) = \frac{1}{1+e^{-z}}$$

and the weights are denoted as θ. It is worthwhile to note that the feature matrix X has been modified to allow for a bias term - ie. the matrix has been appended with a column of 1s.

The algorithm works by minimising the log-likelihood loss, which is defined as:

$$J(\theta) = \frac{1}{m} \cdot \left(-y^T log(h) - (1-y)^T log(1-h)\right)$$

At each step, we compute 1 and then update the weights in the direction of the gradient of the loss. The gradient of the loss can be derived to give:

$$\frac{\delta J(\theta)}{\delta \theta_j} = \frac{1}{m} X^T (g(X\theta) - y)$$

The updated weights θ^* are then obtained using:

$$\theta^* = \theta - \alpha \frac{\delta J(\theta)}{\delta \theta_j}$$

where α is the learning rate.

Here, we experimented with the effect of the learning rate on the training loss.

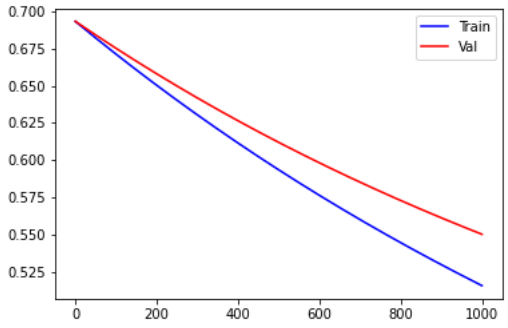

Figure 1: Effect of learning rate on training loss, using Logistic Regression. Learning Rate: 1e-05

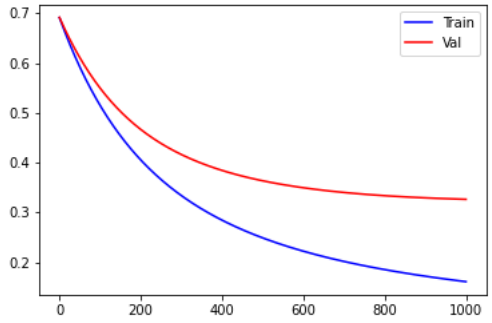

Figure 2: Effect of learning rate on training loss, using Logistic Regression. Learning Rate: 0.0001

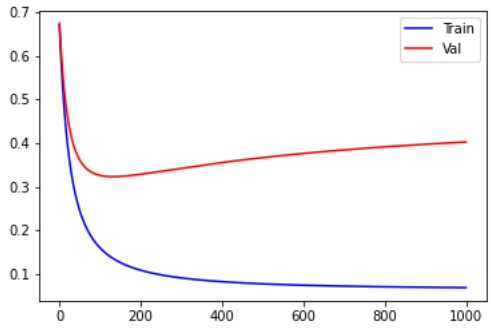

Figure 3: Effect of learning rate on training loss, using Logistic Regression. Learning Rate: 0.001

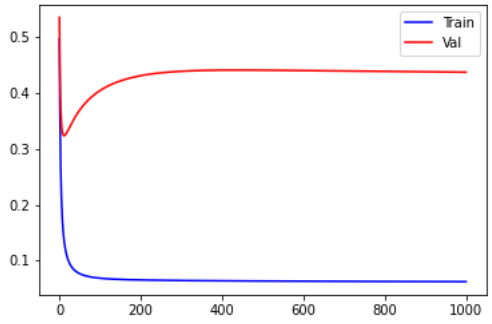

Figure 4: Effect of learning rate on training loss, using Logistic Regression. Learning Rate: 0.01

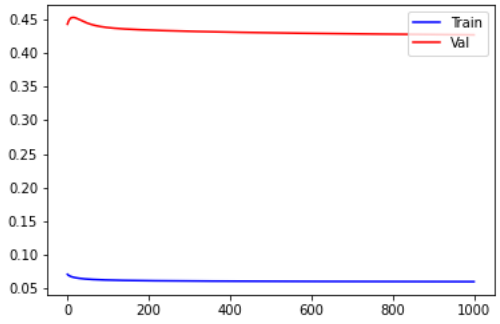

Figure 5: Effect of learning rate on training loss, using Logistic Regression. Learning Rate: 0.1

Looking at the figures, we find that the value of the learning rate has a very significant effect on the training loss: at very small values of α, learning occurs very slowlym not reaching anywhere near convergence in 1000 iterations. As we increase α, the rate of learning increases and we converge much faster. However, we find that when the learning rate is too large, such as in 5, no learning occurs - since we are taking steps which are too large, we are likely to overstep minima. Hence, it is important to select a learning rate which is large enough to accelerate the learning process, but also not so large as to prevent learning altogether.

It is worthwhile to note that from these graphs it is clear that we are over-fitting for the training set and doing poorly on the validation set. Unfortunately, due to the computational restrictions the models are only evaluated on a sample of the validation set, with the sampling strategy outlined in the subsection above. This over-fitting it likely due to the fact that the number of positive examples is still significantly smaller than the number of negative examples, leading to class-imbalance. Hence, the model learns to just predict every example as negative and becomes biased. This can be addressed by either regularisation, ie. penalising weights which are too extreme, or through positive oversampling. An attempt was made here at introducing L2 regularisation but it was found to have very little effect, except for delaying the over-fitting. Both of these methods are explored in more depth in the Neural Network implementation.

We find that the best model achieved the following results on the Validation Set: **mAP:** 0.272 **nDCG:** 0.437

3 LAMBDAMART

3.1 LambdaMART Implemetation

LambdaMART is one of the recent passage-ranking algorithms developed at Microsoft [1], which is boosted tree version of LambdaRank, which in turn is based on RankNet. At the core all three of these algorithms is the use of a pairwise cost function - ie. considering items in pairs and determining the optimal ordering of the items in the pair. Then, by considering all the possible pairs in the dataset we end up with an optimal ordering of all items in the dataset. Hence, to minimize the cost function the algorithm aims to minimize the number of incorrect orderings amongst the pairs. RankNet uses SGD in order to minimize this loss function. Building on RankNet, the key idea behind LambdaRank is that in the training phase the gradients of the cost function, λ, are scaled by the change in nDCG upon swapping each pair of document. The LambdaMART algorithm combines this approach with MART - Multiple Additive Regression Trees. In standard MART approaches, gradient-boosted decision trees are utilised for prediction tasks, while in this case the gradient boosting decision trees have been adapted to use the pairwise cost function from LambdaRank.

In this project, LambdaMART was implemented using the XGBRanker class from the XGBoost library, by setting the `objective` parameters to pairwise loss. In order to attain optimum performance on the validation set, it is necessary to find the optimum combination of hyperparameters, such as maximum depth of the tree, the value of the gamma parameters etc. However, as the performance of the model depends on the interactions between the individual parameters, it is necessary to carry out a grid search in order to find which model combinations work best. Initially, I attempted to achieve this by implementing a randomized grid

search from the sklearn library, which allows for considering a larger combination of values. However, this particular model requires 'group' information about the number of passages which are being considered for each query and proved to be incompatible to incorporate this information into RandomizedSearchCV. Therefore, instead I opted out for initially carrying out some heuristic experimentation to find a suitable parameters range and carried out a few simple linear searches. Next, using the intuitions gathered in this way I performed a grid search over 90 different combinations of 4 hyperparameters: `eta`, `gamma`, `min_child_weight` and `max_depth`. As the training time was quite significant and we found that within the first 50 iterations you could gauge which models would perform best, these 90 combinations were trained and then the best models were trained till convergence. The results of the grid search can be accessed in `grid_search.csv`.

The best hyperparameters found were:

- `eta` : 0.1
- `gamma` : 0.5
- `min_child_weight` : 3
- `max_depth` : 6
- `subsample` : 0.8

We find that the best model, trained for 300 iterations, achieved the following results on the Validation Set: **mAP:** 0.4897 **nDCG:** 0.431

Code: The implementation of LambdaMART and computations using the performance metrics can be found in `Question_3.ipynb`.

4 NEURAL NETWORK MODEL

Logistic Regression can be seen as a feed-forward neural network with no hidden layers. I decided to experiment whether adding some hidden layers and thus creating a feed-forward neural network in PyTorch would lead to better results than just using Logistic Regression. Therefore, I decided to use the same representation, ie. the FastText embeddings which were trained using the entire training dataset.

However, treating a ranking problem as a classification problem turned out to be a challenging but also interesting endeavour. Neural Networks are very powerful and in principle are capable of generalising to learn any non-linear relationship. However, we found that with no regularisation, the network just defaulted to predicting all items to be not relevant, as with so few positive examples this resulted in minimising the pointwise training loss. Nonetheless, our end-goal here is not to minimize loss, but rather to produce a meaningful ranking. A number of techniques was deployed in order to address this - first of all, it was essential to introduce regularisation - this was achieved by using the `weight_decay` parameter in Adam and resulted in a significant improvement in the validation loss. I also found that using drop-out was essential to help with generalisation - I found that dropout of about 0.5 worked best. The learning curves were initially very stochastic and it was only when the learning rate was made very small - 0.00001 - that smooth loss curves were obtained. Comparing the performance of both, Adam and SGD, I found that Adam was by far superior. Additionally, I experimented with some positive oversampling, increasing the number of positive examples in the training dataset, and this gave a large improvement in generalisation loss. Finally, in order to find the best architecture, I performed a grid search over both, 2 and 3 hidden layer architectures, and found that using `64-128-64` hidden neurons in a 3 hidden layer network gave best results. Lastly, I run a small hyperparameter grid search over Adam parameters, specifically over the learning rate and the weight decay, and found that a learning rate of 0.00001 and L2 of 0.08 performed best.

We find that the best model achieved the following results on the Validation Set: **mAP:** 0.456 **nDCG:** 0.410

We find that the feed-forward MLP performs better than logistic regression but worse than LambaMART, which is what we expected. This is because an MLP optimises a pointwise objective - cross-entropy loss with softmax, looking at the problem as though it was a classification problem, rather than a ranking one. As I decided to compare the performance of FastText embeddings over the different models, there were no changes in feature representation for this question. However, a potentially much more powerful model for this problem would be to use passage and query representations where each word is explicitly represented, as there is a significant degree of information loss by averaging the representations, using a model capable of dealing with variable-size inputs, such as a Recurrent Neural Network (RNN).

Code: The implementation of the MLP can be found in `Question_4.ipynb`

5 APPENDIX: INSTRUCTIONS TO RUN THE CODE

Due to the large size of the files, all of the notebooks have been run in Google Colaboratory - the notebooks can be accessed using this link: https://drive.google.com/drive/folders/1sbZZ3rFD6CIxYqpB4_FBNsXO3yT9m25W?usp=sharing.

The copies of the Notebooks for Questions 1-4 have also been submitted, however the data files have not, due to the size being too large for Moodle Upload. I highly recommend running the notebooks on Colaboratory, rather than locally, as a minimum of 25GB RAM is required for many of them, especially for the `Generate Embeddings Dataset.ipynb` notebook.

Running the code on Colab:

- Click 'Add to Drive'
- Change the `PATH` variable to directory where the folder is stored in your drive
- You should then be able to run the code with no further amendments.
- Some of the data and experiments have been pre-processed and the cells contain `True \ False` Statements at the top - should you wish to run a given experiment or pre-processing from scratch, select True. Otherwise, keep the setting at False and a pre-computed file will be loaded from the repository.

Running the files locally:

- In order to generate the embeddings you can either run the entire `Generate Embeddings Dataset.ipynb` notebook, selecting all of the optional statements to True - it took 7-8 hours with 25GB RAM on Colab. Alternatively, you can download the 'data' folder from Colab onto your computer and save it to the Home repository.

REFERENCES

[1] Chris J.C. Burges. 2010. *From RankNet to LambdaRank to LambdaMART: An Overview*. Technical Report MSR-TR-2010-82. https://www.microsoft.com/en-us/research/publication/from-ranknet-to-lambdarank-to-lambdamart-an-overview/