# IRDM Course Project Part II Report

Anonymous ACM Submission

**ACM Reference Format:**
Anonymous ACM Submission. 2022. IRDM Course Project Part II Report. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

This report details the experimentation, model/code setup and results of deliverables generated from Coursework 2 of the COMP0084 Information Retrieval and Data Mining module.

### 1.1 Code Instructions

Please run the scripts in the sequence of the deliverables as files are saved and re-used across tasks. The pip modules and neural network libraries used are listed in *requirements.txt* and should be installed. SpaCy NLP library is used to carry out pre-processing and tokenization of the corpus as well as NLTK's stopwords list import to remove stopwords, both of which require further installation instructions. A distilled FastText *'embeddings.pkl'* file that contains the vector embeddings must be downloaded. Please refer to *instructions.txt*.

## 2 TASK 1 - EVALUATING RETRIEVAL QUALITY

### 2.1 Pre-processing Choices and Justifications

For this task we outline text pre-processing techniques that will be used henceforth for all other tasks. The NLP library used is spaCy which provides very accurate pre-trained tokenization and tagging of tokens with various attributes.

The corpus was first converted to lowercase as typically during query comparison, our vocabulary should be case-insensitive to allow more accurate document ranking. Using spaCy tokenizer, the entire corpus across the training and validation data was tokenized. During this we remove all tokens with the following identified attributes: [punctuation, URLs, emails, numbers, currencies, non-ascii] as these will not typically be queried by a user, or do not contribute to an official vocabulary of English words. Further to this, analysis of part-of-speech (POS) saw removals that may have not been caught out earlier, such as punctuation that may have merged into word tokens, or tokens that spaCy does not recognize coming from the English language. Note, however, tokens found within quotes were still kept as there could be queries by the user of quotes that may not be entirely English but are still relevant.

Finally, all tokens are lemmatized to remove inflectional endings and help return a base dictionary word. This was decided as during ranking whether that be in BM25 or the Neural Network models, these tokens will be embedded and/or intersected between queries and passages. This means that for similarity, tokens with the same base dictionary word that may have different inflexions **should** be classed as being matching to aid ranking. Furthermore, stemming was not applied as stemmers do not take into account the context of words and therefore do not discriminate well between words of similar meanings. A great example is that 'better' lemmatizes to 'good' but stems to 'bet' which is incorrect in the stemming case. Stopwords were also removed as these could negatively impact retrieval quality since many passages and queries will contain them, therefore inducing inflation on similarity of query-document pairs. **Note**: Certain queries/passages became 0 length after processing (e.g URL only passages) and hence did not appear in the final rankings (Q3 and Q4).

### 2.2 Implementation of Mean Average Precision and NDCG

For each task, we compute and store the final, sorted scores in a score matrix which is typically of the form [qid, pid, score, relevancy], where the matrix is extended by the ranking results of a particular task as the query at a given time is computed. From here the implementation of the metrics at a given rank **k** is straightforward.

**mAP@k**
The mean Average Precision (AP) is the mean of all AP's over the queries. The mAP@k is the mAP but with ranking only considered up until the k-th ranked document. Mathematically, it is given as:

$$AP@k = \frac{1}{N(k)} \sum_{i=1}^{k} \frac{TP_{seen}(i)}{i} \tag{1}$$

where,

$$N(k) = min(k, TP_{total})$$

$$TP_{seen}(i) = \begin{cases} 0 & if\ d_{rel}(i) = 0 \\ cumulative\ TP\ at\ i & if\ d_{rel}(i) = 1 \end{cases}$$

At a chosen rank, k, the AP is the cumulative sum of True Positives (relevant documents) seen till rank k, divided by k. This is then divided by either the total TP's seen over all ranks, or k if the total TP exceeds our @k, meaning we do not count TP's outside the range of k we care about. If a documents relevancy is 0, it does not contribute to the AP calculation. This is vectorised in the code as we abuse the fact relevancy $\in \{0, 1\}$ where we compute the cumulative sum of relevancy across sorted rankings and divide each rank's cumulative sum by the rank. The computation is then just that of Eq. 1. Sometimes, when no relevant documents occupy the top ranks, the precision calculation raises zero-division problems ($\frac{0}{0}$). This is dealt with by converting the resultant NaN value to 0.

**mNDCG@k**
Discounted Cumulative Gain (DCG) uses graded relevance as a measure of usefulness. Gain is accumulated as the ranks decrease but are discounted for relevant documents being placed lower. We then normalise this by the optimal DCG, which is the DCG of the best attainable ranking, one where all relevant documents occupy the top ranks of the scores to obtain the NDCG. Similar to before, this is averaged over all queries to quantify the ranking performance of our system. It is calculated as such:

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

where,

$$NDCG_k = \frac{DCG_k}{optDCG_k}$$

To accomplish this with the scoring matrix, first the gain (numerator) is computed by converting the relevancy of the ranked documents, and this is discounted by the rank-discount at the respective ranking. The optimal DCG is computed by counting the total number of relevant documents related to the query. We then take the $min(k, T_{rel})$, as DCG@k, can only account for a maximum of k relevant documents. We abuse the fact relevancy is binary once again and construct an array of ones of size of the minimum and compute the DCG on that, as this mimics the best possible ranking with relevant documents in top ranks. The NDCG@k is then computed using Eq. 2 and averaged across all queries.

## 2.3 BM25 Implementation and Evaluation Results

BM25 does not require any training to act as a scorer for ranking and as such we run it directly on the validation dataset to compute our metrics. The implementation is similar to that of the previous coursework, however, we now have relevancy information for each qid-pid pair and incorporate that into the BM25 equation:

$$= \sum_{\forall i \in Q \cap D} log\left[\frac{(r_i + 0.5)/(R - r_i + 0.5)}{(n_i + -r_i 0.5)/(N - n_i - R + r_i + 0.5)}\right] \cdot \frac{(k1+1)f_i}{K + f_i} \cdot \frac{(k_2+1)qf_i}{k_2 + qf_i} \tag{3}$$

where N is the size of the corpus, $n_i$ is the term frequency within the corpus, $r_i$ is frequency of relevant documents containing the term, $f_i$ is the document TF, $qf_i$ is the query TF and R is the total relevant documents pertaining to the query. The other variables are pre-set constants given as $k1 = 1.2$, $k2 = 100$, $b = 0.75$ and K is calculated from the ratio of the document length to the average document length. *The rankings can be found in bm25.txt.*

**Table 1: Evaluation metrics for BM25 model.**

| | BM25 | | | | | |
|---|---|---|---|---|---|---|
| | mAP | | | mNDCG | | |
| k | @3 | @10 | @100 | @3 | @10 | @100 |
| Metric Value | 0.2 | 0.236 | 0.25 | 0.218 | 0.294 | 0.369 |

Table 1 shows that BM25 performs well across our evaluation metrics. The NDCG scores increase as k increases to a peak of 0.369 which compared to later models is quite impressive. The AP also increases as k increases but has a lower magnitude at 0.25, meaning the model did not prioritise the relevant documents to be in the top rankings as well as it could have. However, with relevant documents only making up 0.1% of the validation set, this performance is still good for a baseline.

# 3 TASK 2 - LOGISTIC REGRESSION

## 3.1 Dataset Pre-processing and Word Embedding

**Training Dataset Sampling**

Due to the large size of the training dataset, to trade-off computation time for more training points, a sampled subset of the training dataset is used henceforth for all future models (LR, LM and NN). The sampling technique is simple; 1 million entries from the training set are to form the subset. Due to large class-imbalances and to make the most of the positive examples for learning, all relevant documents in the training set are preserved in the subset, with the remaining spots filled by sampling uniformly at random entries from the non-relevant documents, $d_{rel} \neq 1$. Shuffling was performed at each stage and *the sampled subset can be found in 'sampled_train_set.csv'.*

**Query and Passage Word Embeddings**

Before embedding the query and passage entries for the datasets, each dataset is pre-processed using the same tokenization method outlined in Section 2.1. This helps clean the inputs by removing undesired attributes and through the lemmatization, helps reduce inflexional endings on our tokens for better quality retrievals and embedding.

Embedding is carried out using the pre-trained FastText English model. We are representing words as vectors as to capture hidden information surrounding the semantics of our inputs and as such using a pre-trained model on a very large corpus such as Wikipedia leads to high-quality embeddings and can help us with high-quality ranking. FastText is especially good at dealing with OOV terms unlike other word models as words are represented by the sum of their substrings, therefore we minimise loss of information when embedding.

Two hashmaps for queries and passages were created, which map each query/passage to its tokenized form. Each token in each mapping is then embedded into a feature vector $\mathbb{R}^{300}$ using FastText and we use the idea of centroid average term embeddings [1] to average the term embeddings over the query/passage token vectors:

$$\overrightarrow{v}_d = \frac{1}{|d|} \sum_{t \in d} \frac{\overrightarrow{v_t}}{\left\|\overrightarrow{v_t}\right\|} \tag{4}$$

This is repeated for the training and validation sets and saved so that the vectors can be used for future tasks.

## 3.2 Logistic Regression Implementation and Evaluation Results

**Training Details**

The logistic regression model is quite simple in implementation. Our parameters follow our input dimensions and are $\{w \in \mathbb{R}^{300}; b \in \mathbb{R}^1\}$. The model predicts using a sigmoid activation on the regressive prediction $\hat{y} = \sigma(\mathbf{w}^\mathbf{T}\mathbf{x})$, and $\sigma(\cdot) = (1 + e^{-\cdot})^{-1}$. Mini-batch gradient descent is used for updating our parameters to give less noisy updates and faster convergence. The model loss function is the cross-entropy loss, given as:

$$L_{cross} = -(ylog(\hat{y}) + (1-y)log(1-\hat{y})) \tag{5}$$

In which to update via mini-batch SGD, we apply the partial derivative of the loss w.r.t our parameters which gives us an update of:

$$\Delta\theta_j = \frac{\delta L_{cross}}{\delta \theta_j} = \frac{1}{N}\sum_{i=0}^{N}(\hat{y}_i - y_i)x_{i,j} \tag{6}$$

$$\begin{aligned} w &\leftarrow w - \alpha\Delta\theta_w \\ b &\leftarrow b - \alpha\Delta\theta_b \end{aligned} \tag{7}$$

where $\alpha$ is our learning rate for the gradient updates. For even better optimisation, a simpler version of the Adam optimiser is used to apply our gradient updates to our parameters. Adam uses adaptive gradient rescaling [2] to reduce noisy updates and help avoid local minimas. The optimiser keeps track of the first and second moment of our gradient updates and on each update, these moments too are updated. On a new update, $\Delta\theta_{grad}$ we first update our moments:

$$\mu \leftarrow (1-\beta_1)\Delta\theta_{grad} + \beta_1\mu$$
$$\nu \leftarrow (1-\beta_2)(\Delta\theta_{grad})^2 + \beta_2\mu$$

then the final gradient update is:

$$\Delta\theta_{final} = \alpha\frac{\mu}{\epsilon+\sqrt{\nu}} \tag{8}$$

where $\beta$ defines a soft horizon for the per-weight moments, $\epsilon$ increases robustness of the updates to numerical issues and these statistics are computed for all model parameters. For simplicity, we hard-code set $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-4}$ and this forms our Adam update step.

During training we also keep track of metrics to evaluate convergence of our model. For this we track the loss of each final update before an epoch ends, as well as the accuracy, precision, recall and F1-score. The model batch-size was optimally found to be best when set to 128 as we trade-off epoch train times for less noisy updates.

### Class Imbalance and Input Formatting

Input formatting into the model took two stages. Firstly, the training data still had large class imbalances which Logistic Regression just cannot deal with. There are two ways to tackle this: weighted LR looks to weight loss contributions to be higher for the under-represented class label. The second way is to perform multiple updates for the under-represented labels in comparison to the over-represented ones, which takes the form of duplicating the under-represented entries of the dataset until the desired ratio is attained. Both ways were trialled and both had similar effects but for simplicity the data duplication approach was used. To balance with over-duplicating entries, the ratio of relevant to non-relevant labels was set empirically to 1:2 so that the relevant labels comprise at least $\frac{1}{3}$ of the training set.

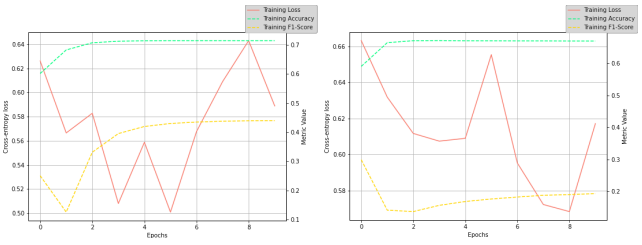


**Figure 1: Figures showing the comparison of cosine similarity (left) vs. concatanation (right) of inputs. Key: Loss = orange, Training Accuracy = green, F1-score = yellow.**

Secondly we consider the vector input format. Figure 1 shows two methods for inputs into the model that were trialed; first the input was a simple cosine similarity score between the query and passage vectors which performed okay but the F1-score saturate below what was expected as the model was too simple and loss started increasing towards the end of training. The second method concatenated the query and passage vectors which lead to a very low F1-score but smoother update graph possible due to very high feature dimension of 600. The best method found was to average the query and passage vectors and also concatanate the cosine similarity score to obtain an input vector $\in \mathbb{R}^{300+1}$ which combines the good F1-score of the cosine similarity feature and also increases the dimension and parameterisation of the model.

### Learning Rate Analysis

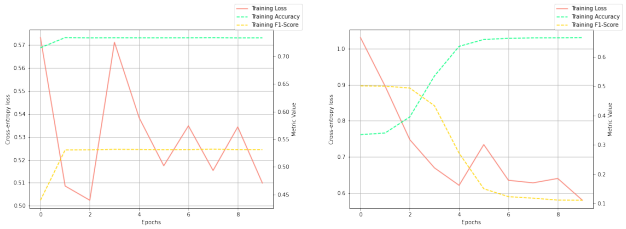


**Figure 2: Figures showing the comparison of 1e-1 (left) vs. 1e-5 (right) of inputs. Key: Loss = orange, Training Accuracy = green, F1-score = yellow.**

The LR model was trained at learning rates of $\{10^{-1}, 10^{-3}, 10^{-5}\}$ for 10 epochs as the model has little updates beyond this. Large learning rates like $10^{-1}$ meant quicker convergence of the F1-scores but very noisy updates as the loss spikes drastically and never goes below 0.5 meaning the parameters are taking step sizes large enough to constantly over-shoot the minima. Very small step sizes like $10^{-5}$ had much smoother updates but the very small step size lead to decreasing F1-score during training and in general much slower convergence with the loss not decreasing below 0.6. The best learning rate was around $10^{-3}$ which converged quickly within around 4 epochs but also did not exhibit very noisy updates and reached the lowest loss of around 0.47. This learning rate was used going forward for evaluation on the validation set.

### Evaluation Results

**Table 2: Evaluation metrics for Logistic Regression model.**

| | Logistic Regression | | | | | |
|---|---|---|---|---|---|---|
| | mAP | | | mNDCG | | |
| k | @3 | @10 | @100 | @3 | @10 | @100 |
| Metric Value | 0.078 | 0.097 | 0.109 | 0.084 | 0.124 | 0.193 |

The final LR model performed worse than the BM25 model with a maximal mNDCG@100 and mAP@100 observed to be 0.193 and 0.109, respectively. This is expected due to the point-wise objective function optimisation which only looks to maximise accuracy of scores over individual qid-pid pairs, not ranking list-wise over the qid's candidate passages. *The ranking results can be found in 'LR.txt'.*

| Feature # | Description | Feature # | Description |
|---|---|---|---|
| 0 | Total TF of matching query terms $q \cap d$ in passage | 8 | Cosine similarity score of TF-IDF vectors between pair |
| 1 | Ratio of Total TF of matching query terms to passage length | 9 | Language model Dirchlet Smoothing score |
| 2 | Sum IDF of query terms $q_i \in q$ | 10 | Length of query term |
| 3 | Sum IDF of passage terms $p_i \in p$ | 11 | Length of passage |
| 4 | Sum TF-IDF of matching query terms in this pair for query | 12 | Ratio of length of query terms to passage terms |
| 5 | Sum TF-IDF of matching query terms in this pair for passage | 13 | Ratio of number of unique matching query terms in passage to passage length |
| 6 | BM25 score for this pair | 14 | Ratio of number of unique matching query terms in passage to query length |
| 7 | Cosine Similarity of average term embeddings between pair | - | - |

**Table 3: Feature vector descriptors for manual feature engineering.**

# 4 TASK 3 - LAMBDAMART

## 4.1 Dataset Pre-processing and Feature Engineering

LambdaMART is a pairwise ranking model that is built upon boosted decision trees architecture. For this task we build our own feature-set of inputs via manual feature engineering on the query-passage pairs. Once again, dataset pre-processing is the same as previous tasks. Referencing Microsoft's Learning to Rank Datasets [3], and the hand-engineered features they used, our qid-pid pairs were each assigned a feature vector $\in \mathbb{R}^{15}$. The selected features are shown in Table 3.

It is not expected that every feature will contribute tremendously to the model performance, however we select features that are unique to the qid-pid pair to allow our LambdaMART model to extract underlying patterns in the decision splits and potentially give good ranking performance. *The constructed feature sets for both training and validation sets were saved to 'p3_features.csv' and 'p3_features_test.csv' respectively.*

## 4.2 Model Training and Hyperparameter Tuning

Before training can occur the features of our inputs were first normalised to be zero-mean with standard deviation of 1. This helps scale all features to be similar such that no one feature's gradient explodes in comparison to another and allows for better convergence. The features were normalised along the feature axis in accordance with $x_i = \frac{x_i - \mu_i}{\sigma_i}$.

Furthermore, to enable ranking via XGBoost we must set the objective function to be 'rank:pairwise' and this means we are enabling pairwise ranking objective which follows LambdaMART. This also means when we construct the DMatrix input into the model we must be careful to set the groupings for each query, else the model will attempt to rank across multiple queries. With this we perform hyperparameter tuning over the maximum depth of a single tree in our ensemble and the boosting learning rate $\eta$.

To find the best set of hyperparameters 3-fold cross-validation was performed over a parameter search space. Cross-validation uses parts of the training set in a given moment to train on, while evaluating on a held-out portion of the training set. This aims to eliminate bias during the hyperparameter search stage. 3-fold CV takes the training set query IDs and splits these into 3 folds, where in each iteration one fold is used for validation while the other two are used for training. The search is performed over all folds and the best parameter found in each fold is averaged and returned. With these best identified parameters, the final model is trained over the entire training set and evaluated against the validation set. The search space comprised of $depth \in 2, 3, .., 6$ and $\eta \in 0.01, 0.14, ..., 0.8$. Other parameters were kept as default.

## 4.3 Evaluation Results

**Table 4: Evaluation metrics for LambdaMART model.**

| | LambdaMART | | | | | |
|---|---|---|---|---|---|---|
| | mAP | | | mNDCG | | |
| k | @3 | @10 | @100 | @3 | @10 | @100 |
| Metric Value | 0.198 | 0.238 | 0.251 | 0.218 | 0.300 | 0.371 |

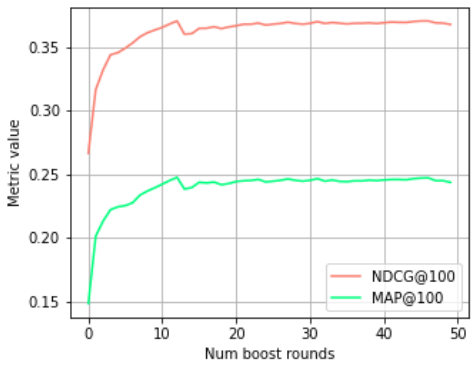


**Figure 3: Test-set evaluation metrics during training.**

From cross-validation, the best found parameters were depth = 4 with a learning rate, $\eta = 0.67$. The final model was trained under these settings and the evaluation metrics over the boosting rounds are shown in Figure 3. Table 4 shows that LambdaMART scores a peak mAP and mNDCG of 0.251 and 0.371, respectively, beating out BM25. This is not surprising given the BM25 score is itself a feature in our model, and paired with further qid-pid features aids to increase model performance.

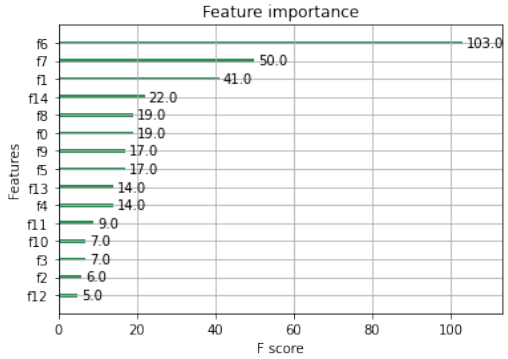


**Figure 4: LambdaMART feature importance scores.**

Conducting further analysis, we can visualise the feature importance our LM model places over our data in Figure 4. Indeed LM considers the BM25 feature to be of great weighting when performing decision slices. It also finds the cosine similarity of the query-passage embedding vectors and the percentage of passage terms that are found in the query to be important features also. The model seems to find many of the features to contribute to ranking, meaning the chosen features were quite good and the model performance speaks to that, although further improvements could see feature ablation and cross-validating over more parameters and search ranges. *The ranking results can be found in 'LM.txt'.*

# 5 TASK 4 - NEURAL NETWORK MODEL

## 5.1 Neural Network Library Choice

The NN library of choice is TensorFlow (TF) for two reasons: one, the author has prior experience with the library, secondly, TF has the TF Ranking (TFR) and TF Recommenders (TFRS) extensions that deal specifically with Learning-To-Rank tasks with ranking based loss functions such as Approximate NDCG Loss, and ranking based metrics to track model performance during training, hence made a good choice to build our ranking model from.

## 5.2 Context-Absent Neural Ranking GAM

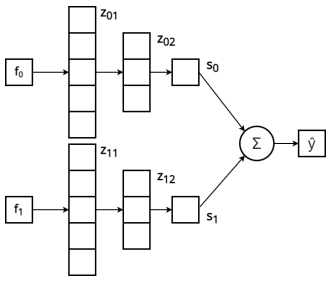


**Figure 5: Context-absent GAM architecture illustration (2-feature version).**

The model of choice for this task was the context-absent neural ranking Generalised Additive Models (GAM) proposed in the "Interpretable Ranking with Generalised Additive Models" paper [4]. The paper proposed a version of GAMs that are able to perform ranking tasks and comprise of simple piece-wise linear functions that are efficient and straightforward to build.

The formulation is simple. Our ranking data, $\mathfrak{D} = \{\{\mathbf{X}, \mathbf{y}\}\}_1^N$, comprises sets of qid-pid feature vectors for N queries in the training set. The GAM acts as a scorer in our final model. Figure 5 shows the network. Each feature in a qid-pid example $\{f_0, f_1, ..., f_K\}$ is fed to an independent feed-forward feature sub-network that scores that feature. For each hidden layer $i \in \{1, ..., H\}$ in the sub-network, we compute

$$\mathbf{z}_{j,i} = \sigma(\mathbf{W}_{j,i}\mathbf{z}_{j,(i-1)} + \mathbf{b}_{j,i}), \forall j \in \{1, ..., K\}, i \in \{1, ..., H\} \quad (9)$$

where $\mathbf{z}_{j,i}$ is the output of the i-th hidden layer for the j-th feature with $\mathbf{z}_{j,1} = \sigma(\mathbf{W}_{j,1}f_j + \mathbf{b}_{j,1})$, the feature input itself; $\mathbf{W}_{j,i}$ and $\mathbf{b}_{j,i}$ are the weight matrix and bias vector for the i-th hidden layer and finally $\sigma(\cdot)$ is a non-linear activation function, RELU in our case. The score of a sub-network is a final dense projection layer of the last hidden layer

$$f_j(x_j) = s_j = \mathbf{W}_j\mathbf{z}_{j,H} + \mathbf{b}_j$$

where we do not apply an activation function as we want a pure logit output. Finally, the scoring function sums the sub-scores across each sub-network to give the score of that qid-pid vector

$$\hat{y} = F(\mathbf{x}) = \sum_j s_j \quad (10)$$

For the problem at hand we wish to directly compare the performance of the traditional tree-based learning algorithms such as LambdaMART, to the supposedly more powerful Deep Learning based models. For this, the same feature set of qid-pid pairs generated in Section 4.1 (Table 3) were used. Since we have no given 'context', that is, the query and passage embeddings are not separate but merged, this architecture was chosen as it supports context-absent ranking, in which the query weighted sum simply becomes a sum over the feature sub-networks. Possible problems with the model could be that feature sub-networks are independent of each other when scoring the input. This lack of feature-interaction could compromise model performance [4].

**Building the Ranking Model**

In TFRS, the model structure typically includes two main components: a scoring model, which we use the GAM, and a loss model, which we instantiate a TFRS ranking task for. The GAM model was constructed empirically (although more experimentation on this is presented in Section 5.5) under TFRS recommendations with each sub-network comprising of 2 hidden layers of 32 and 16 dimensions, respectively. The TFRS ranking layer takes in a score list and the corresponding relevance label, then under the user-set loss function and metrics will compute the loss value for the ranking and also update training metrics. We essentially wrap our GAM model with this ranking task and backpropagate the actual gradients of the scorer w.r.t the chosen ranking loss function. The model is compiled with the Approximate NDCG Loss, a differentiable NDCG based loss function, and track the mAP and NDCG across the entire ranking set. (Later we will evaluate using the standard @3, 10, 100 like before). A custom ranking model class was used to build the final model which a custom train step and model instantiation was created to

facilitate proper training. *Please see "task4.py" for implementation details.*

## 5.3 Dataset Pre-processing and Feature Representation

We use the same feature dataset created for LM and once again normalise the inputs to be zero-mean and with singular standard deviation. Since we are performing ranking, a custom TF Dataset wrapped generator was created that yields an entire query's candidate qid-pid pairs per step. This enables ranking during training as each pass through of the network only computes the listwise loss for that given query set. The inputs were also padded with zeros to ensure all query sets fit 1000 examples, so that no ragged tensors appear during training and evaluation. The training set was split into a train/val set again using an 80/20 split as before, with validation evaluation performed at the end of each epoch.

## 5.4 Training Results and Model Evaluation

The first experiment looked to identify the best hyperparameters for the NN model. In general, the main hyperparamters we can control for the model are the type of optimiser used, the learning rate for that optimiser and the number of epochs to train for. Since we are not batching inputs we do not consider batch size. The TFRS tutorial recommends Adagrad, however preliminary testing showed that Adam performed better under default configurations and training for 3 epochs with both optimisers converging quickly but Adagrad scoring a test-set mNDCG of **0.377** while Adam scored a test-set mNDCG of **0.384**. Therefore Adam was chosen going forward.

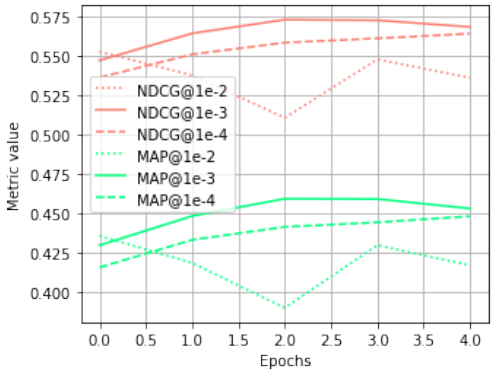


**Figure 6: NDCG and mAP of validation set for different learning rates during training.**

Figure 6 shows the comparison between different orders of learning rate from 1e-2 and 1e-4. All models were trained for 5 epochs. We can see the best learning rate occurs around 1e-3 with the optimal epoch count being 2 epochs before the validation metrics begin declining.

The final model was then trained on the entire train set for 2 epochs at 1e-3 as identified above. The reason for the low epoch count is due to the fact the model is updated with frequency equivalent to the number of queries in the train set for each epoch, and so only a few cycles are needed to converge.

**Table 5: Evaluation metrics for Neural Network model.**

| | Neural Network | | | | | |
|---|---|---|---|---|---|---|
| | mAP | | | mNDCG | | |
| k | @3 | @10 | @100 | @3 | @10 | @100 |
| Metric Value | 0.221 | 0.261 | 0.272 | 0.240 | 0.323 | 0.392 |

Table 5 shows the Neural Network model outperforms all previous models for **all** evaluation metrics against the test-set! It even defeats LambdaMART which was previously the best model for ranking. This could be due to the GAM DNN being able to score qid-pid pairs more accurately through its learned weighting when optimising for maximal NDCG scores.

## 5.5 Further experimenting with GAM Models

In this experiment we trial a wider and deeper GAM scoring model. The configuration was updated to feature 4 dense layers with new output sizes of [256, 128, 64, 32], feeding into the final scoring dense logit layer. The same optimal parameters were used as before.

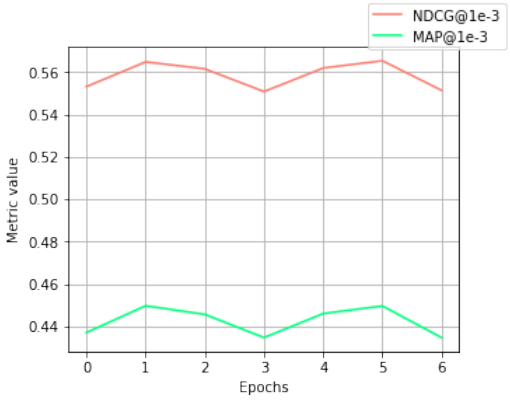


**Figure 7: NDCG and mAP of training set with deeper scoring model.**

Figure 7 shows that the deeper model has no beneficial impact on model performance, in fact the metrics appear to be worse in comparison to the shallow GAM scorer. Experiments were also run which weighted the feature sum with the feature importance weightings generated from the LambdaMART model, but no gain in performance was observed. Gradient accumulation (GA) was also trialed due to the notion that our model weights are only updated once per query, almost enforcing a SGD setting. GA would enforce mini-batch updating but this also limited performance with a peak NDCG observed at **0.294**. *Both LM-enhanced and GA model classes can be found in 'nn_further_experiment_models.py'.*

In conclusion the best model was found to be the NN model, followed by LambdaMART, BM25 and finally Logistic Regression.

## REFERENCES

[1] E. Nalisnick, B. Mitra, N. Craswell, and R. Caruana, "Improving document ranking with dual word embeddings," 2016.
[2] K. et al., "Adam: A method for stochastic optimization," 2014.
[3] T. Q. et al., "Introducing LETOR 4.0 datasets," *CoRR*, vol. abs/1306.2597, 2013.
[4] Z. et al., "Interpretable learning-to-rank with generalized additive models," 2020.