Information Retrieval and Data Mining: Coursework 2

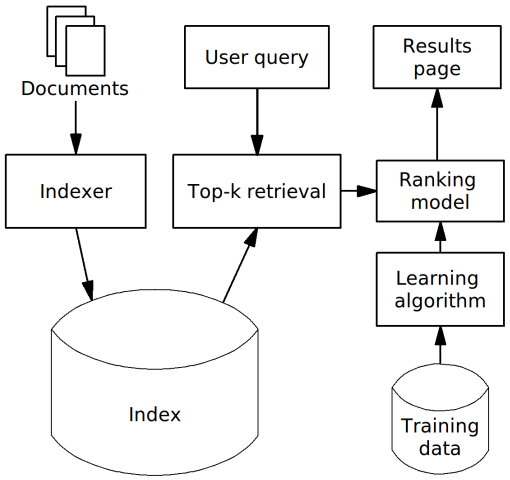

Figure 1: Flowchart for Learning to Rank (Source:Wikipedia)

ABSTRACT

The main idea in this coursework was to try different methods to rank documents based on queries and understand how the method works. This report contains answers to multiple tasks given as an individual coursework for IRDM. It contains explanations on various decisions taken while going through the tasks.

KEYWORDS

ndcg, average precision, dcg, logistic regression, lambdamart, xgboost, neural networks, learning to rank

ACM Reference Format:

. 2018. Information Retrieval and Data Mining: Coursework 2. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

With more and more data being collected, automatic information retrieval systems are becoming really important in every field. New techniques and methods are being developed to give more relevant data on retrieval. The basic process of ranking in Information Retrieval is shown in figure 1. Documents are indexed and stored. The user query is used to get top-k documents for a specific query. These k documents are sent to a ranking model that has been trained on similar data with help of a learning algorithm. After ranking, the results are displayed to user on a results page in a specific order. In this report, I've used similarities based on query and document embedding to rank documents when certain query is given. These rankings have been compared using Normalized Discounted Cumulative Gain (NDCG) and mean Average Precision (mAP). I've used cosine similarity, document length and query length as features for Logistic regression, LambdaMART and Neural Network models.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

2 TASKS

2.1 Evaluating Retrieval Quality

In this task, we were expected to calculate NDCG and Average precision for BM25 model that was implemented in Task 1.

2.1.1 Preprocessing. For pre-processing, I made a function to remove punctuation followed by tokenizing passages, converting them to lower case. Any token, that is a number has been removed.For this coursework, we will be making embedding for documents and queries, so having punctuation won't make much sense, that is why it has been removed. Words like "RNA" and "rna" are same, but a computer would treat them different and we'll have two embeddings for it, instead of one. To avoid such scenario, all words are converted to lowercase. After tokenization, any token that is a number makes no sense in uni-gram, it's just noise. It would make sense if there was context (eg. in bi-gram, tri-gram etc.). At last the stop words like "is","the","are" etc. are removed as they just increase size of data and don't add that much information.

2.1.2 BM25: A probabilistic model. BM25 is a probabilistic model which is based on binary independence model. I've used parameters that were pre-defined in coursework 1 of this module. As done previously, I've ranked the documents based on their ranks for each query. This time for calculating metrics I'm using all of the "validation data" instead of just top 100 passages. This is only done for Task 1.

2.1.3 NDCG: Normalized Discounted Cumulative Gain. NDCG is a popular metric for checking quality of retrieved results from a search. As term suggests, NDCG is a normalized form of DCG. The NDCG metric is interpreted as extent to which ranking of model is in agreement to ideal ranking. The formula for DCG is:

$$DCG_p = \sum_{i=1}^{p} \frac{2^{rel_i} - 1}{log_2(i+1)}$$

Figure 2: Discounted Cumulative Gain Formula

In the formula given in Figure 2, "rel" is the relevance. The numerator in DCG is called "Gain" and the denominator is called "Discount". Here, I'm using non-linear gain formulation for calculations.
The NDCG is given by:

$$\mathrm{nDCG_p} = \frac{DCG_p}{IDCG_p}$$

Figure 3: Normalized Discounted Cumulative Gain Formula

In figure 3, IDCG is means Ideal DCG. This means we normalize DCG against the best possible DCG we can get i.e. the perfect ranking for query. This is useful because NDCG is between 0 and 1 but DCG has no upper bound.

For the validation set, I first calculate BM25 scores for the whole data and then rank everything by BM25 scores. Using these ranks (calculated) and relevance (provided in data), I calculate NDCG for all the queries. Then, I divide the sum of all ndcg by number of queries (unique) to get average NDCG for BM25 ranking which comes out to be 0.3814

2.1.4 Average Precision. Average Precision means average of precision of relevant documents. The higher its score, the better is scoring. The precision is given by:

$$Precision = \frac{TP}{TP + FP}$$

Figure 4: Precision Formula

In figure 4, TP means true positive which in our case means relevant document. FP means not relevant for us. Thus, precision is calculated for each data point. To get average precision for each query, the sum of these precision (precision for all documents in one query) is divided by total number of relevant documents available for query. We have to obtain mean Average Precision for dataset which is given by formula in Figure 5.

$$\boldsymbol{mAP = \frac{1}{n}\sum_{k=1}^{k=n} AP_k}$$
$$\boldsymbol{AP_k = \text{ the AP of class } k}$$
$$\boldsymbol{n = \text{ the number of classes}}$$

Figure 5: Mean Average Precision Formula

In figure 5, classes mean unique queries for us. Thus, to get mean Average precision, we divide sum of all average precisions by total number of unique queries.
The mean Average Precision for BM25 ranking on validation set is 0.2434

2.1.5 Results. The code for this task is in file task1.py or task1.ipynb. For validation set, NDCG = 0.3814 and Average Precision = 0.2434

2.2 Logistic Regression (LR)

Logistic regression is a famous generalized linear model (GLM) in statistics which is used for classification tasks in machine learning. This algorithm models the probability of a discrete outcome given input variable(s). In our case, we are using binary outcome model as our target variable is relevancy which outputs 1 or 0 based on if document is relevant or not, respectively.

$$\ln\left(\frac{P}{1-P}\right) = a + bX$$
$$\frac{P}{1-P} = e^{a+bX}$$
$$P = \frac{e^{a+bX}}{1+e^{a+bX}}$$

Figure 6: Logit link function to logistic function

In Figure 6, the first equation is a logit function, which is mapped to the linear model. The final equation is known as logistic function. The regression equation is given by:

$$Ln\left(\frac{P}{1-P}\right) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_k X_k$$

Figure 7: Logistic Regression Equation

In Figure 7, the LHS is known as "odds-ratio". This means that the more this ratio is the more is the probability for event to occur. If this ratio is greater than 0.5 (generally accepted threshold value, unless we have domain knowledge), I considered that document is relevant while less than 0.5 means document is not relevant. The odds ratio is considered as scoring criteria to rank the documents. While the labels for relevant, thus calculated are used in calculation of metrics.

2.2.1 Datasets. We are given two datasets- Training data and validation data. Due to memory constraints, I'm doing a negative sampling on the training data throughout the coursework and will be using whole validation data for testing and reporting the scores and ranks of top 100 documents for each query in final '.txt' files. Also, there are some queries who don't have 100 documents, so for them, all documents available have been ranked.

2.2.2 Negative Sampling. In a nutshell, negative sampling assumes that if a token from document has never occurred together with a token from query which resulted in a relevant result, is not relevant. I've made a function that takes data and number of negative samples (k) we want as input and outputs a sample from data provided which has 1 example of relevant document for each query and 'k' not relevant documents. In further tasks, this function has been slightly modified to include 'r' positive samples instead of just 1.

2.2.3 Inputs/Features. Traditional L2R models include handcrafted features that encode IR insights. For this coursework, I'm using 3 features in total. One query dependent feature -query length , One query independent feature- document length and one dynamic feature- cosine similarity. I have used these features as I wanted to use at least 1 feature of every kind and explore them. I did make small changes in code to see whether they have an impact or not. It turns out, document length and query length are important as it's giving better generalization result.

2.2.4 Embedding. For this coursework, I'm using GloVe embedding - 50d.GloVe stands for Global Vectors. GloVe is an unsupervised learning algorithm for obtaining vector representations for words. These have been uploaded to the memory using Gensim package. Then, converted to Word2Vec and representation are extracted. Each passage and query of sampled training data and validation data, have first been pre-processed to get passage and query tokens. These 50d embedding for each of the tokens is then extracted from the package and averaged to get one 50d embedding for each query and passage. To make the process faster, I first store the embedding for each unique pid/ qid as key-value pair in a dictionary and then map it back to dataframe for ease. This process significantly improves the speed of whole process.

2.2.5 Logistic Regression Implementation. Once, I have the embeddings for the query and passage. I calculate cosine similarity for each pair. This becomes one of the features. The document length and query length are 2 other features used. Then, I make a class to implement logistic regression, which has basic function like init to initialise the object, fit to train the model and predict to finally predict the outcome variable using the parameters calculated during training phase. So, first, I made two arrays xTr and yTr corresponding to training data for X and Y. This is then passed through the fit functions to get the weights (or parameters/coefficient). The training process requires a choice of some hyperparameters like number of iterations and learning parameter. I have fixed iterations to 1000 as values didn't change much after that but better hyperparameter is surely possible.

2.2.6 Effect of learning rate. To test the model for different learning rates, I checked accuracy in the prediction of outcome variable. I did this due to time constraints. This isn't a really good metric here as the data is highly imbalanced and so oversampling or undersampling is required and metrics like precision or recall might give better results. But for now, I've just used accuracy. For the various models created, I found that as we increase learning rate the training accuracy obviously increases but the testing accuracy doesn't necessarily increasing. I believe this is the bias-variance trade-off here. Smaller learning rates give slightly lower training error but better testing errors as opposed to larger learning rates. For me, the best learning rate was 0.001.

2.2.7 Results. The code for this task is in file task2.py or task2.ipynb. The final file for results of top 100 documents for each query is 'LR.txt' which has the same format as specified in assignment pdf. For validation set comprised of top 100 documents for each query, NDCG = 0.139 and Average Precision = 0.016

2.3 LambdaMART Model (LM)

LambdaMART is a mix of LambdaRank and Multiple Additive Regression Trees (MART). MART used gradient boosted decision trees but instead LambdaMART used the rank function in LambdaRank as cost function. The overall LambdaMART algorithm is described below. This image has been taken from Microsoft Research website.

Algorithm: LambdaMART
set number of trees N, number of training samples m, number of leaves per tree L, learning rate η
for $i = 0$ to m **do**
  $F_0(x_i) = \text{BaseModel}(x_i)$ //If BaseModel is empty, set $F_0(x_i) = 0$
end for
for $k = 1$ to N **do**
  for $i = 0$ to m **do**
    $y_i = \lambda_i$
    $w_i = \frac{\partial y_i}{\partial F_{k-1}(x_i)}$
  end for
  $\{R_{lk}\}_{l=1}^{L}$ // Create L leaf tree on $\{x_i, y_i\}_{i=1}^{m}$
  $\gamma_{lk} = \frac{\sum_{x_i \in R_{lk}} y_i}{\sum_{x_i \in R_{lk}} w_i}$ // Assign leaf values based on Newton step.
  $F_k(x_i) = F_{k-1}(x_i) + \eta \sum_l \gamma_{lk} I(x_i \in R_{lk})$ // Take step with learning rate η.
end for

Figure 8: LambdaMART Algorithm

LambdaRank updates the weights for queries after examining them and the final decisions are computed using all data that falls to that node. Thus, LambdaMART updates few parameters at once using all data. This is useful because this enables LambdaMART to choose splits and leaf values that might be not as good for individual query but overall it's great.

2.3.1 Data. As above, here, I've used sampled training data. Though the sample here has been changed as now I chose to have more positive samples , instead of just one. Though the imbalance problem is still there.

2.3.2 Negative Sampling. As explained above, slight change is done to the negative sampling function to include more positive samples. Now, for each query I have 10 positive samples and 20 negative samples. I did this because I wanted to increase size of training data but I'm not able to use full data as it's too large. But, even after increasing the data we have less training data as compared to testing data.

2.3.3 xgboost LambdaMart - Input Processing. The xgboost library has many L2R algorithms. These are implement using objective. The objective for ranking algorithm is 'rank', this is followed by the metric we want to use. I chose NDCG metric as it involves list-wise ranking in which NDCG is maximized. So, the objective function that I'm using is 'rank:ndcg'. In xgboost library, we have special data containers called DMatrix. It's an internal data structure used by XGBoost. Also, there's a grouping attribute that needs to be set for training and testing data. This is nothing but a way of classifying that there are 'n' documents for a query. It is used to set group size of DMatrix which is important in ranking tasks. Once, I set the group for both training and testing (here, testing means the 'validation data' provided to us), I trained the model based on training data and evaluated it on validation data.

2.3.4 Hyperparameter Tuning. There are many hyperparameters in the model. This is because it is evolved from decision trees which have many parameters like leaf, node, pruning, max depth, min depth and many more. I just chose two parameters which I think affect the model more than the rest of the parameters. They are - Max depth and learning rate. I chose max depth because in a tree based model, this leads to overfitting (no max depth set) and underfitting (max depth is so low that training is not done properly). The reasoning with learning rate is also similar along with the fact that we always want better generalization which comes at cost of training error.
I tested various models, by fixing one of the two parameters and calculating the validation ndcg. From the experiments, I found that with increase in learning rate, validation NDCG doesn't necessarily increase every time. For max depth, validation NDCG did increase initially but then it started decreasing. I found that for my data the best hyperparameters are - eta (learning rate parameter) = 0.01 and max depth = 5.

2.3.5 Results. The code for this task is in file task3.py or task3.ipynb. The final file for results of top 100 documents for each query is 'LM.txt' which has the same format as specified in assignment pdf. For validation set comprised of top 100 documents for each query, NDCG = 0.201 and Average Precision = 0.067

2.4 Neural Network Model (NN)

Neural Networks are known for their amazing performance on all the tasks. These are really famous because they reduced the dependence on handcrafted features. They are able to make features on their own with the help of neurons involved in the process. A simple neural network is shown in figure 9.

2.4.1 Data. The data used here is same as task 3 data. Apart from slight changes in sampling.

2.4.2 Negative Sampling. This is same as task 3 meaning we have 10 positive samples and 20 negative ones. The new thing here is that I've used a package called imblearn which handles imbalanced data. Through this package, I've done random undersampling which will resample the majority class (here class 0: not relevant documents). This will help to get better results as now the positive samples and negative samples are comparable.

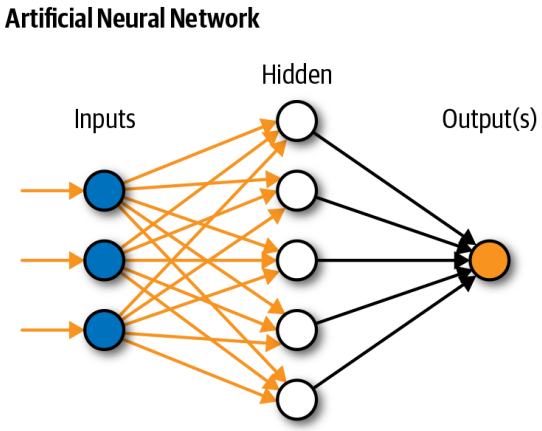

Figure 9: A simple neural network (source: Stackoverflow)

2.4.3 Model. For my network, I'm using a sequential feed-forward network which has non-linearity introduced by dense ReLU layers. There is dropout between layers to avoid overfitting. The summary for my model is shown in figure 10.

```
Model: "sequential_12"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_48 (Dense)            (None, 20)                80

 dense_49 (Dense)            (None, 20)                420

 dense_50 (Dense)            (None, 20)                420

 dense_51 (Dense)            (None, 1)                 21

=================================================================
Total params: 941
Trainable params: 941
Non-trainable params: 0
```

Figure 10: Model Summary

The final dense layer has a sigmoid activation to give predictions/outputs. The optimizer used is adam. This is because, firstly it works really well on almost all tasks and secondly, I tried using different optimizers but adam works the best. The loss function used here is binary crossentropy.

2.4.4 Metric. The metric used here is precision as it is unaffected by imbalance and is a good measure if we want to determine how many predicted relevant documents were actually relevant. This is what we want to maximize and thus I chose this as my metric.

2.4.5 Hyperparameter Tuning. Epochs and batch size were hyperparameters in this case. There are other hyperparameters as well but I am just choosing two. The batch size divides data in batches and parameters are updated after every batch is done. Epochs is the number of iterations of the whole process. I also did a validation split to choose hyperparameters. This split is different from the 'validation-data' provided which acts as just test data. So, in training, I'm using 90% data to train the model and 10% for validation to choose best hyperparameters.

2.4.6 Why this model? I chose a simple feedforward model to get the results. This is because I wanted to see how if I use features, my

ranking model is changing. I do understand that in neural networks, I don't require to send features. I can directly send the embedding and perform CNN, which might give better results. But I wanted to explore how a simple feedforward neural network would perform on that task as compared to other ranking methods.

2.4.7 Results. The code for this task is in file task4.py or task4.ipynb. The final file for results of top 100 documents for each query is 'NN.txt' which has the same format as specified in assignment pdf. For validation set comprised of top 100 documents for each query, NDCG = 0.164 and Average Precision = 0.034

3 REFERENCES

The UCL Slide Deck for COMP0084 Images from Wikipedia, StackOverflow