Information Retrieval and Data Mining Coursework 2

Anonymous

1 Introduction

The end-to-end run time is pretty long for this coursework, because it includes calculating BM25 for a large dataset and other feature extractions. Two BM25 files (in csv) are included so examiners could use them directly instead of training from the beginning. Packages required to run the files:

python matplotlib numpy nltk sklearn gensim xgboost

Put all the data in the submitted folder and there should be no problem running all the files following the instruction. The py files might produce intermediary data files.

2 Task 1: Evaluating Retrieval Quality

2.1 Implementation of BM25

The code for this part is modified from coursework 1. The only difference is that the relevance information (The parameters R and r) is given in coursework 2, which was not available previously.

2.2 Implementation of metrics

The implementation of metrics are in the file `metrics.py`. I create two dictionaries to store the information that I need. One dictionary called `relevant_dict` stores the relevant qid-pid pairs, i.e. the keys of this dictionary are all the qids available and the values are the related pids (ie. Each qid might have 1-2 related pids). Another dictionary called `evaluate_dict` stores the top 100 ranking of all the qids, i.e. the keys of this dictionary are all the qids available, and the values are lists of the pids with top 100 ranking scores in order (ie. Each qid has a list of 100 related qids sorted by the ranking scores).

For each qid in `evaluate_dict`, I can find the actual relevant pids by referring to `relevant_dict`. Since the top 100 ranking pids are stored in `evaluate_dict` in order, I can find the index of the actual relevant pids.

Mean Average Precision (mAP) First, I calculate the average precision for each query. If the i^{th} retrieved passage is relevant, the precision will be,

$$precision_i = \frac{\text{total relevant passages up to } i}{i}$$

The average precision is then the average of $precision_1 + precision_2 + \cdots + precision_i$. Finally, the mAP is the mean of the average precision of all the queries.

Normalized Discounted Cumulative Gain (NDCG) Since all our relevance score is either 1 or 0, the gain will also be 1 or 0. The discounted gain is the gain multiply by $1/log_2(rank+1)$. Summing up all the discounted gain of a query gives the discounted cumulative gain (DCG) of the query. Also, the optimal DCG is retrieving relevant passages without any error, which can be calculated by,

$$optDCG = \sum_{i=1}^{\text{total relevant pids}} \frac{1}{log_2(i+1)}$$

Finally, $NDCG = \frac{DCG}{optDCG}$.

2.3 BM25 performance

The average precision and NDCG metrics of BM25 are shown in Table 1.

Table 1: Evaluation on BM25

BM25			
	@3	@10	@100
mAP	0.205	0.245	0.259
NDCG	0.229	0.309	0.380

3 Task 2: Logistic Regression (LR)

3.1 Word embedding choice: Word2Vec

I choose Word2Vec as my word embedding method. First, instead of using a pre-trained Word2Vec model, I tokenized all the queries and passages to train my own Word2Vec model. My trained Word2Vec model can convert any term in the queries and passages into a 100 units long vector, while pre-trained Word2Vec model online might have missing terms.

3.2 Sub-sampling on training data

In this section, I perform sub-sampling on non-relevant pairs and duplicate relevant pairs. For each query, there are at most 1000 pairing passages, while most of them are not relevant, and this results in an extremely unbalanced dataset. Unbalanced dataset could highly limit the performance of logistic regression. Thus, I only keep the first 200 non-relevant query-passage pairs, and duplicate relevant query-passage pairs 200 times. The dataset is shuffled before training. Finally, the relevant data vs non-relevant data ratio goes from 1:1000 to nearly 1:1.

3.3 Input processing/ Feature extraction

The input features of this section is the average embedding of all the terms in the queries and passages. After tokenizing the validation data and using the Word2Vec model trained by the train data, I convert the validation queries and passages from sentences into lists of vectors. Average the vectors to get the final representation of the queries and passages, and this will be the training data.

3.4 Implementation of LR

The implementation of LR is in the file `LogisticRegression.py`. First, I initialize the weights as zeros. Next, take the dot product of the training data and the weights and substitute it in a Sigmoid function as shown below.

$$sig(x) = \frac{1}{1+e^x}$$

Next, assume the difference between the calculated Sigmoid and the training labels are y', take the dot product of the training data and y' to get the gradient, g. Finally, add $g \times learning\ rate$ to the weight to perform updates for the next iteration.

3.5 LR performance

The average precision and NDCG metrics of LR are shown in Table 2. Note that a single valued vector could not fully represent a whole sentence, therefore, the results of LR is expected to be worse than BM25.

Table 2: Evaluation on LR

LR			
	@3	@10	@100
mAP	0.008	0.010	0.012
NDCG	0.009	0.014	0.030

3.6 Effects on learning rate

From Fig. 1 we can see that appropriate learning rate could speed up the training process and save time. A worth noting point is that very large learning rate (ie. 10) will cause extremely high loss which could not be plotted on the same figure.

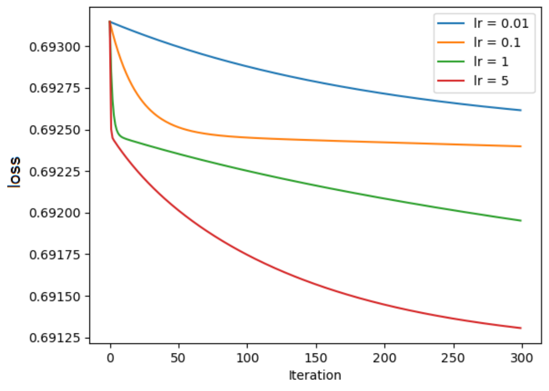

Figure 1: Learning rate effects on training LR models

4 Task 3: LambdaMART (LM) Model

4.1 Building and tuning LM model

To perform LambdaMART algorithm, I set the parameter ‘objective’ to ‘rank:pairwise’. By doing some research, I found some typical ways of doing hyper-parameter tuning in XGBoost (Aarshay, 2020), (Spark, 2019). First, I do a grid search on the two parameters, ‘max_depth’ and ‘min_child_weight’, as they have the most significant impact on the model. Tuning them at the same time gives the balance of variance and bias as they control the complexity of the model. Next, I do a grid search on another two parameters, ‘eta’ and ‘gamma’, as they both have impact on the model but not as much as the previous two parameters.

4.2 Sub-sampling on training data

Sub-sampling is done when calculating BM25 scores on the train data. There are initially over 4000 queries and each query has at most 1000 passages attached. I sub-sampled 2000 queries with at most 500 passages attached for each query. The train data BM25 scores are calculated by `BM25_train.py`.

4.3 Input processing/ Feature extraction

The training features for LM and NN models are exactly the same, which include BM25 scores, query-passage cosine-similarity, and query-passage tf-idf ratio of the train data.

The file that generates the training data representation is `data_representation.py`. Using sklearn library, I calculate the cosine-similarity of the query-passage pairs. Also using sklearn library, it is simple to calculate the tf-idf of each term in a sentence. Summing up all the tf-idf of each term in a sentence gives the final tf-idf score of the query/passage. My reason of using query-passage tf-idf ratio is that relevant query-passage pairs might have similar tf-idf values, therefore tf-idf ratio might be a reasonable feature.

I discard some features after training multiple models as those features do not improve the performance. Initially, I included the query/passage length as features, however, I eventually removed them since they acted like noise and even worsen the performance. BM25 and tf-idf ratio are two main features that ensure the good outcome of this LM model. The effect of cosine-similarity of the query-passage pairs is not significant but usually including it as a feature gives a slight push to the metrics.

4.4 LM performance

The average precision and NDCG metrics of LM are shown in Table 3. Since the BM25 scores are included as one of the features, and other reasonable features are included, the results of LM should be better than the results of BM25. Otherwise there might be negative feature which should be removed.

Table 3: Evaluation on LM

LM			
	@3	@10	@100
mAP	0.218	0.256	0.270
NDCG	0.241	0.319	0.389

5 Task 4: Neural Network (NN) Model

5.1 Justify the choice of the NN model

The goal of this section is build a NN model that could re-rank passages and improve the metrics. I believe the marks of this section is evaluated by metrics improvements instead of building a complex but worse performing NN model. Thus, after extensive online research into ranking models, I found that ranking models such as TensorFlow Ranking uses its own embedding system which prefers to take sentences as input. This is ideal for raw text data for ranking. However, we are asked to build a NN network based on the features we extracted previously, which made me double think about building an ordinary NN model instead of using existing ranking models.

My features are proven useful by my LM model, which imply that building a feed forward NN model might be a good decision as feed forward NN is designed to find relationship between different variables. I build a Multi-layer Perceptron (MLP) model that could predict the probability of whether the query-passage pairs are relevant. Parameter tuning is done by grid searching different learning rates and hidden layer size, and each parameter pairs are examined by a 3-fold cross validation. Taking the probabilities as the ranking scores, I am able to re-rank passages with improved metrics.

5.2 Input processing/ Feature extraction

The data representation in this section is identical to what I used in task 3. Refer to section 4.3 for full details.

5.3 NN performance

The average precision and NDCG metrics of LM are shown in Table 4. Compare the results in Table 1 3 4, it is clear that our LambdaMART model outperforms the baseline BM25 model, and our Neural Network model outperforms our LambdaMART model, which is the outcome that we expected.

Table 4: Evaluation on NN

NN			
	@3	@10	@100
mAP	0.222	0.262	0.275
NDCG	0.245	0.323	0..392

References

Aarshay. 2020. Xgboost parameters: Xgboost parameter tuning.

Cambridge Spark. 2019. Hyperparameter tuning in xgboost.