Information Retrieval and Data Mining Course Project Part II

Anonymous

ACM Reference Format:
Anonymous. 2018. Information Retrieval and Data Mining Course Project Part II. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

This report is a coursework assignment for the Information Retrieval and Data Mining course, discussing the use of different methods to improve the search and ranking of textual data. The report investigates several approaches, including BM25, word embeddings, LambdaMART, and neural networks, to understand their performance in information sorting. The goal is to learn the calculation methods of average precision and NDCG metrics, the construction methods of different models, and their characteristics, among other content.

2 TASK 1

This task involves assessing retrieval quality and entails implementing techniques to calculate the average precision and NDCG metrics. Additionally, it requires evaluating the performance of utilizing BM25 as the retrieval model on the validation data using these metrics.

2.1 Implementation

First, preprocess the data by lowercasing the entire text, expanding abbreviations, removing punctuation and Unicode characters, tokenizing, removing stop words, and lemmatizing. This is an improvement from what was done in coursework 1.

The second step is to load the dataset. This step involves returning the entire dataset text, including the query text with qid and queries, as well as the passage text with pid and passage.

The third step is to build the inverted index, which is an index mapping vocabulary to documents, and to calculate the average length of the document set. This inverted index records the following:

- **ni** (`vocabulary[term][0]`): The number of documents in which the term appears, indicating how many documents in the collection contain the term at least once.
- **idf** (`vocabulary[term][1]`): The Inverse Document Frequency of the term, a metric to measure a term's general importance across a document set. The idf calculation depends on the `compute_idf` function, usually a logarithmic function of the ratio of total documents N to the number of documents containing the term ni.

Additionally, it returns `avg_len`, the average number of terms across all documents, used in subsequent calculations.

The fourth step uses BM25 to calculate scores between every query and its passages, with inputs including the query, document, inverted index dictionary (for term's inverse document frequency), BM25 parameters (k1, k2, b), and the average document length (`avdl`). The output is the BM25 score.

Next, implement calculations for the Average Precision (AP) and Normalized Discounted Cumulative Gain (NDCG) metrics.

Average Precision (AP) measures the model's ability to rank relevant documents higher. To compute AP, it first identifies relevant documents. AP is the average of the precision at the positions of all relevant documents.

The process and related formulas for `compute_mAP` function are as follows:

(1) **Sorting and Truncation**: For each query ID (*qid*), documents are sorted in descending order based on their scores. Subsequently, documents ranked within the specified cutoff value are selected for evaluation. This step directly influences the calculation of mAP (mean Average Precision), as mAP is the mean of the Average Precision (AP) values for these selected documents.

(2) **Computing Average Precision (AP)**: For each query, its AP is calculated as follows. Initially, for each relevant document, its precision (i.e., the number of relevant documents retrieved up to that point divided by the total number of documents retrieved) is calculated. Then, the precisions of all relevant documents are summed up and finally divided by the total number of relevant documents.

(3) **Computing mAP**: The mAP is calculated as the average of the AP values across all queries.
For each query q, its Average Precision $\mathrm{AP}(q)$ is represented as:

$$\mathrm{AP}(q) = \frac{\sum_{k=1}^{n} (P(k) \times \mathrm{rel}(k))}{\text{Number of relevant documents for } q}$$

where $P(k)$ is the precision at cutoff k, $\mathrm{rel}(k)$ is an indicator function that equals 1 if the document at cutoff k is relevant, and 0 otherwise. n is the cutoff point, i.e., the number of documents considered.
Subsequently, the mAP is the average of AP across all queries:

$$\mathrm{mAP} = \frac{1}{Q} \sum_{q=1}^{Q} \mathrm{AP}(q)$$

where Q is the total number of queries.

Normalized Discounted Cumulative Gain (NDCG) measures the quality of document ranking, considering the relevancy of documents and their positions. NDCG gives more weight to highly relevant documents, especially when they are ranked higher. The relevant calculation formulas are as follows:

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

$$DCG@k = \sum_{i=1}^{k} \frac{2^{rel(i)} - 1}{\log_2(i+1)}$$

$$IDCG@k = \sum_{i=1}^{|REL@k|} \frac{2^{rel(i)} - 1}{\log_2(i+1)}$$

$$NDCG@k = \frac{DCG@k}{IDCG@k}$$

where:

- $rel(i)$ is the relevance level of the i^{th} document. - $|REL@k|$ is the set of documents sorted by relevance level in the top k documents.

The process for `compute_NDCG` function is:

(1) **Select the top rank_number documents**: Similar to the function to compute ap, it starts by selecting the top-ranked documents.
(2) **Calculate DCG**: Next, it calculates the Discounted Cumulative Gain (DCG) by scoring each document based on its `relevancy` and applying a logarithmic discount for the ranking. Higher scores contribute more to DCG when at the top.
(3) **Calculate IDCG**: For normalization, it calculates the Ideal DCG (IDCG), assuming all documents are sorted by descending `relevancy`. This represents optimal sorting performance.
(4) **Compute NDCG**: Finally, NDCG is calculated by dividing DCG by IDCG. If IDCG is 0 (no relevant documents), NDCG is defined as 0 to avoid division by zero.

The last step implements the `validate_bm25_model` function, which applies the BM25 model to each query in the validation dataset, ranks paragraphs based on their BM25 scores, and calculates AP and NDCG for a specified ranking cutoff, coordinating the evaluation process.

2.2 Evaluation

Average Precision (AP) Scores: The mAP value of the BM25 model is 0.2377.

NDCG Scores: The NDCG value of the BM25 model is 0.3544.

3 TASK 2

This task is to represent queries and passages using word embedding methods such as Word2Vec, GloVe, FastText, or ELMo. Calculate the embeddings of queries and passages by averaging the embeddings of all words. Utilize these query and passage embeddings as input to implement a logistic regression model to assess the relevance of passages to given queries. Describe how to perform input processing & representation or features used. Report the model's performance on the validation data using the metrics AP and NDCG. Analyze the impact of learning rate on the model training loss.

3.1 Data Processing

3.1.1 Word Embeddings. The Word2Vec word embedding method was utilized to represent queries and passages. The Gensim library was employed to load a pre-trained Word2Vec model (trained on the Google News dataset), the dataset can be download at GoogleNews-vectors-negative300 To accommodate memory constraints, only the vectors for the top 20,000 most frequent words were loaded. For each query or passage, it was first tokenized into words, and then the Word2Vec model was used to obtain embeddings for each word. Subsequently, the `compute_avg_sentence_vector` function was utilized to calculate the average of these word embeddings to obtain the overall embedding representation for the query or passage. This function computes the average vector for words present in the Word2Vec model, while ignoring those that are not. The results were saved in a dictionary for subsequent training and evaluation processes.

3.1.2 Negative Sampling. Due to the limitations of my laptop's processing capabilities for handling large datasets, I employed the negative sampling method to preprocess the data. Excessive negative samples also could lead the model to bias towards predicting the negative class. This approach not only alleviates the pressure on memory usage but also mitigates the impact of data imbalance. It enhances the model's ability to distinguish between positive and negative samples. The logic behind implementing this method involves generating a balanced training sample set for each query ID ('qid'), including 1 positive sample ('relevancy' equals 1, indicating relevance) and up to 'k' negative samples ('relevancy' equals 0, indicating irrelevance). If the number of negative samples for a particular query is less than 'k', then all available negative samples are selected. This strategy ensures that at least one positive sample is included in the training for each query, while also achieving a balance between positive and negative samples by limiting the number of negative samples per query.

3.2 Model Implementation & Training

Logistic regression utilizes the Sigmoid function (or logistic function) to map the output of a linear regression model to the [0,1] interval. The mathematical expression of the Sigmoid function is $\sigma(z) = \frac{1}{1+e^{-z}}$, where z is the model's linear prediction (i.e., the linear combination of weights and input features plus a bias). Through this mapping, the model can output a probability representing how likely a specific sample belongs to the positive class. Then, based on this probability and a predefined threshold (such as 0.5), the model decides the class of the sample (positive or negative).

Here is a detailed description and key components of the model's implementation process in this task:

Initialization: By creating a `LogisticRegression` class, the basic structure of the model is defined, including the learning rate (`learning_rate`), number of iterations (`n_iterations`), weights (`weights`), bias (`bias`), and loss history (`cost_history`). These are key parameters and properties that make up the logistic regression model, where the model's weights and bias are initialized to 0 before starting training. This is to begin the model's optimization process from a uniform starting point.

Training: The Word2Vec word embedding method is used to represent queries and passages, averaging the embedding vectors of all words to obtain the vector representation of the query or passage. The processed query and passage vector representations (`X_train`) and the relevancy labels (`y_train`) are used as input features and output targets to train the model. Different learning rates (`2, 1, 0.1, 0.01, 0.001, 0.0001`) are employed to observe their effects on model performance. The model is trained through the set 1000

iterations, updating the model's weights and bias in each iteration to minimize the loss function. The cross-entropy loss function, a commonly used loss function in logistic regression, is employed to quantify the difference between the model's predicted probabilities and the actual labels. The model uses the gradient descent algorithm to update parameters in each iteration to minimize the average loss across the entire training dataset.

Prediction and Evaluation: The model uses the trained parameters to predict the relevancy probability of each instance in the validation dataset (`X_val`) and evaluates the model's ranking performance on the validation dataset using previously implemented mAP and NDCG metrics. These metrics help us understand the model's effectiveness in actual ranking tasks.

3.3 Learning Rates and Training Loss

The analysis of training loss data under different learning rates is presented. Table 1 showcases the Model Performance Across Different Learning Rates, and Figure 1 illustrates the Training Loss Evolution Across Different Learning Rates.

At learning rates of 2 and 1, the model's training loss is relatively low, at 0.3025 and 0.3031, respectively. This indicates that higher learning rates can enable the model to quickly converge to a lower value of the loss function. However, from the performance metrics of mAP and NDCG, these learning rates do not lead to optimal model performance. This might be due to the fact that although a high learning rate promotes rapid convergence, it also increases the risk of the model overfitting or missing the optimal solution during training.

When the learning rate is set to 0.1, a slight increase in training loss is observed (0.3054), but the model performs relatively well on the mAP and NDCG metrics. This suggests that a moderate level of learning rate might provide a better balance during the model training process, avoiding the risks of too rapid convergence while maintaining a relatively high level of performance.

With further reduction in the learning rate, the training loss significantly increases, especially at the lowest learning rate of 0.0001, where the training loss reaches 0.6683. This trend indicates that lower learning rates, although making the training process more stable and avoiding overfitting, also lead to slower convergence and might even result in the training process getting stuck in local minima.

Table 1: Model Performance Across Different Learning Rates

Learning Rate	mAP	NDCG	Final Training Loss
2	0.0142	0.0399	0.3025
1	0.0157	0.0413	0.3031
0.1	0.0156	0.0412	0.3054
0.01	0.0156	0.0412	0.3137
0.001	0.0156	0.0412	0.5134
0.0001	0.0156	0.0412	0.6683

3.4 Evaluation Metrics

The evaluation results showed that the mAP value of the model is 0.0156, and the NDCG value is 0.0412 when the learning rates are

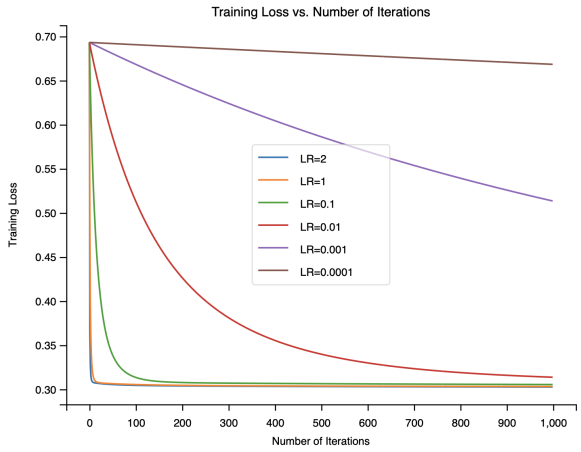

Figure 1: Training Loss Evolution Across Different Learning Rates

0.1, 0.01, 0.001, 0.0001. When learning reta is 2, the mAP value of the model is 0.0142, and the NDCG value is 0.0399. When learning reta is 1, the mAP value of the model is 0.0157, and the NDCG value is 0.0413.

4 TASK 3

This task involves using the XGBoost library to implement the LambdaMART model (a variant of LambdaRank), which is a learning-to-rank algorithm designed to re-rank text paragraphs. By setting the `objective` parameter to `rank:pairwise`, we instruct XGBoost to utilize the LambdaMART algorithm for ranking. During this task, I performed hyperparameter tuning to seek the best-performing model and reported the model's performance on the validation data using metrics I implemented myself.

4.1 Input Processing and Feature Representation

- **Data Loading**: Training and validation data were loaded using NumPy's `load` method from `.npy` files. These data were pre-processed into a format required by the model.
- **Feature Representation**: The input data is represented as sequences of integers, where each integer represents a word in the vocabulary. This representation method offers a concise way for the model to process text data.

4.2 Hyperparameter Tuning

Hyperparameter tuning was accomplished through cross-validation on a predefined grid of parameters. The grid consisted of various values for `max_depth` (maximum depth) and `min_child_weight` (minimum child weight), employing a grid search strategy to systematically explore combinations within the predefined range. Specifically, values for `max_depth` and `min_child_weight` ranged from 1 to 9.

- `max_depth` (Maximum Depth): This parameter determines the maximum depth of the trees in the model. Deeper trees can capture more complex patterns but may also lead to overfitting. By adjusting this parameter, the aim was to find an optimal depth that prevents overfitting while allowing the model to learn complex relationships.
- `min_child_weight` (Minimum Child Weight): This parameter controls the growth of the decision tree. A larger value for the parameter `min_child_weight` makes the model more conservative, thereby preventing overfitting to minor fluctuations in the data. By optimizing this parameter, the aim is to strike a balance between the model's ability to generalize and its capacity to capture significant patterns in the data.

Ultimately, the combination of parameters that resulted in the highest mean average precision (MAP) was selected as the best parameters to guide the final model training, specifically `max_depth=1` and `min_child_weight=5`.

4.3 Evaluation

The evaluation results showed that the mAP value of the model is 0.0083, and the NDCG value is 0.0248. This indicates that there is room for further optimization in the model's handling of the ranking task.

5 TASK 4

In this task, a neural network-based model was constructed to re-rank text paragraphs using the TensorFlow framework, with a particular focus on its Convolutional Neural Network (CNN) components.

5.1 Model Architecture and Rationale

The model architecture comprises several layers, each serving a specific function in the processing and learning from the input data:

(1) **Embedding Layer**: Transforms input integer sequences into fixed-size embedding vectors. The maximum vocabulary size is set to 10,000, with each word vector having a dimension of 128.
(2) **Conv1D Layer**: Utilizes 128 filters and a kernel size of 5 for one-dimensional convolutional operations. The ReLU activation function is introduced to incorporate non-linearity.
(3) **GlobalMaxPooling1D Layer**: Applies global max pooling to the convolutional layer's output, which helps in reducing the number of parameters and mitigating overfitting.
(4) **Dense Layer**: Serves as the output layer, being a fully connected layer that produces a score.

The choice of a maximum vocabulary size of 10,000 ensures the model can cover most common words while keeping the model size and computational requirements manageable. A dimension of 128 for word vectors offers sufficient capacity to capture complex semantic relationships between words. CNNs are particularly suited for text processing tasks due to their ability to capture local relevance and patterns within text data. The configuration of 128 filters and a kernel size of 5 enables the model to learn diverse features from the text, striking a balance between model complexity and performance.

5.2 Data Processing and Feature Representation

The input data is preprocessed into integer sequences, with each integer representing a specific word in the text. The `pad_sequences` function is used to standardize the length of all sequences to 600, allowing the model to handle inputs of varying lengths efficiently.

5.3 Evaluation

Evaluation Result: The mAP value of the NN model is 0.0097. The NDCG value of the NN model is 0.0293.

6 CONCLUSION

This report examines various machine learning models for text ranking and finds that each model has its strengths. It highlights the importance of careful adjustments, such as selecting the appropriate learning rate, for achieving good results. By comparing models using average precision and NDCG, it identifies the different applicabilities of each model in information retrieval.

7 REFERENCE

[1] Information Retrieval and Data Mining Course Resources
[2] XGBoost Documentation: https://xgboost.readthedocs.io/en/latest/