COMP0084 Coursework 2 Report

Student number : 18027527

1 INTRODUCTION

An information retrieval model is an essential component for many applications. This projects aims to develop an information retrieval model that solves the problem of passage retrieval, i.e., a model that can effectively and efficiently return a ranked list of passages relevant to a given query.
This report first discusses the data used for the project (Data) and goes on to explain the tasks conducted for this project. This project focuses on solving 4 subtasks:

(1) Evaluating Retrieval Quality
(2) Logistic Regression (LR)
(3) LambdaMART Model (LM)
(4) Neural Network Model (NN)

2 DATA

2.1 Data Description

There were 2 datasets used for this project. 'train_data.tsv' was used for training the models and 'validation_data.tsv' was used to assess the models' performance. 'train_data.tsv' consisted of 4,364,339 rows whereas 'validation_data.tsv' consisted of 1,103,039 rows. Both datasets contained the 5 columns:

(1) **qid** - Unique identifier of the query
(2) **pid** - Unique identifier of the passage retrieved
(3) **query** - Query text
(4) **passage** - Passage text
(5) **relevance** - Relevance of the passage to the query (1 if relevant, 0 if not relevant)

Table 1 reveals that only 0.1% of query passage pairs (rows) from both datasets were relevant. Therefore, the data has a class imbalance issue.

	train_data.tsv	validation_data.tsv
relevance=1	4,797 pairs	1,208 pairs
relevance=0	4,359,542 pairs	1,101,831 pairs

Table 1: Number of relevant passages and query pairs

2.2 Subsetting the data

As the data provided is relatively large, subsets of 'train_data.tsv' and 'validation_data.tsv' were used to implement the tasks. The subset of data used for training consisted of 100,000 rows from 'train_data.tsv', and the subset of data used for validation consisted of 50,000 rows from 'validation_data.tsv'. As there was a class imbalance, all the rows with a relevance score of 1 (4,797 pairs) and the first 95,203 rows with a relevance score of 0 were used to construct the training data subset. Similarly, all the relevant query-passage pairs (1,208) and the first 48,792 non-relevant query-passage pairs were used to create the subset of data used for validation.

2.3 Data Pre-processing

The selected subset of data was pre-processed prior to carrying out the tasks. For task 1, the subset of data was converted to lowercase, stripped off punctuation, tokenised, stopwords removed and stemmed. For tasks 2,3 and 4, the subset of data was converted to lowercase, stripped off punctuation, tokenised, and stopwords were removed.

2.3.1 ***Converting the Data to Lowercase.*** One of the most common text pre-processing steps is to convert the text into the same case. For this project, all the queries and passages were converted to lowercase.

2.3.2 ***Removing Punctuation.*** Punctuation was removed from the queries and passages by replacing the punctuation characters with empty strings.

2.3.3 ***Tokenisation.*** Tokenisation refers to chopping up a document unit into smaller units called tokens. These tokens can be sentences, words etc. Word tokenisation was used for this project (i.e. each query and passage was converted to a list of words). For example, the query 'Euless Texas is in what county?' will be tokenised to the list of words [Euless, Texas, is, in, what, county?]

2.3.4 ***Stopword Removal.*** Stopwords are commonly used words. A few examples of stopwords in English are the, an, to, so and then. As such words carry little or no meaning, they do not add any value to the analysis. Therefore, I used the NTLK library to remove stop words from the analysis. However, removing stopwords may remove some meaning from the data.

2.3.5 ***Stemming.*** Stemming reduces inflected (or derived) words/tokens to their word stem (root form). For example, the words organising and organising will be stemmed to organis. I used the Porter stemmer from the NLTK library to stem the queries and passages.
Note: Stemming was used for the first subtask but not in subsequent subtasks as the word stems may not be vocabulary words.

3 EVALUATING RETRIEVAL QUALITY

3.1 Objective

Implement methods to compute the average precision and NDCG metrics. Compute the performance of using BM25 as the retrieval model using these metrics.

3.2 Methodology

Note: The code for this task can be found in Q1.py

3.2.1 ***BM25 Model.*** The BM25 model is a popular and effective ranking algorithm based on the binary independence model. The BM25 score for a query containing words $t_1, ..., t_T$ is calculated as follows:

$$\text{BM25 score} = \sum_{i=1}^{T} \log \frac{(r_i+0.5)/(R-r_i+0.5)}{(n_i-r_i+0.5)/(N-n_i-R+r_i+0.5)} \cdot \frac{(k_1+1)f_i}{K+f_i} \cdot \frac{(k_2+1)qf_i}{k_2+qf_i} \quad (1)$$

where

- r_i is the number of relevant passages that contain the term t_i
- R is the total number of passages relevant to the query
- n_i number of passages that contain the term t_i
- N is the total number of passages
- k_1, k_2, K and b are parameters whose values are set empirically. $K = k_1((1-b)) + b \cdot \frac{dl}{avdl}$ where dl is the number of terms in the passage and $avdl$ is the average passage length. . Typical values for k_1 is 1.2, k_2 varies from 0 to 1000 and b = 0.75.
- f_i and qf_i are the number of times the term t_i occurs in the passage and the query respectively

The BM25 score was calculated for each of the query-passage pairs of the validation data subset. Table 2 reveals the average BM25 scores for relevant and non-relevant query-passage pairs in the subset of the validation data used. The mean for relevant pairs is higher than the mean for non-relevant pairs.

	Mean BM25 score
relevance=1	25.0859
relevance=0	14.4082

Table 2: Average BM25 scores

3.2.2 ***Average Precision.*** All the query-passage pairs in the subset of data belong to one of the categories in table 3.
Precision refers to the number of relevant passages that were

	Relevant	**Not-relevant**
Retrieved	True Positive (TP)	False Positive (FP)
Not retrieved	False Negative (FN)	True Negative (TN)

Table 3: Contingency table

retrieved. Therefore, precision is defined in Equation 2.

$$\text{Precision} = \frac{\text{True Positive (TP)}}{\text{True Positive (TP)+False Positive (FP)}} \tag{2}$$

Precision at k refers to the fraction of relevant passages in the top k retrieved passages [11](the function calculate_precision_at_k calculates this). The average precision (AP) is a metric that measures the average of precisions at relevant passages [12]. If a system does not retrieve a relevant document, the precision is assumed to be 0 [12]. The average precision is defined in Equation 3.

$$\text{Average Precision} = \sum_{k=1}^{n} \text{Precision at k} \cdot \mathbb{1} \tag{3}$$

where n is the number of passages retrieved for a given query and $\mathbb{1}$ is an indicator function that equals 1 if the i^{th} passage is relevant to the query and 0 otherwise.
The function calculate_average_precision calculates the average precision of the passages retrieved for a query.
Mean average precision (MAP) is the average of APs over all the queries in the validation data subset. Therefore, MAP is the sum of AP values for all queries divided by the number of queries. MAP is calculated using the function calculate_mean_average_precision.

3.2.3 ***Normalised Discounted Cumulative Gain (NDCG).*** NDCG is a metric commonly used to measure the ranking quality. NDCG measures the relative goodness of the output of the ranking algorithm. This metric takes values between 0 and 1, with a 1 denoting that the algorithm has optimally ordered the passages for a query and a 0 denoting that the passages have been reverse ordered. NDCG is the ratio of the Discounted Cumulative Gain (DCG) of the retrieved order to the DCG of the ideal order [11] i.e

$$\text{Query NDCG} = \frac{\text{Query DCG}}{\text{Query iDCG}} \tag{4}$$

NDCG of the model is calculated using the calculate_NDCG function. This is the mean NDCG of all the queries.
Discounted Cumulative Gain (DCG) uses graded relevance to measure the usefulness/gain from examining a document. Gain is accumulated starting at the top of the ranking and may be reduced or discounted at lower ranks [12]. DCG of a query can be calculated by either equation 5 or 6 [3].

$$\text{Query DCG} = \sum_{i=1}^{n} \frac{relevance_i}{log_2(i+1)} \tag{5}$$

$$\text{Query DCG} = \sum_{i=1}^{n} \frac{2^{relevance_i} - 1}{log_2(i+1)} \tag{6}$$

where n is the number of passages retrieved for a query.
As the relevance scores are binary, both equations 5 and 6 yields the same result for DCG [3].

3.2.4 ***Performance of the BM25 Model.***

- Mean Average Precision (MAP) = 1.0
- Normalised Discounted Cumulative Gain (NDCG) = 0.7756

4 LOGISTIC REGRESSION (LR)

4.1 Objective

Represent passages and query based on a word embedding method, (such as Word2Vec, GloVe, FastText, or ELMo). Compute query (/passage) embeddings by averaging embeddings of all the words in that query (/passage). With these query and passage embeddings as input, implement a logistic regression model to assess relevance of a passage to a given query. Describe how you perform input processing & representation or features used. Using the metrics you have implemented in the previous part, report the performance of your model based on the validation data. Analyze the effect of the learning rate on the model training loss.

4.2 Methodology

Note: The code for this task can be found in Q2_P1.py and Q2_P2.py

4.2.1 ***Word Embedding Method - Word2Vec.*** I used Word2vec as the word embedding method to represent the passages and queries. This method uses a shallow neural network to learn word embeddings and can be done using 2 methods - Common Bag Of Words (CBOW) and Skip Gram [6]. I used the CBOW model as it takes the context of each word as the input and tries to predict the word corresponding to the context and because it is faster and has a better representation for frequent words [6]. The model representing the passages and queries as word embeddings was created

using word2vec from gensim.models and the subset of training data. The arguments used when building the word2vec model are given below

- vector_size refers to the number of dimensions of the embedding, i.e. the length of the dense vector to represent each token (word) [1]. I used vector_size = 50 to reduce computational resources and time.
- min_count refers to the minimum number of times a word should occur when training the model; words that occur less than this number will be ignored [1]. min_count = 2.
- window refers to the maximum distance between a target word and words around the target word [1]. window = 5
- workers refer to the number of threads to use while training [1]. workers= 4

4.2.2 ***Input Processing and Features Used.*** Both the training data subset and the validation data subset were pre-processed as mentioned in Data Pre-processing.
The word embeddings for the validation data subset was created using the word2vec model built.
The query/passage embeddings for both the train data subset and the validation data subset were computed by averaging the word embeddings of all the words in the query/passage of the subset. If a word was not in the model vocabulary, it was represented by a randomly generated 50 dim vector with values between 0 and 1.
The initial feature matrices created using the train and validation subsets (x_train and x_validation, respectively) were NumPy arrays of shape (100000, 2, 50) and (50000, 2, 50), respectively. These arrays were then reshaped to create the feature matrices X_train and X_validation of shape (100000, 100) and (50000, 100), respectively. Therefore, the dimensions of the feature and target, training, and validation matrices are below

- X_train is a NumPy array of shape (100000, 100)
- Y_train is a NumPy array of shape (100000, 1). Y_train consists of 0s and 1s
- X_validation is a NumPy array of shape (50000, 100)
- Y_validation is a NumPy array of shape (50000, 1). Y_validation consists of 0s and 1s

4.2.3 ***Introduction to Logistic Regression.*** Logistic regression is a machine learning algorithm used for classification. I implemented logistic regression to classify if a passage was relevant (y=1) or not relevant (y=0) to a query.
The hypothesis function for logistic regression is

$$\hat{y} = g(w.x + b) = \frac{1}{1 + e^{-(w.x+b)}} \tag{7}$$

g(z) is the sigmoid function or the logistic function. g(z) = $\frac{1}{1+e^{-z}}$. As g(z) returns a value between 0 and 1, $0 \leq \hat{y} \leq 1$. The value of $\hat{y}$ is the estimated probability that y=1 (the passage is relevant to the query) given x (passage embedding and query embedding), parameterized by w and b [10]. Therefore,

$$P(y = 1|x; w, b) = \hat{y} \text{ and } P(y = 0|x; w, b) = 1 - \hat{y} \tag{8}$$

Thus, the cost function can be defined as

$$Cost(\hat{y}, y) = \begin{cases} -log(\hat{y}) & \text{if } y = 1 \\ -log(1 - \hat{y}) & \text{if } y = 0 \end{cases} \tag{9}$$

The above cost function can be compressed to create a single cost function $J(w, b)$

$$J(w, b) = -\frac{1}{n}\sum_{i}^{n}[y^{(i)}log(\hat{y}^{(i)}) + (1 - y^{(i)})log(1 - \hat{y}^{(i)})] \tag{10}$$

where n is the number of rows (query passage pairs) in the training data subset.
Gradient descent can be used to find the optimal parameters for w and b. Update the values for w and b as follows

- w = w - lr*dw where $dw = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}^{(i)} - y^{i})x^{(i)}$
- b = b - lr*db where $db = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}^{(i)} - y^{i})$

4.2.4 ***Implementing Logistic Regression.*** The logistic model was trained using X_train and Y_train to find optimum values for w and b (w is an array of shape (100,1), and b is a scalar). A learning rate of 0.1 was used when training. First, w and b were initialised to be random values. Then for each iteration, the cost function J(w,b) was calculated in the forward pass, and the gradients dw and db were calculated in the backward pass. Then w and b were updated as mentioned above.
The predicted score for X_validation was calculated using the optimum values found for w and b . If the predicted score was ≥ 0.5, the query passage pair received a prediction of 1 (that the pair was relevant), and if the predicted score was < 0.5, the query passage pair received a prediction of 0. **Note**: Using negative sampling yielded similar values for MAP and NDCG. Therefore, I did not use negative sampling.

4.2.5 ***Performance of the Logistic Regression Model.*** MAP and NDCG are 2 metrics that can be used to evaluate the performance of the logistic regression model. Refer AP and NDCG for more information regarding these metrics.

- Mean Average Precision (MAP) = 0.0022
- Normalised Discounted Cumulative Gain (NDCG) = 0.0042

4.2.6 ***Analyze the Effect of the Learning Rate (lr) on the Model Training Loss.*** From Figures 1 and 2, it can be observed that the choice of learning rate is critical. From Figure 1, if the learning rate is large (greater than 1), the training loss does not converge to 0. When the lr was 5 and 10, the training loss oscillates. From Figure 2, the training loss converges to 0 for learning rates 1, 0.1 and 0.01. The training loss for lr=1 converges to 0 the fastest, and the training loss for learning rates 0.1 and 0.01 drop drastically in the first 5-150 iterations. However, the training loss of very small learning rates (lr=0.001) requires more iterations to converge to 0.

5 LAMBDAMART MODEL (LM)

5.1 Objective

Use the LambdaMART [2] learning to rank algorithm from XGBoost gradient boosting library to learn a model that can re-rank passages. You are expected to carry out hyper-parameter tuning in this task and describe the methodology used in deriving the best performing model. Using the metrics you have implemented in the first part, report the performance of your model on the validation data. Describe how you perform input processing, as well the representation/features used as input.

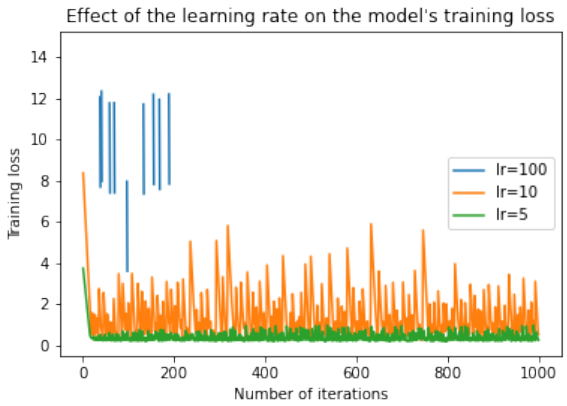

Figure 1: Effect of the learning rate on the model training loss - learning rate > 1

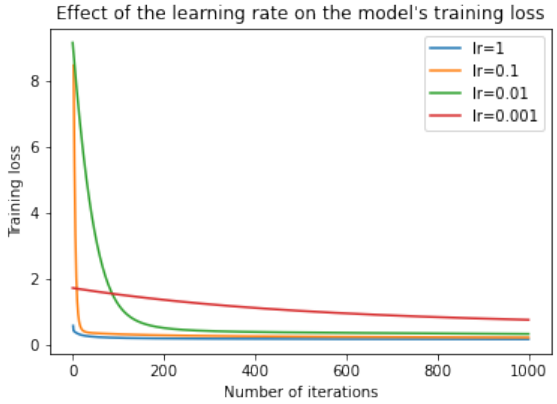

Figure 2: Effect of the learning rate on the model training loss - learning rate ≤ 1

5.2 Methodology

Note: The code for this task can be found in Q3.py

5.2.1 ***Introduction to LambdaMART*** . LambdaMART combines LambdaRank and MART (Multiple Additive Regression Trees). MART uses gradient boosted decision trees for prediction tasks. LambdaMART improves this by using gradient boosted decision trees using a cost function derived from LambdaRank to order any ranking task [5].

5.2.2 ***Input Processing and Features Used***. The training data subset was used to train the lambdaMART model and carry out hyper-parameter tuning, whilst the validation data subset was used to assess the model's performance.
XGBoost uses DMAtrices. DMatrices contain both the features and the target [9]. Therefore, I created the DMatrices train_mat using X_train & Y_train and val_mat using X_validation & Y_validation (Refer Logistic Regression Input Processing and Features Used to see how these NumPy arrays were created).

5.2.3 ***Hyper-parameter Tuning and Deriving the Best Performing Model***. I used a grid search to conduct hyper-parameter tuning. I first created a list with the parameter/ parameter pairs. Then for each parameter/ parameter pair in the list I implemented 10 fold cross-validation on train_mat using XGBoosts's cross validation-function (xgboost.cv) and recorded the mean-test-map.max() value. I then selected the optimum parameter/ parameter pair by maximising the mean-test-map.max() value. The parameters I fine-tuned are as follows:

(1) **max_depth** is the maximum number of nodes allowed from the root to the farthest leaf of a tree and is a parameter that adds constraints on the architecture of the trees. Deeper trees can model more complex relationships by adding more nodes, but as we go deeper, splits become less relevant and are sometimes only due to noise, causing the model to overfit [9].
(2) **min_child_weight** is the minimum weight (or the number of samples if all samples have a weight of 1) required to create a new node in the tree. This is also a parameter that adds constraints to the architecture of the trees. A smaller min_child_weight allows the algorithm to create children that correspond to fewer samples and create more complex trees but is more likely to overfit [9].
(3) **eta** is the learning rate. This corresponds to the shrinkage of the weights associated with features after each round [9].
(4) **subsample** refers to the fraction of rows to subsample at each step. This parameter controls the sampling of the dataset that is done at each round of boosting. By default, it is set to 1.
(5) **colsample_bytree** is the fraction of columns to use. This parameter controls the sampling of the dataset that is done at each round of boosting. By default, it is set to 1.

As both max_depth and min_child_weight add constraints on the tree architecture, I tuned them together. Similarly, I tuned the parameters subsample and colsample_bytree together as they control the dataset sampling that is done at each round of boosting. Therefore, the parameter values that maximised the MAP were

- max_depth = 9
- min_child_weight = 1
- eta = 0.3
- subsample = 1
- colsample_bytree = 1

5.2.4 ***Implementing LambdaMART using XGBoost***. XGBoost can be commanded to use the LambdaMART algorithm for ranking by setting the objective parameter to 'rank:pairwise'. I used the code xgboost.train(parameters, train_mat) where parameters is a dictionary with the values above to create the LambdaMART model. Predictions were then obtained for the validation data using this model.

5.2.5 ***Performance of the LambdaMART model***. MAP and NDCG are 2 metrics that can be used to evaluate the performance of the LambdaMART model. Refer AP and NDCG for more information regarding these metrics.

- Mean Average Precision (MAP) = 0.0016

- Normalised Discounted Cumulative Gain (NDCG) = 0.0036

6 SUB TASK 4 - NEURAL NETWORK MODEL (NN)

6.1 Objective

Using the same training data representation from the previous question, build a neural network based model that can re-rank passages. Justify your choice by describing why you chose a particular architecture and how it fits to our problem. Using the metrics you have implemented in the first part, report the performance of your model on the validation data. Describe how you perform input processing, as well as the representation/features used.

6.2 Methodology

6.2.1 ***Neural Network Based Model that Re-ranks Passages - Manhattan LSTM*** . I used Manhattan LSTM to re-rank the passages. Manhattan LSTM is a siamese recurrent architecture that learns sentence similarity [4][8]. For this task, the model will learn the similarity between the queries and the passages and rank the passages.

Long Short-Term Memory (LSTM) models are particularly successful in language translation, and text classification tasks [7]. LSTM models are built upon basic RNN models but avoid the key limitation of vanishing gradients for long sequences in RNNs [7].

Siamese networks refer to networks with two or more identical subnetworks. Siamese networks perform well on similarity tasks and have been used for tasks like sentence semantic similarity, recognizing forged signatures and many more. Siamese network are also easier to train because it shares weights on both sides [4].

6.2.2 ***Manhattan LSTM Architecture***. The model architecture excluding the passage and query pre-processing section is illustrated in Figure 3. As observed from Figure 3, in Manhattan LSTM, the identical sub-network (siamese network) is from the embedding up to the last LSTM hidden state [4].

Inputs to the network are zero-padded sequences of word indices. These inputs are vectors of fixed length, where the initial 0s are ignored, and the nonzeros are indices that uniquely identify words [4]. These vectors are then passed into the embedding layer. This

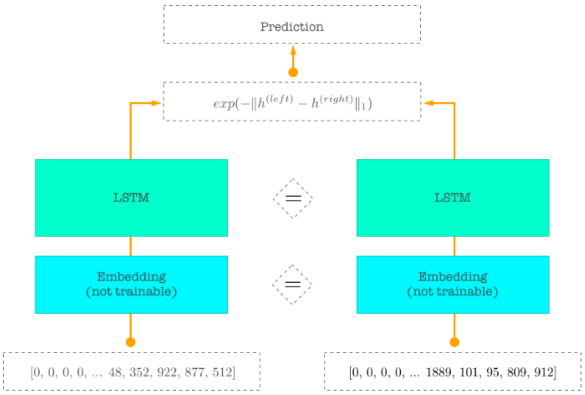

Figure 3: Manhattan LSTM architecture from [4]

layer looks up the corresponding embedding for each word and encapsulates them into a matrix. This matrix represents the given text as a series of embeddings [4]. I use the word2vec model created in Task 2 to create the embeddings. The embedding process can be observed in Figure 4. We now obtain 2 embedded matrices

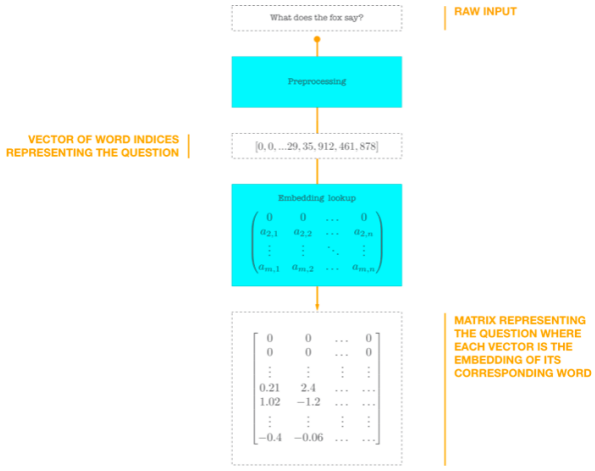

Figure 4: Embedding process from [4]

representing two possibly related query passage pairs. This is fed into the LSTM. The final state of the LSTM for each question is a 25-dimensional vector (denoted by h in Figure 3). The LSTM is trained to capture the semantic meaning of the passage and query pairs.

Now, we have the 2 vectors that hold the semantic meaning of each query and passage. This is passed through the similarity function. This similarity function calculates the Manhattan difference between the 2 vectors. Since we have an exponent of a negative, the output/ prediction will be between 0 and 1.

$$\text{Similarity function} = e^{-||h^{(left)} - h^{(right)}||_1} \tag{11}$$

6.2.3 ***Input Processing and Features Used***. Both the training data subset and the validation data subset were pre-processed as mentioned in Data Pre-processing.

Convert each passage and query to a list of word indices. This is done by creating a vocabulary_dict where the keys are words (str), and values are the corresponding indices (a unique id as int) and an inverse_vocabulary_lst which is a list of words (str) where the index in the list is the matching id (from vocabulary_dict) [4]

6.2.4 ***Implementing the Manhattan LSTM Model***. The optimizer of choice in the article is the Adadelta optimizer, which can be read about in this article. We also use gradient clipping to avoid the exploding gradient problem [4].

REFERENCES

[1] Jason Brownlee. 2020. How to Develop Word Embeddings in Python with Gensim. Retrieved April 20, 2022 from https://machinelearningmastery.com/develop-word-embeddings-python-gensim/

[2] Christopher J. C. Burges. 2010. From RankNet to LambdaRank to LambdaMART: An Overview.

[3] Pranay Chandedakar. 2020. Evaluate your Recommendation Engine using NDCG. Retrieved April 20, 2022 from https://towardsdatascience.com/evaluate-your-recommendation-engine-using-ndcg-759a851452d1

[4] Elior Cohen. 2017. How to predict Quora Question Pairs using Siamese Manhatton LSTM. Retrieved April 21, 2022 from https://blog.mlreview.com/implementing-malstm-on-kaggles-quora-question-pairs-competition-8b31b0b16a07

[5] Nikhil Dandekar. 2016. Intuitive explanation of Learning to Rank (and RankNet, LambdaRank and LambdaMART). Retrieved April 20, 2022 from https://medium.com/@nikhilbd/intuitive-explanation-of-learning-to-rank-and-ranknet-lambdarank-and-lambdamart-fe1e17fac418

[6] Dhruvil Karani. 2018. Introduction to Word Embedding and Word2Vec. Retrieved April 20, 2022 from https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa

[7] Gautam Karmakar. 2018. Manhatton LSTM model for text similarity. Retrieved April 21, 2022 from https://medium.com/@gautam.karmakar/manhattan-lstm-model-for-text-similarity-2351f80d72f1

[8] Jonas Mueller and Aditya Thyagarajan. 2016. Siamese Recurrent Architectures for Learning Sentence Similarity. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI Press.

[9] Cambridge Spark. 2017. Hyperparameter tuning in XGBoost. Retrieved April 20, 2022 from https://blog.cambridgespark.com/hyperparameter-tuning-in-xgboost-4ff9100a3b2f

[10] Suraj Verma. 2021. Logistic Regression From Scratch in Python. Retrieved April 20, 2022 from https://towardsdatascience.com/logistic-regression-from-scratch-in-python-ec66603592e2

[11] Benjamin Wang. 2021. Ranking Evaluation Metrics for Recommender Systems. Retrieved April 20, 2022 from https://towardsdatascience.com/ranking-evaluation-metrics-for-recommender-systems-263d0a66ef54

[12] Emine Yilmaz. 2022. UCL COMP0084 Information Retrieval Evaluation Slides.