# CS510 Project Report

# Implementation of Pairwise Learning to Rank in MeTA

**Project Track:** Software Track

**Team Members:**
Mihika Dave (mhdave2@illinois.edu)
Anthony Huang (kesongh2@illinois.edu)
Rachneet Kaur (rk4@illinois.edu)

**Link to the code:**
https://github.com/mihikadave/meta/tree/spd
https://github.com/mihikadave/meta/tree/ltr_rerank

**Pull request to 'develop' branch in 'meta':**
https://github.com/meta-toolkit/meta/pull/194

# 1. Introduction & Impact

MeTA is a C++ data science toolkit widely used by students of UIUC, Coursera as well as the TIMAN Research Group. But MeTA doesn't currently support Learning To Rank. Learning to rank is a machine learning task for ranking objects that can be employed in numerous areas. We extended the MeTA toolkit with Learning to Rank. The idea is that pairwise learning to rank problem can be reduced to learning a binary classifier. We implemented 2 algorithms.

1. Stochastic Pairwise Descent (SPD) algorithm. MeTA supports SGD SVM classifier which can be thus utilized to implement SPD.
2. We also implemented pairwise learning to rank as RankSVM, with the help of libSVM wrapper in MeTA.
3. In the following sections, we use SPD to refer to implementation with SGD SVM and use RankSVM to refer to implementation with libSVM.

We also implemented a new ltr_ranker in 'ranker' package, which can take weights from file specified in configuration file, and use the learned weight to score documents based on multiple features.

Ranking is the core problem in IR, and using machine learning techniques to rank is an emerging research area. These algorithms are widely used by commercial web search engines. It is useful for tasks in document retrieval, collaborative filtering, sentiment analysis and many more. RankNet is known to be the central algorithm used by Bing. Other than IR, LETOR has multiple applications in computational biology and recommender systems. Considering the emerging impact of LETOR and MeTA, it is extremely useful to build a LETOR algorithm for MeTA.

# 2. Algorithm Description

Pairwise Learning to Rank methods explore all the $O(n^2)$ pairs for training size of n samples. Even by letting the pairwise preferences to be valid only within a given query results in super-linear dependencies on n.

## 2.1 Objective

In a dataset D with labelled examples (x, y, q) where $x \in R^n$ is the location of the example in the n-dimensional space, y is the rank and q denotes the query the example belongs to, we consider the set of tuples $((a, y_a, q_a), (b, y_b, q_b))$ where $y_a \neq y_b$ and $q_a = q_b$ as the candidate pairs.

Let P denote the set of all candidate pairs in D. The objective then is to find a weight

vector w $\in R^n$ that minimizes the hinge loss over P with little computational effort.

Mathematically, the goal can be written as follows:

$$\textbf{Minimize } \frac{\lambda}{2}||w||^2 + \frac{1}{|P|}\left(\sum_{((a, y_a, q_a), (b, y_b, q_b))} Hinge\ Loss\ ((a-b),\ sign\ (y_a - y_b),\ w)\right) \quad \textbf{(1)}$$

where $\lambda$ is the regularization parameter.

## 2.2 Stochastic Pairwise Descent Algorithm

SPD attempts to reduce the pairwise learning to rank problem to the learning a binary classifier problem, via stochastic gradient descent. The binary classifier we implemented is Support Vector Machines. The method is explained in the paper [1] by D.Sculley. The core objective is to sample candidate pairs from P for the stochastic steps.

The algorithm iteratively samples a random pair of labelled examples from the indexed dataset. To train the dataset, Support Vector Machine classifier is used on the difference of feature vectors of sampled pair ((a, $y_a$, $q_a$), (b, $y_b$, $q_b$)) s.t. $y_a \neq y_b$ and $q_a = q_b$.

The classification result of +1 indicates that (a, $y_a$, $q_a$) must be relatively ranked higher than (b, $y_b$, $q_b$) and vice versa. After learning the weights w from the classifier that minimize the Hinge loss described in (1), the score for a test example x is computed by $< w, x >$. These score values determine the ranking of the documents.

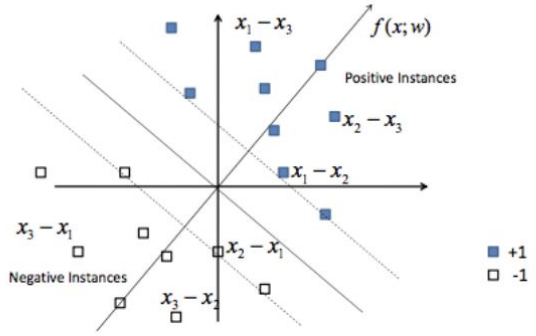


**Figure 1: Transformation to pairwise SVM**

### Algorithm 1: Pseudocode for the Stochastic Pairwise Descent algorithm

**Step 1:** Index the dataset D using a nested Hash table to $D_{index}$

$D_{index} \leftarrow$ **Create Index (D)**

**Step 2:** Initialize first weight

$W_0 = \oslash$

**Iterate from i = 1 to t (Fixed no. of iterations):**

**Step 3:** Randomly sample a pair of labelled examples from $D_{index}$

**$((a, y_a, q_a), (b, y_b, q_b)) \leftarrow$ Get Random Pair ($D_{index}$)**

**Step 4:** Compute the difference of the feature vectors for the sampled pair

**$x \leftarrow (a - b)$**

**Step 5:** Compute the sign of the difference of ranks for the sampled pair

**$y \leftarrow$ sign ($y_a - y_b$)**

**Step 6:** Compute the next weight using the Stochastic Gradient algorithm

$w_i \leftarrow$ **Stochastic Gradient Step (** $w_{i-1}, x, y, i$)

**end**

**Step 7:** Return the value for final weight $w_t$

**return** $w_t$

## 2.3 Sampling

We have implemented Indexed Sampling approach for the Get Random Pair ($D_{index}$) function in our code.

- **Indexed Sampling:**

In this approach, the dataset is indexed using nested hash tables and pairs are uniformly sampled.

### Algorithm 2: Pseudocode for the Indexed Sampling algorithm

**Goal 1:** To index the dataset D into a nested hash table P

Let Q denote the unique values in D

**Step 1:** For q ∈ Q, map Y[q] to set of unique y values s.t. (x, y, q') ∈ D with q = q'

**Step 2:** For q ∈ Q and y ∈ Y [q], map P[ q ][ y ] to (x, y', q′) ∈ D with q = q′ and y = y′

**Goal 2:** Sample from P in O(1) time

**Step 1:** Uniformly sample a query q from Q

**Step 2:** Select $y_a$ uniformly at random from Y [q]

**Step 3:** Select $y_b$ uniformly at random from Y [q] - $y_a$

**Step 4:** Select (a, $y_a$, $q_a$) uniformly at random from P [q][$y_a$ ]

**Step 5:** Select (b, $y_b$, $q_b$) uniformly at random from P [q][$y_b$ ]

**Step 6:** Return the pair ( (a, $y_a$, $q_a$), (b, $y_b$, $q_b$))

These randomly sampled pairs are then used for training the SGD SVM classifier. The indexed sampling performs faster when the dataset fits in the memory.

## 2.4 RankSVM

To compare results from SPD algorithm, we implemented pairwise learning to rank with RankSVM using the existing libSVM wrapper in MeTA, which has a different training process from SPD.

After reading nested datamap from train data, we convert it into a pairwise dataset. Each entry in this pairwise dataset is a pair containing a label and a feature vector. The label can be either 1 or -1, indicating the sign of difference of labels of the two query-doc pairs. The feature vector is the subtraction of feature vectors for these two pairs. Then we pass the pairwise dataset for libSVM to train, after which process the libSVM will write this SVM classifier into file. We then need to load trained weights from the classifier file into memory for computing ranking scores later.

In testing phase, the loaded trained weights are used to compute ranking score of each test feature_vector. Therefore, the testing and evaluation phase is similar for RankSVM and SPD implementation. The difference between them is that the SPD is trained on fixed number of iterations while the RankSVM is trained on all samples in training set, indicating that the training time of SPD should be steady regardless of size of training set. Also, since the SPD is trained in a stochastic way using gradient descent, the performance (Precision, MAP and nDCG) of SPD may be slightly worse than that of RankSVM. We will show comparison of SPD and RankSVM on various metrics in detail in Results Comparison section.

## 2.5 Evaluation on ranking

The evaluation metrics discussed above evaluates the trained model either using SPD or RankSVM on the validation and testing set corresponding to training set provided by LETOR. However, we also want to test the trained learning to rank model to rank real documents. Therefore we implement a new ltr_ranker in meta::index namespace and registered it in the ranker_factory.

This ltr_ranker takes weights_path and briefs_path as two arguments to construct, then loads value of trained weights from file at weights_path and loads string brief corresponding to each weight from file at briefs_path. In the overridden score_one function, the ltr_ranker retrieves ranking scores from okapi_bm25 ranker, absolute_discount ranker, dirichlet_prior ranker, and jelinek_mercer ranker. Then it search for the weights corresponding to these feature values (feature type corresponding to each weight is specified by input briefs). Finally the score is computed as sum of each weight multiplying corresponding feature value.

The overridden score_one function in ltr_ranker will be called by rank function in ranker, which will then return sorted documents based on scores computed by ltr_ranker. Since the ltr_ranker has been registered in ranker_factory, it can be used just like other

existing rankers in MeTA. Details of configuration to use ltr_ranker will be discussed in tutorial of our implementation.

# 3. Implementation Details

## Algorithm 3: Pseudocode for implementation

**Step 0:** User specifies to use SPD or RankSVM for training and testing

**Step 1:** Read the dataset

**If use SPD**

**Step 2a:** Index the dataset in a nested hash table

**Step 3a:** Using Algorithm 2 for Indexed Sampling, randomly sample a pair of tuples with feature vector, rank label and query ID

**Step 4a:** With the difference of the sampled pairs as a data sample for the classifier, train the SGD SVM classifier for 100,000 iterations

**If use RankSVM**

**Step 2b:** Build a pairwise dataset from nested datamap

**Step 3b:** Pass the pairwise dataset to libSVM wrapper to train

**Step 4b:** Load weights trained by libSVM wrapper into memory for scoring

**Step 5:** Validate the trained classifier

**Step 6:** Test on the unseen samples

**Step 7:** Evaluate the performance of the model based on Precision, MAP and nDCG

**Step 8:** Evaluate document rankings for given queries between ltr_ranker and other existing rankers

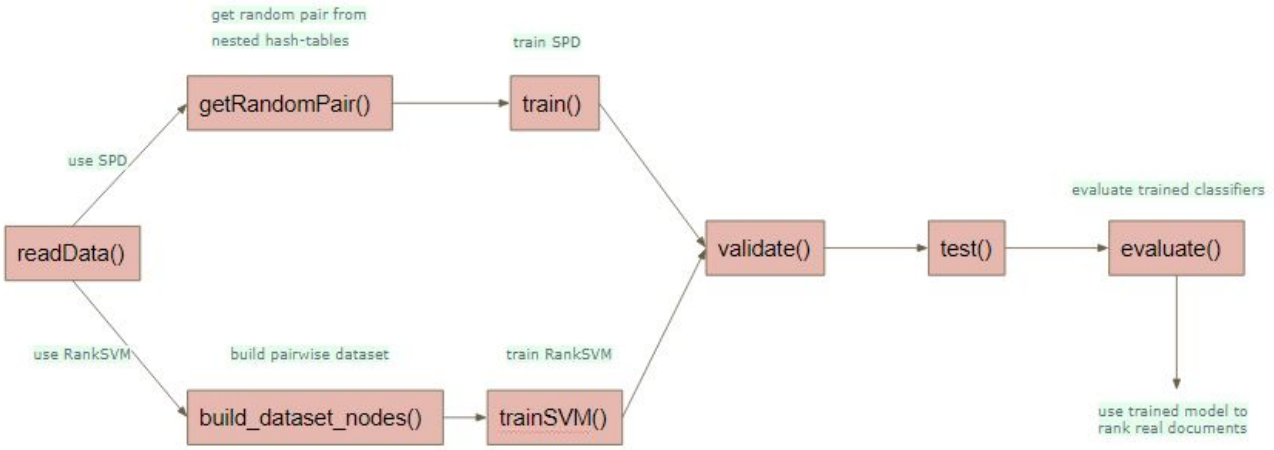


**Figure 1: Flow of Implementation and Evaluation**

### Experimental Details:

**Datasets used:** TD2003, TD2004, NP2003, NP2004, HP2003, HP2004, OHSUMED in the LETOR 3.0 datasets; MQ2007 and MQ2008 in the LETOR 4.0 datasets.
**Training iterations:** 100, 000 (fixed for all the datasets)
**Optimization:** Stochastic Gradient Descent
**Sampling:** Indexed Sampling
**Evaluation:** Precision @1, @3, @5, @10; nDCG @1, @3, @5, @10; MAP; Run Time in seconds
**Compared with:** RankSVM implemented with libSVM wrapper in MeTA
**Evaluation on real ranking:** Compare document rankings given 50 queries between ltr_ranker and other rankers

Either the SPD or RankSVM implementation will save value of learned weights into "letor.weights" file. To test our trained learn to rank model (weights) to rank real documents, we specify in the configuration file to use the ltr_ranker, and we specifies "letor.weights" as the file that the ltr_ranker will use to read value of weights from. We also make a "letor.briefs" file, in which file each line represents the brief description of the feature that each weight in "letor.weights" file will be multiplied upon.

We then use ./interative-search config.toml to perform 50 queries on ltr_ranker, and store the ranking of top 5 documents for each query. The same queries are tested on okapi_bm25 ranker, absolute_discount ranker, dirichlet_prior ranker, and jelinek_mercer ranker, and rankings for each query with each ranker are recorded. After that we compare the document rankings for 50 queries between ltr_ranker and each of the other four existing rankers.

# 4. Organization of Contribution

We split our work into two branches.

1. The `spd` branch contains implementation of 2 pairwise learning to rank model. One is called Stochastic Pairwise Descent and another is RankSVM. First uses SVM with SGD and another uses libSVM.
2. The `ltr_rerank` branch contains implementation of a ltr_ranker that takes weights specified in 'config.toml' and scores documents by dot product between trained weights and features.

We have split them and will make two pull requests for each branch, so that in each pull request we are focusing on implementation changes in one aspect.

# 5. Results Comparison with Methods

We compared the Stochastic Pairwise Descent algorithm for the LETOR 3.0 and LETOR 4.0 benchmark dataset against the original RankSVM algorithm implemented with libSVM. The results and comparisons are discussed as follows.

| **Algorithm** | **NDCG @1** | **NDCG @3** | **NDCG @5** | **NDCG @10** | **Prec @1** | **Prec @3** | **Prec @5** | **Prec @10** | **MAP** | **Train Time (s)** |
|---|---|---|---|---|---|---|---|---|---|---|
| RankSVM | 0.2466666 | 0.2962288 | 0.2928726 | 0.2950134 | 0.2466666 | 0.2718518 | 0.24 | 0.1746666 | 0.2018968 | 36.839884 |
| **SPD** | **0.22** | **0.30653** | **0.2958528** | **0.2963046** | **0.22** | **0.2703704** | **0.216** | **0.1526666** | **0.2244122** | **0.17703** |

**Table 1: Results on TD2003 dataset**

| **Algorithm** | **NDCG @1** | **NDCG @3** | **NDCG @5** | **NDCG @10** | **Prec @1** | **Prec @3** | **Prec @5** | **Prec @10** | **MAP** | **Train Time (s)** |
|---|---|---|---|---|---|---|---|---|---|---|
| RankSVM | 0.4 | 0.3578892 | 0.3303004 | 0.3090948 | 0.4 | 0.3466666 | 0.2986668 | 0.2466666 | 0.2242972 | 49.72276 |
| **SPD** | **0.2799998** | **0.2874816** | **0.2734334 9** | **0.2627157** | **0.2799998** | **0.286666** | **0.2506668** | **0.212** | **0.1913158** | **0.210498** |

**Table 2: Results on TD2004 dataset**

| **Algorithm** | **NDCG @1** | **NDCG @3** | **NDCG @5** | **NDCG @10** | **Prec @1** | **Prec @3** | **Prec @5** | **Prec @10** | **MAP** | **Train Time (s)** |
|---|---|---|---|---|---|---|---|---|---|---|
| RankSVM | 0.5570116 | 0.7221504 | 0.7718352 | 0.7957908 | 0.5570116 | 0.2543552 | 0.1740842 | 0.0938058 8 | 0.6784266 | 9.95652 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **SPD** | **0.533589** | **0.756404** | **0.7896262** | **0.8105264** | **0.533589** | **0.2652194** | **0.1740384** | **0.0939310 4** | **0.6751454** | **0.16488** |

**Table 3: Results on NP2003 dataset**

| **Algorithm** | **NDCG@1** | **NDCG@3** | **NDCG@5** | **NDCG@10** | **Prec@1** | **Prec@3** | **Prec@5** | **Prec@10** | **MAP** | **Train Time (s)** |
|---|---|---|---|---|---|---|---|---|---|---|
| RankSVM | 0.575238 | 0.7789668 | 0.8086056 | 0.8312058 | 0.575238 | 0.273968 | 0.1782856 | 0.0971429 | 0.7007574 | 7.233768 |
| **SPD** | **0.5219046** | **0.792232** | **0.8293892** | **0.8411878** | **0.5219046** | **0.2831748** | **0.1862858** | **0.0972381 6** | **0.6862656** | **0.182406** |

**Table 4: Results on NP2004 dataset**

| **Algorithm** | **NDCG@1** | **NDCG@3** | **NDCG@5** | **NDCG@10** | **Prec@1** | **Prec@3** | **Prec@5** | **Prec@10** | **MAP** | **Train Time (s)** |
|---|---|---|---|---|---|---|---|---|---|---|
| RankSVM | 0.6779822 | 0.783445 | 0.7880292 | 0.8094076 | 0.6779822 | 0.3126862 | 0.1943242 | 0.1045875 8 | 0.7364058 | 11.56542 |
| **SPD** | **0.7201532** | **0.786548** | **0.8105894** | **0.8242756** | **0.7201532** | **0.3100468** | **0.200935** | **0.1066616** | **0.7548992** | **0.1676** |

**Table 5: Results on HP2003 dataset**

| **Algorithm** | **NDCG@1** | **NDCG@3** | **NDCG@5** | **NDCG@10** | **Prec@1** | **Prec@3** | **Prec@5** | **Prec@10** | **MAP** | **Train Time (s)** |
|---|---|---|---|---|---|---|---|---|---|---|
| RankSVM | 0.5895238 | 0.7457278 | 0.7693514 | 0.7951498 | 0.5895238 | 0.2787302 | 0.1809524 | 0.1001905 4 | 0.6966816 | 9.562482 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **SPD** | **0.60380 96** | **0.7116 806** | **0.76 4688 4** | **0.784 8518** | **0.60380 96** | **0.265 0794** | **0.183 8098** | **0.098 76194** | **0.684 9238** | **0.187 434** |

**Table 6: Results on HP2004 dataset**

| **Algorithm** | **NDC G@1** | **NDC G@3** | **NDC G@5** | **NDCG @10** | **Prec @1** | **Prec @3** | **Prec @5** | **Pre c @10** | **MAP** | **Train Time (s)** |
|---|---|---|---|---|---|---|---|---|---|---|
| RankSVM | 0.5 406 496 | 0.49 4239 8 | 0.45 7924 8 | 0.437 9256 | 0.667 7492 | 0.612 7998 | 0.541 6796 | 0.4 840 04 | 0.43 656 | 55.19 7056 |
| **SPD** | **0.51 138 54** | **0.46 7027 8** | **0.44 8665 8** | **0.442 5568** | **0.620 9092** | **0.56 0678 4** | **0.540 6492** | **0.5 041 558** | **0.44 8827 4** | **0.162 1212** |

**Table 7: Results on OHSUMED dataset**

| **Algorithm** | **NDC G@1** | **NDC G@3** | **NDC G@5** | **NDCG @10** | **Prec @1** | **Prec @3** | **Prec @5** | **Pre c @10** | **MAP** | **Train Time (s)** |
|---|---|---|---|---|---|---|---|---|---|---|
| RankSVM | 0.47 036 24 | 0.46 7041 6 | 0.47 3038 6 | 0.5102 584 | 0.545 5812 | 0.49 8738 8 | 0.473 4686 | 0.4 445 588 | 0.53 6280 4 | 36.2 0937 2 |
| **SPD** | **0.4 360 204** | **0.44 5161** | **0.45 4808 4** | **0.485 355** | **0.510 353** | **0.47 3675 8** | **0.450 0434** | **0.41 330 8** | **0.513 754** | **0.177 366** |

**Table 8: Results on MQ2007 dataset**

| **Algorithm** | **NDC G@1** | **NDC G@3** | **NDC G@5** | **NDCG @10** | **Prec @1** | **Prec @3** | **Prec @5** | **Pre c @10** | **MAP** | **Train Time (s)** |
|---|---|---|---|---|---|---|---|---|---|---|
| RankSVM | 0.4 | 0.48 | 0.53 | 0.6109 | 0.576 | 0.531 | 0.490 | 0.41 | 0.59 | 9.06 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 606452 | 9747 | 01098 | 202 | 2972 | 7204 | 7624 | 8414 | 24202 | 29244 |
| **SPD** | **0.41656546** | **0.470119 2** | **0.5142168** | **0.5896466** | **0.5417664** | **0.508908 6** | **0.4801892** | **0.4 067 104** | **0.57 0077 8** | **0.162 396** |

**Table 9: Results on MQ2008 dataset**

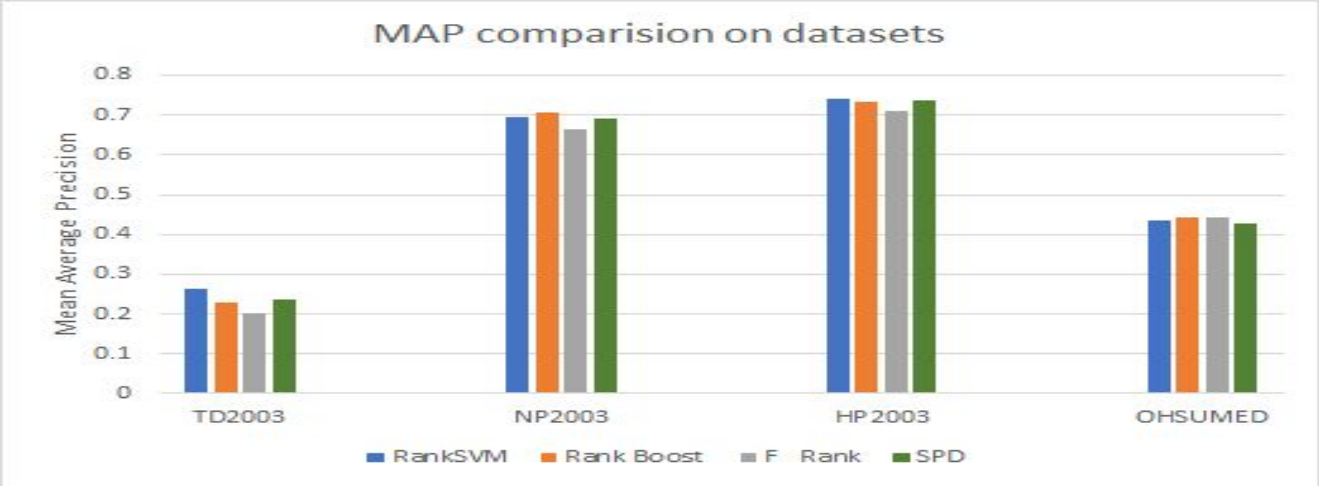


**Figure 2: Comparison of results**

It can be noted that the results show the ranking performance of the SPD algorithm is at least as good as the benchmark methodologies tried namely, Rank SVM, Rank Boost and F Rank on the LETOR 3.0 datasets.

We also perform evaluation of our trained learning to rank model on real queries. In this evaluation we simply make 50 queries to ltr_ranker, and record document rankings for each query. Then we make the same 50 queries using okapi_bm25, absolute_discout, dirichlet_prior, and jelinek_mercer ranker respectively. Then for each existing ranker, we take its document rankings as the true ranking, and calculate MAP and nDCG at position 5 of the document rankings from ltr_ranker. In this way, we can see the ranking difference between ltr_ranker and other existing rankers.

| **Base Ranker** | **okapi_bm25** | **absolute_discount** | **dirichlet_prior** | **jelinek_mercer** |
|---|---|---|---|---|
| **MAP** | 0.9432 | 0.931933333 | 0.916466667 | 0.9744 |
| **NDCG@5** | 0.99388675 | 0.981475069005 | 0.979288626 | 0.99819562 |

**Table 10: Document rankings of 50 queries**

# 6. Implementation Challenges

In this section we discuss the few challenges we encountered while implementing the algorithms:

1. The biggest challenge was integration with MeTA toolkit. We had earlier implemented our algorithms in a procedural manner which would train and test for a particular dataset. In order to integrate it with the project structure of MeTA, we had to change it into Object Oriented approach. This could make the ranker re-usable for future users who can now use pre-trained models and also create their own rankers and test it on a wide range of datasets.
2. In Stochastic Pairwise Descent, we kept the number of iterations fixed at 100,000 which was suggested in the paper [1]. This SGD SVM iteration count was good for the tested datasets but we have no formal reasoning for the same. It may not be optimal for other datasets. We need to fully validate the samples to tune the iteration number parameter.
3. The benchmark datasets sometimes had only 1 document for a given query, which led to arbid results earlier. We fixed the code to later handle such cases.
4. We needed to decide a suitable method for indexing the dataset. We decided to read the whole training dataset and process it. Sampling values uniformly was tricky.

# 7. Using the software

## Setting up the directory

Follow the steps given below will install “MeTA” toolkit along with our implementation of Pairwise Learning to Rank:

1. Clone the repository

```
git clone https://github.com/mihikadave/meta.git
```

2. Change directory to meta

```
cd meta
```

3. Switch to branch ltr_rerank

```
git checkout ltr_rerank
```

4. Pull for branch ‘spd’

```
git pull
```

4. The next few steps for the set-up instructions depend on the OS you are using.

Follow the instructions in the `README.md` to set up meta and build it, depending on your operating system:
https://github.com/mihikadave/meta/blob/master/README.md#project-setup

## Running Learning to Rank

### Pairwise_letor_main.cpp :

We provide `pairwise_letor_main.cpp` to demonstrate how users can integrate the pairwise learning to rank algorithm that we have implemented (pairwise_letor.h and pairwise_letor.cpp) in their own functions.

`Pairwise_letor_main.cpp` takes 2 command line arguments:

a. to provide path to the training, validation and testing dataset to be used for training the pairwise ranker
b. Number of features in each sample

After building meta, you can run the learning to rank algorithm as follows:

5. Change to build directory and build

```
cd build
```

```
make
```

6. To run letor, you need to provide the `path to the dataset` and the `number of features` in each sample

### Usage:

```
./pairwise_letor_main [directory_path] [num_features]
```

7. Running the above command will take two inputs from the user:
First, choose whether you want to use a pre-trained model:
*Enter 1 for pre-trained model, and 0 for training a new model. If you chose 1, enter the path to the existing model.*

8. Second, choose the ranking method:
Enter 1 for training with SPD and 0 for training with RankSVM. Entering 0 will prompt to enter the path to libSVM.
The program will now save the LETOR model and print out the MAP, NDCG values for the test data

## Manual Tests

Example datasets from LETOR3.0 can be found in the `data` folder in `meta` directory.

### Example 1:

Every data sample in `MQ2007` dataset (part of Letor3.0 dataset) has `46` features.
To use the `MQ2007` dataset, run the following command from `build` folder

1. Run the main file

```
./pairwise_letor_main PATH-TO-META/meta/data/MQ2007/Fold3/ 46
```

2. For a new model, enter 0.
3. For RankSVM, enter 0.
4. Enter the path to libSVM

```
PATH-TO-META/meta/deps/libsvm-modules
```

### Example 2:

Every data sample in `MQ2007` dataset (part of Letor3.0 dataset) has `46` features.
To use the `MQ2007` dataset, run the following command from `build` folder

1. Run the main file

```
./pairwise_letor_main PATH-TO-META/meta/data/MQ2007/Fold3/ 46
```

2. For a pre-trained model, enter 1.
3. Enter the path to the model

```
letor_svm_train.model
```

4. For RankSVM, enter 0.

### Example 3:

Every data sample in `OHSUMED` dataset (part of Letor3.0 dataset) has `45` features.
To use the `OHSUMED` dataset, run the following command from `build` folder

1. Run the main file

```
./pairwise_letor_main PATH-TO-META/meta/data/OHSUMED/QueryLevelNorm/Fold4/ 45
```

2. For a new model, enter 0
3. For SPD, enter 1.

### Example 4:

Every data sample in `OHSUMED` dataset (part of Letor3.0 dataset) has `45` features.
To use the `OHSUMED` dataset, run the following command from `build` folder

1. Run the main file

```
./pairwise_letor_main PATH-TO-META/meta/data/OHSUMED/QueryLevelNorm/Fold4/ 45
```

2. For a pre-trained model, enter 1.
3. Enter the path to the model

```
letor_sgd_train.model
```

4. For SPD, enter 1.
5. Enter 1 to continue training this model, else 0.

## Documentation of the code:

*Doxygen documentation for the code (if required for review) can be found at the end of the report.*

## Sample Tutorial:

We have implemented a sample file for implementing the class pairwise_letor.
The sample file is pairwise_letor_main.cpp. As we saw earlier, it requires the path of training dataset and number of features in the data.

**Sample usage of the run Command:**

```
./pairwise_letor_main PATH-TO-META/meta/data/OHSUMED/QueryLevelNorm/Fold4/ 45
```

Given below we show some snippets from pairwise_letor_main.cpp to demonstrate some use cases of our implementations:

## I. Pre-trained model

A pre trained model can be used rather than training a new model.

```
cout << "Do you want to load trained model from file? 1(yes)/0(no)" << endl;
cin >> hasModel;
if (hasModel) {
        cout << "Please specify path to your model file" << endl;
        cin >> model_file;
        cout << "Path to your model file is: " << model_file << endl;
    }
```

If '1': User is prompted to enter the path for the file containing the trained model
Else: A new model is trained.

## II. Options of using different rankers

RankSVM (libSVM) or SPD (SVM with SGD)

```
cout << "Please select classification method to use: 0(libsvm), 1(spd)" << endl;
cin >> selected_method;
if (selected_method == 0) {
        train_libsvm(data_dir, num_features, hasModel, model_file);
    } else {
        train_spd(data_dir, num_features, hasModel, model_file);
    }
```

## III. Handling pre-trained models

As shown in I, for pre-trained models, the path to the model_file is obtained from the user, else model_file is null. The training step is skipped in this case.

## IV. New letor model with SPD

```
  pairwise_letor letor_model(num_features, pairwise_letor::SPD, hasModel, model_file);
```

```
letor_model.train(data_dir);
```

SPD is trained on 100,000 iterations using the Hinge Loss function.

### V. New letor model with libSVM:

User is prompted to provide path of libSVM modules and the model is trained.

```
pairwise_letor letor_model(num_features, pairwise_letor::LIBSVM, hasModel, model_file);
```

```
letor_model.train_svm(data_dir, svm_path);
```

### VI. Validating the model:

```
letor_model.validate(data_dir);
```

### VII. Testing the model:

The trained and validated model is tested. The test documents are ranked and evaluated on Precision, MAP and nDCG.

```
letor_model.test(data_dir);
```

In order to use ltr_ranker to rank, you need to checkout branch ltr_rerank. Then in the config.toml under build/, at the [ranker] part, write:
Method = “ltr_ranker”
Weights = “letor.weights”
Briefs = “letor.briefs”
Then use ./interactive-search config.toml
The letor.weights is automatically saved after each time you run ./letor_main.
The letor.briefs containing briefs for each weight (what this feature stands for). For example, “tf_doc”, “bm25_doc”.

# 7. Contribution of Team Members

We have contributed to the project in regular intervals and organized multiple group meetings to collaborate on various project requirements.

Mihika: worked on the random sampling method and train method for spd, made the code object oriented, integrated with the meta project structure, fixed style issues, added comments and created doxygen documentation, worked on the software documentation, etc.

Anthony: implemented ranksvm, tested on LETOR datasets, implemented indexing methods, compared method performance, various parts of software description in the report, add ltr_ranker in meta::index namespace to use trained learning to rank model in real document ranking

Rachneet: Contributed towards the implementation for SPD, worked on the technical background for the implementation, documentation, generated graphs, contributed to tutorial

# 8. References

[1] Sculley, D. "Large scale learning to rank." *NIPS Workshop on Advances in Ranking,* 2009.

[2] Liu, Tie-Yan, et al. "Letor: Benchmark dataset for research on learning to rank for information retrieval." *Proceedings of SIGIR 2007 workshop on learning to rank for information retrieval*. Vol. 310. 2007.