CS 276 / LING 286 Spring 2015

# Programming Assignment 4 – Learning to Rank

Due: Thursday, May 28, 2015, 11:59 PM PDT

In the previous assignment, we have examined various ways of ranking documents given a query; however, weights for different features were not learned automatically but set manually. As more and more ranking signals are investigated, integrating more features becomes challenging as it would be hard to come up with a single ranking function like BM25 for arbitrary features. In this assignment, you will be investigating different approaches to the learning to rank task that you have learned: (1) the pointwise approach using *linear regression* and (2) the pairwise approach employing *support vector machines*. The goal is to let these algorithms learn weights automatically for various features.

## 1 Overview

This assignment should be done in teams of two or individually. More specifically, it involves the following tasks:

1. Implement an instance of the pointwise approach with *linear regression* based on basic tf-idf features (§2).

2. Implement an instance of the pairwise approach, the *ranking SVM* method, using basic tf-idf features (§3).

3. Experiment with more features such as BM25 and PageRank (§4).

4. Write up a summary report and answer questions we asked for each task.

5. For extra credit, you can implement other ranking approaches, e.g., instances of the pointwise/pairwise/listwise methods (§5).

We use the same training data as in PA3 which consists of two sets: (a) training set (295 queries) and (b) development set (97 queries). We cleaned up PA3 training data by removing headers that are longer than 20 words. The development set is used to tune your model parameters, choose features, etc. The idea is to avoid over-fitting parameters that may yield good performance on training data, but perform poorly on new unseen data. You are required to **train your models on the training set**, and **report performance evaluated on the development set** throughout. The evaluation metric used is **NDCG** as in PA3. There will be a **hidden test set** used in evaluating your work.

Please **read the grading guideline** in Section 6 carefully to make sure you accomplish what we expect.

The description below on machine learning algorithms for the learning to rank task is largely based on Hang Li's tutorials `http://research.microsoft.com/en-us/people/hangli/li-acl-ijcnlp-2009-tutorial.pdf`.[1]

## 2 Pointwise Approach and Linear Regression (Task 1)

In ranking, each query $q_i$ will be associated with a set of documents (like a group), and for each document $j$, we extract a query-document feature vector $\boldsymbol{x_{i,j}}$ as illustrated in Figure 1. There is also a label $y_{i,j}$ associated with each query-document vector $\boldsymbol{x_{i,j}}$.

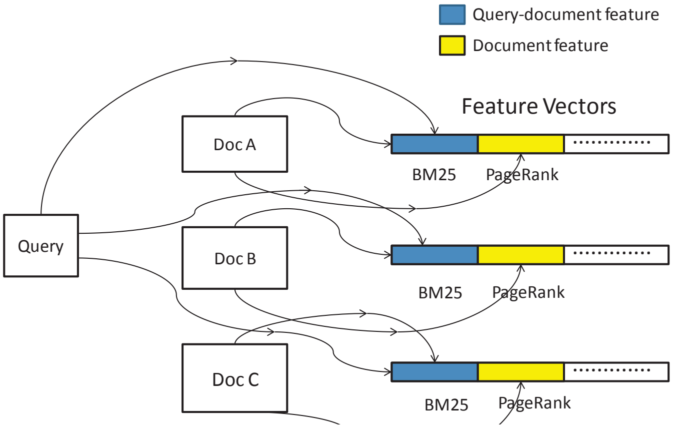

Figure 1: Feature Extraction.

In the pointwise approach, such group structure in ranking is ignored, and we simply view our training data as $\{(\boldsymbol{x_i}, \mathrm{y}_i)\}$, i=1 . . . $m$, where each instance consists of a query-document feature vector $\boldsymbol{x_i}$ and a label $\mathrm{y}_i$ (which is a relevance score as in PA3). The ranking problem amounts to learning a function $f$ such that $f(\boldsymbol{x_i})$ closely matches $y_i$.

[1] There is also a book version of the tutorial which is freely accessible on the Stanford network `http://www.morganclaypool.com/doi/abs/10.2200/S00348ED1V01Y201104HLT012`.

In this task, we consider a very simple instance of the pointwise approach, the *linear regression* approach. That is, we will use a linear function $f$ which gives a score to each query-document feature vector $\boldsymbol{x}$ as follows $f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x} + b$. Here, the weight vector $\boldsymbol{w}$ and the bias term $b$ are parameters that we need to learn to minimize a loss function:

$$\sum_{i=1}^{m} (f(\boldsymbol{x_i}) - y_i)^2 \tag{1}$$

This formulation is also referred to as the *ordinary least squares* approach.

### 2.1 What to do

Here are the specific things you need to do for this task:

1. Represent each query-document pair as a five-dimensional vector of tf-idf scores, each of which corresponds to a field – url, title, header, body, and anchor.

   Specifically, given a query vector $q$ (idf scores) and a term frequency vector $t_f$ of a document field $f$, the tf-idf score is $q^\top t_f$. (Note: you could try using either the raw or the normalized term frequency vectors as in PA3.)

2. Train a linear regression model. We strongly recommend using the Weka package `http://www.cs.waikato.ac.nz/ml/weka/` (see Section 8 for more details).

3. Given a learned weight vector $\boldsymbol{w}^*$, you can now directly compute score for each query-document vector $x$ as $f(x) = \boldsymbol{w}^{*\top} \boldsymbol{x} + b$ and rank based on that score. **Report the NDCG performance** achieved on the development test data.

## 3 Pairwise Approach and Ranking SVM (Task 2)

To recap, in the pairwise approach, ranking is transformed into a pairwise classification task in which a classifier is trained to predict the ranking order of document pairs. In other words, instead of giving a numeric rank to each document, e.g., rank 1, 2, 3 for documents A, B, C respectively, the classifier will judge if a document is better than another for each pair of documents, e.g., if A is better than B, B is better than C, and A is better than C. This is the idea behind the Ranking SVM method, an instance of the pairwise approach as illustrated in Figure 2. In this task, you will be applying your knowledge about Support Vector Machines (SVMs) to build such a classifier.

Specifically, instead of working in the space of query-document vectors, e.g, $\boldsymbol{x_1}$, $\boldsymbol{x_2}$, and $\boldsymbol{x_3}$, we transform them into a new space in which a pair of documents is represented as the difference between their feature vectors (with respect to a query), e.g. $\boldsymbol{x_1}-\boldsymbol{x_2}$, $\boldsymbol{x_2}-\boldsymbol{x_3}$, and $\boldsymbol{x_1}-\boldsymbol{x_3}$. For each vector $\boldsymbol{x_i}-\boldsymbol{x_j}$, a label $+1$ is assigned if document $i$ is more relevant than document $j$ and for the reverse case, a label $-1$ is used.

Note that we do not make pairwise ranking facts out of either pairs of documents with the same relevance score or pairs of documents that were returned for different queries. If $\boldsymbol{x_i}-\boldsymbol{x_j}$ is a positive example, $\boldsymbol{x_j}-\boldsymbol{x_i}$ could be used as a negative one and vice versa, so

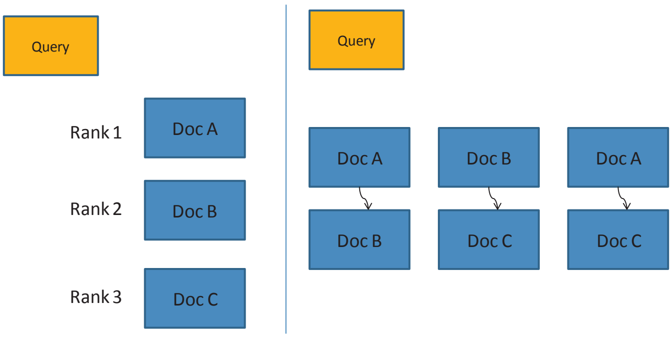

Figure 2: **Pairwise Classification**: if we have 3 documents with ranks as shown in the left side picture, then we produce pairwise ranking facts as in the right side picture, from which we proceed to train a classifier.

using either $\boldsymbol{x_i} - \boldsymbol{x_j}$ or $\boldsymbol{x_j} - \boldsymbol{x_i}$ is sufficient.[2] However, for SVM to work, it is important to **equally distribute** your training examples into classes.

## 3.1 Linear SVM Formulation

Formally, training data for the ranking SVM is given as $\{(\boldsymbol{x_i}^{(1)}, \boldsymbol{x_i}^{(2)}), y_i)\}, i = 1 \dots m$ where each instance consists of two feature vectors $(\boldsymbol{x_i}^{(1)}, \boldsymbol{x_i}^{(2)})$ and a label $y_i \in \{+1, -1\}$. The learning is framed as a Quadratic Programming problem:

$$\begin{gathered} \min_{\boldsymbol{w}, \boldsymbol{\xi}} \tfrac{1}{2}||\boldsymbol{w}||^2 + C \textstyle\sum_{i=1}^{m} \xi_i \\ \text{s.t. } y_i \boldsymbol{w}^\top \left(\boldsymbol{x_i}^{(1)} - \boldsymbol{x_i}^{(2)}\right) \geq 1 - \xi_i \\ \xi_i \geq 0, i = 1 \dots m, \end{gathered} \tag{2}$$

where $\xi_i$ are slack variables for soft-margin classification as you have learned in the SVM lecture and $C$ is a regularization term to control over-fitting. The weight vector $w$ corresponds to a linear function $f(x) = w^\top x$ which can score and rank documents.

## 3.2 Non-linear SVMs

General SVM formulations replace $\boldsymbol{x_i}$ in Eq. 2 by $\phi(\boldsymbol{x_i})$ to achieve the effect of mapping training vectors $\boldsymbol{x_i}$ into a higher (maybe infinite) dimensional space through the function $\phi$. By lifting the feature vectors into higher dimensional spaces, we hope to make the separation of training examples easier. To keep the computation reasonable, in practice, we do not need to compute $\phi(\boldsymbol{x_i})$ explicitly but rather use a kernel trick to only compute $K(\boldsymbol{x_i}, \boldsymbol{x_j}) = \phi(\boldsymbol{x_i})^\top \phi(\boldsymbol{x_j})$. In the linear case, $K(\boldsymbol{x_i}, \boldsymbol{x_j}) = \boldsymbol{x_i}^\top \boldsymbol{x_j}$. For non-linear kernels, one popular choice of non-linear kernels is the radial basis function (RBF):

$$K(\boldsymbol{x_i}, \boldsymbol{x_j}) = \exp(-\gamma||\boldsymbol{x_i} - \boldsymbol{x_j}||^2), \gamma > 0$$

[2] With respect to the Quadratic Programming formulation in Eq. 2, a negative instance essentially yields the same constraint as the positive one.

The RBF function is also the default kernel used in the LibSVM library `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`.

### 3.3 What to do

Here are the specific things you need to do for this task:

1. Using the same basic tf-idf features as in Task 1, construct training data for the pairwise approach.

   Before computing the difference vectors for pairs of documents, you are **required to perform standardization** over query-document feature vectors. That is, to scale values of each feature type to have a mean of 0 and a standard deviation of 1. You need to **scale both train and test (development) data**. See Section 8.3 for details on how to do such standardization in Java.[3]

2. Train a *linear* SVM classifier. We strongly recommend using LibSVM wrapper in Weka `http://weka.wikispaces.com/LibSVM` (see Section 8 for more details). Given a learned weight vector $\boldsymbol{w}^*$, you can now directly compute the score for each query-document vector $x$ as $f(x) = \boldsymbol{w}^{*\top}\boldsymbol{x}$ and rank based on that score. **Report the NDCG performance** achieved on the development data.

3. Train a *non-linear* SVM classifier with the RBF kernel. Unfortunately, you cannot perfom prediction by extracting the weight vector like in the linear SVM case. The correct way to perform prediction is by performing pairwise comparisons (your non-linear SVM classifier is trained to tell you which document is better in a pair) and infer rankings from the comparisons. **Report the NDCG performance** achieved on the development data.

   Note: You might want to do a grid search over your paramemters. In the case of the RBF kernel, the two parameters to tune are $C$ (in the SVM formulation) and $\gamma$ in the RBF formula. As a starting point, we suggest you do a grid search for $C = \{2^{-3}, 2^{-2}, \dots, 2^{3}\}$ and $\gamma = \{2^{-7}, 2^{-6}, \dots, 2^{-1}\}$. A more detailed guide can be found in `http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf`.

## 4 More Features and Error Analysis (Task 3)

With the machine learning algorithms built in previous tasks, let's try to use more signals. We provide a list of feature categories for you to try below:

1. BM25F value derived with your best weights in PA3 Task 2.[4]

[3] The reason for performing standardization is because SVMs are not scale invariant, i.e., if a feature type has a large range of values, SVMs might over-optimize that particular feature and take long time to train. Note that, you might get a slightly lower NDCG score with standardization, but that is fine. Standardization will make it easier to compare among your systems as well as keep you free from worrying about some features having large values.

[4] You might want to consider a formula with/without PageRank and try adjusting only the $\mathrm{K}_1$ parameter in Eq. (4) of PA3 write-up.

2. Smallest window feature(s).

3. PageRank.

*Hint*: you might want to do an **ablation test**, i.e. start out by using all features, and try not consider a feature at a time. This will give you an idea of what signals are useful and what are not.

### 4.1 What to do

Here are the specific things you need to do for this task:

1. From the list of features above, find out which combinations of features help boost performance. Note that we expect you to try **all feature categories in the suggested list**. Report the **NDCG scores** achieved on the development test data for different combinations of features, e.g., those that give you progressive improvements.

2. Examine your ranking output, list at least 2 types of errors that your system tends to make and propose features that help fix them. For example, if your system tends to rank wrongly URLs that link to PDF documents, perhaps you might want to add a binary feature to indicate if a URL ends in ".pdf". Do the new features help fix the problems you observed? What about the NDCG scores achieved on the development test data? **Report your finding**.

Note that not all features will be as helpful as you might have expected to your overall NDCG score (C'est la vie!). However, if a signal does not improve rankings for certain queries, we would still be interested to know that you have tried. We will evaluate based on the depth of your analysis. Simply list rankings of documents for a query without telling us why a document is perferred to another will not give you full credit.

## 5 Extra Credit

**Please consider whether it is wise to attempt extra credit problems at this point in the quarter versus to practice exam problems for your final exam**.

### 5.1 Other Learning Methods

For extra credit, we suggest you to look at other learning methods as below and **pick at most two extensions**:

1. Use SVM Regression feature provided by LibSVM for the pointwise approach. See `http://alex.smola.org/papers/2003/SmoSch03b.pdf` for reference.

2. Experiment with other approaches: (a) Pairwise (RankNet, RankBoost) and (b) Listwise (AdaRank, ListNet). RankLib `http://sourceforge.net/p/lemur/wiki/RankLib/` provides implementations for these models.

3. Use word vectors released in `https://code.google.com/p/word2vec/` to help improve your system.[5]

   We will later provide a file containing a list of all words occuring in the train/dev/test data over different fields together with their vectors extracted from `https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit?usp=sharing`.

For the extensions you tried, report the NDCG performance achieved.

# 6 Grading

We will be evaluating your performance on a different test dataset, which will have queries drawn from the same distribution as the training set. The format for the dataset is the same as *queryDocTrainData*.

**Task 1 [10%]**: 10% if your NDCG score on the development data is above 0.83.

**Task 2 [25%]**: 25% if your NDCG score on the development data is above 0.845.

**Task 3 [20%]**: 20% if your NDCG score on the development data is above 0.86.

**Report [35%]**: 15% for the various design choices you made in the different tasks and reporting detailed NDCG scores in a clear manner, i.e., it should be clear from your report the performances achieved for each task and for different feature combinations. 20% for the error analysis and discuss effects of different features in Task 3 (scaled by the depth of your analysis).

**Code [10%]**: A check to ensure that your code is doing the right things.

**Extra Credit [5-15%]**: 5% for each extension tried together with your justification of why your extension works or does not work. 5% for improved performance achived over task 2 with your extension(s).

We will give extra credit for best ranking systems in the entire class, which is based on the NDCG scores computed on our hidden test data. 10% for the top 5 systems and 5% for the next 15 systems.

# 7 Deliverables

## 7.1 Input/Output format

The starter code contains a script named *l2r.sh* which can be invoked as below. **Please read this carefully for a smooth submission process**.

[5] To find out if two words have similar meaning or to add more features such as the similarity scores between the query text and the texts in different fields.

```
$ ./l2r.sh <train_signal_file> <train_rel_file> <test_signal_file>
          <task> [out_file]
```

The arguments are:

- *train_signal_file*: training signal file with the same format as files *pa4.signal.train*.
- *train_rel_file*: training relevance file with the same format as files *pa4.rel.train*.
- *test_signal_file*: test signal file with the same format as file *pa4.signal.dev*.
- *task*: either 1 (Task 1), 2 (Task 2), 3 (Task 3), or 4 (Extra Credit).
- *out_file*: output file for ranked queries. This argument is optional, and if no file is specified, the output will be printed to **stdout**.

**Do not alter the arguments and make sure you use the best set of features that produces the best NDCG score on the development test data for each task**. If we cannot reproduce similar NDCG performances as you reported, marks will be deducted.

You can read in whatever training data you need to build your model but we will not pass that as inputs to the script. The script should output (to **stdout**) for each query, the query followed by the documents (in the form of urls) in decreasing rank order specified by your ranking. You can print anything you want to stderr.

For example, if query $q1$ has three documents and the file is listed as follows:

```
query: q1
  url: http://xyz.com
  url: http://def.edu
  ...
  url: http://ghi.org
```

If your ranking algorithm gives a rank of 1 to ghi.org, 2 to xyz.com and 3 to def.edu, then you should output the order in the following format:

```
query: q1
  url: http://ghi.org
  url: http://xyz.com
  url: http://def.edu
  ...
```

In your experiments, you could execute the following commands:

```
$ ./run.sh <train_signal_file> <train_rel_file> <test_signal_file>
          <test_rel_file> <task>
```

which will give you performance results on the development data.

### 7.2 Report

Answers to questions we asked for each task should be included in a write-up file *report.pdf*. If there is any important design choices undertaken or different configurations used in your machine learning algorithms that boost performances, you could report them. It must be a maximum of **4 pages long**. Reports that are too terse or those that do not contain enough analysis will not get full credit.

### 7.3 Partner

A list of people who worked together on the assignment, in *people.txt*, e.g.:
<sunet id1>
<sunet id2>

## 8 Code Guide

We encourage you to use the *Weka* package in Java for this assignment as it has a wide range of libraries for various machine learning algorithms including those mentioned in this assignment. It will save you from intensive coding and you will benefit from it not only in PA4, but also in the long run. The corresponding jar files have already been included in skeleton package, so if you are using eclipse as your IDE, you should import the project through *build.xml*, and then all necessary external packages can be invoked by normal.

We provide a skeleton code package, in which *Learning2Rank.java* is the entry point of the entire project. Running as follows:

```
$ java -cp bin:lib/weka.jar
       cs276.pa4.Learning2Rank <train_signal_file> <train_rel_file>
       <test_signal_file> <task> [out_file]
```

### 8.1 Linear Regression

Here we provide some code snippets and walk you through steps of training a linear regression model:

**Step 1** – build training data $X$ and labels $y$:

```
Instances dataset = null;

/* Build attributes list */
ArrayList<Attribute> attributes = new ArrayList<Attribute>();
attributes.add(new Attribute("url_w"));
attributes.add(new Attribute("title_w"));
attributes.add(new Attribute("body_w"));
attributes.add(new Attribute("header_w"));
attributes.add(new Attribute("anchor_w"));
attributes.add(new Attribute("relevance_score"));
dataset = new Instances("train_dataset", attributes, 0);

/* Set last attribute as target */
dataset.setClassIndex(dataset.numAttributes() - 1);

/* Add data */
double[] instance = {1.0, 1.0, 1.0, 1.0, 1.0, 1.0};
Instance inst = new DenseInstance(1.0, instance);
dataset.add(inst);
```

Here, the dataset contains 6 columns. We set the last column (relevance score) as the target we want to predict, and all other columns would be the features. Then we add one row to the dataset, in which all fileds are set to 1.0.

**Step 2** – train a linear regression model:

```
import weka.classifiers.functions.LinearRegression;

...

LinearRegression model = new LinearRegression();
model.buildClassifier(dataset);
```

**Step 3** – build testing data: The process is the same as Step 1. Notice that for the target column, you can set whatever value you want. This column won't be involed in prediction process.

**Step 4** – predict labels:

```
instances test_dataset = ...;   /* The dataset you built in Step 3 */
double prediction = model.classifyInstance(dataset.instance(i));
```

## 8.2 Support Vector Machines

In Weka, to create an SVM model, use:

```
LibSVM model = new LibSVM();
```

The default kernel of the SVM model in Weka is **RBF kernel**. We also encourage you to try other kernels and explore their performance, especially **Linear kernel**. To change the kernel, use:

```
model.setKernelType(new SelectedTag(LibSVM.KERNELTYPE_LINEAR,
                                    LibSVM.TAGS_KERNELTYPE));
```

There are also several other parameters you can play with, for example, the regularization term $C$. You can set these parameters using:

```
model.setCost(C);
model.setGamma(gamma); // only matter for RBF kernel
```

### 8.3 Standardization

Below are some code to do standardization:

```
Standardize filter = new Standardize();
Instances X = ... /* construct dataset */
filter.setInputFormat(X);
Instances new_X = Filter.useFilter(X, filter);
```

### 8.4 Weights

To retrieve weights of the model, use:

```
double[] weights = model.coefficients();
```

For the linear regression model, the weights array contains (number of features + 2) coefficients. The 2 additional weights come from one coefficient for the target attribute, which should always be 0, and one coefficient as intercept.

For SVM model, notice weights are only meaningful if you use linear kernel. We modified the weka.classifiers.functions.LibSVM to return the weights for you based on the guideline here `http://www.csie.ntu.edu.tw/~cjlin/libsvm/faq.html#f804`. The weights array contains one coefficient for each feature and one for the intercept.

### 8.5 NDCG Score

We've updated NdcgMain.java in PA3. Now you can directly use it in your code rather than through a script. This should be helpful when you tune parameters:

```
NdcgMain ndcg = new NdcgMain(relFile);
double ndcgScore = ndcg.score(rankedFile);
```

For more details, please refer to Weka manual: `http://sourceforge.net/projects/weka/files/documentation/3.7.x/WekaManual-3-7-11.pdf` and documentation: `http://weka.sourceforge.net/doc.dev/`

## 9 Submission

Make sure the shell script *l2r.sh* runs as explained in Section 8. Then, call the below command and select an option to submit (report, tasks 1, 2, 3, or extra credit):

```
python submit.py
```

The submit script will execute your model on the development and test data on your machine. It will also check to see whether you have the right number of query-document pairs for each data set. Remember, it is **an honor code violation** to knowingly take a look at the test set.

Since *NdcgMain* will be used to evaluate the NDCG scores of your model on the test data, your output file should conform to the format mentioned in 7.1. Additionally, we have provided starter code in Java for your convenience.

As a last step, you will have to submit your code through Coursera. Zip your assignment directory using a zip archiver (without the PA1 corpus), name it as *SUNetid1_SUNetid2.zip* and upload it on the Coursera assignments page for the "Code" section using the simple uploader you have used before. Do not worry about making submissions for the other parts as we will populate those with scores using your reports and code.

Only one person in the group needs to submit the assignment (Please submit everything from the same member's SUNet ID and with the same *people.txt* file)