
Executive Summary

The objective of this project is to improve the matching functionality of software named “Subsift” which is designed to automatically match submitted papers and potential reviewers. I make improvement to Subsift by learning from reviewers’ feedbacks on the ranked list of papers provided to each reviewer by Subsift and incorporating the learning result into Subsift, so that the matching result of reviewers with new incoming papers will match better their individual preferences.

This project is of 50% theoretical investigation and 50% software implementation. The problem of learning from different formats of feedbacks falls into the Machine Learning field “learning to rank”. I did a bunch of background research into algorithms in this field. Then I implemented a web application on which the reviewers can give three formats of feedbacks. The algorithms to process feedbacks are selected from those I closely studied during the background research phase.

In the research review, I expounded important algorithms in four major branches of “learning to rank”. In this phase, my achievements are listed as follows:

- Designed three feedback formats which are practical for the reviewers to give and easy to be learned by existing learning algorithms (see section 6, 7, 8).
- Designed scientific working flow to obtain feedbacks; implemented user-friendly interfaces (see section 4).
- Designed java classes with high cohesion and low coupling to complete the data processing work along the working flow efficiently (see section 5, 6, 7).
- Did thorough study of two algorithms: Prank and Ranking SVM (see section 6.1 and section 7.1); Implemented code dealing with pointwise and pairwise feedbacks by applying these two algorithms respectively (see section 6.2 and section 7.2). Note that the training algorithm of Ranking SVM is from an external java library called LIBSVM, which I explained in section 7.2.3.
- Creatively transformed listwise feedback into pointwise and pairwise feedback formats and processed it with Prank or Ranking SVM based on the user’s choice (see section 8).
- In consideration of the noisiness of reviewers’ feedbacks, I implemented a jsp page asking reviewers to give secondary feedback on the renewed ranked list produced after incorporating feedback; tuned the degree of feedback’ influence according to the reviewers’ own evaluation on the renewed list (see section 9.1).
- Designed an evaluation model to assess and compare the effectiveness of the algorithms (see section 9.2); did input parameter optimization based on the results of the evaluation model (see section 9.3).
- Solved the problem of communicating with Subsift from my application, including getting/posting a considerable amount of data, creating/deleting folder, etc (see section 3).
- Proposed future improvement directions (see section 10).

Acknowledgements

I would like to thank:

- . My Project Supervisor, Prof. Flach for introducing me to this fascinating area of Learning to Rank and for his patient guidance, he is the best supervisor I have ever met or heard, I can't thank him more
- . My parents for their comfort when I got depressed
- . My friend Kevin, for his company throughout this difficult time
- . My friend Edith, for her encouragement

Table of Contents

1	Introduction.....	1
1.1	Aims and Objectives	1
1.2	Theoretical Investigation.....	2
1.3	Software Implementation.....	2
2	Background Research on Learning to Rank.....	3
2.1	Conventional Approach	3
2.2	Pointwise Approach	4
2.3	Pairwise Approach	4
2.4	Listwise Approach.....	5
3	Subsift.....	6
3.1	What is Subsift	6
3.2	Theoretical basis	7
3.2.1	Vector Space Model	7
3.2.2	Representational State Transfer (REST).....	8
3.3	Subsift REST API	8
3.4	REST API Explorer	11
3.5	Improvement on Subsift Using Reviewers' feedback	13
4	Main Working flow.....	14
5	Basic Class Structure	17
6	Feedback type 1 --- Top n and Bottom n Papers	20
6.1	Algorithm Explanation.....	20
6.2	Software Implementation.....	22
6.2.1	Illustration for JSP Pages	22
6.2.2	Illustration for Class PointWiseFeedbackProcessing.java.....	25
7	Feedback type 2 --- n ordered instance pairs	33
7.1	Algorithm Explanation.....	33
7.1.1	SVM.....	33
7.1.1.1	Maximize Margin.....	33
7.1.1.2	Lagrangian formulation for linear separable case.....	34
7.1.1.3	Lagrangian formulation for nonlinear separable case.....	36
7.1.1.4	Optimality Criteria for SVM problem	37
7.1.2	Cast Preference learning as Ordinal Regression Problem	38
7.1.3	Preference Learning based on SVM	40
7.2	Software Implementation.....	41
7.2.1	Illustration for JSP Pages	41
7.2.2	Illustration for Class PairWiseFeedbackProcessing.java.....	43
7.2.3	Illustration for LIBSVM	48
8	Feedback type 3 --- Full list of n papers	49
9	Algorithm Evaluation and Parameter Optimization	50

9.1 Evaluation from Reviewer	50
9.2 Evaluation Model.....	51
9.2.1 Evaluation Steps.....	51
9.2.2 Evaluation Algorithm.....	52
9.2.3 Evaluation Result	53
9.3 Parameter Optimization	54
10 Future Work	55
10.1 Larger Scale Experiment.....	55
10.2 Parameter optimization	56
10.3 Input Inspection.....	56
10.4 Interface Optimization	57
11 Conclusion	57
Bibliography.....	58

1. Introduction

As college students or researchers, many of us have the experience of submitting papers to journals or conferences in order to publish them. We can imagine that there must be a group of reviewers (instead of just one person) to assess those papers. After receiving a serious amount of papers, it would be an extremely time-consuming task for the committee chair to match those papers with potential reviewers based on their professional fields if the work is done manually.

Subsift is a tool coming to their rescue. It matches submitted papers to potential reviewers based on the similarity between those papers and the reviewers' published works which can be gotten from online databases such as Google Scholar [1]. At present, this software applies a simple Vector Space Model to perform the matching function, which doesn't have any learning capability.

1.1 Aims and Objectives

The ultimate goal of this project is to add learning capability to Subsift. An application is designed to enable Subsift learn from reviewers' feedbacks on the ranked list of papers produced by Subsift to each of them so as to better match new coming submissions to reviewers. Since this project is of 50% investigation and 50% software implementation, the aim can be divided into two parts.

In the investigatory part, my objective is to have a good understanding of algorithms available solving "learning to rank" problem, design feedback formats based on the study, and to have exhaustive study of algorithms dealing with the formats of feedbacks that I designed. Note that the feedback formats designed should be possible to be dealt with by existing learning algorithms and practical for the reviewers to give in the same time.

In the programming part, my objective is to implement an application on which the reviewers are able to give 3 formats of feedbacks that I designed in the investigatory part and evaluate the effectiveness of the feedback processing algorithms. Specifically, first I need to implement a user-friendly interface; second I need to convert the general, theoretical expression of algorithms solving "learning to rank" problem that I studied in the investigatory part into code and decide the parameters in the algorithms giving full consideration to the practical situation, which involves both thorough understanding of the algorithms and certain degree of creativity; last but not least, I need to design reasonable methods to evaluate and compare the effectiveness of feedback processing algorithms.

1.2 Theoretical Investigation

Let us first have a wide guess on what kind of feedbacks we can ask the reviewer to give after providing him with a ranked list of papers. We might invite him to give each paper a score within the range of $[1, 10]$; or we can ask him to give us several pairs of papers indicating that he prefers the first ones comparing to the second ones; it is also possible for the reviewer to give his own ordered list of all (or some of the) papers. Although those feedbacks are of different formats, they all specify some partial orders between items in the feedback.

“Learning to rank” is a Machine Learning problem in which the goal is to construct a ranking function from training data, where the training set is a list of items with some partial order specified between items of the list. It is obvious that learning from reviewers’ feedbacks falls exactly into the area “learning to rank”. Therefore, I did a bunch of background research into all kinds of algorithms designed to solve “learning to rank” problem.

The algorithms in this field are divided into conventional approach, pointwise approach, pairwise approach and listwise approach based on what kind of training data is dealt with. Algorithms in conventional approach solve the problem of ranking which regards binary judgments or multi-valued discrete as “non-ordered” categories or real values; Algorithms in pointwise approach fit to training sets containing individual instances with rankings(or say, scores) attached to them; Algorithms in pairwise approach are suitable for training sets with pairwise instances and there is label on each pair indicating preference; Algorithms in listwise approach apply to training sets in which a list of full ranked documents associated with a query serves as an instance.

I did a detailed study of algorithms in this field before designing and implementing the feedback processing application, which is summarized in section 2.

1.3 Software Implementation

The software I implemented is a web application. It can deal with three types of feedbacks described above. In pointwise approach, it asks a reviewer to select top n and bottom n papers from a ranked list of m papers provided by Subsift to him. Then the software deals with this type of feedback using an algorithm called “prank”. In pairwise approach, it requires reviewers to provide n pairs of papers indicating they prefer the former ones to the latter ones. This kind of feedback will be converted to a traditional SVM problem and processed by Ranking SVM. In listwise approach, a ranked list of n papers will be required from the reviewer. This problem can be either solved by Prank or Ranking SVM. If the later one is chose, the list will be converted

into $n-1$ pairs of instances: (paper1, paper 2), (paper 2, paper 3)... (paper 9, paper 10).

The outcomes of those algorithms are the same, which is a “local weight” vector for terms in the reviewer’s profile. After updating the reviewer’s profile with the vector calculated from the reviewer’s feedback, the matching result of this reviewer with future incoming papers will suit better to his individual preference.

In order to evaluate the effectiveness of those algorithms, after the “local weight” vector being updated, a new ranked list of m papers will be calculated and presented to the reviewer and he will be asked whether feeling happy about the list. Based on the evaluation of the reviewer, the parameters of the algorithm will be slightly altered to trim the influence of the reviewers’ first feedback on the reviewer’s profile. For example, if the reviewer says that he is extremely unsatisfied with the result, then the value added or subtracted from the base value of local weights will be lessened ten times to minimize the influence of the feedback.

I also designed an evaluation model to evaluate the effectiveness of the feedback processing algorithms, as well as the difference in ranking accuracy improvement by having different numbers of training instances.

2 Background Research on Learning to Rank

In this section, I will expound briefly the background research work I did. The algorithms I used (Prank and Ranking SVM) to deal with feedbacks in my application are selected from the algorithms that I have closely studied in this phase.

2.1 Conventional Approach

In conventional approach classification or regression methods are directly applied to classify instances into two classes (relevant and irrelevant). With a simple post-process to S-CART, we can directly apply regression tree to ranking problem [2]; there are also some models straightforwardly applying classification methods in solving ranking problem, including the BIR model [3], language model [4], which fall into the category of generative classification; and Maximum Entropy model [5], Support Vector Machine [3], which fall into the category of discriminative classification.

Because those models depend on the assumption of independency of instances which is very likely to be untrue in the real world [6]; besides, it doesn’t give consideration to unique properties of ranking for information retrieval (for example, the relative order might be more important rather than predicting accurate category or value), we proceed to more advanced ranking models which are categorized into pointwise approach, pairwise approach and listwise approach.

2.2 Pointwise Approach

Algorithms in this approach learn from an ordinal training set to tune thresholds of the ordinal categories and support vectors to make ranking produced by the ranking rule fit to the real ranking of instances in the training set. This is just slightly different from the conventional approach as the objects it deals with are discrete classes instead of real values and the output is ordered classes rather than non-ordered classes.

The training process is to solve the problem of *ordinal regression* which can be referred as a learning paradigm as follows: Given a training sample sequence $(\vec{x}^1, y^1), (\vec{x}^2, y^2), \dots, (\vec{x}^t, y^t)$ donated by $S_{\vec{x}, Y}^n$, and a ranking rule H from \vec{x} to Y . (\vec{x}^t, y^t) are instance-ranking pairs. \vec{x}^t represents an instance which is in \mathbb{R}^n ; y^t donates corresponding rank which is an element of a finite sample set $S = \{r^1, r^2, r^3 \dots r^n\}$ with “>” as a total order relation which means r^i is preferred over r^{i+1} . The learning procedure selects a mapping rule h^L using the predefined loss L . L defines the difference between $h^L(\vec{x})$ and Y . Problem of ordinal regression is to tune parameters in h^L so as to minimize the risk functional $R_{\text{emp}}(h^L, S)$ so that the instances in the training sample $S_{\vec{x}, Y}^n$ are mapped into the right intervals. This is based on the principle called Empirical Risk Minimization (ERM) [7, 8].

In pointwise approach, I did close study of Prank, an algorithm trying to cast instances into the correct intervals [7, 9], which is based on ordinal regression; Ranking with large ranking principle [10], an algorithm trying to maximize the margins between classes which the instances fall in, there are two different strategies: “fix margin” strategy and “sum of margin” strategy, which are both based on SVM.

As explained above, the input of training is a sequence $(\vec{x}^1, y^1), (\vec{x}^2, y^2), \dots, (\vec{x}^t, y^t)$. This kind of models can't deal with the circumstances under which the feedback is a pair or a list of documents instead of a single one. Those situations are tackled by algorithms in pairwise and listwise approaches.

2.3 Pairwise Approach

Pairwise approach uses pairwise user feedback to training the learning algorithm. In this approach, I studied carefully Ranking SVM [8], which applies preference learning to ordinal regression problem. It converts pairwise problem into a traditional SVM problem. RankBoost is another algorithm specially designed for pairwise feedbacks which gives different degrees of attention to each pairs depending on their weights (importance). It figures out the final ranking by combining many “weak rankings” through by a “weak learner”, specific scores are of no use in this algorithm, instead, it totally relies on relative orders [11]. There are many other algorithms falling in this category, such as RankNet, a system using cross entropy as loss function and gradient

descent as algorithm to train a Neural Network model to produce an ordered list.

Pairwise approach has many advantages. First, it makes use of existing classification models (SVM, boosting, Neural Network). Also, pairwise preference is easier to given comparing to giving precise scores; moreover, in some sense, pairwise instances fit better to the circumstance in which feedbacks' associated scores have different semantics as they're from different people (70 might mean "excellent" for one reviewer while it means "not bad" for another reviewer).

A shortcoming of pairwise approach is that it treats every feedback pair identically, which means the group structure is ignored. For example, in ranking SVM, the position of the pair in the ranking list is invisible to the loss function (31). Another shortcoming is that the computation cost is relatively high as there are so many possible pairs in a training set [12].

2.4 Listwise Approach

For methods in listwise approach, in the learning process, unlike algorithms in pointwise approach and pairwise approach, which treat a single document associated with a ranking or an ordered pair of documents as a learning instance, algorithms in listwise approach take a list of perfectly ranked documents associated with a query as one learning instance, so that group structure is taken into consideration. More formally, listwise approach can be described as follow:

Donate $d^{(i)} = (d_1^{(i)}, d_2^{(i)}, \dots, d_n^{(i)})$ to be the document list associated with query $q^{(i)}$, and $y^{(i)} = (y_1^{(i)}, y_2^{(i)}, \dots, y_n^{(i)})$ to be the corresponding degree of relevance(scores) of $d^{(i)}$ to $q^{(i)}$. Future vector $x_j^{(i)} = \varphi(d_j^{(i)}, q^{(i)})$, $j = 1, 2, \dots, n_i$, $i = 1, 2, \dots, m$. So that each instance has a list of features $x^{(i)} = (x_1^{(i)}, x_2^{(i)}, \dots, x_n^{(i)})$, together with corresponding scores. $(x^{(i)}, y^{(i)})$ is viewed as one learning instance. So that the learning space is donated by $\tau = \{ (x^{(i)}, y^{(i)}) \}_{i=1}^m$ [14].

In learning, for each feature vector $x_j^{(i)}$, the learning algorithm will assign it a score $f(x_j^{(i)})$, so scores form a list $z^{(i)} = \{f(x_1^{(i)}), f(x_2^{(i)}), \dots, f(x_n^{(i)})\}$. The objective of the learning algorithm is to minimize: $\sum_{i=1}^m L(y^{(i)}, z^{(i)})$ [12].

In this approach, I had a detailed study of one algorithm: ListNet. It maps the ranking produced by the ranking rule and the real ranking to two probability distributions using Top One probability model, then it takes Cross Entropy as the loss function, uses Neural network to compute the score list and Gradient Descent to minimize the loss in each round [12].

Listwise approach takes the group structure into account and it solves the ranking problem straightforwardly. ListNet is a primary model in listwise approach which minimizes loss by minimizing the difference of corresponding probability distribution. There're other models following this train of thought such as listMLE, which instead maximizes sum of the likelihood function of two probability distributions, it also uses Neural Network as ranking model, but it chooses Stochastic Gradient Descent(SGD)

as the algorithm to train the model. [13] proves its effectiveness through experiments.

3. Subsift

Peer review of written works is important for academic process which should give feedback of high quality to submissions to conferences, journals, funding bodies, etc. No need to say, making a good match between submissions and reviewers is an important link to ensure the quality. In the past, the matching process consisted of basically two steps. In step 1, the reviewers were given all submissions and they bided on the ones they would like to review. In step 2, the committee chair made allocation decisions depending on the bids of reviews [1].

Obviously, doing the matching and allocating job in a complete manual way is quite demanding. For reviewers, going through more than hundreds abstracts in order to select a few papers on which to place a high bid is too time-consuming; for the committee chair, with only the reference of the bids of reviewers, making a good match is still difficult, especially for the submissions with too many or too few bids.

3.1 What is Subsift

Subsift, short for submission sifting, was designed to assist the matching and allocating process. After deploying Subsift, there would be three steps in allocating submissions to corresponding reviewers. Step 1, for any paper, each reviewer's bid is produced by Subsift based on textual similarity between the papers' abstract and the reviewers' publication titles, which can be found in the DBLP bibliographic database. Step 2, each reviewer is sent an email containing a personalized Subsift-generating list of papers ordered by their similarity to his own published works, then the reviewer can bid on the ones he would like to review based on the list. Step3, after all reviewers' bids are submitted, the committee chair will be able to allocate paper according to reviewers' bids produced in step 2 while consulting a similarity ranked list of reviewers for each paper produced in step 1 to assist them in allocating papers with too few or too many bids [1].

Subsift magnificently reduces the work load for reviewers since they only need to check whether the list produced by Subsift is reasonable instead of looking for papers which they might be interested in from a sea of submissions. Meanwhile, it also simplifies the allocating work of the committee chair by providing them a similarity ranked list of reviewers for each paper as reference when the decision is hard to make only with the bids of reviewers.

The Subsift tools were originally designed to help with the matching process of peer review for the ACM SIGKDD'09 data mining conference. Later it has been used in many major data mining conferences. Now Subsift is not only applied to the field of academic paper review, it is developed into a tool with more general usage such as personal discovery, mashup, etc. It is now applied in some interesting fields, for

example, it is used by some profiling research groups and organizations [1].

3.2 Theoretical basis

The next question is: how does Subsift match submissions and reviewers, and thus generate a similarity-ordered list of reviewers for a particular submission (and vice versa)? The theoretical basis of the matching functionality of Subsift is the well known *vector space model* from information retrieval [14]. In addition, Subsift software relies on the *representational state transfer (REST)* design pattern for web services [15]. In this section, I'll give brief overview of these topics in turn.

3.2.1 Vector Space Model

The canonical task in information retrieval is, given a query q in the form of a list of terms, rank a set of text documents D in order of their similarity to the query. The vector space model is a common approach to solving this problem.

Vocabulary V is the set of distinct terms in D , which defines a vector space with dimensionality $|V|$. Each d in D is represented by a set of terms occurring in d . Thus each document d_j can be represented as vector $\vec{d}_j = [w_{1,j}, w_{2,j}, w_{3,j}, \dots, w_{N,j}]^T$ in the space, and query q can be represented as vector \vec{q} in the space. The angle θ between vectors \vec{d}_j and \vec{q} represents the similarity between query q and document d_j , which can be measured by its cosine, namely, the dot product of the vectors scaled to unit length [16]:

$$s(\vec{q}, \vec{d}_j) = \cos(\theta) = \frac{\vec{q} \cdot \vec{d}_j}{\|\vec{q}\| \cdot \|\vec{d}_j\|} \quad (1)$$

However, we can't directly use raw terms count \vec{q} and \vec{d}_j because in that case the result will treat all documents equally important and deviate towards long documents. Instead, we use *frequency-inverse document frequency (tf-idf) weighting scheme* in which the element of the feature vector is based on the frequency that a word appears in a document plus the penalty on that word depending on how many documents it appeared in. More formally, *term frequency* tf_{ij} of term t_i in the document d_j , and *inverse document frequency* idf_i of term t_i are defined as [16]:

$$tf_{ij} = \frac{n_{ij}}{\sum_k n_{kj}} \quad idf_i = \log_2\left(\frac{|D|}{df_i}\right) \quad w_{ij} = tf-idf_{ij} = tf_{ij} * idf_i \quad (2)$$

Where n_{ij} donates how many times are term t_i appears in document d_j , df_i donates how many documents in D in which term t_i appears [16].

In this way, the similarity between query q and document d_j can be calculated as:

$$S(\vec{q}, \vec{d}_j) = \frac{\sum_{i=1}^N w_{ij} * w_{iq}}{\sqrt{\sum_{i=1}^N w_{ij}^2} * \sqrt{\sum_{i=1}^N w_{iq}^2}} \quad (3)$$

In Subsift, we compare every document in one collection D1 (e.g. abstracts) with every document in another collection D2 (e.g. reviewer bibliographies) to produce a ranked list for each document. It aims at the importance of each term over the union of documents folders D1 and D2, so that $tf-idf_{ij}$ values are calculated based on the union of D1 and D2 [16].

3.2.2 Representational State Transfer (REST)

REST is a design pattern for web services that you can send requests to web sources and get response from it using HTTP protocol [4]. In REST, URIs represent resources and HTTP request methods define operations on those resources. The operations are implemented by adding verbs into the URIs, such as “get”, which returns a webpage in the form of an HTTP response which the browser displays. Table 1 shows five most significant HTTP request methods [16].

HTTP Method	Usage in REST	Changes Resource
GET	show and list operations	No
HEAD	exists operations to check if a resource exists	No
POST	create and compute operations	Yes
PUT	update and recompute operations	Yes
DELETE	destroy operations	Yes

Table 1 HTTP request methods, Source: Subsift Website

3.3 Subsift REST API

Subsift has an Application Programming Interface (API) which allows the functionality of Subsift to be incorporated into other softwares, namely API enables others' programs to call functions of Subsift. Subsift API follows the design principles of REST.

API is organized based on folders which are divided into 3 types: documents folder, profiles folder and Match Folder. A document is usually an external resource such as the content of a webpage or the abstract of a conference paper, which is an item of a Documents Folder. A profile is a summary of the unique features of a document comparing to other documents in the same documents folder, which is grouped into

Profile Folder. Two Profile Folders compare with each other to produce a Match Folder in which there is a match item produced using Vector Space Model for each profile item in the two profiles folder. For each term in each match item the *tf-idf cosine similarity*, and various related statistics are recorded. Figure 1 shows how information is stored in the Document Folder, Profiles Folder and Match Folder respectively. Knowing the inner structure of those folders is essential for the program to communicate with Subsift.

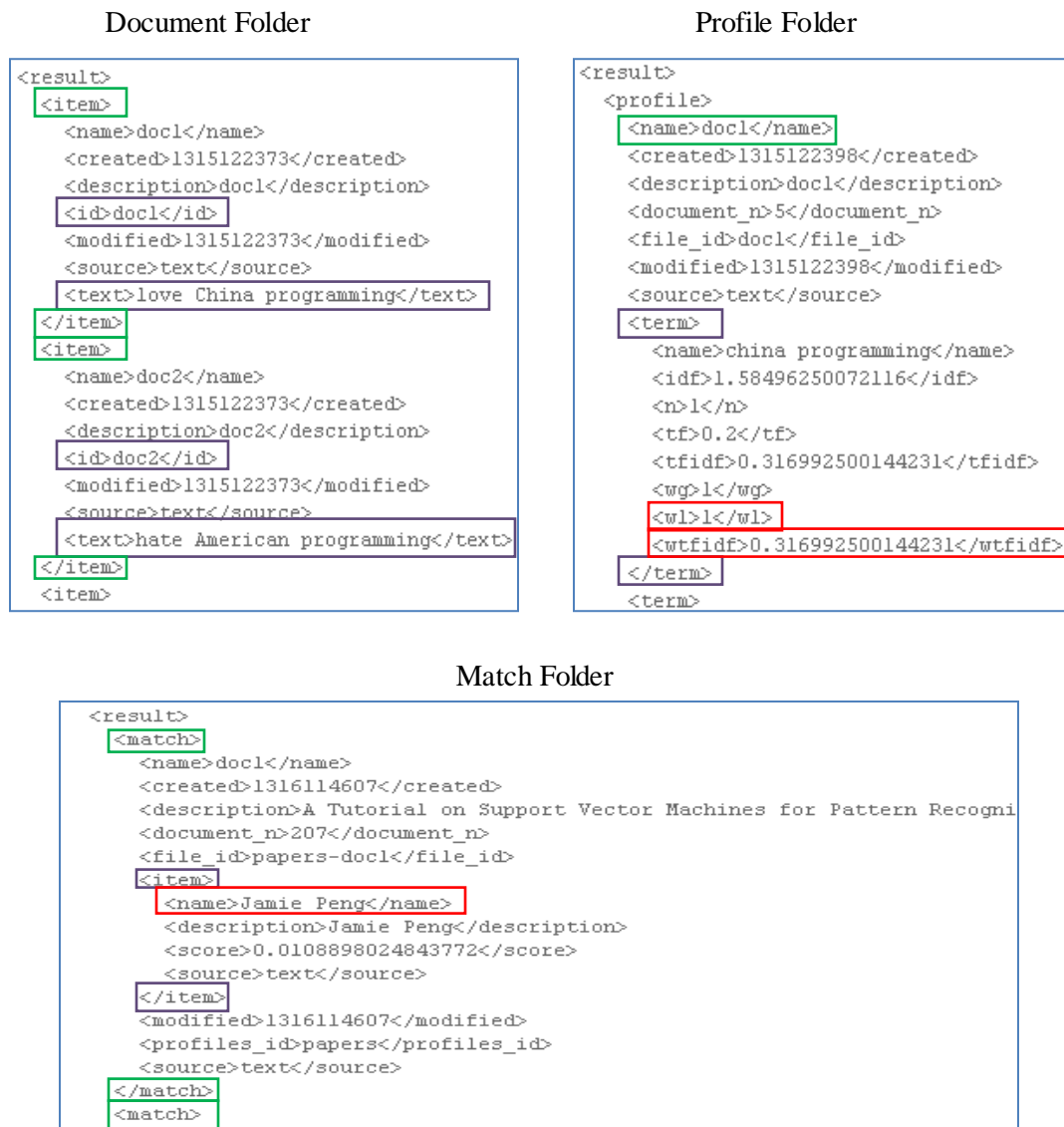


Figure 1 Inner Structure of 3 types of folders in Subsift, Source: Subsift Website

From Figure 1, we can see that from Document Folder, we can get the text of each document item. In Match Folder, there is a list of items from one Profile Folder (reviewers) for each item in the other Profile Folder (papers).

Profile Folder stores the *tf-idf* value of each term in each profile item. Besides *wtfidf* value, tag `<wl>` `</wl>` stores the value of local weight of the term; local term

weights apply to a specific reviewer or document. Match scores for terms are multiplied by their local term weights. Specifying term weights allows the importance of terms to be scaled down. A term weight of zero is equivalent to adding the term to the stop words list. A term weight of one is equivalent to not specifying any weight. Local weight (tag `<wl></wl>`) is the place where the result of feedback being incorporated.

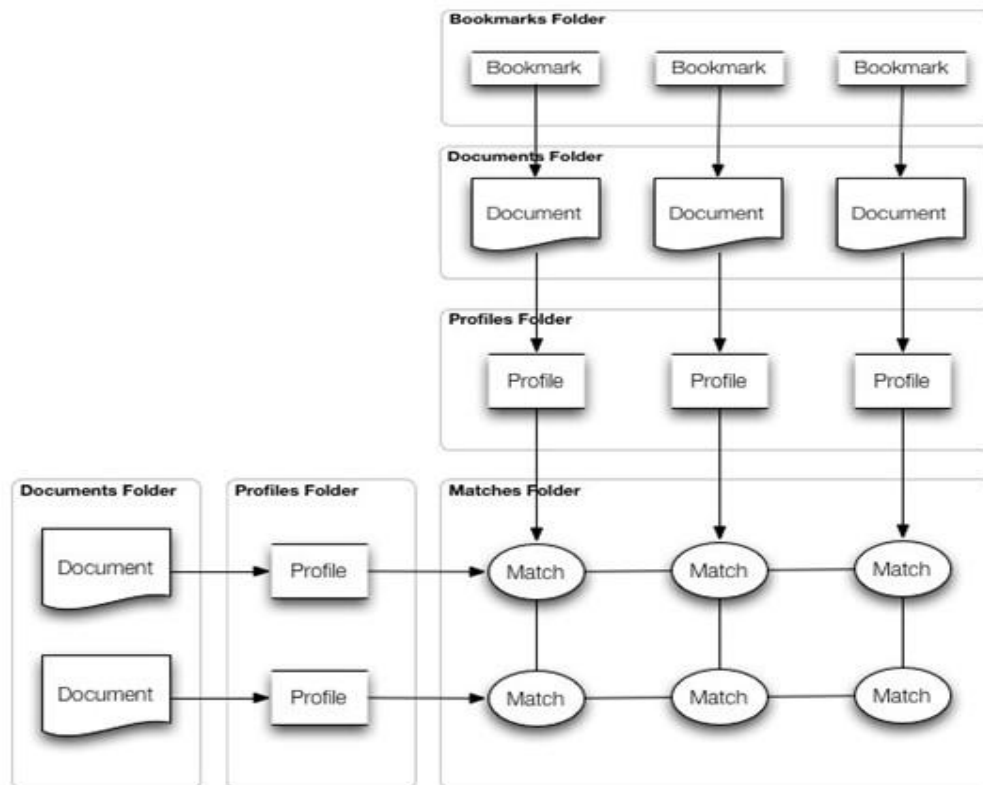


Figure 2 workflow of Subsift API, Source: Subsift Website

Figure 2 depicts the workflow of the transformation of documents from two Document Folders (e.g. submissions versus PC members' published works) into a folder of matching statistics. The process involves all kinds of API methods. Also, useful data and metadata can be obtained at each step in the process via methods such as "profile items show", "matches show". In order to improve the functionality of Subsift, I need to obtain the data of Document Folder items, Match Folder items; delete and create Profile Folders and items, etc.

3.4 REST API Explorer

REST API Explorer (which is located at Home/Demos/REST API Explorer) is a simple, but quite useful tool to try kinds of API methods. It shows you the HTTP query sent to the server and the HTTP response returned. Figure 3 shows the interface of creating document items in a specific Document Folder:

The screenshot shows the REST API Explorer interface. At the top, the 'Method' is set to 'POST' and the 'URL' is 'http://subsift.ilrt.bris.ac.uk/sp0890/documents/test5/items.xml'. Below this, the 'Parameters' section contains two entries: 'Name 1: folder_id' with 'Value 1: test', and 'Name 2: items_list' with 'Value 2: "doc1", "An example document", "Lorem ipsum dolor sit amet..." "doc2", "Another example document", "Ut enim ad minima veniam..."'. There is an 'Add Parameter' button to the right. Below the parameters, the 'Token' field contains '090ca46e3bee554c09dffa530d02b0e320b1lala'. A 'Submit' button is located below the token field. At the bottom, the 'HTTP Response' section shows 'Status: 201'.

Figure 3 Create Document Folder items in REST API Explorer, Source: Subsift Website

From the figure above, we can see that apart from the URL string, two parameters are needed: folder_id and item_list, additionally, token is added implicitly to the HTTP header. Similarly, in the program, parameters are added to the URL String:

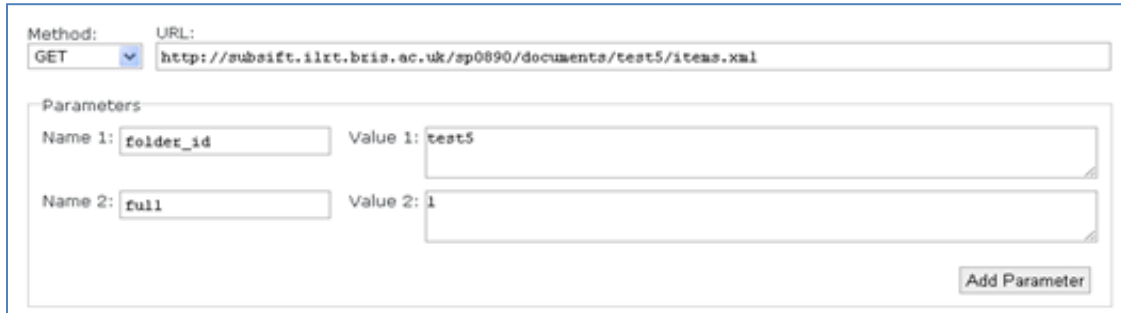
```
protected static void postContent(ArrayList<PostParameter> params, String urlStr, String token){
    HttpClient client = new HttpClient();
    PostMethod method = new PostMethod(urlStr);
    method.addRequestHeader("token", token);
    // Add POST parameters
    Iterator<PostParameter> paramsIterator = params.iterator();
    while(paramsIterator.hasNext()){
        PostParameter param = (PostParameter) paramsIterator.next();
        String name = param.getName();
        String value = param.getValue();
        method.addParameter(name, value);
    }
    // Send POST request
    try {
        statusCode = client.executeMethod(method);
    }
```

Figure 4 Post content into a specific folder, Source: my program

I import package HttpClient to help post content onto Subsift. Comparing figure 4 to figure 3, we note that token is added using method.addRequestHead(); parameters are added using method.addParameter(); and the status code is returned after the method

is executed by `client.addParameter()`, by which we can know whether the request is processed successfully.

Hereafter is an example of HTTP GET method. Figure 5 shows the way to get the content of a document item in REST API Explorer:



The screenshot shows the REST API Explorer interface. The Method is set to GET. The URL is `http://subsift.illrt.bris.ac.uk/sp0890/documents/test5/items.xml`. Under the Parameters section, there are two parameters: Name 1: `folder_id` with Value 1: `test5`, and Name 2: `full` with Value 2: `1`. An "Add Parameter" button is visible at the bottom right.

Figure 5 Get items' information from a Document Folder, Source: Subsift Website

We note that parameter “full = 1” is added, which tells the server to include the text data in the returned representation of the items (content between `<text>` and `</text>` tags in figure 1, Document Folder). Using GET method, I can obtain all the information that feedback processing algorithms need to analyze feedbacks. Figure 6 and figure 7 show how to obtain data from Subsift in my program:

```
if(paperID != null && paperID.length() > 0){
    urlStr = "http://subsift.illrt.bris.ac.uk/sp0890/documents/papers/items/"+paperID+
    ".xml?folder_id=papers&item_id="+paperID+".xml&full=1";
}
documentContent = SubsiftInteraction.getContent(urlStr);
// fetch the content of tag <text>
int textIndexBegin = documentContent.indexOf("<text>") + 6;
int textIndexEnd = documentContent.indexOf("</text>");
documentContext = documentContent.substring(textIndexBegin, textIndexEnd);
return documentContext;
```

Figure 6 Method `getDocumentContext()`

In method `getDocumentContext()`, the url string is formed, which includes all the parameters (note that `full = 1` is added). This string serves as the input parameter of method `getContent()` in figure 7. In method `getContent()`, I import package `URLConnection` to help get content from Subsift. Comparing figure 7 to figure 5, we note that token is added using `conn.setRequestProperty()` and the data returned is converted into a string, which is returned to the calling method and later analyzed by the feedback processing algorithms. I will introduce how to use the string in detail in section 6 and section 7.

```

protected static String getContent(String urlStr){
    String text = "";
    try
    {
        URL url = new URL(urlStr);
        URLConnection conn = url.openConnection ();
        conn.addRequestProperty("token", "090ca46e3bee554c09dffa530d02b0e320b11a1a");
        BufferedReader br = new BufferedReader(new InputStreamReader((InputStream) conn.getContent()));
        StringBuffer sb = new StringBuffer();
        String line;
        while(( line = br.readLine()) != null){
            sb.append(line);
        }
        br.close();
        text = sb.toString();
    } catch (Exception e){
        e.printStackTrace();
        System.out.println("IO exception occurs");
    }
    return text;
}

```

Figure 7 Get Content from certain http address

Here I list some other API methods dealing with the Document Folder that I made use of in the program. The way to create, update, destroy Profile Folder, Match Folder and the items in those folders are much the same with Document Folder:

- Document Folder Create
 HTTP Method: POST
 URL: http://Subsift.ilrt.bris.ac.uk/user_id/documents/folder_id.format
 HTTP Response: success: 201
- Document Folder Destroy
 HTTP Method: DELETE
 URL: http://Subsift.ilrt.bris.ac.uk/user_id/documents/folder_id/items.format
 HTTP Response: success: 200
- Document Folder Update
 HTTP Method: PUT
 URL: http://Subsift.ilrt.bris.ac.uk/user_id/documents/folder_id.format
 HTTP Response: success: 200

3.5 Improvement on Subsift Using Reviewers' feedback

Vector Space Model is the theoretical basis of ranking reviewers for a specific submission based on similarity. One of the shortcomings of this model is that term-weights are empirically tuned and the model provides no theoretical basis for computing optimum weights. "Learning to rank" is to use Machine Learning techniques to train the ranking model in order to improve its effectiveness. Although

there are different training algorithms available to tackle different types of ranking models, the outcomes of those algorithms are the same, which is a “Local Weight” vector for terms in the profile of the reviewer giving feedback, we denote it: \vec{v}_{lw} . This vector will be stored in the reviewer’s profile (see figure 1, Profile Folder, content in between lable <wl></wl> which is circled by red line) and multiplied the next time it is compared with the profiles of new incoming papers. Namely, after incorporating “local weight” vector into reviewers’ profiles, the cosine similarity will be calculated as follows:

$$s(\vec{q}, \vec{d}_j) = \cos(\theta) = \frac{\vec{q} * \vec{d}_j * \vec{V}_{lw}}{\|\vec{q}\| * \|\vec{d}_j\| * \|\vec{V}_{lw}\|} \quad (4)$$

Note that an additional vector \vec{v}_{lw} is multiplied after the feedback information being incorporated into reviewers’ profiles.

4 Main Working flow

There are mainly three approaches to dealing with different types of feedbacks. Pointwise approach aims at solving the ranking problem of individual instances; Pairwise approach targets the ground truth of pairwise preference; Listwise approach tackles ranking for a list of instances with partial or total order.

Before Introducing 3 types of feedbacks in detail respectively, let us first have an overview of the main working flow of the web-based application, which can be illustrated by the figure below:

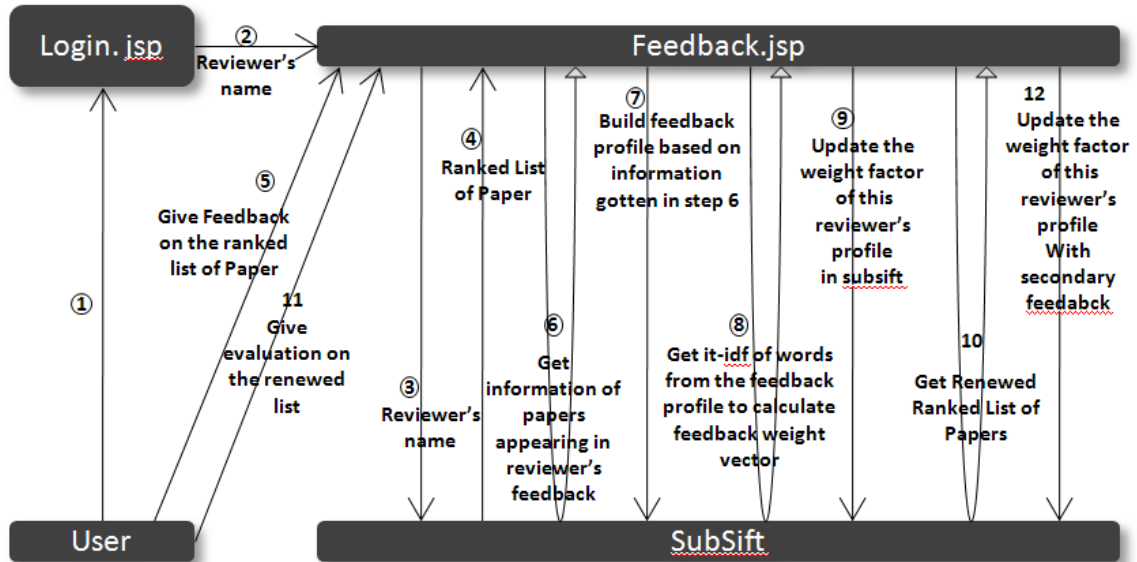


Figure 8 Working flow of my software

Please note that Feedback.jsp is actually the abbreviation of four jsp pages: ActionPageOfGivingPointWiseFeedback.jsp and

ActionPageOfCompletingPointWiseFeedback.jsp, ActionPageOfRenewedList.jsp and ActionPageOfFinalCompletingFeedback.jsp. In the rest of this section, I will explain the working flow, as well as showing the user interface on each step.

On the login page, the reviewer is required to input his name and select the type of feedback he wants to give, as shown in figure 9:

Figure 9 Login Page

After clicking on one of the three buttons in Figure 9, the reviewer will be directed to the feedback page, step 2, step 3 and step 4 in figure 8 will be completed at this stage. Figure 10 shows the pointwise feedback interface:

Figure 10 Pointwise feedback Page

Clicking on the titles of papers in figure 10, the reviewer will be directed to a page displaying the abstract of that paper (as shown in figure 11), which can help the reviewer know better about those papers. Those abstracts are gotten from the corresponding Document Folder items using http request as explained in section 3.4.

```
- <result>
- <item>
  <name>doc2</name>
  <created>1314700365</created>
  <description>Discriminative Models for Information Retrieval </description>
  <id>doc2</id>
  <modified>1314700365</modified>
  <source>text</source>
- <text>
  Discriminative models have been preferred over generative models in many machine learning tasks due to some of their attractive theoretical properties. In this paper, we explore the performance of two popular discriminative models, support vector machines and maximum entropy, on ad-hoc retrieval. We compare their performance with that of language modeling, the state-of-the-art. Experiments on ad-hoc retrieval indicate that although maximum entropy is significantly better than support vector machines are on par with language models. We argue that the main reason for this is their ability to learn arbitrary features automatically as demonstrated by our evaluation task of TREC-10.
</text>
</item>
</result>
```

Figure 11 Webpage including the abstract of a specific page after clicking on its title

If the reviewer selects “Discard Result of Previously Given Feedbacks” as shown in figure 10, the result of previously given feedbacks stored in the reviewer’s profile will be erased and completely replaced by the new given feedback result.

However, if the reviewer selects “Merge with Result of Previously Given Feedbacks”, the information obtained from the newly given feedback will be merged with previously obtained feedback information stored in the reviewer’s profile to form a new version of feedback information.

After selecting the top 3 and bottom 3 papers and clicking on the OK button, step 5, step 6, step 7, step 8 and step 9 of figure 8 has been accomplished. The reviewer will be provided the option to view the renewed paper ranking which is calculated after taking his feedback into consideration:

Thanks, your feedback has been processed.

[view the renewed ranked list of papers](#)

Figure 12 Webpage providing the option to view the renewed paper ranking

After clicking on the button in figure 12, the reviewer will be directed to the following page to give evaluation on the renewed ranked list:

No.	Title of Paper
1	SubSift: a novel application of the vector space model to support the academic research process
2	SubSift web services and workflows for profiling and comparing scientists and their published works
3	Query-Level Stability and Generalization in Learning to Rank
4	SoftRank: Optimising Non-Smooth Rank Metrics
5	Learning to Rank under Multiple Annotators
6	Yahoo! Learning to Rank Challenge Overview
7	Query-Level Learning to Rank Using Isotonic Regression
8	A Support Vector Method for Optimizing Average Precision
9	Novel Tools To Streamline the Conference Review Process: Experiences from SIGKDD'99

☐ Very Satisfied
☐ Satisfied
☐ Unsatisfied
☐ Very Unsatisfied

ok

Figure 13 Webpage inviting reviewer to give secondary feedback

After clicking on “ok” in figure 13, step 10, step 11 and step 12 of figure 8 are accomplished. The whole feedback process is down after the reviewer gives his secondary feedback, i.e. whether he is happy with the renewed ranking of papers. The parameters of algorithms will be slightly altered according to the reviewer’s feeling about the renewed ranking, and then the “local weight” vector is calculated for one more time, update the reviewer’s profile with this vector.

In the following sections, I will explain the algorithms and code implementation details to deal with 3 formats of feedbacks. In order to make it easier to understand, let us first have an overview of the basic class structure and supportive classes.

5 Basic Class Structure

The class design follows a simple rule: a type of feedback will be tackled by a class, `PointWiseFeedbackProcessing.java` is responsible for processing pointwise feedback, and similarly, `PairWiseFeedbackProcessing.java` deals with pairwise feedback. Listwise feedback can be either converted to pointwise feedback or pairwise feedback, so that there is no need to have an extra class to deal with it.

There are many common procedures in processing different formats of feedbacks, such as building Feedback Profile Folder, getting the content of Feedback Profile Folder. Those actions are done by methods written in the super class `FeedbackProcessing.java` to reduce code duplication. The inheritance relationship is shown in figure 14.

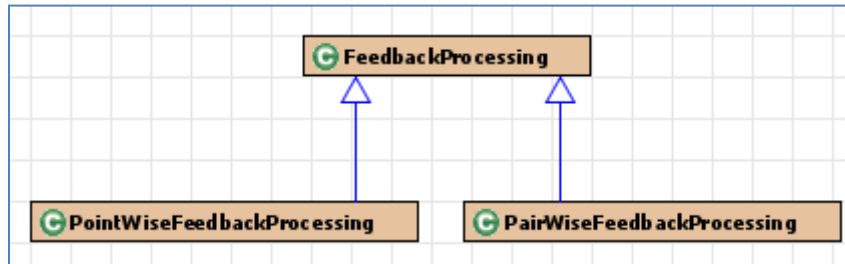


Figure 14 Inheritance relationship

Apart from main feedback processing classes, there are many other supporting classes.

Terms are the basic elements for algorithms to deal with. An instance of class TermInfo.java stores information of a term, the string representation is indispensable in initializing a new term. Optional information is the unique code assigned to a term and its it-idf value, which is quite essential in building training instances. Because TermInfo.java is a basic supporting class, it will be initialized in various situations and for different purposes. As a consequence, it has various constructors as shown in figure 15.

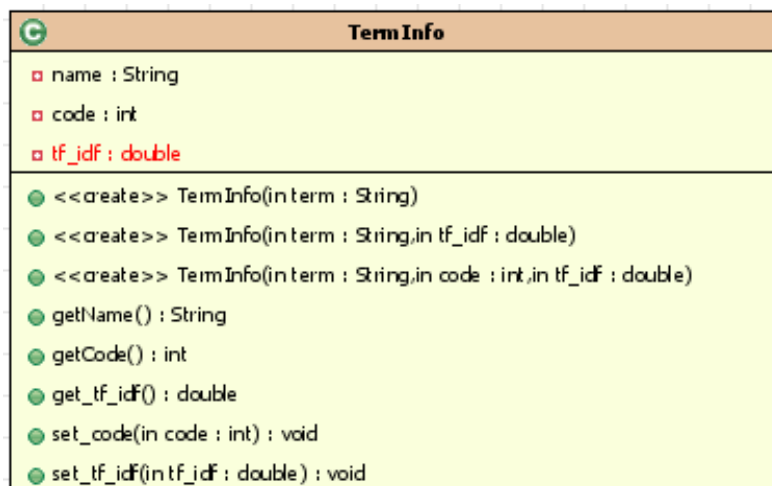


Figure 15 Class TermInfo.java

There should also be a class to store information of a “paper”, which is actually a term list. This list can be the terms appearing in a Document Folder item, a Profile Folder item, or a Match Folder item. Possible properties of a paper are: name, ID, rank, context and its vocabulary list. Like class TermInfo.java, There are various constructors for this class. Figure 16 shows the fields and methods in PaperInfo.java:

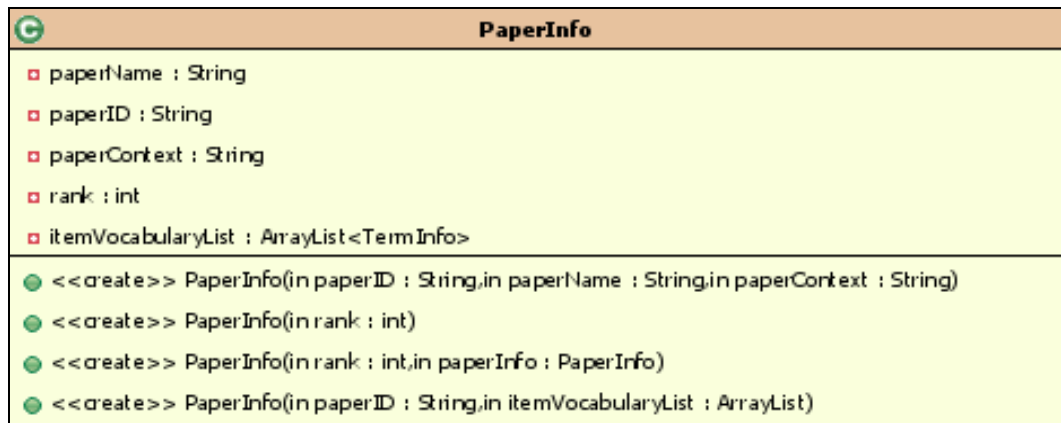


Figure 16 Class PaperInfo.java

Another supporting class SubsiftInteraction.java is responsible for the communication between my program and Subsift. As explained in section 3.4, I use methods in class URLConnection to get content of a folder (or a folder item) from Subsift website, as well as post folders onto Subsift when it doesn't involve parameters with value being a long string. I use methods in class HttpClient to post content onto Subsift folders in the case that the values of certain parameters are very long strings, such as the value of "item_list" parameter when creating Profile Folder for reviewers.

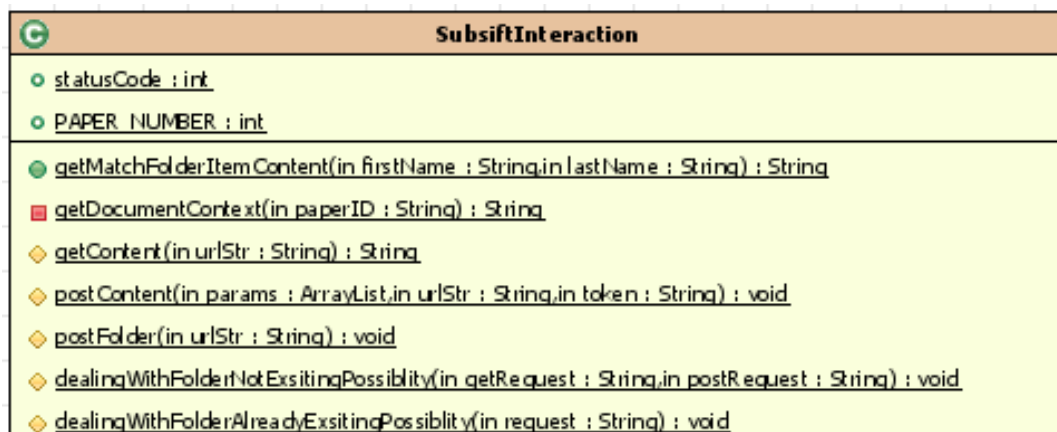


Figure 17 Class SubsiftInteraction.java

Before posting content to a folder, dealingWithFolderNotExsitingPossiblity() will be called. To judge whether such a folder exists, a GET request will be sent, if the status code returned is 404, then it means such folder doesn't exist. In this case, such a folder will be created before posting content to it. Similarly, Before creating a new folder, dealingWithFolderAlreadyExsitingPossiblity() will be called. If the returned status code of GET request is 200, it means such folder already exists. In this case, the old folder will be deleted before posting a new folder with that name.

Another thing to note is that function postContent() needs a parameter list to obtain all the parameters needed. Every (parameter name, parameter value) pair is an instance of class PostParameter.java (see figure 18).

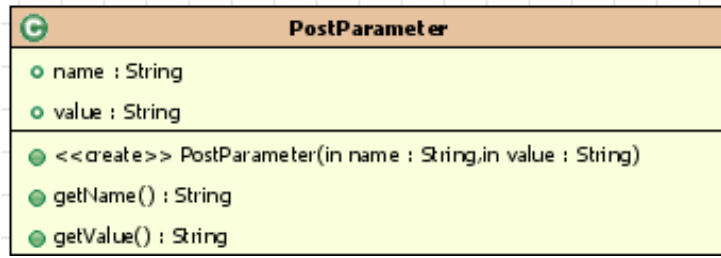


Figure 18 Class PostParameter.java

There are other classes, such as Prank.java, which is written to implement pointwise prank algorithm; svm_train.java from external java library LIBSVM, which is responsible for training the SVM instances in pairwise feedback processing. The explanation for those classes will be given along the illustration of processing procedures of each type of feedbacks.

In the following sections, I will show the algorithms behind the user interface to obtain and process 3 types of reviewer's feedbacks. The first format of feedback is the top n and bottom n papers selected by the reviewer from a ranked list of N papers provided by Subsift to him, which is dealt with by an algorithm in pointwise approach called "prank". In the second type of feedback, the reviewer provides n pairs of papers indicating they prefer the former ones to the latter ones. This is a typical pairwise feedback, which will be converted to a SVM problem and solved by training algorithm in LIBSVM. In listwise approach, a ranked list of n papers will be required from the reviewers. This format of feedback can be solved either by Prank or LIBSVM.

6 Feedback type 1 --- Top n and Bottom n Papers

After the top n and bottom n papers are selected by the reviewer, scores are assigned to those papers, (top1, 20), (top2, 19), (top3, 18),, (bottom 3, 3), (bottom 2, 2), (bottom1, 1). Those (instance, score) pairs are typical training instances for algorithms in pointwise approach. First let me give explanation to Prank, the pointwise algorithm I select to deal with the first type of feedback. It is not complicating, however, quite effective.

6.1 Algorithm Explanation

Prank is a learning algorithm based on *ordinal regression* (see definition in section 2.2) which is motivated by the perceptron algorithm for classification. Here y^t is from $S = \{1, 2, 3, \dots, k\}$ with the total order relation ">" [7].

The ranking rule $H: \mathbb{R}^n \rightarrow S$ combines perceptron weights $\vec{w} \in \mathbb{R}^n$ and threshold vector $\vec{b} = (b_1, \dots, b_{k-1})$ with $b_1 \leq b_2 \leq \dots \leq b_k = \infty$, b_k is not included because

it is infinite. Given a new instance \mathbf{x} , the rank is defined as the index of the smallest threshold b_r for which $\vec{w} \cdot \vec{x} \leq b_r$, thus all the instances that satisfies the condition $b_{r-1} \leq \vec{w} \cdot \vec{x} \leq b_r$ are assigned with the same rank r . Formally, given a ranking rule defined by \vec{w} and \vec{b} the predicted rank of a instance \mathbf{x} is [7]:

$$H(\mathbf{x}) = \min_{r \in \{1, \dots, k\}} \{r : \vec{w} \cdot \vec{x} - b_r \leq 0\} \quad (5)$$

The objective of the PRank algorithm is to find a perceptron weight vector \vec{w} and a threshold vector \vec{b} so that \vec{w} will cast all the instances in the training set into the k subintervals defined by \vec{b} . It works in the rounds in which prediction mistake happens:

Initialize: Set $\mathbf{w}^1 = 0$, $b_1^1, \dots, b_{k-1}^1 = 0, b_k^1 = \infty$.

Loop: For $t = 1, 2, \dots, T$

- Get a new rank-value $\mathbf{x}^t \in \mathbb{R}^n$.
- Predict $\hat{y}^t = \min_{r \in \{1, \dots, k\}} \{r : \mathbf{w}^t \cdot \mathbf{x}^t - b_r^t < 0\}$.
- Get a new label y^t .
- If $\hat{y}^t \neq y^t$ update \mathbf{w}^t (otherwise set $\mathbf{w}^{t+1} = \mathbf{w}^t$, $\forall r : b_r^{t+1} = b_r^t$) :
 1. For $r = 1, \dots, k-1$: If $y^t \leq r$ Then $y_r^t = -1$
Else $y_r^t = 1$.
 2. For $r = 1, \dots, k-1$: If $(\mathbf{w}^t \cdot \mathbf{x}^t - b_r^t)y_r^t \leq 0$ Then $\tau_r^t = y_r^t$
Else $\tau_r^t = 0$.
 3. Update $\mathbf{w}^{t+1} \leftarrow \mathbf{w}^t + (\sum_r \tau_r^t) \mathbf{x}^t$.
- For $r = 1, \dots, k-1$ update: $b_r^{t+1} \leftarrow b_r^t - \tau_r^t$

Output : $H(\mathbf{x}) = \min_{r \in \{1, \dots, k\}} \{r : \mathbf{w}^{T+1} \cdot \mathbf{x} - b_r^{T+1} < 0\}$.

Figure 19 illustration of updating rule, Source: ref. [7]

On round t , the algorithm gets an instance \vec{x}^t . It predicts a rank \hat{y}^t and compares it with the correct rank y^t . If $\hat{y}^t \neq y^t$, it updates the ranking rule (namely, trim the value of \vec{w} and \vec{b}) to minimize the number of thresholds between \hat{y}^t and y^t (the loss in ordinal regression). The pseudo code is given in Figure 19 [7, 9]. Here we explain step1, 2, 3 in Figure 13. For simplicity, we omit the index of round when referring to an instance-ranking pair (\vec{x}, y) . Here we expand the rank y into $k-1$ virtual variables y_1, y_2, \dots, y_{k-1} and set $y_r = +1$ for the case $\vec{w} \cdot \vec{x} > b_r$ and $y_r = -1$ for the case $\vec{w} \cdot \vec{x} < b_r$. Therefore, the rank value indicates a vector $\vec{y} = (y_1, \dots, y_{k-1}) = (+1, +1, \dots, -1, -1)$ for which the maximum index r that satisfies $y_r = +1$ is $y-1$. And the prediction rule is correct if $y_r(\vec{w} \cdot \vec{x} - b_r) > 0$ for all r . However, if the ranking algorithm makes a mistake by ranking \vec{x} as \hat{y} , ($\hat{y} \neq y$), namely, for all r , $\hat{y}_r(\vec{w} \cdot \vec{x} - b_r) > 0$, but for some r , $y_r(\vec{w} \cdot \vec{x} - b_r) < 0$. Then there is at least one threshold for which the value of $\vec{w} \cdot \vec{x}$ is at the wrong side of b_r . On the round where the mistake happens, We modify the values of b_r for which $y_r(\vec{w} \cdot \vec{x} - b_r) < 0$ by replace them with $b_r - y_r$. And we replace \vec{w} with $\vec{w} + \sum y_r \vec{x}$, where $\sum y_r$ donates the total number of thresholds between \hat{y} and y . The principle is shown in Figure 20 [7, 9].

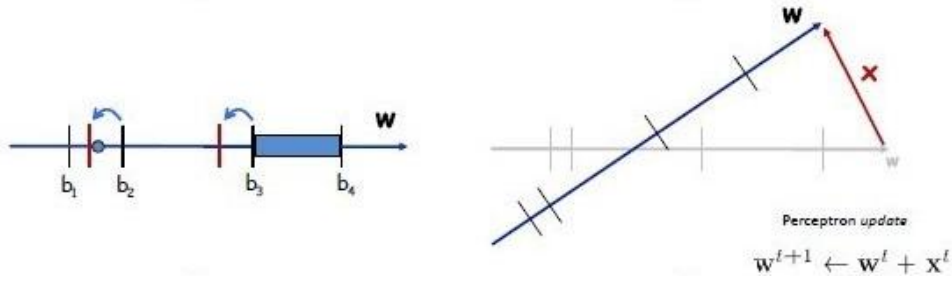


Figure 20 Illustration of updating b_r and \bar{w} , Source: ref. [6]

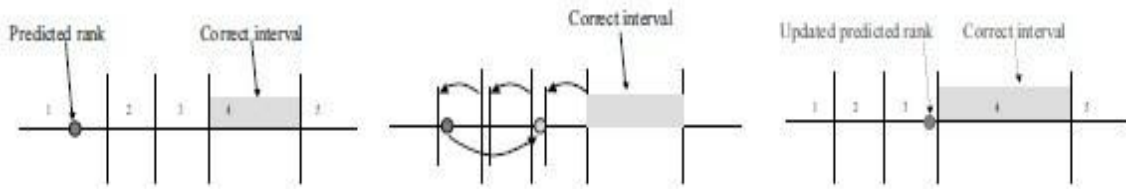


Figure 21 illustration of updating rule, source: ref. [7]

Figure 21 gives an example of an updating round. The correct rank y is 4, however, the predicted rank \hat{y} is 1, so that b_1, b_2, b_3 are sources of error, so their values are replaced with $b_1 - 1, b_2 - 1, b_3 - 1$. \bar{w} is modified to be $\bar{w} + 3\bar{x}$. So that the inner product of \bar{w} and \bar{x} increases by $3\|\bar{x}\|^2$. After the modification, the predicted rank is 3, which is much closer to the correct rank [7].

We find that once we obtain \bar{w} in formula (41) we can work out the scores of incoming instances. Therefore this is the vector which we need to calculate from the pointwise feedback and incorporate into the reviewer's profile.

6.2 Software Implementation

JSP is a technology to dynamically generate web pages based on HTML. It allows Java code to be interleaved with static web markup content with the resulting page being compiled and executed on the server to deliver an HTML document. The web application to obtain reviewers' feedbacks is implemented as JSP pages. Each page initializes corresponding java classes and calls functions in those classes to realize the functionality of that page.

6.2.1 Illustration for JSP Pages

There are five jsp pages involved to deal with pointwise feedback which are listed as follow:

- ✓ index.jsp (see figure 9);
- ✓ ActionPageOfGivingPointWiseFeedback.jsp (see figure 10);
- ✓ ActionPageOfCompletingPointWiseFeedback.jsp (see figure 12);
- ✓ ActionPageOfRenewedList.jsp (see figure 13);
- ✓ ActionPageOfFinalCompletingFeedback.jsp.

The functionalities each jsp page implements are listed as below:

➤ Index.jsp (figure 9)

- a) Display the components on login page using HTML, CSS.
- b) Add onclick events to buttons to direct reviewer to the feedback page in based on the feedback type he selected, pass parameters (reviewer's first and last name) to the next page.

➤ ActionPageOfGivingPointWiseFeedback.jsp (figure 10)

- a) Get parameters (reviewer's first name and last name).
- b) call static GetMatchFolderContent() in SubsiftInteraction.java (see figure 17) to get the content of the Match Folder item of that reviewer and store it in a string, the structure of returned string is shown in figure 22.

```
<result>
  <match>
    <name>Jamie Peng</name>
    <created>1316114607</created>
    <description>Jamie Peng</description>
    <document_n>487</document_n>
    <file_id>reviewers-Jamie_Peng</file_id>
    <item>
      <name>doc8</name>
      <description>Subsift: a novel application of the vector
      <score>0.0978011707747715</score>
      <source>text</source>
    </item>
    <item>
      <name>doc10</name>
      <description>SubSift web services and workflows for prof
      <score>0.09461520401692</score>
      <source>text</source>
    </item>
  </item>
</match>
```

Figure 22 Content of Match Folder item Jamie Peng

Note that the documents in the Match Folder item Jamie Peng have already been ranked by their match scores. So that we can get each document's ID, description, as long as its ranking in the match folder item of Jamie Peng from this string.

- c) Call static buildPaperArrayFromMatchFolder() in FeedbackProcessing.java to build an ArrayList from the string representation of the match folder content. Each element is an object of class PaperInfo, figure 23 shows the pseudo code of this function:

```

ArrayList<PaperInfo> papers = new ArrayList<PaperInfo>();
while(rank < PAPER_NUMBER){
    rank ++;
    get paperID from matchFolderContent;
    get paperName name from matchFolderContent;
    get paper's context using SubsiftInteraction.getDocumentContext(paperID);
    papers.add(new PaperInfo(paperID, paperName, paperContext));
}
return papers;

```

Figure 23 Pseudo code of function buildPaperArrayFromMatchFolder()

Note that we use the constructor `PaperInfo(paperID, paperName, paperContext)`. This constructor is specially designed to create instance for Document Folder items. The ranking of each paper in the match folder item of reviewer Jamie Peng is just its position in this `ArrayList` plus 1.

- d) Creating an instance of class `PointWiseFeedbackProcessing.java`. Assign the paper list gotten from step c to a static `ArrayList<PaperInfo>` static field `top_n_papers` of class `FeedbackProcessing.java` (as shown in figure 27), so that it can be used by all instances of `FeedbackProcessing.java` and its child calsses, Note that field `top_n_papers` stores the information of Document Folder items, including their IDs, Names, Context and rankings in the match folder item of the reviewer giving feedback (see Table 2).

- e) Display the ranked list of papers and the feedback frame (see figure 10). As explained in step b, the papers have been ranked in the Match Folder item, so that just display the name of papers in the `ArrayList top_n_papers` in turn in the table.

Note that click on the title of a paper, the reviewer can access the abstract of that paper (see figure 11). This is realized by using super link in html, the url incorporates paper ID gotten dynamically during runtime using java code.

➤ **ActionPageOfCompletingPointWiseFeedback** (figure 12)

- a) Get parameters (reviewer's selection of top n and bottom n papers) and store them in an `ArrayList<Integer>` named "rankList".
- b) Create an instance of class `PointWiseFeedbackProcessing.java`; Call function `go(ArrayList<Integer> rankList)` with the "rankList" gotten from step a as parameter. The process of building a "local weight" vector and incorporating it into Subsift is done in class `PointWiseFeedbackProcessing.java`. The detailed explanation can be found in section 6.2.2.

➤ **ActionPageOfRenewedList.jsp** (figure 13)

- a) Create an instance of class `FeedbackProcessing.java`; get the first name and last name of the reviewer which is stored in the static field of this class.
- b) Get the Match Folder item content of this reviewer and store it into a string using `SubsiftInteraction.getMatchFolderContent(firstName, lastName)`.
- c) Build a paper `ArrayList<PaperInfo>` by fetching information from the string gotten

from step b using static method `buildPaperArrayFromMatchFolder(content)` of class `FeedbackProcessing.java`.

- d) Display the list of papers in a table.
- e) Display a feedback frame allowing the reviewer to give feedback on the renewed ranked list.

➤ `ActionPageOfFinalCompletingFeedback`

- a) Get the parameter of the radio boxes on `ActionPageOfRenewedList.jsp` which represents the result of secondary feedback and assign a value to variable “opinion” based on the string value of the parameter, as shown in figure 24:

```
double opinion = 1;
if(request.getParameter("secondFeedback").equals("Very Satisfied"))
    opinion = 2;
if(request.getParameter("secondFeedback").equals("Satisfied"))
    opinion = 1.5;
if(request.getParameter("secondFeedback").equals("Unsatisfied"))
    opinion = 0.5;
if(request.getParameter("secondFeedback").equals("Very Unsatisfied"))
    opinion = 0.1;
```

Figure 24 Select Double value for “opinion” based on value of parameter

- b) Create an instance of class `FeedbackProcessing.java`; Call function `incorporateSecondaryFeedback(Double opinion)` of the instance. This function is in class `FeedbackProcessing.java`, which can be shared by both pointwise and pairwise feedback processing approach when secondary feedback is given. the pseudo code is shown as below:

```
public void incorporateSecondaryFeedback(double opinion){
    for(every term appearing in usefulTermTable){
        double current_value = usefulTermTable.get(current_term);
        current_value = 0.5 + (current_value - 0.5) * opinion;
        if(current_value < 0) current_value = 0;
        if(current_value > 1) current_value = 1;
        usefulTermTable.put(current_term, current_value);
    }
    incorporateFeedbackIntoSubsift(firstName, lastName, usefulTermTable);
}
```

Figure 25 pseudo code for function `incorporateSecondaryFeedback()`

Note that the part greater or less than 0.5 of local weight value for each term is timed by “opinion”, so that the influence of the feedback is magnified or shrank based on the reviewer’s evaluation of the renewed ranked list of papers.

6.2.2 Illustration for Class `PointWiseFeedbackProcessing.java`

First let us have an overview of fields and methods in PointWiseFeedbackProcessing.java and its super class FeedbackProcessing.java. They are shown in figure 26 and figure 27:



Figure 26 Class FeedbackProcessing.java

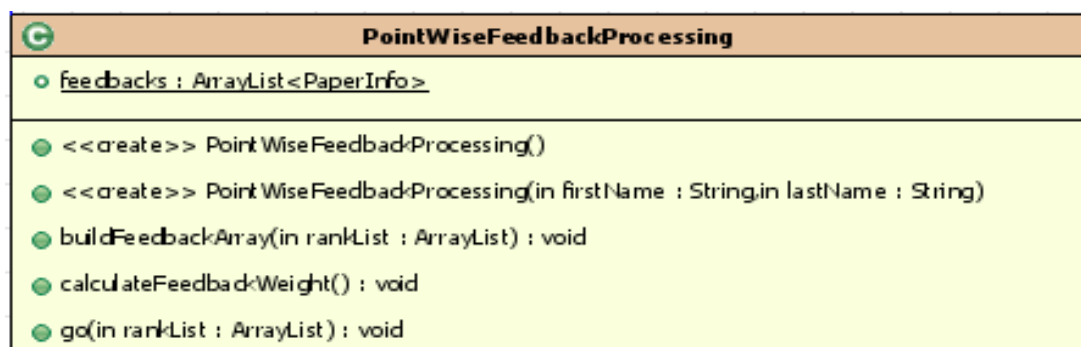


Figure 27 Class PointWiseFeedbackProcessing.java

One important thing to notice here is that I use data type ArrayList to store all kinds of input information in classes dealing with feedbacks. This is because ArrayList is an extensible data type, so that no change will be needed to be made to the classes when I try different numbers of input instances in the parameter optimization phase (see section 9.3). Static Field *feedbacks* of class PointWiseFeedbackProcessing.java is an

ArrayList with elements of type PaperInfo.java. Each element stores the score of a paper appearing in the reviewer's feedback, as well as the paper's ID and name. The score of each instance is essential input of Prank algorithm.

For convenience's sake, table 2 below gives illustration to static fields of class feedbackProcessing.java, which can serve as a useful reference in later narration.

Name	Illustration
<i>PAPER_NUMBERS</i>	Final static field to specify the number of papers gotten from the Match Folder and displayed in the table in figure 9. Currently the value is 20, namely, select top 20 papers from the Match Folder item of this reviewer.
<i>mergeResult</i>	Boolean field to specify whether to merge newly obtained feedback information with history feedback information. If the value is false, then the previously obtained feedback information will be discarded.
<i>top_n_papers</i>	An <PaperInfo> ArrayList to store information of a list of document items. The information of each element includes <u>paper ID</u> , <u>paper name</u> and <u>paper context</u> . The <u>rank</u> of each paper in the Match Folder item of the reviewer giving feedback is its position in this ArrayList + 1.
<i>paperRanks</i>	A <Integer> ArrayList to store the ranks of papers appearing in the feedback. Knowing a paper's rank, we can locate it in ArrayList <i>top_n_papers</i> , so as to get the paper's information.
<i>vocabularyList</i>	A <TermInfo> ArrayList to store information of all terms in the Profile Folder which contains profiles of documents appearing in the reviewer's feedback. The information of each element includes <u>term's string representation</u> , and <u>its tf-idf value</u> , which is initialized to be "0.5".
<i>selected_papers_list</i>	A <PaperInfo> ArrayList to store information of papers appearing in the feedback. The information of each element includes <u>paper ID</u> and it is <u>vocabulary list</u> , with <u>it_idf</u> values gotten from the profile of that paper.
<i>usefulTermTable</i>	A <String, Double> Hashtable to store <u>terms</u> and their <u>local weight values</u> . This is the final outcome of the feedback processing algorithm, which will be used to form the "local weight" parameter value.

Table 2 explanation of static fields of class FeedbackProcessing.java

After calling function `go()` of a newly built `PointWiseFeedbackProcessing.java` instance from `ActionPageOfCompletingPointWiseFeedback.jsp`, a list of functions in `go()` will be executed, as shown in figure 28. I will give illustration to this list, as the way to explain the pointwise processing procedure.

```
public void go(ArrayList<Integer> rankList){
    buildFeedbackArray(rankList);
    buildFeedbackProfileFolder("PointWise");
    String feedbackProfileFolderContent = getFeedbackProfileFolderContent("PointWise");
    buildVocabularyVector(feedbackProfileFolderContent);
    buildPaperList();
    calculateFeedbackWeight();
    incorporateFeedbackIntoSubsift(firstName, lastName, usefulTermTable);
    rebuildMatchFolder();
}
```

Figure 28 Function list in `go()` in class `PointWiseFeedbackProcessing.java`

➤ `buildFeedbackArray(rankList)`: Function in `PointWiseFeedbackProcessing.java`

From step d in `ActionPageOf GivingPointWiseFeedback.jsp`, we have assigned an `ArrayList` to the static field `top_n_papers`. For every number in the `rankList`, I can find the corresponding paper in `top_n_papers`. So that I can create an instance of class `PaperInfo.java` with the constructor `PaperInfo(int rank, PaperInfo paperInfo)` and add it to the end of static `ArrayList` field `feedbacks` of class `PointWiseFeedbackProcessing.java` (as shown in figure 23). Figure 29 shows the code of this part:

```
if(listWiseFeedback == false){
    int halfSize = rankList.size()/2;
    //top ones
    for(int i = 0; i < halfSize; i++){
        PaperInfo paper = new PaperInfo(i + 1,
            top_n_papers.get(rankList.get(i)-1));
        feedbacks.add(paper);
    }
    //bottom ones
    for(int i = 0; i < halfSize; i++){
        PaperInfo paper = new PaperInfo(PAPER_NUMBER - (halfSize - i) + 1,
            top_n_papers.get(rankList.get(halfSize + i) - 1));
        feedbacks.add(paper);
    }
} else {
    //listwise feedback
    for(int i = 0; i < rankList.size(); i++){
        PaperInfo paper = new PaperInfo(i + 1,
            top_n_papers.get(rankList.get(i)-1));
        feedbacks.add(paper);
    }
}
```

Figure 29 Function `buildFeedbackArray()`

The value of field `listWiseFeedback` specifies whether this `rankList` is from the action page giving pointwise feedback or the action page giving listwise feedback.

The parts circled by the red line are the algorithms to decide ranks (or say, scores) in Prank for top papers and bottom papers. The part circled by blue line is the algorithm that determines scores for papers in listwise feedback, which I will explain in section 8.

➤ buildFeedbackProfileFolder(“PointWise”): Function in FeedbackProcessing.java

This function is to build a Profile Folder for documents in ArrayList “feedbacks”.

The parameter feedbackType (“PointWise”) is used to specify the name of Document Folder and Profile Folder:

```
String documentFolderName = feedbackType + "_Feedback_Of_"+reviewerName;
```

The Steps to build a Profile Folder are as follows:

- a) Build a Document Folder for the documents appearing in field *feedbacks* using static postFolder() in SubsiftInteraction.java
- b) Post documents items into this folder using static postContent() in SubsiftInteraction.java, the information needed to build document items (including id, description, context) can be achieved from field *feedbacks*.
- c) Build Profile Folder from Document Folder using static postFolder() in SubsiftInteraction.java

➤ getFeedbackProfileContent(“PointWise”): Function in FeedbackProcessing.java

This function is to get the content of the Profile Folder build in the last step using getConent() method in class SubsiftInteraction.java as introduced in section 3.4. After getting the string representation of the folder content, trim the string to get rid of all folder ids which shouldn't be included in the term vocabulary.

➤ buildVocabularyVector():Function in FeedbackProcessing.java

This function is to build an ArrayList for the vocabulary of the Profile Folder from the string gotten from the last step and store it in the static field *vocabularyList* of class FeedbackProcessing.java (see table 2).

➤ buildPaperList():Function in FeedbackProcessing.java

This function is to build an ArrayList for each profile item in the Profile Folder of documents appearing in the reviewer's feedback, and then store the list into a static field of class FeedbackProcessing .java named *selected_papers_list* (see table 2). The pseudo code is shown below:

```

for(every number in ArrayList paperRank){

    find corresponding paperID from top_n_papers;
    get profile item content, store it into a string;
    clone a vocabulary list;
    for(every term in the profile item content string){
        fetch term and its wtfdidf value;
        find such an item from cloned vocabularyList;
        set tf_idf value;
    }
    add index to the cloned vocabulary list's items; // used in pairwise approach
    PaperInfo newPaper = new PaperInfo(paperID, itemVocabularyList);
    add the new paper(the cloned vocabularyList) to static field selected_papers_list;
}

```

Figure 30 Pseudo code of function buildPaperList ()

From figure 30 we see that I first clone a *vocabularyList* that I built in the last step which contains all terms appearing in the profile folder of feedback papers. Note that every term in the cloned list is with the default tf-idf value 0.5. Then I traverse the profile item string gotten from the Profile Folder of feedback papers, fetch each term and its it-idf value v , locate the term in the cloned vocabulary list, set the term's it-idf value to be $v + 0.5$, for tf-idf values exceed 1 after the addition, they will be trimmed to be 1; for terms with tf-idf values less than 0, they are assigned with tf-idf value 0.. Note that vocabulary lists for feedback profile items are of the same length (the length of the *vocabularyList*). For terms not appearing in one profile item, the it-idf values would be the default value set in the last step (0.5) in the ArrayList built for that feedback profile item.

After building an ArrayList<TermInfo> for one paper in the feedback, a new instance of PaperInfo.java will be created with constructor PaperInfo(String papered, ArrayList<TermInfo> itemVocabularyList) to represent that paper and it will be added to the static field *selected_papers_list* (see table 2).

➤ calculateFeedbackWeight(): function in PointWiseFeedbackProcessing.java

After the field *selected_papers_list* has been added with information of papers from the feedback, we can now calculate the local weight vector using Prank. The algorithm is completed by Prank.java. The pseudo code is shown in figure 31 ,where parameter vocabularyList of function go() is the weight vector \vec{w} needed to be optimized in figure 19; parameter selected_papers_list stores the training instances \vec{x}^t and from parameter feedbacks we can obtain the score of each training instance; in PrankPerRound(), the parameter settings are as follows (see figure 19 for the definition of the parameters): $k = 21$, since there are 20 possible ranks; $t = 6$, since there are 6 training instances gotten from the reviewer's feedback..

The prank algorithm has been explained thoroughly in section 6.1. What we need to notice here is that after the local weight vector (*vocabularyList*) has been

calculated, the weight values needs to be tuned so that the influence of feedback on the local weight would be reasonable and within the scope of [0, 1] as shown in the red circled part of Figure 31. The terms with weight value of 0.5 are not included in the *usefulTermTable* because all terms will be set to be the default value of 0.5 when rebuilding the profile of the reviewer, there is no need to add terms with default value to the parameter “terms_weight”.

After the weight values of terms in the *vocabularyList* resulting from the Prank algorithm are tuned, the term list and the terms’ weight values are restored in a static `Hashtable<String, Double>` field *usefulTermTable* of class *FeedbackProcessing.java*.

```
public class Prank {
    public Hashtable<String, Double> go(ArrayList<TermInfo> vocabularyList,
        ArrayList<PaperInfo> selected_papers_list, ArrayList<PaperInfo> feedbacks){

        set the tf-idf values of vocabulary list to be 0;
        for(every paper in the selected_paper_list){
            get the rank of the paper;
            PrankPerRound(vocabularyList, currentItemList, rank);
        }

        for(every term in the vocabulary list){
            String word = vocabularyTerm.getName();
            double weight = vocabularyTerm.get_tf_idf();
            weight = 0.5 + weight*0.00001;
            if(weight > 1) weight = 1;
            if(weight < 0) weight = 0;
            if(weight != 0.5)
                usefulTermTable.put(word, weight);
        }
        return usefulTermTable;
    }

    private void PrankPerRound(ArrayList<TermInfo> vocabularyList,
        ArrayList<TermInfo> currentItemList, int feedbackRank){
        see figure 19, change the tf-idf values in the vocabularyList
    }
}
```

Figure 31 Pseudo code for Class Prank

➤ **IncorporateFeedbackIntoSubsift():** function in *FeedbackProcessing.java*

In this step, we need to update the local weights of terms in the reviewer’s profile based on the *usefulTermTable* we’ve gotten from the last step.

Subsift API allows users to specify the local weights of certain terms when creating Profile Folder from Document Folder (as shown in Figure 33). I make use of this feature of Subsift, specify the local weights of terms and recreate the Profile Folder of the reviewer using static function `postContent()` of class *SubsiftInteraction.java*. The pseudo code is shown in figure 32:

```

protected void incorporateFeedbackIntoSubsift
(String firstName, String lastName, Hashtable<String, Double>termTable){
    get profile content and store terms and corresponding tfidf values
    in hashtable termTableOfReviewerProfile;
    // form the term_weights parameter string
    String term_weight = "";
    term_weight = firstName + " "+lastName + ",";
    for(every term in termTable) {
        double weight = feedbackTermTable.get(str);
        if( this term exist in the termTableOfReviewerProfile){
            double history_tf_idf = termTableOfReviewerProfile.get(str);
            // whether to merge result with previously given feedbacks
            if(mergeResult == true && history_tf_idf != 1)
                weight = 0.5 + ( (history_tf_idf - 0.5) + (weight - 0.5))/2;
        }
        term_weight = term_weight + str + "," + String.valueOf(weight) + ",";
    }
    term_weight = term_weight.substring(0, term_weight.length()-1);
    post new profile folder with specified local weights and default weight value (0.5);
}

```

Figure 32 pseudo code of incorporateFeedbackIntoSubsift()

Method:	URL:
POST	http://subsift.ilt.bris.ac.uk/sp0890/profiles/reviewers/from/reviewers.xml
Parameters	
Name 1: folder_id	Value 1: reviewers
Name 2: term_weights	Value 2: Jamie Peng,machine learning,0.9,learning,0.7,machine,0.7
Name 3: term_weight_default	Value 3: 0.5
Add Parameter	

Figure 33 Build Profile Folder with term weights in Subsift, Source: Subsift Website

The part circled by red line in figure 32 shows the way to merge new coming feedback with previously given feedback. For each term in the *usefulTermTable*, see whether it is in the table containing the terms in the reviewer's profile. If so, then judge from field *mergeResult* whether the reviewer wants to merge the newly obtained feedback information with existing feedback information in the reviewer's profile. If so, the term will be assigned with the average of its value in the *usefulTermTable* and the historical tf-idf value.

➤ rebuildMatchFolder(): function in FeedbackProcessing.java

This function is to rebuild Match Folder by matching the papers' Profile Folder with the updated reviewers' Profile Folder using SubsiftInteraction.postFolder().

7 Feedback type 2 --- n ordered instance pairs

The learning algorithms in this approach is given a training set with linear ordering (top m papers) as the base level information about the ranking task just like in the pointwise approach; then the algorithm is also provided with information about which pairs of instances in the training set should be ranked above or below each other as feedback, in other words, pairwise feedback.

The classification model (such as ranking SVM in conventional approach) can be trained with the pairwise feedback together with the linear ordering so that it will result in a ranking that orders wrongly as few instances pairs in the feedback as possible, which is referred to the problem of *preference learning (PL)* [13].

The algorithm I select to dealing with pairwise feedback is Ranking SVM, which is a method making uses of Support Vector Machines (SVM) to building the classification model [8].

7.1 Algorithm Explanation

In this section we first introduce SVM. Then we prove that ordinal regression problem can also be expressed by the ordered instance pairs, so that pairwise feedback can be processed by SVM. After that we will explain how to solve the problem of ordinal regression using SVM based on training set with instance-ranking pairs.

7.1.1 SVM

An SVM represents instances as points in the space, which are mapped so that those points are divided into two categories by a gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to which category based on which side of the gap they fall on.

7.1.1.1 Maximize Margin

As shown in figure 34, there are many lines that can separate two groups of points.

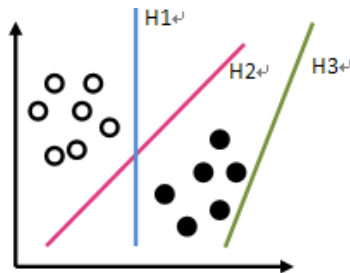


Figure 34: hyper-plane illustration, Source: Wiki. *H3 doesn't separate the two classes; H1 does, with a small margin and H2 with the maximum margin. Hence H2 is the hyper-plane*

Our goal is to find a hyper-plane (H2 in figure 34) so that the points are divided with the maximum margin. More formally [17]:

- Given data $(\vec{x}_1, y_1), (\vec{x}_2, y_2), \dots, (\vec{x}_n, y_n)$
- Finding a separating hyperplane can be posed as a constraint satisfaction problem:

Any $i \in (1, n)$, find \vec{w} such that :

$$\vec{w}^T * \vec{x}_i + b \geq +1, \quad \text{if } y_i = +1$$

$$\vec{w}^T * \vec{x}_i + b \leq -1, \quad \text{if } y_i = -1$$

$$\text{or equivalently, } \forall i, y_i(\vec{w}^T * \vec{x}_i + b) - 1 \geq 0 \quad (6)$$

- The margin of a classifier is defined as this: Take a hyper-plane(P0) that separates the data; put a parallel hyper-plane (P1) on a point in class 1 closest to P0; put a second parallel hyper-plane (P2) on a point in class -1 closest to P0; The margin is the perpendicular distance between P1 and P2. \vec{w} is prescaled so that the margin between two categories is $2/|\vec{w}|$ (see the detailed explanation in figure 35) [17].

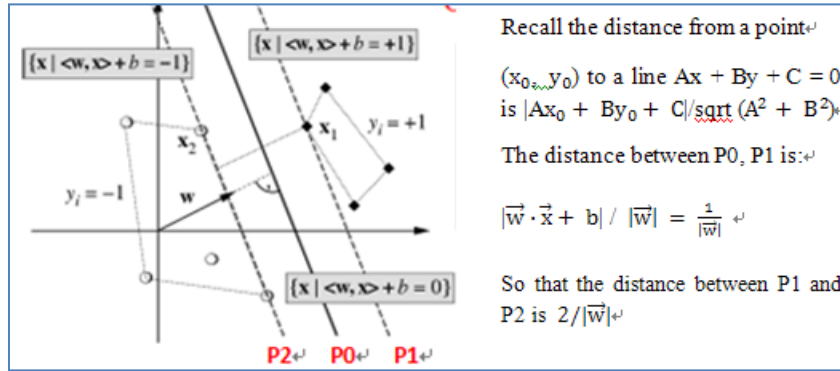


Figure 35 Margin for Canonical Hyperplane

Therefore the SVM constraint optimization problem becomes: minimize $f: |\vec{w}|^2$ so that $g: y_i(\vec{w}^T * \vec{x}_i + b) - 1 = 0$. This is a *quadratic programming problem*, which can be solved by Lagrangian multiplier method.

7.1.1.2 Lagrangian formulation for linear separable case

To solve the QP-programming problem, we need to flatten f with superimposed constraint g . This is shown vividly in the figure below:

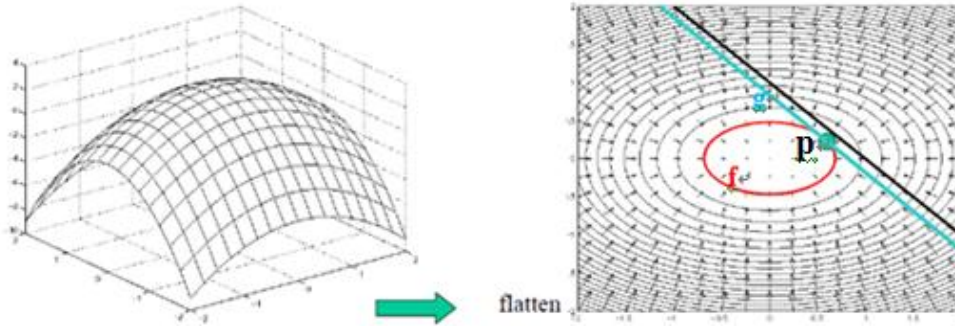


Figure 36 find minimum value of f with constrain g , Source: ref.[17]

From the Figure 36 we can see that f is minimized when the constraint line g is tangent to the inner ellipse contour line of f . At the tangent solution P , gradient vectors of f , g are parallel, or say, gradient of f must be in the same direction as g . Therefore, the problem can be converted to: Looking for solution point P that meets the parallel normal constraint $\nabla f(P) = \nabla a * g(P)$ and $g(x) = 0$. Combine those two which will result in Lagrangian L , the derivative of which is required to be zero [17]:

$$L(x, a) = f(x) + g(x) \quad \text{where} \quad \nabla(x, a) = 0 \quad (7)$$

In formula (7), partially derivatives with respect to x recovered the parallel normal constraint; partially derivatives with respects to λ recovers $g(x) = 0$. In general there is:

$$L(x, a) = f(x) + \sum_{i=1}^n a_i g_i(x) \quad (8)$$

In our case, $f: |\vec{w}|^2$ and $g: y_i(\vec{w}^T * \vec{x}_i + b) - 1$, substitute into formula (8):

$$L(\vec{w}, b, \vec{a}) = 1/2 |\vec{w}|^2 - \sum_{i=1}^m a_i (y_i [\langle \vec{w}, \vec{x}_i \rangle + b] - 1) \quad (9)$$

At the saddle point of $L(\vec{w}, b, \vec{a})$, it will reach its minimum value, there are [17, 18]:

$$\frac{\partial}{\partial b} L(\vec{w}, b, \vec{a}) = 0, \quad \frac{\partial}{\partial \vec{w}} L(\vec{w}, b, \vec{a}) = 0 \quad (10)$$

$$\text{This gives the condition:} \quad \sum_{i=1}^n a_i y_i = 0, \quad \vec{w} = \sum_{i=1}^n a_i y_i \vec{x}_i \quad (11)$$

Substitute into $L(\vec{w}, b, \vec{a})$, it is now converted to the *dual problem*:

$$\text{Maximize } W(\vec{a}) = \sum_{i=1}^n a_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n a_i a_j y_i y_j \langle \vec{x}_i, \vec{x}_j \rangle \quad (12)$$

$$\text{Subject to} \quad a_i \geq 0, \quad i = 1, 2, \dots, n \quad \sum_{i=1}^n a_i y_i = 0.$$

Dot product

Note that the constraints are replaced by constraints on the Lagrangian multipliers and the training data will only occur as dot products [17, 18].

The *decision boundary* separating two categories is expressed as:

$$f(\vec{x}) = \text{sgn}(\langle \vec{w}, \vec{x} \rangle + b) \quad \text{Substitute} \quad \vec{w} = \sum_{i=1}^n a_i y_i \vec{x}_i$$

$$\text{Namely: } f(\vec{x}) = \text{sgn}(\sum_{i=1}^n a_i y_i \langle \vec{x}_i, \vec{x} \rangle + b) \quad (13)$$

Instead of considering all points in the training process like linear regression or naïve Bayes, in SVM, only “difficult points” close to the decision boundary will influence the optimality. Those points form the *support vector*, such as points on P1 and P2 in figure 34. Formally, Support vector is \vec{x}_i which satisfies $y_i[\langle \vec{w}, \vec{x}_i \rangle + b] - 1 = 0$ [17, 18].

7.1.1.3 Lagrangian formulation for nonlinear separable case

In reality, the training set is often noisy, namely, the instances are more likely to be non-linear separable. The noisy problems are best addressed by allowing some instances to violate the margin constraint. Those potential violations are represented using slack positive slack variables $\varepsilon = (\varepsilon_1, \dots, \varepsilon_n)$. This extension of SVM is called “Soft Margin SVM” [17, 18].

$$\varepsilon_i = \begin{cases} 0 & \text{if } x_i \text{ is correctly classified} \\ \text{distance to margin} & \text{if } x_i \text{ is not correctly classified} \end{cases} \quad (14)$$

So that the constraint optimization problem becomes [17, 18]:

$$\text{Given data } (\vec{x}_1, y_1), (\vec{x}_2, y_2), \dots, (\vec{x}_n, y_n)$$

$$\text{Minimize } P(\vec{w}, \vec{\varepsilon}, b) = \frac{1}{2} |\vec{w}|^2 + C \sum_{i=1}^n \varepsilon_i \quad (15)$$

$$\text{subject to: } y_i(\vec{w}^T * \vec{x}_i + b) \geq 1 - \varepsilon_i, \quad \varepsilon_i \geq 0$$

Because the data is no longer linear separable, we can't use dot product $\langle \vec{x}_i, \vec{x}_j \rangle$ directly as in formula (12). Imagine that there is a function ϕ that maps that non-linear separable data into a linear separable space as shown in figure 37 [17]:

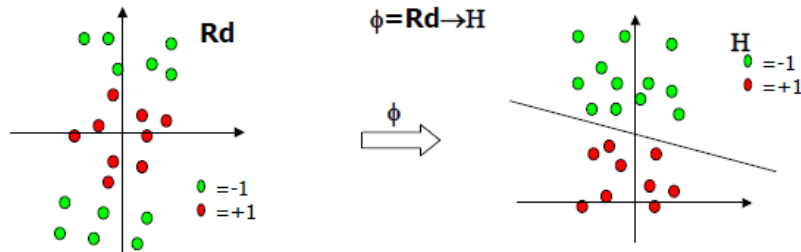


Figure 37 Imaginary mapping function ϕ , Source: ref. [17]

Then the dot product $\langle \vec{x}_i, \vec{x}_j \rangle$ in linear case can be replaced by $\langle \phi(\vec{x}_i), \phi(\vec{x}_j) \rangle$ in

non-linear case. Imagine there is a “kernel function” K such that $K(\vec{x}_i, \vec{x}_j) = \phi(\vec{x}_i) \cdot \phi(\vec{x}_j)$, then we don't need to know ϕ explicitly. Actually there are such Kernel functions, and the most commonly used one is *RBF kernel*, which can implement any continuous decision boundary [19].

$$k_r(\vec{x}_i, \vec{x}_j) = e^{-r|\vec{x}_i - \vec{x}_j|^2} \quad (16)$$

The corresponding *dual formation* of this soft-margin problem becomes [17, 18]:

$$\text{Maximize } W(\vec{a}) = \sum_{i=1}^n a_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n a_i a_j y_i y_j K(\vec{x}_i, \vec{x}_j) \quad (17)$$

$$\text{Subject to} \quad 0 \leq a_i \leq C, \quad i = 1, 2, \dots, n \quad \sum_{i=1}^n a_i y_i = 0.$$

$$0 \leq a_i \leq C \text{ can also be written as : } y_i a_i \in [A_i, B_i] = \begin{cases} [0, C] & \text{if } y_i = 1 \\ [-C, 0] & \text{if } y_i = -1 \end{cases} \quad (18)$$

The corresponding *decision function* becomes:

$$f(\vec{x}) = \text{sgn}(\sum_{i=1}^n a_i y_i K(\vec{x}_i, \vec{x}) + b) \quad (19)$$

7.1.1.4 Optimality Criteria for SVM problem

The problem left is to find the $\vec{a}^* = (a_1^*, \dots, a_n^*)$ that would maximize $W(\vec{a})$ in formula (17). Here we present the optimality criteria in order to find \vec{a}^* [18].

Let $\vec{g}^* = (g_1^*, \dots, g_n^*)$ be the products of derivative of formula (17), there is:

$$\vec{g}^* = \frac{\partial W(\vec{a}^*)}{\partial a_i} = 1 - y_i \sum_{j=1}^n y_j a_j^* K(\vec{x}_i, \vec{x}_j) \quad (20)$$

Assume that for all subscript i , there is $y_i a_i^* < B_i$, for all subscript j , there is $y_j a_j^* > A_j$, where A_j, B_i are defined in (18). Based on (i, j) , we define $\vec{a}^\delta = (a_1^\delta, \dots, a_n^\delta) \in \mathbb{R}^n$ as [18]:

$$a_k^\delta = a_k^* + \begin{cases} \delta y_k & \text{if } k = i \\ -\delta y_k & \text{if } k = j \\ 0 & \text{otherwise} \end{cases} \quad (21)$$

Where δ is a positive and quite small. Now we will have:

$$W(\vec{a}^\delta) - W(\vec{a}^*) = \mathcal{E}(y_i g_i^* - y_j g_j^*) + o(\delta) \quad (22)$$

Note that \vec{a}^* is the solution for Maximize $W(\vec{a})$, so that $w(\vec{a}^\delta)$ must be smaller than

$w(\vec{a}^*)$, which means that $y_i g_i^* - y_j g_j^*$ has to be less than 0, namely, $y_i g_i^* < y_j g_j^*$. Then we obtain the following *optimality criterion* [18, 19]:

$$\exists \rho \in \mathbb{R} \text{ such that } \max_{i \in I_{\text{up}}} y_i g_i^* \leq \rho \leq \min_{j \in I_{\text{down}}} y_j g_j^* \quad (23)$$

Where $I_{\text{up}} = \{i \mid y_i a_i < B_i\}$ and $I_{\text{down}} = \{j \mid y_j a_j > A_j\}$.

The common situation is that there are always some a_k^* which are between upper and lower boundaries and the corresponding $y_k g_k^*$ is on both sides of inequity, then there is only one possible value for ρ [18, 19]:

$$\exists \rho \in \mathbb{R} \text{ such that } \forall k, \begin{cases} \text{if } g_k^* > y_k \rho & \text{then } a_k^* = C \\ \text{if } g_k^* < y_k \rho & \text{then } a_k^* = 0 \end{cases} \quad (24)$$

$$\text{From formula (11) we have: } \vec{w}^* = \sum_{i=1}^n a_i^* y_i \Phi(\vec{x}_i) \quad (25)$$

$$\text{We pick: } b^* = \rho \quad \varepsilon^* = \max \{0, g_k^* - y_k \rho\} \quad (26)$$

Note that \vec{w}^* , b^* , ε^* above satisfies the constraints in formula (15).

Now we need to prove that optimality criteria (23) and (24) are enough to minimize $P(\vec{w}, b, \varepsilon)$ and maximize $W(\vec{a})$ in the same time, namely $P(\vec{w}^*, b^*, \varepsilon^*) = W(\vec{a}^*)$ (see formula (15) for definition of $P(\vec{w}^*, b^*, \varepsilon^*)$). A simple derivation using (20) gives [18]:

$$P(\vec{w}^*, b^*, \varepsilon^*) - W(\vec{a}^*) = C \sum_{k=1}^n \varepsilon_k^* - \sum_{k=1}^n a_k^* g_k^* = \sum_{k=1}^n (C \varepsilon_k^* - a_k^* g_k^*) \quad (27)$$

From (24) and (26), we can easily get that $C \varepsilon_k^* = a_k^* (g_k^* - y_k \rho)$, namely, $C \varepsilon_k^* - a_k^* g_k^* = -y_k a_k^* \rho$ is always true, no matter g_k^* is less than, equal to or greater than $y_k \rho$. Then we have [18]:

$$P(\vec{w}^*, b^*, \varepsilon^*) - W(\vec{a}^*) = -\rho \sum_{k=1}^n y_k a_k^* \quad (28)$$

Substitute constraint $\sum_{i=1}^n a_i y_i = 0$ to (28), we know that $P(\vec{w}^*, b^*, \varepsilon^*) - W(\vec{a}^*) = 0$ always holds, namely, $P(\vec{w}^*, b^*, \varepsilon^*) = W(\vec{a}^*)$.

Therefore the optimality criteria in (23) and (24) are enough for us to obtain $(\vec{w}^*, b^*, \varepsilon^*)$ that minimizes $P(\vec{w}^*, b^*, \varepsilon^*)$ and \vec{a}^* that maximizes $W(\vec{a})$.

7.1.2 Cast Preference learning as Ordinal Regression Problem

In the ordinary regression problem, the input space is $\vec{X} \in \mathbb{R}^n$ and the output space $Y = \{r^1, r^2, r^3 \dots r^n\}$ with ordered rank " $>_y$ ". The training set $S = \{(\vec{x}_i, y_i)\}_{i=1}^n$. The mapping

relation $H = \{h: \vec{x} \rightarrow Y\}$. This indicates an ordering " $>_x$ " on the input space by the follow rule [8]:

$$\vec{x}_m >_x \vec{x}_n \Leftrightarrow h(\vec{x}_m) >_y h(\vec{x}_n) \quad (29)$$

In this way, the ordinal regression problem turns to be finding the optimal h , so that for all $\vec{x}_m >_x \vec{x}_n$, it results in minimum number of $h(\vec{x}_m) <_y h(\vec{x}_n)$, the case of inversion resulting from h can be expressed by the risk function $R(h)$ [8]:

$$R(h) = E(L(h(\vec{x}_1), h(\vec{x}_2), y_1, y_2)) = E(L(\hat{y}_1, \hat{y}_2, y_1, y_2)) \quad (30)$$

$$\text{With: } L(\hat{y}_1, \hat{y}_2, y_1, y_2) = \begin{cases} 1 & \text{if } y_1 >_y y_2 \text{ and } !(\hat{y}_1 >_y \hat{y}_2) \\ 1 & \text{if } y_2 >_y y_1 \text{ and } !(\hat{y}_2 >_y \hat{y}_1) \\ 0 & \text{else} \end{cases}$$

As introduced in section 2.2, according to the ERM principle, h^L should minimize the accumulated reversing risk $R_{\text{emp}}(h^L, S)$. Applying to training set S , it can be done as follows:

$$R_{\text{emp}}(h^L, S) = \frac{1}{n^2} \sum_{i=1}^n \sum_{j=1}^n L^S(h(\vec{x}_i), h(\vec{x}_j), y_i, y_j) \quad (31)$$

$\hat{S}: X \times X \times \{-1, +1\}$, with elements of pairs of instances, derives from S as follows:

$$\hat{S} = \{(\vec{x}_i^{(1)}, \vec{x}_i^{(2)}), \varphi(y^{(1)}, y^{(2)})\}_{i=1}^n \quad (32)$$

$$\varphi(y^{(1)}, y^{(2)}) = \text{sign}(y^{(1)} \ominus y^{(2)}) \quad (33)$$

Where $\vec{x}_i^{(1)}, \vec{x}_i^{(2)}$ donates the first instance and second instance of a pair. \ominus is the preference on them (+1 or -1). Now we deduce the empirical risk $R^{0-1}_{\text{emp}}(h^L, \hat{S})$ for set \hat{S} from the risk function $R_{\text{emp}}(h^L, S)$ for set S [8]:

$$\begin{aligned} R^{0-1}_{\text{emp}}(h^L, \hat{S}) &= \frac{n^2}{t} R_{\text{emp}}(h^L, S) \\ &= \frac{1}{t} (L_{0-1} \varphi(h(\vec{x}_i^{(1)}), h(\vec{x}_i^{(2)})), \varphi(y^{(1)}, y^{(2)})) \\ &= \frac{1}{t} (L_{0-1} P(\vec{x}_i^{(1)}, \vec{x}_i^{(2)}), \varphi(y^{(1)}, y^{(2)})) \end{aligned} \quad (34)$$

The mapping function h for set S defines a mapping function p for set \hat{S} in this way:

$$P(\vec{x}_i^{(1)}, \vec{x}_i^{(2)}) = \varphi(h(\vec{x}_i^{(1)}), h(\vec{x}_i^{(2)})) \quad (35)$$

So that the empirical risk base on the L_{0-1} loss of the mapping p on the sample \hat{S} is equivalent to the empirical risk of mapping h on a sample S multiplying a constant factor n^2/t which is unrelated to p or h . Therefore, the problem of ordinal regression can be solved by classifying pairs of instances, in other words, *preference learning*

[8].

7.1.3 Preference Learning based on SVM

We now introduce how to solve the problem of ordinal regression with preference learning based on SVM and theory in the last section. The new training set \hat{S} defined by formula (25) can be represented as n labeled vectors [8]:

$$\hat{S} = \{(\vec{x}_i^{(1)} - \vec{x}_i^{(2)}), Z_i\}_{i=1}^n \quad (36)$$

We take \hat{S} as training set and construct a SVM model that can sign label either $Z_i = 1$ or $Z_i = -1$ to any vector $\vec{x}_i^{(1)} - \vec{x}_i^{(2)}$, namely: $Z_i = \varphi(y^{(1)}, y^{(2)})$. The hyper-plane passing through each instance pair is defined as [8]:

$$Z_i [\vec{w} \cdot (\vec{x}_i^{(1)} - \vec{x}_i^{(2)})] \geq 1 - \varepsilon_i \quad i = 1, 2, \dots, n \quad (37)$$

Using (37) as constraint, we apply QLP formulation to optimize the hyper-plane as introduced in section 7.1.1.3:

$$\min_{\vec{w}, \varepsilon_i} = \frac{1}{2} |\vec{w}|^2 + C \sum_{i=1}^n \varepsilon_i \quad (38)$$

As explained in section 7.1.1.3, (38) leads to the *dual problem* of finding the $\vec{\alpha}^{\text{opt}}$, which will result in:

$$\begin{aligned} \max [\vec{1}^T \vec{\alpha} - \frac{1}{2} \vec{\alpha}^T Z^T Q Z \vec{\alpha}] \\ \text{Subject to} \quad 0 \leq a_i \leq C, i = 1, 2, \dots, n \quad \sum_{i=1}^n a_i y_i = 0. \end{aligned} \quad (39)$$

Where $\vec{z} = (z_1, \dots, z_n)$, $Z = \text{dig}(\vec{z})$, $Q_{ij} = (\vec{x}_i^{(1)} - \vec{x}_i^{(2)})^T (\vec{x}_j^{(1)} - \vec{x}_j^{(2)})$ [8][20]. Note that formula (39) is actually the matrix representation of formula (16). Solve the dual problem using the optimality criterion in section 7.1.5, we can obtain the \vec{a}^* . In reality, *SVM problem* has been solved by many existing software, LIBSVM is the one I choose to assist calculating the optimal weight vector, which I will introduce in section 7.2.3.

After $\vec{\alpha}^{\text{opt}}$ is solved, the \vec{w}^{opt} can be solved by calculating the sum of the differences of all the vectors from the training set \hat{S} [20]:

$$\vec{w}^{\text{opt}} = \sum_{i=1}^n \vec{a}^* z_i (\vec{x}_i^{(1)} - \vec{x}_i^{(2)}) \quad (40)$$

Now we need to work out the position of hyper-planes (thresholds) based on the known \vec{w}^{opt} and instances pairs in the training set. We first define a mapping function set F so that for each mapping function h in H for training set S there will be

a $U \in F$ with [20]:

$$h(\vec{x}_i) = r_i \Leftrightarrow U(\vec{x}_i) \in [\theta(r_{i-1}), \theta(r_i)] \quad (41)$$

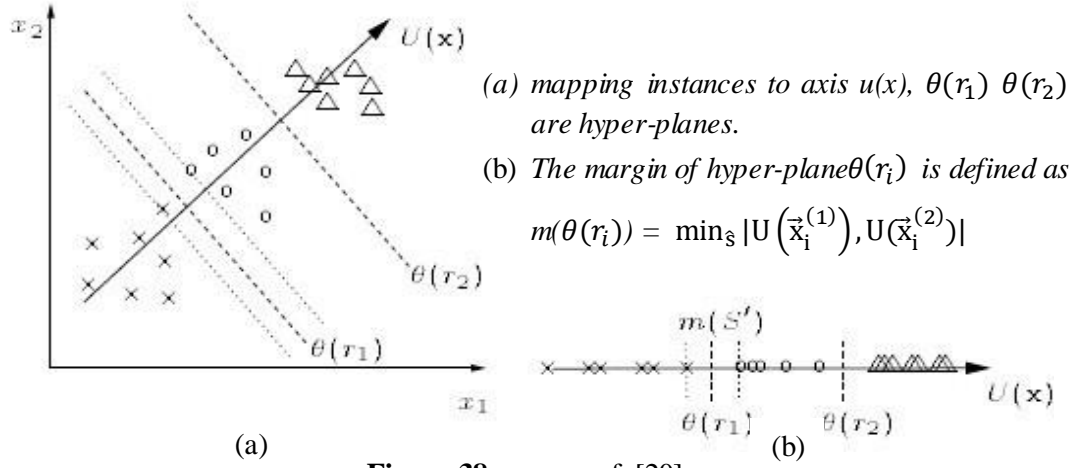


Figure 38 source: ref. [20]

As shown in figure 38, the optimal hyper-plane($\theta(r_k)$) for rank r_k is in the middle of the instances of r_k and r_{k+1} with minimum margin $m(\theta(r_k))$, formally [20]:

$$\theta(r_k) = \frac{U(\vec{x}_1; \vec{w}^{\text{opt}}) + U(\vec{x}_2; \vec{w}^{\text{opt}})}{2} \quad (42)$$

With $(\vec{x}_1, \vec{x}_2) = \arg \min_{h(x_i)=k; h(x_j)=k+1} |U(\vec{x}_1; \vec{w}^{\text{opt}}) - U(\vec{x}_2; \vec{w}^{\text{opt}})|$

We find that \vec{w}^{opt} in formula (41) represents the rule of classifying instances into different classes. Therefore this is the vector which we need to calculate from the pairwise feedback and incorporate into the reviewer's profile.

7.2 Software Implementation

The pairwise feedback processing procedure is symmetric to pointwise approach. Although they adopt different algorithms, they share lots of things in common, such as the functions in class `SubsiftInteraction.java` and `FeedbackProcessing.java`. I will brush slightly over stuff that I explained in section 6 and focus on the distinct features of pairwise feedback processing.

7.2.1 Illustration for JSP Pages

The reviewer needs to interact with 5 jsp pages in order to give pairwise feedback:

- ✓ `index.jsp`;
- ✓ `ActionPageOfGivingPairWiseFeedback.jsp`;

- ✓ ActionPageOfCompletingPairWiseFeedback.jsp;
- ✓ ActionPageOfRenewedList.jsp;
- ✓ ActionPageOfFinalCompletingFeedback.jsp.

The second and third jsp pages are specially written for pairwise feedback obtaining and processing.

➤ ActionPageOfGivingPairWiseFeedback.jsp

Do the same thing as ActionPageOfGivingPointWiseFeedback.jsp does in section 6.2.1. The only difference is the layout of components in the frame, which is designed for the reviewer to give pairwise feedback, as shown in the figure below:

Figure 39 Interface giving pairwise feedback

From figure 39 we know that the reviewer is invited to give 5 ordered pairs of papers in the feedback frame.

➤ ActionPageOfCompletingPairWiseFeedback.jsp

- a) Get parameters (reviewer's selection of 5 ordered pairs of papers) and store them in an static `ArrayList<PaperPair>` field named *paperPairs* of class `PairWiseFeedbackProcessing.java`. `PaperPair.java` is a class to store the order information of an ordered pair of papers. The class structure of class `PaperPair` is shown in figure 40.

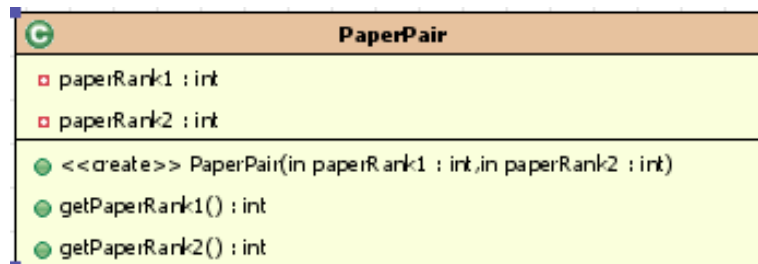


Figure 40 Class PaperPair.java

Field `paperRank1` and `paperRank2` stores the ranking of two papers in the Match Folder item of the reviewer.

- b) Create an instance of class `PairWiseFeedbackProcessing.java`; Call function `go(ArrayList<Integer> rankList)` with the `ArrayList` gotten from step a as the parameter. The process of building a local weight vector and incorporating it into `Subsift` is done in class `PointWiseFeedbackProcessing.java`. The detailed explanation can be found in section 7.2.2

7.2.2 Illustration for Class `PairWiseFeedbackProcessing.java`

Figure 41 shows the fields and methods in class `PairWiseFeedbackProcessing.java`

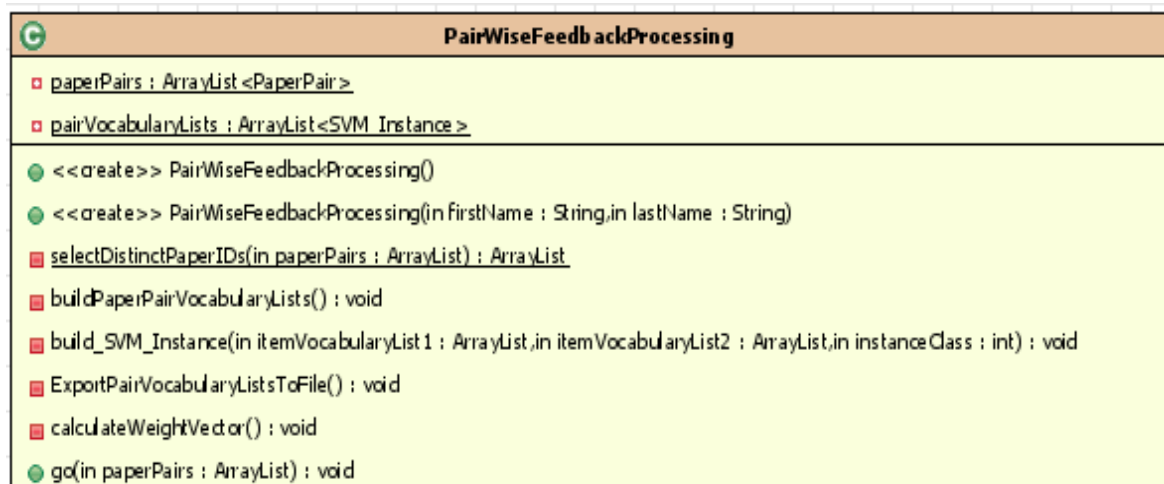


Figure 41 Class `PairWiseFeedbackProcessing.java`

Static field `paperPairs` stores n pairs of feedbacks, every element in Field `pairVocabularyLists` stores SVM instances (see figure 40) created from the reviewers' pairwise feedback.

After calling function `go()` of a newly built `PairWiseFeedbackProcessing.java` instance from `ActionPageOfCompletingPairWiseFeedback.jsp`, a list of functions in `go()` will be executed, as shown in figure 42.

```

PairWiseFeedbackProcessing.paperPairs = paperPairs;
paperRanks = selectDistinctPaperIDs(paperPairs);
buildFeedbackProfileFolder("PairWise");
String feedbackProfileFolderContent = getFeedbackProfileFolderContent("PairWise");
buildVocabularyVector(feedbackProfileFolderContent);
buildPaperList();
buildPaperPairVocabularyLists();
ExportPairVocabularyListsToFile();
svm_train svm = new svm_train();
String[] args = new String[1];
args[0] = "svm.txt";
try {
    svm_model model = svm.start(args);
    calculateWeightVector(model);
} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
incorporateFeedbackIntoSubsift(firstName, lastName, usefulTermTable);
rebuildMatchFolder();

```

Figure 42 Function list in go() in class PairWiseFeedbackProcessing.java

SelectDistinctPaperIDs() is first called to get distinct IDs of papers appearing in the reviewer's feedback.

Then Function buildFeedbackProfileFolder() is called to get document content of papers with IDs gotten from the last step and build Profile Folder for those papers using information in field *top_n_papers*.

After that the content of the feedback Profile Folder is fetched by function getFeedbackProfileFolderContent() and passed to buildVocabularyVector() to build a vocabularyVector for the all terms appearing in the feedback Profile Folder, which is stored in the static field *vocabularyList*.

Then buildPaperList() is called to build a list storing information of papers appearing in the feedback, the information of each list element includes paper ID and paper' vocabulary. This list is stored in static field *selected_papers_list*.

Based on the *selected_papers_list*, a list of SVM_instances with the same length can be formed by function buildPaperPairVocabularyLists(). After exporting the list into a txt file, the SVM training algorithm will read the file, train the data and return a result model. Function calculateWeightVector() extracts information from the model to build the local weight vector.

At last, incorporateFeedbackIntoSubsift(); rebuildMatchFolder() will be called to rebuild the Profile Folder of the reviewer with the local weight vector as parameter and recreate the Match Folder by matching papers with the renewed reviewers Profile Folder.

I will give detailed explanation to functions that only appears in pairwise feedback processing.

➤ selectDistinctPaperIDs()

When giving pairwise feedback, the paper IDs in different pairs might overlap, for example, for ordered paper pairs (1,2), (2,3), paper with ID 2 appears in both pairs. In order to build paper list with distinct paper IDs, I implemented this function to select distinct paper IDs from all feedback pairs.

➤ buildPaperPairVocabularyLists()

From formula (28), we know that the inputs of SVM training algorithm are SVM instances, each instance consists of two parts: \vec{z}_i and $(\vec{x}_i^{(1)} - \vec{x}_i^{(2)})$, $i \in [1, n]$.

The aim of this function is to build an ArrayList of SVM instances derived from the pairwise feedback given by the reviewer. This ArrayList will be stored in a static field named *pairVocabularyLists* of class PairWiseFeedbackProcessing.java. Each element of field *pairVocabularyLists* is an instance of class SVM_Instance.java (figure 43 shows the class structure).

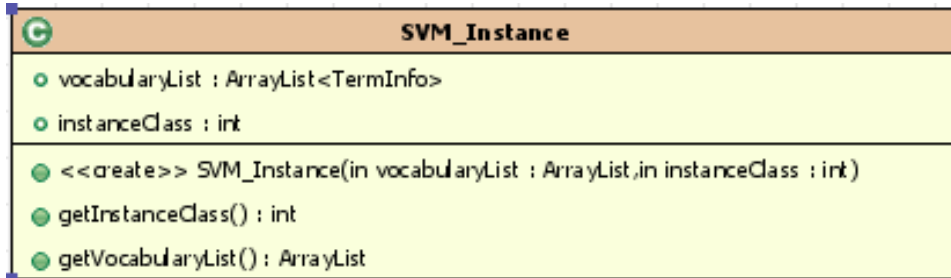


Figure 43 class SVM_Instance.java

In class SVM_Instance.java, field vocabularyList is designed to store $(\vec{x}_i^{(1)} - \vec{x}_i^{(2)})$ and instanceClass is to store \vec{z}_i .

The pseudo code of this function and its support function build_SVM_Instance() is shown in figure 44 and figure 45. Note that we need to obtain a feedback paper's vocabulary information from its ranking in the Match Folder item of that reviewer, since the input is *paperPairs* (remember that the elements of field *paperPairs* were added on page ActionPageOfCompletingPairWiseFeedback.jsp, each pair contains a pair of papers' rankings in the Match Folder item of that reviewer). As shown in figure 44, we first locate the paper's ID in field *top_n_papers* which contains the relationship between a paper's ID and its ranking in the Match Folder item of that reviewer; then obtain the paper's vocabulary list from field *selected_papers_list* based on the paper's ID.

```

private void buildPaperPairVocabularyLists(){
    for(every element in ArrayList paperPairs){

        get rank of each paper in the pair,
        store them in variables paperID_1_rank, paperID_2_rank;

        get IDs of papers in the pair from ArrayList top_n_papers
        identified by paperID_1_rank, paperID_2_rank,
        store them in variables paperID_1, paperID_2;

        find the indexes of papers with paperID_1, paperID_2
        in ArrayList selected_papers_list,
        store them in variables index_paper_1, index_paper_2

        find the vocabulary list of papers identified by index_paper_1, index_paper_1
        in ArrayList selected_papers_list,
        store them in variables itemVocabularyList1, itemVocabularyList2

        build SVM_Instance(itemVocabularyList1, itemVocabularyList2, 1)
        build SVM_Instance(itemVocabularyList2, itemVocabularyList1, -1);
    }
}

```

Figure 44 pseudo code of Function buildPaperPairVocabularyLists()

```

private void build_SVM_Instance(ArrayList<TermInfo> itemVocabularyList1,
                                ArrayList<TermInfo> itemVocabularyList2, int instanceClass){

    clone itemVocabularyList1 into SVM_InstanceVocabularyList;

    for(every term in SVM_InstanceVocabularyList){
        int index = SVM_Instance_VocabularyList.indexOf(currentWord);
        double tfidf_ID1 = SVM_Instance_VocabularyList.get(index).get_tf_idf();
        double tfidf_ID2 = itemVocabularyList2.get(index).get_tf_idf();
        double tfidf_pair = 0.5 + (tfidf_ID1 - 0.5) - (tfidf_ID2 - 0.5);
        if the word only appear in profile ID1
            tfidf_ID1 = tfidf_ID1 + (tfidf_ID1 - 0.5)*2;
        if the word only appear in profile ID2
            tfidf_ID1 = tfidf_ID1 - (tfidf_ID2 - 0.5)*2;
        if tfidf_pair > 1 tfidf_pair = 1;
        if tfidf_pair < 0 tfidf_pair = 0
        currentWord.set_tf_idf(tfidf_pair);
        SVM_Instance_VocabularyList.set(index, currentWord);
    }
    SVM_Instance new_SVM_instance =
    new SVM_Instance(SVM_Instance_VocabularyList, instanceClass);
    pairVocabularyLists.add(new_SVM_instance);
}

```

Figure 45 pseudo code of Function build_SVM_Instance()

The part circled by blue line in figure 44 tells us that each pair produces two SVM_instances: $\{1, (\bar{x}_i^{(1)} - \bar{x}_i^{(2)})\}$ and $\{-1, (\bar{x}_i^{(2)} - \bar{x}_i^{(1)})\}$. Therefore, we would have $2*n$ SVM instances if the reviewer gives n pairs of feedbacks.

The part circled by red line in figure 45 is the code calculating the it-idf value of each element in the term list of one SVM instance. Note that we add extra weight to terms that with non-0.5 weight value in itemVocabularyList1 while with default weight value in itemVocabularyList2. And then I less weight value of terms with non-0.5 weight value in itemVocabularyList 2 while with default value in itemVocabularyList1. I

highlight those terms since they represent one of the biggest differences between these two lists.

➤ exportPaperPairVocabularyListsToFile()

Since the input of SVM training algorithm is a txt file, we need to write the SVM instances in ArrayList *pairVocabularyLists* into a file, one instance per line with the class label going first, as shown in figure 46.

```
1 1:0.0 2:0.0 3:0.0 4:1.0 5:1.0 6:1.0 7:1.0 8:1.0 9:1.0 10:1.0 11:1.0 12:1.0 13:1.0 14:1.0 15:1.0 16:0.0 17:0.0
-1 1:0.0 2:0.0 3:0.0 4:1.0 5:1.0 6:1.0 7:1.0 8:1.0 9:1.0 10:1.0 11:1.0 12:1.0 13:1.0 14:1.0 15:1.0 16:1.0 17:0
1 1:0.0 2:0.0 3:0.0 4:1.0 5:1.0 6:1.0 7:1.0 8:1.0 9:1.0 10:1.0 11:1.0 12:1.0 13:1.0 14:1.0 15:1.0 16:1.0 17:0.0
-1 1:0.0 2:0.0 3:0.0 4:1.0 5:1.0 6:1.0 7:1.0 8:1.0 9:1.0 10:1.0 11:1.0 12:1.0 13:1.0 14:1.0 15:1.0 16:1.0 17:0
1 1:0.0 2:0.0 3:0.0 4:1.0 5:1.0 6:1.0 7:1.0 8:1.0 9:1.0 10:1.0 11:1.0 12:1.0 13:1.0 14:1.0 15:1.0 16:1.0 17:0.0
-1 1:0.0 2:0.0 3:0.0 4:1.0 5:1.0 6:1.0 7:1.0 8:1.0 9:1.0 10:1.0 11:1.0 12:1.0 13:1.0 14:1.0 15:1.0 16:1.0 17:0
1 1:0.0 2:0.0 3:0.0 4:1.0 5:1.0 6:1.0 7:1.0 8:1.0 9:1.0 10:1.0 11:1.0 12:1.0 13:1.0 14:1.0 15:1.0 16:1.0 17:0.0
-1 1:1.0 2:1.0 3:1.0 4:0.08788796854462 5:0.08788796854462 6:0.0 7:0.08788796854462 8:0.0 9:0.08788796854462 10
1 1:1.0 2:1.0 3:1.0 4:0.08788796854462 5:0.08788796854462 6:0.0 7:0.08788796854462 8:0.0 9:0.08788796854462 10
-1 1:0.0 2:0.0 3:0.0 4:1.0 5:1.0 6:1.0 7:1.0 8:1.0 9:1.0 10:1.0 11:1.0 12:1.0 13:1.0 14:1.0 15:1.0 16:1.0 17:0
```

Figure 46 exported txt file content

➤ svm.start()

I use SVM training algorithm in an external Java library named LIBSVM to train the data, obtain the prediction model, and form weight vector in formula (40) using that model. The working principle, parameter selection for LIBSVM can be found in section 7.2.3.

➤ calculateWeightVector()

In the result model of the SVM training algorithm, field SV stores n support vectors (n is the number of input SVM instances); field sv_coef stores the class label for each support vector. The way to calculate local weight vector from SV and sv_coef is shown in figure 47.

```
double[][] sv_coef = model.sv_coef;
svm_node[][] SV = model.SV;
java.util.Iterator<TermInfo> itrVocabularyList1 = vocabularyList.iterator();
while(itrVocabularyList1.hasNext()){
    TermInfo currentTerm = itrVocabularyList1.next();
    String name = currentTerm.getName();
    int code = currentTerm.getCode();
    double weight = 0;
    for(int j = 0; j < SV.length; j++){
        {
            double label = sv_coef[0][j];
            weight = weight + label * SV[j][code - 1].value;
        }
        weight = 0.5 + weight * 0.5;
        if(weight > 1) weight = 1;
        if(weight < 0) weight = 0;
        if(weight != 0.5)
            usefulTermTable.put(name, weight);
    }
}
```

Figure 47 Calculate local weight vector from SVM prediction model

The optimal weight vector is calculated using formula (44). However, the weight values gotten from formula (44) are not within the range of $[-1, 1]$. The part circled by red line in figure 44 is trim the weight values so that they will be shrank within the range of $[0, 1]$.

7.2.3 Illustration for LIBSVM

LIBSVM is a library for support vector machines, which is currently one of the most popular SVM software. A typical use of LIBSVM involves two steps: first, training a data set to obtain a model; second, predict classification result of testing set using the model. In my project, *pairvocabularyLists* serves as the training set, I use LIBSVM to train the data and obtain the result model. Then I can construct local weight vector from that model and incorporate it into Subsift [19].

LIBSVM supports various SVM formulations for classification, regression and distribution estimation, such as C-SVC v-SVC, One-class SVC, etc. From [23], we know that C-SVC is specially designed to solve SVM problem in formula (27), namely, the *dual problem* in formula (16) with decision boundary of formula (17). Therefore, C-SVC would be the svm type I select to train *pairvocabularyLists*.

The working principle of C-SVC is much the same with what I've described in section 7.1.1.4 and 7.1.1.5. The main difference is that LIBSVM is written as the minimization of $-W(\vec{a})$ instead of maximization of $W(\vec{a})$ and the variable $G[i]$ in the source code contains $-g_i$ instead of g_i . LIBSVM's stop criterion is slightly different with formula (23) [18]:

$$\max_{i \in I_{up}} y_i g_i - \min_{j \in I_{down}} y_j g_j < \epsilon \quad (43)$$

Where $I_{up} = \{i | y_i a_i < B_i\}$ and $I_{down} = \{j | y_j a_j > A_j\}$ as in (23) and ϵ is a predefined accuracy

. The training result of LIBSVM is a model with structure shown in figure 48.

```
class svm_model
{
    svm_parameter param;    /* parameter */
    int nr_class;           /* number of classes*/
    int l;                  /* total #SV */
    svm_node **SV; /* SVs (SV[l]) */
    double **sv_coef; /* coefficients for SVs*/
    double *rho;
    double *probA;
    double *probB;
    /* for classification only */
    int *label;
    int *nSV;
    int free_sv;
}
```

Figure 48 Fields in class svm_model.java

Where nr_class is the number of classes, in our case, the value is 2, since we have two

classes {1, -1}. Field l is the number of support vectors, Suppose we have n pairs of feedbacks, the value of l will be 2*n, since we build 2*n SVM instances from n pairs of feedbacks. SV and sv_coef are support vectors and corresponding coefficients. Field label stores the label of each class and field nSV stores the number of support vectors for each class. In our case, label[0] = 1, label[1] = -1, nSV[0] = n, nSV[1] = n.

Recall the way to calculate the optimal weight vector in formula (40), we substitute sv_coef and SV from svm_model into it, there is:

$$\vec{w}^{opt} = \sum_{i=1}^l sv_coef_{[0][i]} * SV[i] \quad (44)$$

Formula (44) is the algorithm to calculate local weight vector in figure 47.

8 Feedback type 3 --- Full list of n papers

In this type of feedback, the reviewer will provide a full ranked list of top n papers. The user interface is shown as below:

Figure 46 Interface of listwise feedback

There is no algorithm specially designed to tackle this type of feedback, instead, a full ranked list of paper can be either converted into the format of pointwise feedback or pairwise feedback based on the preference of the reviewer, then the feedback will be processed by PointWiseFeedbackProcessing.java if “Process as PointWise Feedback” is selected in figure 46, otherwise, the listwise feedback will be processed by PairWiseFeedbackProcessing.java.

➤ Convert full ranked list to pointwise feedback

- Get parameters (reviewer’s selection of full ranked list of papers’ IDs) and store them in an ArrayList named “rankList”.
- Set the static field *listWiseFeedback* of PointWiseFeedbackProcessing to be true.
- Create an instance pointWiseFeedbackProcessing of class PointWiseFeedbackProcessing.

d) Call function go() of variable pointWiseFeedbackProcessing.

The place in PointWiseFeedbackProcessing.java specially written to convert listwise feedback is in function buildFeedbackArray() (as shown in figure 29). The ranking of each paper in prank algorithm is $(i + 1)$, where i is the position of that paper in the full list provided by the reviewer.

➤ Convert full ranked list to pairwise feedback

A full list of n paper can result in $n-1$ ordered pairs of papers. In this way, the listwise feedback is converted into pairwise feedback, the code is shown in the figure below:

```
for(int i = 0; i < 10; i++){
    inputID = "ID" + String.valueOf(i+1);
    paperRank[i] = Integer.parseInt(request.getParameter(inputID));
}
ArrayList<PaperPair> paperPairs = new ArrayList<PaperPair>();
for(int i = 0; i < 9; i++)
    paperPairs.add(new PaperPair(paperRank[i], paperRank[i+1]));
```

Figure 47 Convert listwise feedback to pairwise feedback

The perfect way to convert listwise feedback to pairwise feedback should be constructing $n!$ pairs of feedbacks from the full ranked list of n papers, namely (paper1, paper2), (paper1, paper3),, (paper $n-1$, paper n). However, in that case, the processing time would be intolerably long since there would be $n(n+1)/2$ pairs and $n(n+1)$ SVM instances. Therefore I adopted a simpler way (see figure 47) to convert listwise feedback to pairwise feedback which will result in less SVM instances, however, it will lose some ranking information of the full list. This is one of the place where future improvement can be made, namely, find a way to make use of full information from the feedback list while keeping the running time short enough.

9Algorithm Evaluation and Parameter Optimization

There are mainly two ways to evaluate the effectiveness of the feedback processing algorithm, the first one is the evaluation given by the reviewer, while the second one is done by an evaluation model designed by me.

9.1 Evaluation from Reviewer

The first type of evaluation is on the next to last jsp page ActionPageOfRenewedPointWiseList.jsp (figure 13). Since the reviewers' feedbacks are noisy, I adjust the influence of their feedbacks on the "local weight" vector based on their own evaluation on the renewed ranked list of papers.

For example, if the reviewer selects “Very Satisfied” on `ActionPageOfRenewedPointWiseList.jsp`, the reviewers’ Profile Folder will be updated as shown in figure 24 with the parameter `opinion = 4`. In this way, the influence of this reviewer’s feedback on the local weight vector will be strengthened by 4 times. However, if the reviewer selects “Very Unsatisfied”, the influence of feedback will be weakened to 0.1 times of the previous value.

9.2 Evaluation Model

The model will be explained in 3 parts: the steps to obtain ideal ranking list and give feedbacks to the imperfect list according to the ideal list; the algorithm to evaluate the closeness of the renewed list and the ideal list after incorporating feedback; and the evaluation result on the effectiveness of the processing algorithms obtained from the model.

9.2.1 Evaluation Steps

The second type of evaluation is done with the steps shown as follow:

Step 1: Build an ideal paper list which is Subsift-reproducible.

- 1) Create a local weight vector with terms being those appearing in reviewer Jamie Peng’s profile; Set the “local weight” of each term to be a random value between $[0,1]$.
- 2) Recreate reviewers Profile Folder with the “local weight” vector gotten from step 1) as the value for parameter `terms_weight`; rebuild Match Folder from papers’ Profile Folder and the renewed reviewer’ Profile Folder.
- 3) Get and record the ranked list of paper of Jamie Peng on jsp page: `ActionPageOfGivingPointWiseFeedback.java`. We call it the “ideal list”, which is definitely reproducible by Subsift.

Step 2: Build the paper list to which we will give feedback.

- 1) Rebuild the Profile Folder of Jamie Peng without specifying value for parameter `terms_weight`;
- 2) Rebuild Match Folder from Profile Folder of reviewers and papers
- 3) Get the ranked list of paper of Jamie Peng on jsp page: `ActionPageOfGivingPointWiseFeedback.java`. We call it the “imperfect list”.

Step 3: Give feedback to the imperfect list based on the ideal list

For PointWise feedback, Find the top n and bottom n papers’ names in the ideal list; Locate those papers in the imperfect list; Give feedback with rankings of those papers.

For example, in the ideal list, the top 3 papers' names are doc 8, doc 10 and doc 5. In the imperfect list, the rankings of doc 8, doc 10 and doc 5 are 2, 4, 6. Then in the feedback frame, we select papers with rankings 2, 4, 6 to be the top 3 papers. In the same way, we select the bottom 3 papers.

For PairWise feedback, find 5 pairs of papers in the imperfect list with orders different from ideal list; Give these pairs with orders in the ideal list as feedback to the imperfect list.

For example, in the ideal list, doc4 ranks higher than doc3, while in the imperfect list, doc4 ranks lower than doc 3. Therefore one feedback pair for the imperfect list can be (doc4, doc3), which indicates that doc4 should be preferred over doc3 according to the ideal list we intend to produce.

Step 4: View the renewed list, see how close the list is to the ideal list after incorporating the feedback, so as to evaluate the effectiveness of the feedback processing algorithm.

9.2.2 Evaluation Algorithm

The problem left is to design algorithms to evaluate numerically the closeness of the ideal list and the renewed imperfect list.

Assume that the reviewer is asked to give feedback of top n and bottom n paper to a ranked list of m papers. A paper ranks k in the ideal list while ranking i in the imperfect list. After incorporating the feedback, in the renewed list that paper ranks j . Then the improvement of closeness of the imperfect list after incorporating feedback to the ideal list can be measured as below:

$$I = \left(\sum_{p=1}^m \frac{|k-i| - |k-j|}{m-1} \right) / m \quad (45)$$

In the formula above, $(m-1)$ is the longest distance between two papers in the list, which serves as the distance "reference frame". $|k-i|$ donates one paper's difference of ranking in the imperfect list and in the ideal list. $|k-j|$ donates one paper's difference of ranking in the renewed list and in the ideal list. If $|k-j| < |k-i|$, the improvement of ranking accuracy of that paper will be positive, otherwise it will be negative, which means the paper ends up with a rankings further from where it ranks in the ideal list after incorporating feedback.

Also note that $|k-i| - |k-j|$ is divided by $(m-1)$. This is to make the ranking accuracy improvement proportional with the size of the list.

The accuracy improvement percentage I serves as the measurement of the effectiveness of feedback processing algorithms.

9.2.3 Evaluation Result

In pointwise processing algorithm, give feedback of top 3 and bottom 3 papers, do 10 rounds of evaluation following the steps in 9.2.1 and calculate the result using formula (45), table 3 shows the experiment result:

Expriment Number	1	2	3	4	5	6	7	8	9	10	AVG	STDEV
Improvement In Accuracy (%)	3.31	1.92	3.58	3.03	3.27	3.59	4.08	2.46	3.02	3.23	3.25	0.8

Table 3 Experiment Result of assessing pointwise processing algorithm

Table 3 tells us that although the improvement is not significant, the ranking result gets closer to the ideal ranking after incorporating reviewers' feedback.

We can also notice that the highest accuracy improvement is 4.08% and the lowest one is 1.92%, and the standard deviation of 10 times' experiments is 0.8 %. The instability of accuracy improvement partly results from the randomness of the ideal list. In some rounds of experiments, the ideal list produced is quite close to the imperfect list, in those cases, the improvement in accuracy can't be high even if the renewed list gets to be much the same with the ideal list. However, in some experiments, the ideal list is quite different from the imperfect list, so that the improvement in accuracy is possible to be high.

In pairwise processing algorithm, give 5 ordered pair of papers as feedback, do 10 rounds of evaluation following the steps in 9.2.1 and calculate the result using formula (45), table 4 shows the experiment result:

Expriment Number	1	2	3	4	5	6	7	8	9	10	AVG	STDEV
Improvement In Accuracy (%)	3.31	2.58	2.19	2.51	3.73	2.46	2.39	2.10	2.82	1.69	2.58	0.59

Table 4 Experiment Result of assessing pairwise processing algorithm

Comparing table 4 to table 3, we find that the improvement in ranking accuracy after processing feedbacks of 5 pairs of ordered papers is lower than the improvement after processing feedback of top 3 and bottom 3 papers. We can't say that Prank is better than Ranking SVM since the former type of feedback carries more information than the latter one, since we can obtain 6! pairs of ordered papers from the former type of feedback. We also can't determine that Ranking SVM is more stable than Prank based

on the standard deviation because of the interference of the randomness of the ideal list.

Table 5 shows the evaluation results of processing listwise feedback using Prank and Ranking SVM respectively.

Expriment Number	1	2	3	4	5	6	7	8	9	10	AVG	STDEV
Prank(%)	8.42	9.01	8.56	8.92	9.30	9.02	7.26	8.49	9.52	7.72	8.62	0.70
Ranking SVM(%)	3.67	3.56	3.29	4.80	3.32	3.35	4.92	3.32	3.85	5.34	3.94	0.78

Table 5 Experiment Result of processing listwise feedback using two algorithms

In Table 5 we can see that Prank is more stable than Ranking SVM since it has smaller standard deviation value. the performance of Prank is better than Ranking SVM on the same training set. This is partly because of the way I construct instance pairs, which doesn't carry full information of that list (as I have explained at the end of section 8). Also, I give another direction in improving pairwise processing algorithm in section 10.2.

9.3 Parameter Optimization

Note that in pointwise approach we can give feedback of top n and bottom n papers, here we test the improvement in ranking accuracy by setting n to be 3, 4, 5. Here we also run the evaluation model for 10 rounds. The result is shown in table 6:

Evaluation Result% Parameter	Test NO	1	2	3	4	5	6	7	8	9	10	Average Improvement In Accuracy
n = 3		3.31	1.92	3.58	3.03	3.27	3.59	4.08	2.46	3.02	3.23	3.25%
n = 4		4.02	3.49	4.57	4.73	5.81	6.54	5.02	5.33	4.15	4.88	4.95%
n = 5		9.97	7.83	9.64	8.26	9.38	8.89	10.32	8.82	9.67	8.94	9.17%

Table 6 Parameter optimization of pointwise processing algorithm

From Table 6, we know that as the reviewer gives more information about the ideal list, the renewed list gets closer and closer to the ideal list. This is logically correct, which proves the effectiveness of Prank algorithm from another aspect.

We also need to note that although the improvement in accuracy given different ideal lists while fixing the number of input instances (see every row in table 6) varies a lot, the improvement in accuracy increases stably as the number of number of inputs increases given the same ideal list (see every column in table 6). This shows that without the interference of the randomness of the ideal list, the performance of Prank

is stable.

Based on the experiment result above, I decided to ask the reviewers to give top 5 and bottom 5 papers as the feedback. I rearranged the components on jsp page and changed the input to the algorithm. Since I used arraylists instead of arrays to store information, I needn't to change any part of classes dealing with pointwise feedback, although the number of input instances has been changed.

In pairwise approach we can give feedback of n pairs of papers, here we test the improvement in ranking accuracy by setting n to be 3, 5, 7. Run the evaluation model for 10 rounds. The result is shown in table 7:

Evaluation Result Parameter	Test NO	1	2	3	4	5	6	7	8	9	10	Average Improvement In Accuracy
n = 4		2.31	2.28	2.19	2.31	1.25	3.16	2.23	2.31	2.27	2.15	2.25%
n = 5		3.31	2.58	2.19	2.51	3.73	2.46	2.39	3.10	1.82	1.69	2.58%
n = 6		2.58	3.32	2.64	2.91	3.12	3.13	2.01	2.49	2.48	2.90	2.76%

Table 7 Parameter optimization of pairwise processing algorithm

From Table 7, we find that there will be improvement in accuracy after increasing the number of feedback pairs, although the improvement is not as obvious as in pointwise approach. This is understandable since the information carried by per isolated feedback pair can't compare with the information carried by increasing two elements in an ordered list.

From columns of table 7, we know that the performance of Ranking SVM is not stable, since for some columns (such as column 6, 9, 10), the improvement in ranking accuracy decreases while increasing the number of training instances.

10 Future Work

Larger scale of experiments can be done in the future to test the robustness and effectiveness of the software. Apart from that, there are several aspects in the software where future improvements are needed, including parameter optimization, input inspection, interface optimization, parameter optimization etc.

10.1 Larger Scale Experiment

Currently in the reviewers' Profile Folder, there is only one item "Jamie Peng", and in the papers' Profile Folder, there are 20 paper items. This means that, at present, we can only input "Jamie Peng" on the index.jsp.

All experiments are done upon this scale of testing data. In section 9.2.3, I produce

n different ideal lists by setting random local weight values to the terms in profile of Jamie Peng for n times and then recreate the Match Folder for reviewers' profile folder and papers' profile folder.

Larger scale experiment needs more reviewers' document items and papers document items to be added to the "reviewers" Document Folder and "papers" Document Folder. After that, "reviewers" Profile Folder and "papers" Profile Folder need to be rebuilt. At last, "pc_reviewer" Match Folder can be recreated with ranked list of papers for more reviewers. Then reviewers with their match items existing in the Match Folder can try the application with inputting their own names.

I didn't ask more people to try the application since I think automatic evaluation done in section 9 can assess the effectiveness of the application objectively. I can evaluate the closeness of the improved ranked list after incorporating feedback with the ideal list precisely without the interference of unpredictable (maybe unreasonable) subjective factors from human beings. Although my experiment avoids being influenced by the noisiness of reviewers' feedbacks, I take the noisiness into consideration by incorporating the reviewer's evaluation on the renewed list into Subsift (as shown in section 9.1).

10.2 Parameter optimization

In LIBSVM, There are other parameters can be optimized before training besides the type of svm. LIBSVM provides a simple tool grid.py to check a grid of parameters. From each parameter setting, LIBSVM obtains cross validation accuracy. In the end, the parameter setting with the highest accuracy will be returned. The tool assumes that RBF kernel (Gaussian in Figure 35) is used. In RBF kernel, r is the parameter which can be optimized. Note that in formula (27), C is the parameter available for optimization. So that the parameter setting turns to be (C, r) . Since I didn't find the way to call python function in java code, those parameters haven't been optimized beforehand in my application. This can be a good direction to improve the effectiveness of pairwise feedback processing algorithm in the future [19].

10.3 Input Inspection

On the feedback receiving interfaces (as shown in figure 10, figure 39, and figure 46), I used dropdown boxes to enable reviewers to give their feedbacks. However, I didn't do parameter inspection for inputs, so that the application will not give any warning on invalid inputs. Here are some invalid inputs that we need to tackle properly.

When the reviewer gives pointwise feedbacks, we need to check whether the input numbers of top n papers and bottom n papers are distinct. Although inputting feedback with duplicating elements will not cause crash of the program, it is logically

insane.

When the reviewer gives pairwise feedbacks, apart from checking whether there are duplicating pairs, we also need to check whether he provides invalid feedback pairs. For example, the reviewer gives three pair of feedback (doc2, doc3), (doc 3, doc4), (doc4, doc2), we note here that if doc2 is preferred over doc3, and doc3 over doc4, then it is impossible for doc4 to be preferred over doc2.

10.4 Interface Optimization

The listwise feedback receiving interface can be designed in a more user-friendly way. We can enable the reviewer to move the title of papers around to form the full ranked list of papers. For example, if he wants paper “A Tutorial on Support Vector Machines for Pattern Recognition” to rank 1st, he simply move that title on top of other titles. This can not only bring convenience to the user, but also avoid the risk of having duplicating items in the feedback list.

11 Conclusion

One direction to improve the matching functionality of Subsift is to incorporate reviewers’ individual preference information from their feedbacks. After studying closely of algorithms available in the field “learning to rank”, I designed 3 formats of feedbacks for the reviewers to give, implemented user interfaces, as well as the algorithms behind to process the feedbacks and incorporate the result into Subsift. The result of evaluation demonstrates that the feedback processing algorithms are effective, since the renewed ranking list produced after incorporating reviewer’s feedback got closer to the ideal ranking list.

Bibliography

- [1] Simon Price, Peter A. Flach, Sebastian Spiegler, Subsift: a novel application of the vector space model to support the academic peer review process, *Workshop on Applications of Pattern Analysis (WAPA 2010)*, Windsor, UK. ISSN 19387228, pp. 20–27. September 2010.
- [2] S. Kramer, G. Widmer, et al. Prediction of ordinal classes using regression trees. *Fundamenta Informaticae*, 2000.
- [3] R. Nallapati, Discriminative model for information retrieval, *SIGIR*, 2004.
- [4] Ponte, J. M. and Croft, W. B, A Language Modeling Approach to Information Retrieval, *ACM SIGIR*, 275–281, 1998.
- [5] Robert Malouf, et al. A comparison of algorithms for maximum entropy parameter estimation, <http://delivery.acm.org/10.1145/1120000/1118871/p18-malouf.pdf?key1=1118871&key2=2716394031&coll=DL&dl=ACM&ip=137.222.230.14&CFID=21910382&CFTOKEN=46984588>
- [6] Tie-Yan Liu. Learning to ranking for information retrieval, <http://research.microsoft.com/en-us/people/tyliu/letor-tutorial-sigir08.pdf>
- [7] K. Crammer, Y. Singer, PRanking with ranking, *NIPS*, 2002.
- [8] R. Herbrich, T. Graepel, et al. Support Vector Learning for Ordinal Regression, *ICANN* 1999
- [9] F. Harrington. Online ranking/collaborative filtering using the perceptron algorithm, *ICML*, 2003
- [10] A. Shashua, A. Levin, Ranking with large margin principle: Two approaches, *NIPS*, 2002.
- [11] Y. Freund, R. Iyer, et al. An Efficient Boosting Algorithm for Combining Preferences, *JMLR*, 2003.
- [12] Z. Cao, T. Qin, et al. Learning to Rank: From Pairwise to Listwise Approach, *ICML*, 2007.
- [13] F. Xia. T.-Y. Liu, et al. Listwise Approach to Learning to Rank –Theory and Algorithm, *ICML*, 2008.
- [14] G. Salton, A. Wong, and C. S. Yang, “A vector space model for automatic indexing,” *Commun.ACM*, vol. 18, no. 11, pp.613–620, 1975.
- [15] R. T. Fielding and R. N. Taylor, Principled design of the modern web architecture, *ACM Transactions on Internet Technology*, vol. 2, pp. 115–150, May 2002.
- [16] Simon Price, Peter Flach, Sebastian Spiegler, Christopher Bailey, Nikki Rogers, Subsift web services and workflows for profiling and comparing scientists and their published works, *The 6th IEEE International Conference on e-Science (e-Science 2010)*, Brisbane, Australia. ISBN 9780769542904, pp. 182–189. December 2010.
- [17] R. Berwick, An Idiot's guide to Support vector machines (SVMs), <http://www.cs.ucf.edu/courses/cap6412/fall2009/papers/Berwick2003.pdf>

-
- [18] L éon Bottou, Chih-Jen Lin. Support Vector Machine Solvers, <http://mitpress.mit.edu/books/chapters/0262026252chap1.pdf>
- [19] Chih-Chung Chang, Chih-Jen Lin. LIBSVM: A Library for Support Vector Machines, <https://cs.nmt.edu/~kdd/libsvm.pdf>
- [20] Christopher J.C. Burges. A Tutorial on Support Vector Machines for Pattern Recognition, *Data Mining and Knowledge Discovery*, 2, 121–167, 1998.
- [21] Roger Hartley. *Liner and Nonlinear Programming: An Introduction to Linear Methods in Mathematical Programming*, 1985.
- [22] Mokhtar S.Bazaraa, C.M.Shetty. *Nonlinear Programming*, 1979.
- [23] D. Cossock, and T. Zhang, Subset ranking using regression. *COLT*, 2006.
- [24] Teevan, J. and Karger, D., Empirical Development of an Exponential Probabilistic Model for Text Retrieval: Using Textual Analysis to Build a Better Model, *the 26th Annual ACM Conference on Research and Development in Information Retrieval*, 2003.
- [25] Cao, Yunbo, Xu, Jun , et al. Adapting ranking SVM to document retrieval. *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval. New York, NY, USA , ACM*, 2006.
- [26] A. Shashua and A. Levin. Taxonomy of Large Margin Principle Algorithms for Ordinal Regression Problems. *Technical Report 2002-39*, Leibniz Center for Research, School of Computer Science and Eng., the Hebrew University of Jerusalem.