



Team

- **Ville Valkonen - CSE - 0750931**
 - **WIS: Webcrawling amazon.**
 - **IR: Implementing Dice, Damerau-Levenshtein and combining them, also made some helper methods to speed them up with a large datasets.**
 - **ML: Tf-Idf and making optimization to run it more smoothly with a large dataset. Combining Good/Bad judging on a sense way.**
- **Arjen Meurers - CSE - 0603149**
 - **WIS: Webcrawling amazon.**
 - **IR: Keyword retrieval and filtering.**
 - **ML: Keyword classification.**
- **Javier Salcedo – CSE - 0759906**
 - **IR: GUI accesibility to the keyword retrieval process.**
 - **ML: Association rules extraction based on the A Priori algorithm.**

Introduction

For our **WIS-IR assignment** we have decided to create a program that can help a user to analyze a product based on its costumer reviews. Websites like **amazon.com** that sell all kinds of products often have the option for a user to add a review on how he or she thinks about the product. However these reviews can be added by anyone, even people who wish to promote the product in question. For the person reading these reviews in an attempt to decide whether he wishes to buy this product, it can be hard or even impossible to know which reviews to trust and which not. This fact that user made comments about a product are potentially untrustworthy, are the basis for our tool.

The question now arises, how we can deal with the fact that comments are unreliable?

To illustrate how we intend to tackle this problem we look at a practical example of a user looking to buy a new wireless headphone:

Searching through several websites and hardware stores, a costumer has found a potential headphone he wishes to buy. But the costumer still has his doubts as to whether this is really the headphone that suits him best. At the website of an online store he finds that his headphone has hundreds of reviews. While skimming through several of these reviews, he finds many positive comments, about the quality of the sound, the comfort of the headphone and the longevity of the batteries. But he also notices that some reviews mention noise because of the wireless radio. Investigating further the costumer finds that the headphone actually has an analog receiver instead of a digital one that is common in most modern headphones. Based on this information the costumer decides not to buy the headphone, as he really dislikes noise in his headphone.

This practical example shows us that while a product can be highly praised, it still can have issues that might make it uninteresting to a potential costumer. It is however often difficult to find this information in several hundreds of costumer reviews. As we have mentioned before, many of these reviews might be overly positive, while ignoring important disadvantages of a product, whether this was done intentionally or not.

But the practical example also shows how to deal with this issue. A tool is needed that can scan through all the reviews of a product. Scanning through these reviews it will store all the keywords, or combinations of them. At the end it will count in how many reviews a keyword appears. This will generate a list of keywords in order of number of reviews. The idea is that if a certain fact about a product, for example the noise of a headphone, appears in many reviews, we can assume this fact to be both important and potentially trustworthy. It is important because many individual people talk about it. And it is trustworthy if the keyword goes against the general opinion of the product. If a product is considered very good, but many people mention a certain issue, we can safely assume this issue indeed occurs. The opposite is also true, if a

product is considered bad, the occurrence of a positive keyword, can be safely assumed to be a positive property of the product.

Note that this method also deals with overly positive or negative reviews. Even if someone mentions that the sound quality of the headphone in question is extremely bad, possibly due to a faulty copy, this will not reflect in the result of our tool. Only if a property is mentioned by many different reviewers, can we assume it to have some kind of validity to the whole product.

How to Run the Application

To run the application the user has to double click in the **UCA.jar** file. It is mandatory that the user has the **java 1.6 version** installed in his computer and **python** in the PATH variable.

It is also possible to run the application using the **IDE NetBeans 6.8, or higher**.

The steps to execute this code are:

- **Open IDE NetBeans 6.8** or higher.
- Import NetBeans Project.
- Run the **main.Main.java** file.

The second choice is interesting in order to debug or see more in detail the internal processes of the application. If the user pretends to debug the program instead of opening the NetBeans project, can do it from the command line or console typing: **java -jar UCA.jar**.

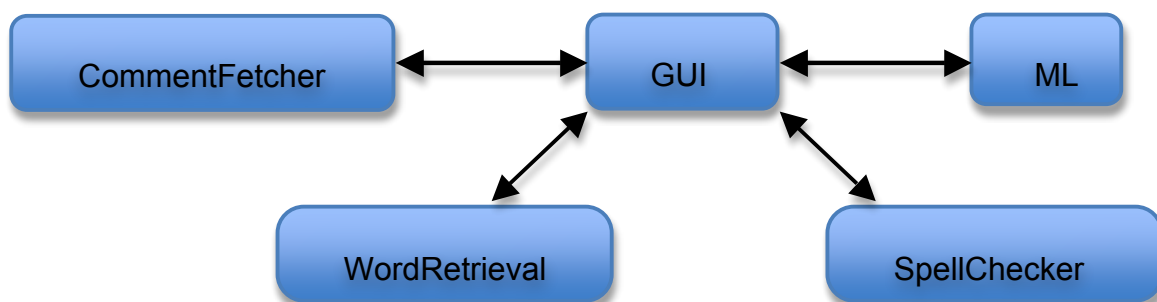
This project has been tested in **Linux, MAC OSX** and **Windows 7**, working in all of them under the requirements mentioned before.

Architecture

The application has been developed and code in **Java** over the **SDK 1.6 version** using the **IDE Net Beans 6.8** for that purpose, and **Python** for the *amazon comment fetching process*.

Thus, the whole GUI has been built using the **Net Beans GUI tool**, generating some extra code because of that, but making the process easier than other IDEs like Eclipse .

The system architecture is displayed as a diagram below and, each one of its components are described further afterwards:



CommentFetcher (WIS)

This module was totally implemented by Arjen Meurers and Ville Valkonen for their WIS task.

At the beginning there were questions

1. How to determine comment boundary
2. Is comment boundary always the same
3. How to parse fields such as Name, Stars, Header and Comment
4. How to save data in a convenient way for later use
5. How to clean up extra characters and information we have, for example HTML-tags
6. How to crawl all pages?
7. Procedure from the beginning to the end

Preface

At the beginning we examined how the comments are structured in a product page of Amazon. For that, there were handy option in Firefox, behind the right button called View Selection Source. Painting a word and then using View Selection Source it quickly revealed awfulness what dynamically generated content can be. A total mess.

How to determine comment boundary?

The very first thing was to detect when a comment changes to another. That was a trivial case. Amazon indicated it by “<!-- BOUNDARY -->”. Thus it was easy to spot and match, they seem to have added extra new lines with spaces to not to make this too easy. To remove multiple spaces regular expression were used.

After little pondering regular expression seemed to be a good catch to match other needed elements from the page too. Regular expressions are used to detect whether there is a next page, how many comments customers have written and to cut down extra character noise, just to name a few of them. For example, if there is a next page, following regular expression is used:

```
re.compile(r"all (\d|,)*\d* customer")
```

This matches combinations like “1, 234”, “1234” and “1”.

How to parse fields such as Name, Stars, Header and Comment

As mentioned in the previous paragraph, regular expressions were used to obtain sensible data such as an ID of a comment and a header of a comment. After reading these values from the comment, these are saved in a vector that contains Comment.py entities, thus data can be accessed more easily.

How to save data in a convenient way for later use

For later use we save comments in following format:

Name: Comment ID

Stars: 5.0 out of 5

Header: Subject of the review, customer defined

Comment:

This is a comment.

Because we remove all non alphabet characters from the comments “---” will be a good sign to detect comment changes.

How to clean up extra characters and information we have, for example HTML-tags

The data collected so far is in useless form because it contains HTML-tags and extra characters made by dynamical page creation. For cleaning out HTML-tags there

is a snippet called `strip_ml_tags()` done by Micah D. Cochran. This was perfect for our use.

How to crawl all pages?

To obtain all the comments we have to crawl through all the pages. Again we use regular expression to detect whether there is a link to next page or not.

Procedure from the beginning to the end

1. Whole process at the beginning to the end is following:
2. User gives a URL which refers to a product main page from Amazon.
3. Program will fetch the page and store it in disk in HTML format
4. If there are reviews, return the line number where the link is and the link
5. Count how many comments exists
6. Loop:
 - a. Parse needed info: Name, Stars, Header, Comment
 - b. Set them to a new entity
 - c. Show information to user: comments fetched / comments total, pages fetched / pages total, estimated time of arrival
 - d. If there is a next page, continue
7. Show time elapsed and exit

Conclusion

Because the pages were generated dynamically there were many situations that couldn't be repeated. This non deterministic operation made debugging highly slow operation. In the end it still works for most of the pages but there might be still some pages and tags that can cause disturbance.

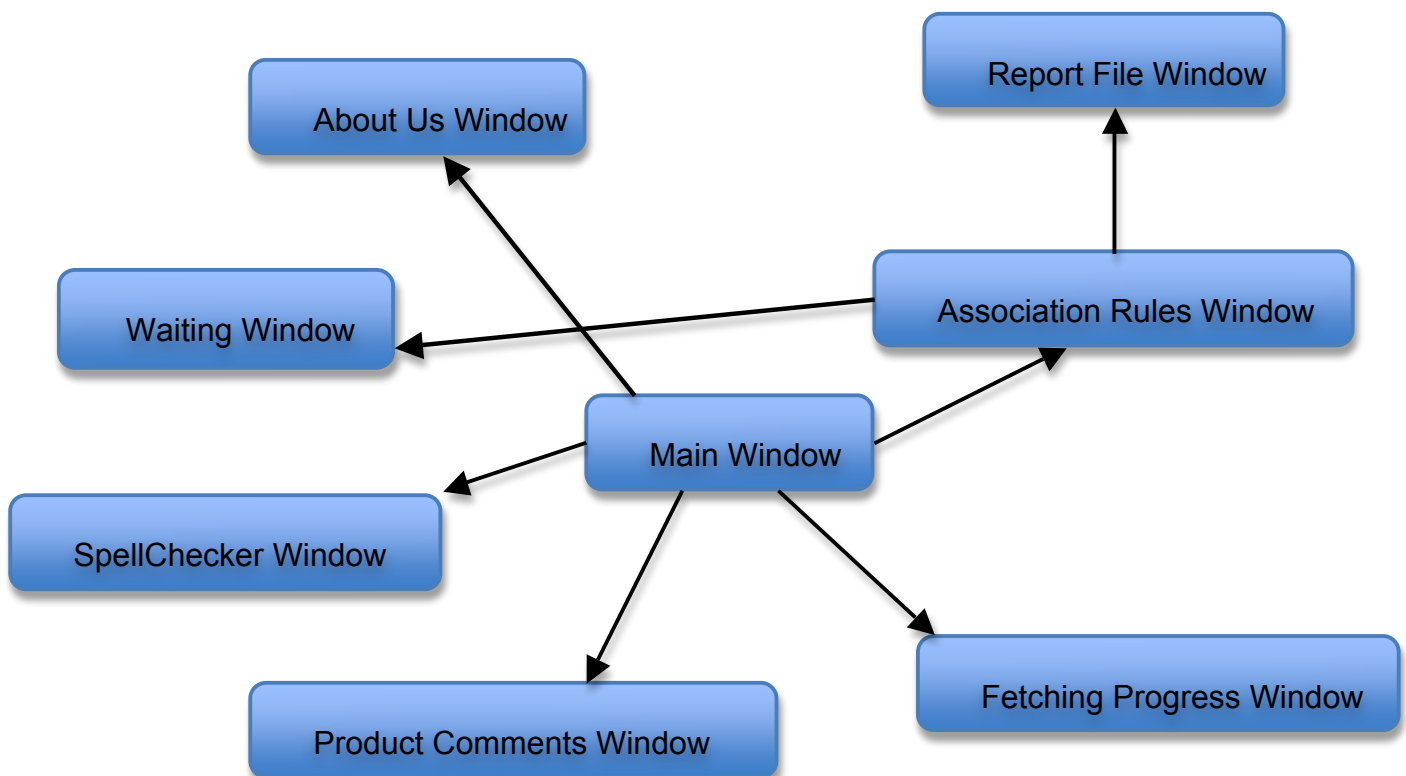
GUI

Represents the application **Graphic User Interface**. This module was totally implemented by Javier Salcedo.

Two important details concerning the coding of this module are:

- Each one of the components have been designed with the **Singleton** design pattern so each one of the components is created only once.
- Also the **thread** matter is also important in order to wait between threads. Each process is done in a separate thread making the application **Thread-Safe**.

This module has the following structure:



Each one of the previous components are briefly explained:

- **About us:** Contains the about us window classes to handle the displaying of the about us window.
- **Association Rules Window:** Handles the association rules Windows.
- **Main Window:** Contains all the methods relating to the main window.
- **ProductCommentWindow:** Contains the text area in which the comments are displayed with the selected keyword in the panel.
- **ProgressWindow:** Handles the progress window which shows the fetching progress.
- **SpellCheckerWindow:** Handles the spell checker tool of the application.
- **WaitingWindow:** Handles the window which is displayed when a progress is executing.

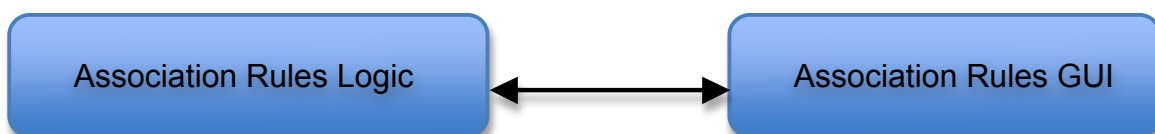
ML (Machine Learning)

Contains the following components:

Association Rules

This is the association rules generator module which handles all the process mentioned before.

The structure is:



The components are:

- **Association Rules Logic:** Contains the methods to execute the **A Priori** algorithm.
- **Association Rules GUI:** Classes to handle the GUI for the association rules. It contains, as we displayed before, the Association Rules Window, which configures the A Priori algorithm and executes it displaying the results in a panel.

Keyword classification

Contains the methods for the keyword classification split in different logical classes in the same package.

This package is further explained in the **Arjen's ML task**.

WordRetrieval

The keyword retrieval module reads and analyses the comments file that is generated by the comment fetcher. It reads the comments word for word. The words are then filtered. The filtering is based on the expression provided by the user. The filter scans through all the words and checks if the current set of words matches the filter tokens. If this is the case, the words are stored together with the number of the comment they appeared in. Once all words in the comments have been read and stored, the words are put through several algorithms. First the **TF-IDF** weight of each keyword is determined. After this the keywords are counted. For each unique keyword the number of comments it appears in is determined. With this information the classification algorithm will now classify the keywords. Both the TF-IDF and the classification algorithm only cover the most occurring keywords, as applying these algorithms to all the acquired keywords would be too expensive. When all the information acquired it is then put together in a data structure that is displayed by the GUI of the program.

Individual Assignments

Ville Valkonen

IR Assignment

Problem Formulation

We wanted to give user an option to check whether there are misspelled or plural words.

Goal

Our preliminary purpose was to make a word substitute of probably misspelled or plural words but we ended up to only show them. Because everyone was making own implementations there were many complex algorithms and data structures that made stemming implementation harder. That's why we ran out of time and ended up only to show these words.

Motivation

Damerau-Levenstein

Damerau-Levenstein distance counts the distance between two words and shows how many changes are needed to change other string to other. Change can be substitution, addition, remove or transposition. To count distance with a plain Levenstein algorithm it tracks changes between additions, removes and modifications. With Damerau addition we are able to track transpositions too.

Dice coefficient

In Dice's coefficient distance can be expressed as a percentage. The result 1.0, can be also read 100%, means that strings are equal.

Experiments design

In example we see what that means:

We have two strings:

dod

dog

Distance between these is 1 because we only need to substitute **d** → **g** to make dod to dog. Damerau addition gives as better change to detect misspelled words than normal Levenstein has. Example follows:

dgo

dog

With plain Levenstein distance this would give 2 as a result when Damerau Levenstein gives 1. Damerau counts **go** → **og** cost of transposition as a 1 and is therefore better to detect a typographical errors.

When distance is 0 strings are equal.

Dice's coefficient

Formula how to calculate Dice's coefficient:

$$s = 2nt / nx + ny$$

Set of bigrams are calculated for both strings. Term **nx** presents bigram of a string **A**, and **ny** presents bigram of a string **B**, while **nt** is intersection of **nx** and **ny**.

Example:

hello → {he, el, ll, lo}

hallo → {ha, al, ll, lo}

Both sets have 4 bigrams. Intersection of them will give result 2, thus “ll” and “lo” can be found from the both sets. Putting all together it will look like this: $(2*2) / (4 + 4) = 0.5$

Evaluation results

After examining couple of algorithms we chose Levenstein distance because it is well known, well examined and used widely by search engines and spell checkers.

Few to mention, compared on Hamming distance which only support strings of the same length, and Longest common substring which support everything else except substitution - Damerau-Levenstein is definitely the most superior algorithm. At least for our use.

Conclusions/Summary including the discussion of main limitations

Already explained in the goal

ML Assignment

Problem Formulation

With a large keyword dataset there will occur slowness if we will not pay attention how we retrieve data from the array. Therefore we are only comparing initials until we will find a match. After that iterate over to a right keyword and count occurrences. This saves time in bigger data sets.

Goal

Tf-Idf is really practical to rule out noisy comments and to give a better view what all the comments are about. Word count in entire comments is not reliable way to do this.

Motivation

In our project Tf-Idf alias Term Frequency-Inverse Document Frequency is used for ranking how important a keyword is. Work flow is following:

- Count how many times the term shows up in the comment and divide by total numbers of words. This is called Term Frequency.
- If the term is really common and appears in many different comments, its occurrence should be limited with Document Frequency. In practice: Count how many times this term can be found from different comments and divide by total amount of comments.

Therefore, we will have the formula:

$$\text{importance} = \text{term frequency} * \log_{10}(1 / \text{document frequency})$$

Experiments design

Example follows:

- If a term “noise” appears 1 time in comment, which includes 28 words, it has $1/28 \sim 0.04 = 4\%$ Term Frequency.
- If “noise” appears in 8 different comments out of 25 total, it will have $8/25 \sim 0.32 = 32\%$ Document Frequency.
- Tf-Idf would be: $\text{termfreq} * \log(1 / \text{docfreq}) \sim 0.18 = 18\%$

Evaluation results

After comparing values of total offurrence of word in the document and Tf-Idf score we normalized keywords to level that we can actually use them for determine product's goodness.

Conclusions/Summary including the discussion of main limitations

By using a boundary value for Tf-Idf-scoring we will rule out words that doesn't have that much weight in comments.

- Then we will append the index number of the each fulfilled word in a set **A**.
- Next set a boundary value and rule out comments that does not fulfill desired Good/Bad ratio. Then again, save the index numbers of fulfilled comments in a set **B**.
- Intersection of **A** and **B** gives us indexes, thus those can be used to verdict whether the product has good rating or not

Arjen Meurers

IR Assignment

Problem Formulation

From a set of comments, the individual keywords for each user comment must be retrieved. Since the single keywords that are retrieved are not always very informative, filtering must be applied.

Goal

We wish to give the user the ability to filter the keywords, by specifying which types or combinations of keywords should be displayed. He should be able to remove uninteresting words from the list like stop words. It should also be possible to show the neighboring keywords of the keyword in question.

Motivation

The list of keywords that will be retrieved by the keyword retrieval algorithm is initially unfiltered. This means that all possible words are shown. Removing the irrelevant words like stop words from the list will clean up the keywords list. After this the user generally wants to know more about keywords he finds interesting. This creates the need to investigate the context of a keyword. The user wishes to be able to see the keywords preceding and following the keyword in question. For example sometimes a keyword is preceded by an adjective, which can be very important to know.

Experiments design

After the **WebCrawler** has retrieved all the comments for a single product, the individual keywords for each user comment must be retrieved. We do this by scanning through each user comment and storing all the keywords, their rating, and the number of the comment in which the keywords appeared. Keywords are normalized by removing special characters and turning upper case characters into lower case. This is done to be able to match as many occurrences keywords together. For example: "Sound", "sound.", "-sound-" and "(sound)" are all turned into the keyword "sound".

To give the user the ability to filter the keywords list, we introduce keyword filters that will enable the user to select which keywords he wishes to see and which not. We use a form of **regular expression** to give the user the ability to precisely express which combinations and types of keyword he wishes to have displayed in the keyword list. The user can give expressions of the following shape to filter the keyword list:

`[] ([] [])`

Where "[]" denotes a keyword of a certain type, and `([] [])` denotes an optional arbitrary amount of additional keywords to be matched. The following *types of expressions* are currently supported by the program:

- **[*]** -> This keyword can be of any kind. No filtering is applied.
- **[+]** -> This keyword can be of any kind, but the keywords are filtered by stop words.
- **[a]** -> This keyword must be an adjective.
- **[c]** -> This keyword must be a connection word(to, and, of, vs,... etc).
- **[;string]** -> This keyword must be equal to the word given by string. This enables the user to make any kind of combination of keywords.

To be able to filter on either stop words, adjectives or connection words, we use **predefined lists** of these words. So for each keyword encountered, the program will check if this keyword is present in either, the stop words list, the adjectives list or the connection words list. The quality of these lists will determine how well this filter will work. If the user wishes to exclude certain keywords from any one of these lists he can edit the lists as they are stored in plain text format.

Evaluation results

After having retrieved all the keywords the result will look like as the left most screen in figure 1.

Product 1		
56: the	2/1: 66.67	[8.3%]
55: i	9/5: 64.29	[13.0%]
54: and	7/3: 70	[10.0%]
49: to	1/0: 100	[11.5%]
48: these	3/0: 100	[6.7%]
46: for	16/1: 94.12	[9.4%]
46: sound	13/1: 92.86	[6.7%]
43: are	0/0:	[8.9%]
43: headphones	8/0: 100	[6.7%]
43: is	0/0:	[4.2%]
43: my	0/0:	[6.7%]
43: they	4/0: 100	[6.7%]
41: of	0/0:	[5.1%]
39: but	11/1: 91.67	[4.5%]
39: that	0/0:	[4.3%]
36: have	0/0:	[5.4%]
35: great	0/0:	[13.3%]

Product 1		
46: sound	13/1: 92.86	[14.3%]
43: headphones	8/0: 100	[16.0%]
35: great	0/0:	[28.6%]
28: good	0/0:	[9.1%]
24: quality	7/2: 77.78	[4.7%]
22: comfortable	1/0: 100	[8.3%]
21: bass	7/2: 77.78	[7.7%]
21: music	0/0:	[12.5%]
18: better	0/0:	[9.1%]
18: dont	0/1: 0	[7.1%]
18: high	1/0: 100	[3.8%]
17: price	2/0: 100	[7.7%]
17: time	0/0:	[5.9%]
15: hd	0/0:	[4.5%]
15: listening	0/0:	[5.6%]
15: open	0/0:	[3.7%]
15: nair	0/0:	[16.3%]

Product 1		
4: good sound	0/0:	[0.0%]
3: better sound	0/0:	[0.0%]
3: great sound	0/0:	[0.0%]
2: loud sound	0/0:	[0.0%]
2: outstanding sound	0/0:	[0.0%]
1: amazing sound	0/0:	[0.0%]
1: clean sound	0/0:	[0.0%]
1: clear sound	0/0:	[0.0%]
1: comfortable sound	0/0:	[0.0%]
1: fantastic sound	0/0:	[0.0%]
1: full sound	0/0:	[0.0%]
1: little sound	0/0:	[0.0%]
1: natural sound	0/0:	[0.0%]

We can see that as expected the program finds a lot of keywords that are of no importance to the user. In this example the word “headphones” is the 9th most found word, while most of the words above it are not interesting. The middle screen in picture 1 shows the result when the stop words are removed by using the expression: **[+]**

We can see that after this filter has been applied most of the irrelevant words have been removed. The word headphones is now second highest of the list, which is to be expected as the product is indeed a headphone.

If the user wishes to focus on one specific keyword he can use the **[;string]** expression. This is useful if the user has identified a specific keyword of interest, and wants to know the relation between this keyword and other keywords. So for example the keyword sound appears high in the list. But the user still does not know whether

the reviewers regard this headphone to have a good or a bad sound. To find this out we can now apply the following filter: [a][;sound]

And as we can see in the right-most screen on picture 1, the majority of occurrences of the keyword sound preceded by an adjective is positive.

ML Assignment

Problem Formulation

The keywords retrieved by the program can have a positive, a negative or a neutral meaning for the product. It is however not possible to determine just from the list of keywords what the relation of the keyword to the product in question actually is.

Goal

We wish to determine what the relation is of certain interesting keywords to the product. The program should show the ratio between the amount of positive and negative occurrences of a keyword.

Motivation

Keywords in themselves contain no information at all. So when a user sees that the word “comfortable” appears in many comments related to a headphone, he still has no idea whether the reviewers find the headphone comfortable or not. To give the keywords some meaning it is crucial to know their relation to the product itself.

Experiments design

Since the keywords themselves contain no information, the algorithm will look at the keyword that precedes the keyword in question. If the keyword is preceded by an adjective, the keyword is often given a positive or negative charge. For example the combination “good sound” indicates that the sound is considered to be positive. This means that if the keyword is preceded by more positive than negative keywords, we can assume the keyword to be considered good.

However the effect an adjective has on a keyword is not always the same. The effect depends on whether a keyword is regarded to be positive or negative in relation to the product. Assume we have a set of keywords based on the reviews of a headphone. The adjective “much” tied to the word “noise” gives a negative charge to the word noise, because noise is regarded negative in relation to headphones. However “much” in combination to “comfort” gives a positive charge to the word comfort because comfort is good for headphones.

Adjectives can be divided in two groups: the qualifiers and the augmenters. We make this division because they have different effects on their accompanying keyword. The qualifiers are adjectives that give a positive or negative charge to a neutral keyword.

The augmenters do not directly give a charge to a keyword, but augments their initial charge. For example the adjective “much” plus the word “noise” augments the charge of the keyword noise. To use these augmenters the algorithm must know the initial charge of a keyword. This is why a user defined list of keywords is used that determines for each of these keywords their initial charge.

In addition to the keyword list, the algorithm uses four lists of adjectives: the positive and negative qualifiers, the increasing and decreasing augmenters.

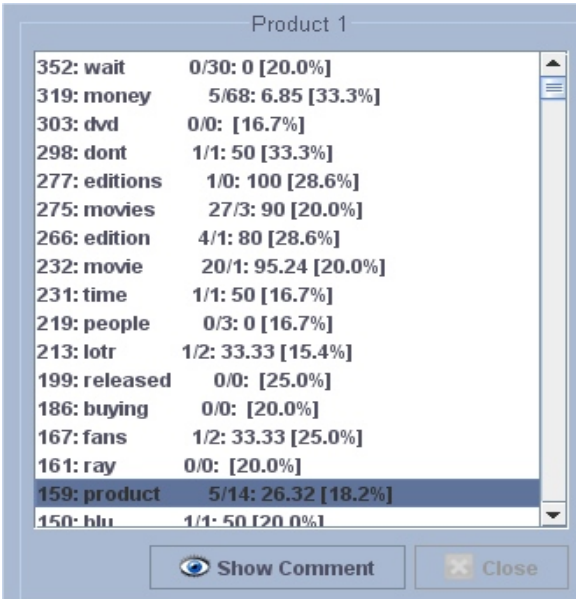
Using the information in the lists, the program can now give a verdict on the rating of a keyword. For each keyword it will find the appearances of this keyword together with one of the adjectives from the adjectives lists. If a combination is found that increases the charge of a keyword, the program increases the value of the posCharge variable. When an adjective is found that decreases the charge of a keyword, the program increases the value of the negCharge variable. Note that for each adjective this happens only once per comment. We do not wish that the multiple appearances of a certain adjective in a single comment influences the final result. Once the values of posCharge and negCharge for a certain keyword have been found, we have an indication of the rating of this keyword. We display the result by giving the percentage of the number of positive occurrences.

Evaluation results

In figure 2 we see the result of the analysis of the reviews for a movie DVD. We can see the keyword movie has 27 positive occurrences and 3 negative ones. Its rating is calculated as follows:

$$(PosCharge / (posCharge - negCharge)) * 100$$

This would give a rating of $(27/30) * 100 = 90\%$. This result shows that for this keyword, the large majority of adjectives + the keyword combinations give a positive charge to the keyword. Since the keyword movie has a higher than 50% rating, we can consider this a good movie. However the product itself is considered to be bad as the rating of the keyword product is 26.32%. Therefore the majority of occurrences of the keyword product are bad.



Keyword	Count	Percentage
352: wait	0/30: 0	[20.0%]
319: money	5/68: 6.85	[33.3%]
303: dvd	0/0: 0	[16.7%]
298: dont	1/1: 50	[33.3%]
277: editions	1/0: 100	[28.6%]
275: movies	27/3: 90	[20.0%]
266: edition	4/1: 80	[28.6%]
232: movie	20/1: 95.24	[20.0%]
231: time	1/1: 50	[16.7%]
219: people	0/3: 0	[16.7%]
213: lotr	1/2: 33.33	[15.4%]
199: released	0/0: 0	[25.0%]
186: buying	0/0: 0	[20.0%]
167: fans	1/2: 33.33	[25.0%]
161: ray	0/0: 0	[20.0%]
159: product	5/14: 26.32	[18.2%]
150: blu	1/1: 50	[20.0%]

Javier Salcedo

IR Assignment

Problem Formulation

Once a user finds a keyword he is interested in, the user wants to find out why this keyword is relevant to the product. The application should link a keyword to the most relevant comments for this keyword such that the user can quickly find the relation between the keyword and the product.

Goal

Find the relation between the keywords obtained and the list of the keywords displayed more easily than watch the keywords one by one in the generated lists or reading all the comments one by one.

Motivation

In order to complete the keyword retrieval task, when the user selects a filter to apply in the list of keywords in the selected products, sometimes is not obvious to determine if that keyword is useful to the him or not.

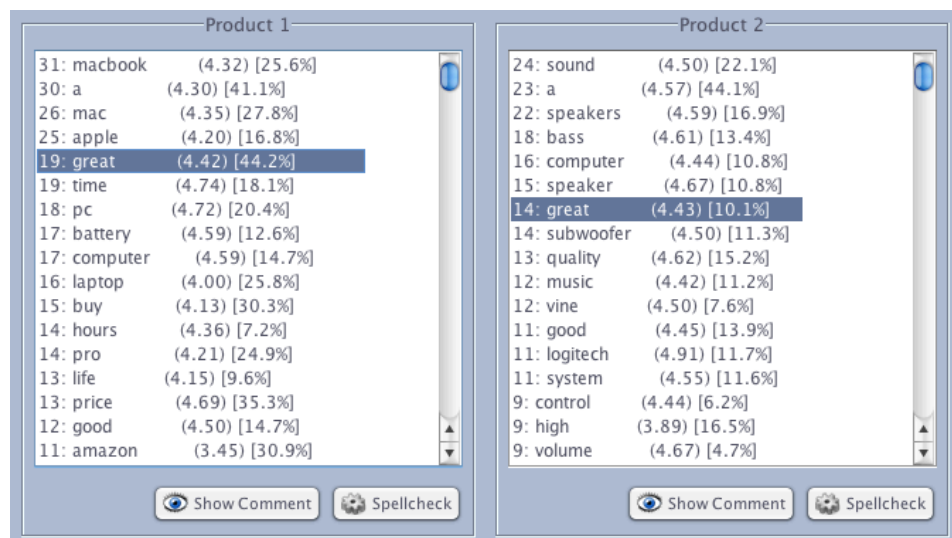
For instance, if the user applies the adjective filter, he obtains a list like: "good, bad, easy..." but, "good what?" or "bad what". And also, does it that adjective appears as well in the other product lists?

Other approaching could be using this tool with the **association rules process** when the results are not as good as expected.

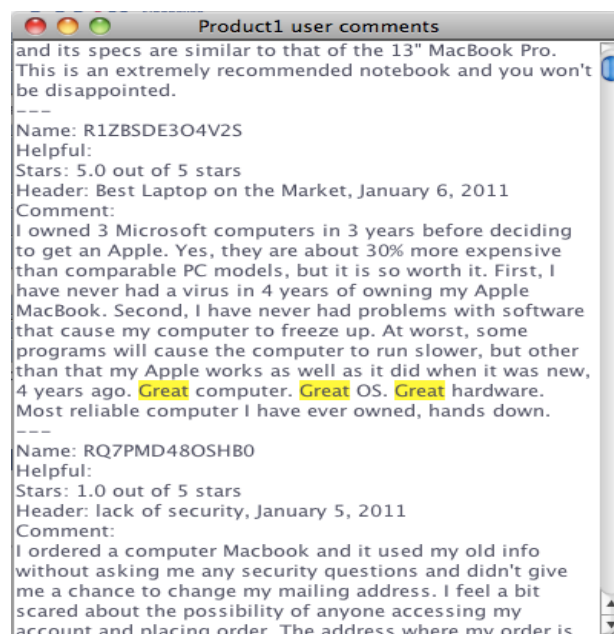
Experiments design

In order to do that, i implemented two main functionalities in the GUI:

- **List Highlighting:** Consist on highlight the selected keyword in one list in the other lists. So the user can check, in a faster way the relationship between that keywords in the products.



- **Comment Highlighting:** Highlights the selected keyword in the comments text file. It displays the comments file generated by the fetching process with the selected keyword in the product list.



Evaluation results

The results have been succesful tested for both sub-tasks mentioned above

When the user selects a keyword it is also highlighted in the other list if appears on them. If not keeps the highlighting in the current list and removes the other highlightings in the other lists if that selected keyword does not appear in that product/s.

Conclusions/summary including the discussion of main limitations

The application allows to select only one word at a time in one of the lists.

Other interesting approaching would be to allow the multiselection in the lists so the user can compare more keywords at a time. I decided not to do it in that way because other functionalities would be badly affected because of that.

We display the list of the keywords, sorted by the number of appeareances as a first handout, and precisely, this IR task was thought to be an addition and not a problem after all.

One the user is get used to the application usage, it is completely useful, and we have observed that actually, it is a very important functionality in our application.

ML Assignment

Problem Formulation

Association rules generation based on the **A Priori algorithm** applied on the appearance of the best 10 keywords in each compared product comments.

Goal

Generate a list of rules, which the user is comparing, based on the appearance of the 10 best keywords of each product with the format "If X appears also appears Y AND Z" in order to find hidden information in the comments thanks to the keywords themselves.

Motivation

Find common features between the products to compare based on the appearance of the keyword list of all the products., like "if Bad appears also appears quality and noise".

In that case, the user can extract some interesting info about the products which he is comparing without the necessity of reading each one of the product comments one by one.

For instance, if the user is comparing 2 headphones and in each one of them the Word "noise" appears a lot and in both products, that indicates that something relevant has to do with the noise.

Experiments design

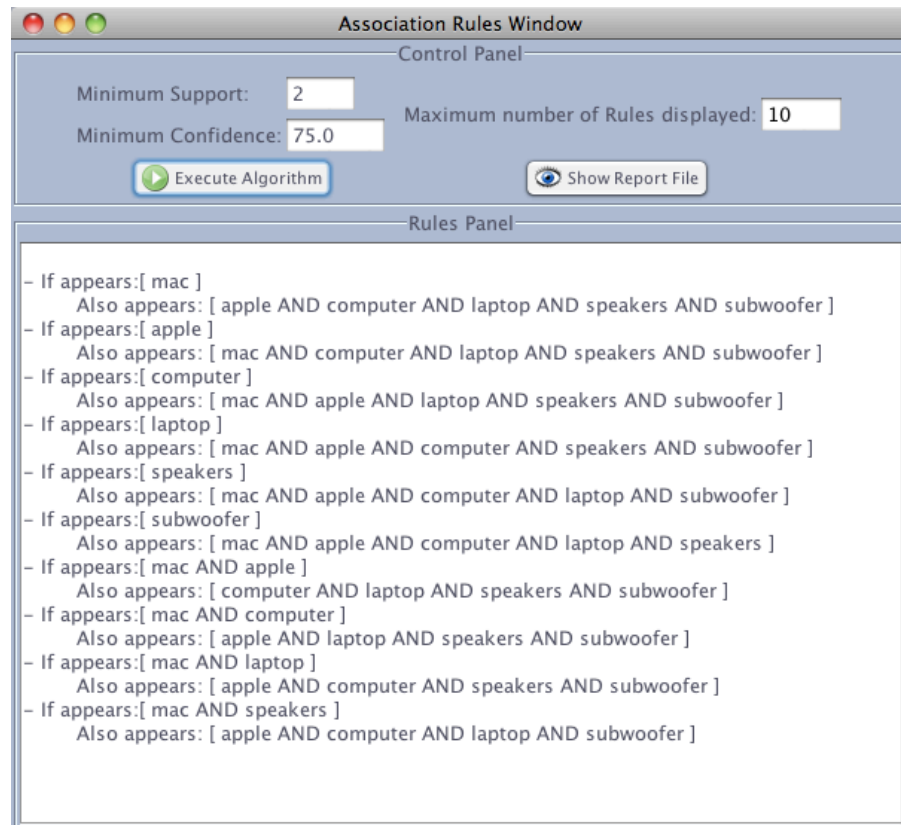
There is a class called **Mining.java** which executes the **aPriori** method to generate the most frequent sets, and afterwards, the association rules themselves. To get that matrix the application parses the keyword lists of the products in the following format:

In order to executes the algorithm, it receives a **false boolean matrix** , represented by an **Integer matrix**, in which 1 → true, and 0 → false. The format is a static bidimensional array, in which the columns are the products and the rows are the appearance of each words in each product comments. For instance: [1,1] means that the keyword X appears in product one and product two, and the same for the rest of the keywords.

Evaluation results

The results obtained depend on the chosen products to compare. Very often, between products with similar features, and therefore, with a lot of keywords in common, the results are useful but using the Javier's IR task mentioned before.

The application can output something like:



"If mac Appears also appears apple AND computer AND laptop AND speakers AND subwoofer".

But computer or laptop what? Then to obtain the desired conclusions the user should **filter the keywords** or just go to the comments themselves and check the words directly with the **highlighting tool**.

But it has to take in account that, this behaviour is expected, because the words are meaningless or weightless.

The algorithm has been tested with a small matrix made by hand and with that the results have been successful with that example made on purpose.

Conclusions/summary including the discussion of main limitations

This tool is useless if you are comparing, for instance, keyword lists which contains stopwords, because it generates a huge amount of frequent itemset which collapses the application.

Also the implementation should be improved to get more than the 10 first keywords from the list, but in that case it would be a problem with the memory heap space.

We assumed that it is more useful to use the most relevant keywords respecting the principle: *"It is more important that a Word appears 1 time in 1000 different comments than 1000 times in the same comment."*

Its behaviour depends directly on the product keywords, and depending on that the results can be useless sometimes and useful in other cases.

References

Spellchecking:

- levenstein:

http://en.wikipedia.org/wiki/Levenshtein_distance

http://en.wikipedia.org/wiki/Damerau%E2%80%93Levenshtein_distance

- dice:

http://en.wikipedia.org/wiki/Dice%27s_coefficient

For the **GUI**: <http://download.oracle.com/javase/tutorial/uiswing>

For the **Association Rules Module**:

Lecture 4 – Association slides used in the lectures during the course.

Repository: <http://github.com/2ID25work/amazon>