Data Warehousing and Data Mining Project

Free University of Bolzano
Faculty of Computer Science

Ondrej Platek and Peteris Nikiforovs

Semester project

Lecturer: M.Sc. Johann Gamper, Ph.D.
Exerciser: M.Sc. Mateusz Pawlik

Autumn semester 2011

Evaluation: Every milestone is graded on a scale of 0 - 30. 25 points is the expected minimum. Project will be evaluated as average of all milestones. Some points will be added/subtracted from the average after the final presentation.

Contents

1. Milestone 3

1.1 Content of Milestone

The main purpose of this milestone is to try out various data mining techniques.

We decided to explore the following data mining techniques in this milestone: classification and prediction. For each technique we evaluated various algorithms described in the lectures and compared them to some others that were available to use in the data mining software packages that we used.

We also extracted some new information using the models during the mining process. We will present the interesting foundations further on which is the goal of the MSR 2012 challange.

Note that we run similar experiments for classification twice. In Section 1.4 we present experiments run in Rapid Miner and Appendix **??** describes how to run the experiments in Weka.

1.2 Data

We chose to mine the Android bug database because the data was already provided by the organizers of the MSR 2012 challange. More specifically, we chose to focus on the bug reports in this database as we believe it is the most important part of the entire database and it should give the most interesting results. The database contains 20169 reported bugs (up to December 3, 2011).

Description of the data

Here is the available data about the bugs.

- Bug ID - unique identifier assigned to each bug
- Title - short description of the bug
- Description - detailed description of the bug without a predefined structure
- Type - type of the bug: Defect (bug report), Enhancement (feature request)
- Status - current status of the bug (Reviewed, New, Duplicate, Declined, NeedsInfo, FutureRelease, Released, Spam, Unreproducible, Question, WorkingAs-Intended, Assigned, Unassigned, UserError)
- Priority - urgency of the bug to be fixed (Small, Medium, High, Critical, Blocker)
- Component - category of the bug based on which part of the system the bug affects
- Stars - number of people that have starred this bug (could be an indicator of popularity)
- Owner - assigned developer to fix the bug

ExampleSet (20169 examples, 1 special attribute, 11 regular attributes)

Role	Name	Type	Statistics	Range	Missings
id	ID	integer	avg = 10107.437 +/- 5841	[1.000 ; 20254.000]	0
regular	Title	text	mode = Q (239), least = N	Q (239), CanQ (110), canQ (22), "Google T	6
regular	Status	polynominal	mode = New (12190), lea	FutureRelease (731), New (12190), Declin	0
regular	Assigned Developer	polynominal	mode = caryand@gtempa	caryand@gtempaccountcom (331), e@goc	17623
regular	ClosedDate	date	length = 1424 days	[Jan 8, 2008 ; Dec 3, 2011]	13824
regular	Type	polynominal	mode = Defect (16118), le	Defect (16118), Enhancement (4050)	1
regular	Priority	polynominal	mode = Medium (20052),	Medium (20052), Critical (10), High (48), Sl	1
regular	Component	polynominal	mode = Applications (895	Framework (689), Tools (782), User (31), D	15438
regular	Stars	integer	avg = 13.635 +/- 135.368	[0.000 ; 8121.000]	0
regular	Reported By	polynominal	mode = dfils@gmailcom	dfils@gmailcom (232), Chris@gmailcom (	0
regular	Opened Date	date	length = 1410 days	[Nov 12, 2007 ; Sep 22, 2011]	0
regular	Description	text	mode = NOTE This form i	NOTE This form is only for reporting user b	0

Figure 1.1: Defined types for imported data in Rapid Miner

- Reported By - person who reported the bug
- Opened Date - when the bug was reported
- Closed On - when the bug was marked as closed

It is also possible to aggregate the comments for a bug to determine additional data about it.

- Comment count - the number of comments for this bug
- Number of commentators - the number of distinct people participating in the discussion about the bug
- How long it took the solve the bug - by subtracting the close date from the open date

Title, description and type are entered by the user. System also automatically generates the bug id, marks it as a new bug, assigns a medium priority to the bug, makes the user as the reporter of the bug and sets the opened date to the date and time of the submission.

Component and owner are later manually assigned by one of the administrators. Status, priority can later be changed, however, these changes are not reflected in the available data, only the latest state of the bug is available.

Processing of the data

The bug database which was provided as a single XML file was converted to a CSV file using a manually written script which contains all bug properties, previously mentioned aggregated properties but doesn't contain any of the bug comments.

This CSV file was later imported in the data mining software that we used and each property was assigned a type as seen in the screenshot below.

Properties of type *text* will be later converted to vectors using the text processing features of the data mining software.

1.3 Tools

To carry out data mining we used two different open source data mining tools: Weka and Rapid Miner. The first tool recommended to us was Weka, but we encountered problems while using it. It prompted us to look for alternative solutions. Luckily, we came across Rapid Miner which we believe is much superior software in terms of usability and capabilities for data mining.

1.3.1 Weka

At first we used Weka to familiarize ourselves with the data mining process and algorithms. It was recommended to us by fellow students and it was also introduced to us in a lab in the Computer Linguistics course here at the university.

Weka has a nice and simple UI that can be used for simple data mining tasks. Unfortunately, Weka GUI requires a lot of memory and contains a few bugs which often made it crash while running data mining tasks with large amounts of data. Fortunately, this problem could be solved by using a command line interface and the command line parameters are displayed everywhere in the UI and can be easily copy and pasted.

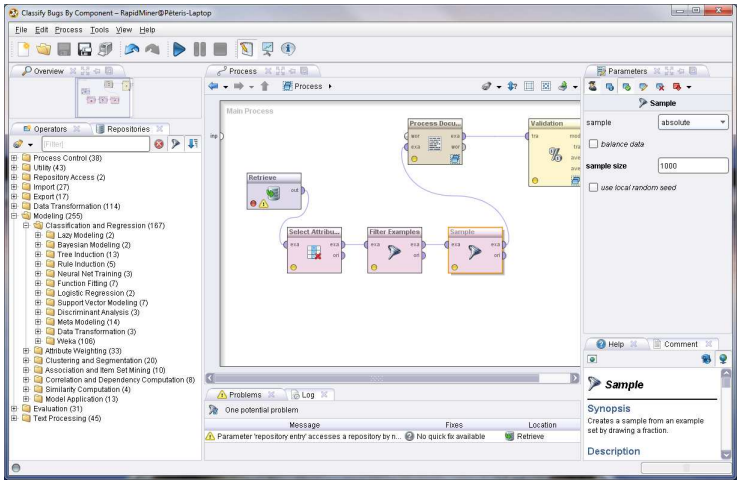
It's a great tool to get started with since it has a very steep learning curve but it wasn't useful when mining large amounts of data.

1.3.2 Rapid Miner

Our preferred data mining tool was Rapid Miner because it offers a very friendly UI and it has many more implementations of data mining algorithms and other capabilities than Weka. In fact, Weka can even be integrated into Rapid Miner. Rapid Miner also offers excellent text processing tools that were available and easy to use which we found immensely useful. To top it all off, it can take advantage of multi-core processors and run computations in parallel.

Rapid Miner provides a workflow model where each process can be dragged and dropped on a canvas and then connected using arrows. It helps visualize the entire data mining task and provides a quick overview of the task at a glance. It's also worth mentioning that the interface works flawlessy, general help as well as contextual help is available at every step and it even has break points like most programming IDEs which allows to inspect intermediate results during a data mining process. It has excellent data import tools as well as reporting and visualization tools.

The only downside that we can see is that at first it's confusing due to the huge amount of features and complexity. However, getting started guides are readily available and there are video tutorials on the website which show basic usage and helps understand the usage patterns of the tool (perhaps the developers used data mining to analyze user actions and develop these tutorials). After watching the videos, Rapid Miner is a treat to use, it's like Eclipse in this regard, confusing at first but great to use after getting to know it.



1.4 Classification

1.4.1 Classification of bugs by component

Task. When a bug report is submitted, someone has to look at it and determine which component it affects and assigned an appropriate label. Currently it is done manually by one of the maintainers. We wondered if it's possible to use data mining to automatically classify new bugs by component.

Attributes used. After inspecting the available data, we came to the conclusion that only title and description should be used to determine the component since they describe the bug but other properties are just metadata. The outcome of the classifier should be the component name that this bug affects.

Training data and classes. We need training data which should already be classified. Only the bugs which already had a component label assigned to them were chosen as the training data for data mining because the bugs with a component assigned to them are already classified and we didn't have to do it manually. There are about 10 different components used in the data, for example, Dalvik, System, Browser, Applications.

Text processing. Classification algorithms can only work with numbers, not text, so it was necessary to preprocess text and convert it to vectors that algorithms can understand. Title and description values were transformed into lower case and then tokenized using non letters as delimiters and then word vectors were generated from them using TF-IDF (term frequency–inverse document frequency - used to evaluate how important a word is to a document in a collection or corpus). Each word then became an attribute with the TF-IDF value. Other vector creation schemas were trialed (term frequency, term occurences) but they showed poor results and it was concluded that TF-IDF is the best choice. It's worth pointing out that if a more sophisticated tokenization algorithm was used which could recognize Java code snippets and tokenize namespaces, class names, method names etc., perhaps it would significantly improve the accuracy of classifiers.

Performance. After the data was pre-processed, we chose various classifying algorithms that were learned about in class and compared their accuracy. Accuracy is used to assess the performance of classifiers. It was mentioned in the

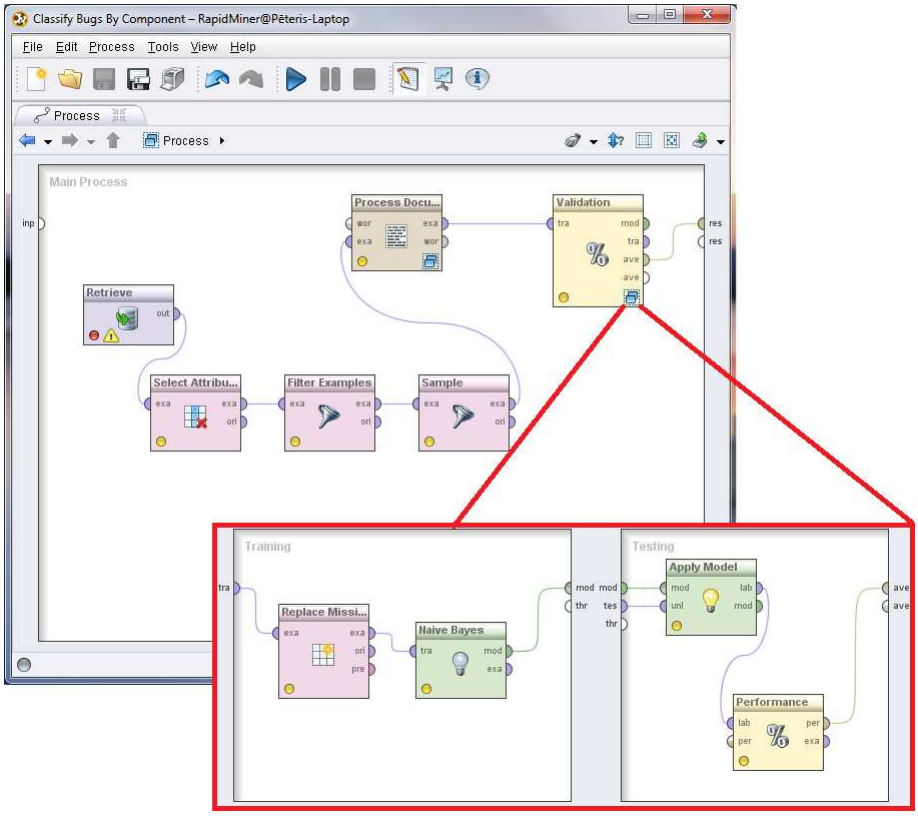


Figure 1.2: Workflow in Rapid Miner: 1. Load data 2. Select attributes 3. Use only labeled data 4. Use a sample 5. Text processing 6. Cross Validation

lectures that a stratified 10-fold cross validation is recommended for estimating accuracy. Because we wanted to try out various algorithms and compare them, we only used a sample of data because it would take too much time to the classifiers to run on our desktop computers.

The graphical representation of the processes of this data mining task can be seen in the following screenshot of Rapid Miner.

Obtained results are displayed in the following table. Type is the class of algorithms, algorithm is a specific classification algorithm (Weka means that it wasn't a Rapid Miner implementation of the algorithm), sample is the number of classified bugs used, time is how long it took to run, accuracy is the stratified 10-fold cross validation result. Starred times indicate that they were run on a different computer.

Type	Algorithm	Sample Size	Time	Accuracy
Decision Trees	Weka J48	500	2:19	44.80%
		1000	10:25	45.50%
Decision Trees	Weka LAD-Tree	500	13:40	39.80%
Bayes Classifiers	Naive Bayes	500	0:08	37.80%
		1000	0:20	39.30%
		2000	0:56	42.40%
		4000	2:53	43.32%
Bayes Classifiers	Bayes Network	500	0:18	40.60%
		1000	1:11	42.50%
		2000	4:06	50.05%
		4000	43:41*	54.45%
Lazy Classifiers	k-NN (k=1)	500	0:09	41.60%
		1000	0:39	40.80%
		2000	3:08	46.65%
		4000	11:14*	49.07%
Lazy Classifiers	k-NN (k=15)	500	0:11	45.40%
		1000	0:38	49.80%
		2000	3:13	55.60%
		4000	11:18*	58.02%
Lazy Classifiers	k-NN (k=30)	500	0:10	45.00%
		1000	0:35	49.20%
		2000	3:01	54.60%
		4000	11:15*	57.98%
Lazy Classifiers	k-NN (k=60)	500	0:08	42.20%
		1000	0:38	46.70%
		2000	3:29	52.50%
		4000	11:21*	56.10%
Support Vector Classifiers	Weka SMO	500	0:53	41.40%
		1000	4:15	47.20%

Conclusion. As can be seen, it is possible to automatically classify bugs by their content more than half of the time. Most of the trialed algorithms could easily reach the 40% accuracy barrier, however, the performance between these algorithms varies. While decision trees provide an acceptable although below average accuracy compared to all other algorithms, their performance is very poor. Bayes Network was the first algorithm to cross the 50% line but it was beaten by k-NN which not only performs much faster but also gives the best accuracy. Therefore the k-NN algorithm was chosen to be run on the whole dataset to find out the best possible accuracy. It was mentioned in the lectures that the k-value should be in tens which clearly holds true in our experiments.

When the k-NN (k=15) was run on the whole dataset (4731 bugs), its accuracy was 58.55%. If a more sophisticated tokenizer was developed and used, it's our belief that the accuracy could be higher, perhaps even as high as 70%. Such accuracy is acceptable although leaves something to be desired and the model could be used to automatically assign the appropriate component when a bug is reported and the maintainers would only need to change it 40% of the time.

1.5 Prediction

1.5.1 Prediction of bug lifespan

Task. The lifespan of a bug is from the time it was reported until it was marked as closed. Essentially it means that is how much time it took to solve the bug. We wondered if it's at all possible to predict how long it will take to solve a bug by looking at its contents and metadata. The idea is that the longer and more complicated the description of the bug, the longer it would take to solve it.

Attributes used. After inspecting the available data, we came to the conclusion that in addition to title and description other attributes such as priority, assigned developer, component, stars (indicates popularity) and type should be used. Status does not seem to make sense in this context since it is changed after a bug is closed.

Training data. Training data consists of all those bugs that have been closed so that it's possible to calculate the lifespan by substracting the opened date from the closed date. The prediction outcome should be the lifespan as a timespan in milliseconds of a bug.

Text processing. Text processing was done exactly like described in the previous section.

Performance. Error rates are used to estimate accuracy of predictors. We are using a small sample size and only 3-fold stratified cross validation because the predictors take a long time to run. Error rates are given in milliseconds as the root of mean squared error.

Algorithm	Sample Size	Error Rate
Linear Regression	50	15809561072.876 (183 days) +/- 1913382829.648 (22 days)
	80	23016235402.190 (266 days) +/- 9169804778.729 (106 days)
Polynomial Regression	50	12954847042.251 (150 days) +/- 5225235366.410 (60.5 days)
	80	15726486894.414 (182 days) +/- 8349272014.051 (97 days)

Conclusion. It took a lot of time for these data mining tasks to run so we were able to test them only on a small sample size. The obtained results were not very convincing nor useful.

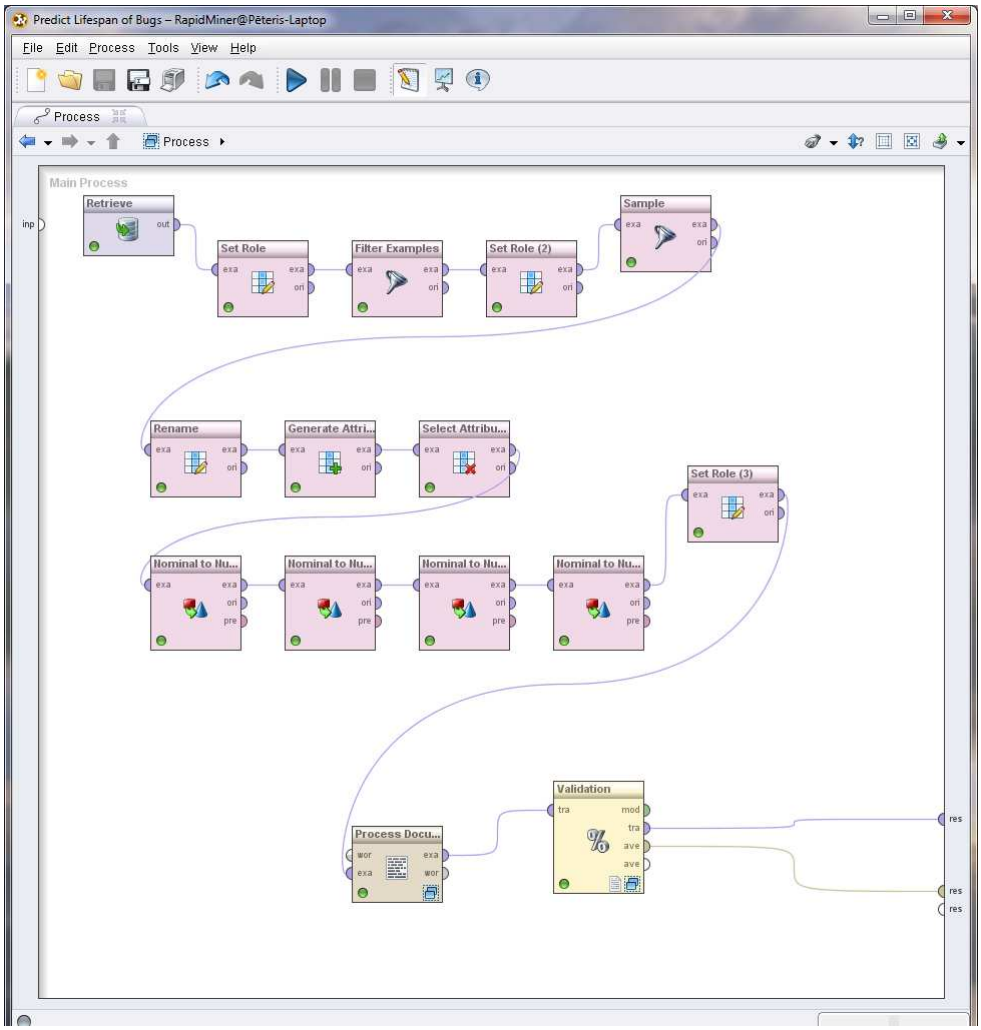


Figure 1.3: Workflow in Rapid Miner: 1. Load data 2-4. Filter non closed bugs 5. Use a sample 6-7. Calculate new Lifespan attribute 8. Select attributes 9-12. Convert polynominal to numeric attributes 13. Mark Lifespan attribute as label 14. Text processing 15. Cross validation