Semester Thesis

Adaptive Weighting of Filters in Spamato

Supervising Professor: Prof. Dr. Roger Wattenhofer
Supervising Assistant: Keno Albrecht

Summer Semester 2006

Dennis Rietmann,
ETH Zürich

October 20, 2006

Table of Contents

1 Introduction 1
2 The DecisionMaker Evaluator 2
2.1 Concepts 2
2.2 User Guide 2
2.2.1 The main window 3
2.2.2 Mailset creation and management 4
2.2.3 Integration and configuration of additional DecisionMakers 5
2.2.4 Test run execution 5
2.2.5 Analysis of a single test run 6
2.2.6 Comparison of multiple test runs 7
2.2.7 Advanced topics 8
2.2.7.1 Implementing custom Metric Providers 8
2.2.7.2 Implementing custom Property Providers 9
3 The AdaptiveSpam DecisionMaker 11
3.1 Concepts 11
3.2 The Weighted Majority Algorithm 11
3.3 Implementing additional strategies 12
4 Future Work 14
5 Conclusions 14
6 References 14
7 Glossary 15
8 Appendices 16
8.1 Required plug-ins 16
8.2 DME folder structure 16

1 Introduction

The goal of this semester thesis was to design and implement a DecisionMaker which is smarter than the current MinSpam DecisionMaker. The MinSpam DecisionMaker decides for spam if a certain fixed number of filters voted for Spam. Despite its simplicity, it works reasonably well for the currently implemented filters. The reason for this is that the false positive and false negative rates of the current filters are really low. Therefore if a filter votes for spam or ham, which means the FilterResult is not of type *UNKNOWN*, there is a high probability that the filter is right.
So why is there a need for another DecisionMaker? First of all, we hope that we can do even better. Second, in the future it is planned to implement a bunch of lightweight, simple spam rules which act as filters, but which are much less reliable than the current full-grown filters like the *Bayesianato* or the *Comha* filter. Each of these rules looks for specific characteristics or patterns often found in spam mails. But considered in isolation, each rule only gives a clue that a mail is potentially spam. What gives them power is their combination. Examples of such rules can be found in the *SpamAssassin* [1] spam filter project. It can be clearly seen that in such a scenario the current MinSpam DecisionMaker will no longer work reliable, since the amount of filters voting for spam varies in a greater range and is less stable. Therefore it would be impossible to find a fixed limit from where on a mail is considered as spam. What is needed is a more flexible and dynamic DecisionMaker.
This is where the idea of the adaptive weighting of filters might help. The DecisionMaker should somehow weight the FilterResults of the involved filters based on how good they performed in the past. For instance if a filter is very accurate its influence over the final decision will grow over time. But even a filter with a good performance fails sometimes, for example it can be fooled by spammers with a specific type of mail. If in such a case enough of the other filters, like the rule-based ones described above, vote for spam, the aggregated result can still be correct even though they have an individual lower influence. Developing such a DecisionMaker is one goal of this thesis.
But once this new DecisionMaker has been developed how should we measure its performance? How can we compare it with the existing DecisionMaker or compare the same DecisionMaker with different settings? Of course comparison could be done manually, but it would be a very tedious and error-prone task.
These demands lead to the next goal of this thesis: The design and implementation of an evaluation framework for different DecisionMakers. It should be possible to compare different DecisionMakers, as well as analyzing an individual one in an appropriate manner. To facilitate its usage it should be quick and easy to use but still remain extendible. Furthermore it should also assist the developer during development of a new DecisionMaker. For example the developer should be able to monitor the interesting parts of the state of the DecisionMaker to check whether it is working as expected.

2 The DecisionMaker Evaluator

This section describes the *DecisionMaker Evaluator* (*DME* for short) which represents the above mentioned evaluation framework for DecisionMakers. The introduction of the evaluation concepts used in the tool is followed by a user guide which gives a hands-on explanation on how to use its functions.
As SPAMATO itself, the *DME* is also written in Java and it uses the *Eclipse Standard Widget Toolkit* (*SWT*) [2] to create the user interface. Some data can be visualized in a chart form. To create charts, the *JFreeChart* chart library [3] has been used. Because the *DME* is not integrated as a plug-in into SPAMATO, it runs as a standalone application. To gain access to SPAMATO, it sets up its own SPAMATO instance by using the `SpamatoFactory` class. For a list of plug-ins that need to be present at minimum see Appendix 8.1. For a description of the folder structure see Appendix 8.2.

2.1 Concepts

The *DME* can be helpful in the following two application areas:

1. Assist the developer during implementation of a new DecisionMaker
2. Compare different DecisionMakers

For both purposes at least one *mailset* is required. A mailset consists of an arbitrary number of mails together with a correct classification (ham or spam) for each mail. Such a classification has to be made manually by the user. We decided that mailset creation could be handled with every regular mail client for which a SPAMATO extension exists (*Mozilla Thunderbird* was used during development), instead of creating a custom solution. For this purpose, a small SPAMATO plug-in, called the *DME Initialiser*, has been developed. It needs to run in the SPAMATO instance used by the mail client which is used for mailset creation (see Section 2.2.2 for information on how to use it). Mailset creation involves selecting the mails which should be part of the mailset and defining for each mail whether it is spam or ham via SPAMATO's report/revoke functionality.

The second important concept is the *test run*. A test run is always executed against a specific mailset and uses a specific DecisionMaker. During test run execution, each mail in the mailset is fed as input to the DecisionMaker under test. The test run object records for each mail which decision has been made by the DecisionMaker. This is essentially what we need for later analysis.
To compare different DecisionMakers, we can now simply execute a test run for each DecisionMaker on the same mailset and compare the total number of correct decisions, false positives, false negatives etc. or compare how these metrics evolve over time. It is even possible to implement custom metrics (see Section 2.2.7.1 in the user guide). By using different mailsets, we can analyze how the DecisionMakers perform on different types of mails.
During development of a new DecisionMaker, the tool can be used to easily check whether the DecisionMaker works correctly. By use of customized Property Providers (see Section 2.2.7.2 in the user guide), developers are able to monitor the relevant parts of the internal state of the DecisionMaker under test. Of course, it is also useful for tuning and tweaking a DecisionMaker, like comparing how the same DecisionMaker performed with different settings.

2.2 User Guide

After a presentation of the main window, the tasks which are typically part of *DME* usage are described in more detail. Since the user guide is written in a task-oriented manner, it can also serve as a reference.

2.2.1 The main window

In the main window, the test runs can be maintained and additional information can be retrieved. It is also the starting point for most other tasks, which can be launched via the buttons on the toolbar or separate controls, as described in the following sections. For illustration purposes, a screenshot is provided in Figure 1 below.

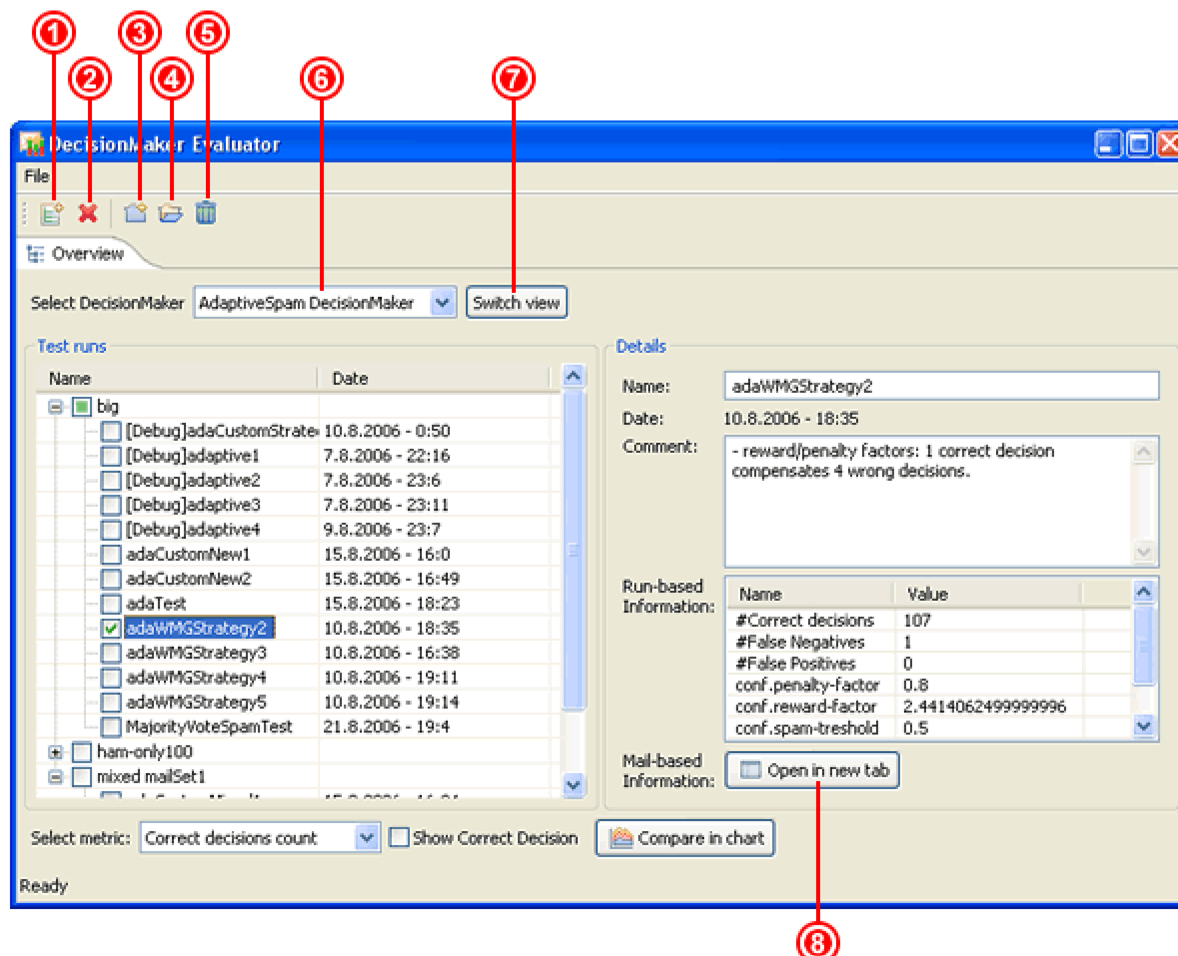


Figure 1 - The main window showing the DecisionMaker-centric view

The window is vertically split into two areas. On the left side, an overview of already executed test runs is given. Test runs can be browsed in two different ways:

- In the *mailset-centric* view the user can select a mailset in the combo box at the top (6 in Figure 1), and only test runs which used the selected mailset are displayed in the tree view below. This view is also displayed by default on startup.
- In the *DecisionMaker-centric* view, as shown in Figure 1, a DecisionMaker can be selected in the combo box (6 in Figure 1) and only the runs which used the selected DecisionMaker are displayed in the tree view.

To switch between views, use the *Switch view* button (7). For convenience, the test runs are grouped by DecisionMaker in the *mailset-centric* view and by mailset in the *DecisionMaker-centric* view in the tree navigation.
Test runs can be deleted by checking them in the tree view and pressing the corresponding button in the toolbar (2).
Additional information is available for each test run on the right side in the *Details* section. It can be displayed by clicking on a test run. This information consists of the name of the test run, the date when it was executed, a comment, and the run-based properties, which are displayed in a table. The name and comment fields are fully editable and changes are saved instantly. The mail-based information, like the metrics and the mail-based properties, can be

displayed in a separate tab by clicking on the *Open in new tab* button (8) (more detailed information is available in Section 2.2.5).
The controls at the bottom are used to compare multiple DecisionMakers and are explained in Section 2.2.6.

2.2.2 Mailset creation and management

As seen in Chapter 2.1, mailsets are a vital part of the *DME* since no test runs can be executed without at least one mailset, and for meaningful results carefully created mailsets are from uttermost importance.
Mailset creation consists of two parts: First, Selecting the mails which should be part of the mailset and second, importing the mailset file into the *DME* to make it available for test run execution. To facilitate the first task, a small SPAMATO plug-in, called *DME Initialiser*, has been created. When this plug-in is installed and activated, a regular mail client running SPAMATO can be used to create the mailset file. It contains the selected mails together with their correct classifications.

Create a new mailset

1. Start up the mail client and make sure that every mail, which will be part of the mailset, has been checked by SPAMATO and the corresponding ActivityResult still exists.
2. To add mails, the plug-in re-uses the report/revoke mechanism of SPAMATO
   a. To add a spam mail, simply *report* it
   b. To add a ham mail, simply *revoke* it
3. Once all mails have been added, close the mail client. A dialog will pop up where you can specify the location where the mailset file should be saved to. After confirmation, the mailset file will be created in the designated location and depending on its size, a progress bar may appear to indicate the current progress. The storage location can be configured through the WebConfig (initially it is set to `C:\dme_initialiser\`).

Import a mailset in the DME

1. Launch the *Add Mailset Wizard* by clicking on the corresponding button on the toolbar (3 in Figure 1).
2. Then select the mailset file, which has been created by the *DME Initialiser*. Currently there is a restriction that it needs to be located in the *DME* base directory. Therefore either move it to this location manually or, for more convenience, set the path in the *DME Initialiser* accordingly (so that it will be directly saved to this directory). Configuration is handled through the WebConfig.
3. Enter a name for the mailset, which will be displayed in the GUI (mandatory).
4. Optionally the mailset can also carry a description. This is an ideal place to characterize the mails which are contained in the mailset or add other information which could be helpful to distinguish this mailset from others.
5. When all information has been entered, press *Finish* and the mailset is ready to use in the *DME*.

Mailset Management

To delete a mailset simply select it in the combo box (6 in Figure 1) and press the corresponding button on the toolbar (5 in Figure 1).
To browse the available mailsets, the mailset browser view has been provided. It is accessible via the toolbar (button 4 in Figure 1) and opens in a new tab. A screenshot is shown in Figure 2.
The following information can be accessed In the Mailset Browser view:

- In the *Mail Set Details* section, all available meta-data, like the name and the description of the selected mailset, is shown.

- In the *Mail Details* section, the individual mails contained in the selected mailset can be inspected. Besides the information about the mail itself, like the body and subject text, some statistics are displayed. They include the correct decision as specified by the user during mailset creation and the individual filter decisions, which have been created by SPAMATO.

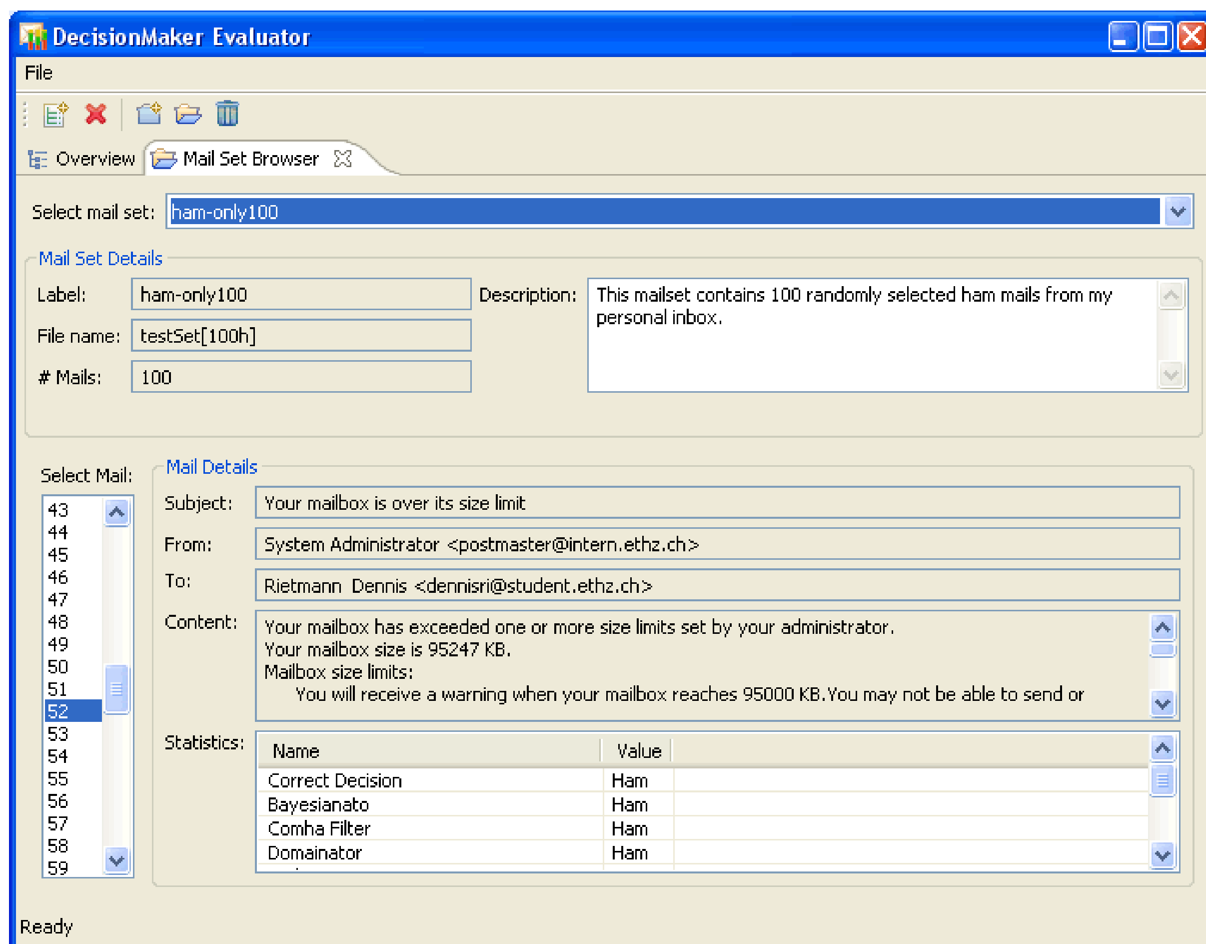


Figure 2 – The mailset browser view

2.2.3 Integration and configuration of additional DecisionMakers

Since the *DME* uses a stripped down SPAMATO system, additional DecisionMakers can be installed exactly the same way as in a regular SPAMATO system: Simply copy its plug-in folder (which contains the `plugin.jar` and `plugin.zip`) as a subfolder into `default_plugins\decisions` and add an entry to the `versions.txt` file. The next time the *DME* is started, the DecisionMaker should be installed (watch the `error_log.txt` located in the `userHome` directory for error messages). Before executing a test run, make sure the DecisionMaker is properly configured e.g. via WebConfig (accessible via *File – Open WebConfig* or open it directly at `http://localhost:8574/`). If you are testing your own DecisionMaker you might be interested in creating custom Property Providers for your DecisionMaker. See Section 2.2.7.2 for more information about what they can do for you and how to implement them.

2.2.4 Test run execution

Prerequisites:
At this point you should have the mailset(s) you want to use imported into the *DME* (see *2.2.2 Mailset creation and management*) and the DecisionMaker which should check the mails should be installed an properly configured (see *2.2.3 Integration and configuration of additional DecisionMakers*).

The *TestRun wizard* guides you through the test run execution and it can be started from the toolbar (1 in Figure 1).
The first page of the wizard is then displayed. Enter a name for the test run in the corresponding field. The comment field holds additional information about the test run provided by the user but can also be left blank. It might be helpful to characterize the test run, describe the goal(s) of it, or mention any special settings (if not captured by Property Providers). This information can also be edited later in the overview tab. In the list of DecisionMakers select the one you would like to test and press *Next*.

On the next page select one or more mailsets from the list; a test run will be created for every selected mailset. Once you are done, press *Finish*. The test runs are then executed asynchronously in the background and the current progress is indicated in the status bar in the lower left corner of the window (see Figure 3). Once a test run has been executed, it is accessible from the GUI.

Figure 3 – Test run execution status indicted the status bar

2.2.5 Analysis of a single test run

Analyzing a single test run takes place in the metrics view. To open the metrics view, select a test run in the tree view and click on the *Open in new tab* button (8 in Figure 1).
The table, depicted in Figure 4, lists all the available run-based information for the selected test run. This consists of the metrics and the mail-based properties, whose values are displayed in a row each. A value in a column represents either a metric value for the mail indicated in the header or a property value which has been generated after the DecisionMaker has been called with this mail. The run-based information can be viewed in the overview tab (see Section 2.2.1). Note that only the metrics of the Metric Providers which are registered for the DecisionMaker used during the test run execution are displayed. The same also goes for the Property Providers, for instance the filter-weight Property Provider is only applicable to runs which used the AdaptiveSpam DecisionMaker.

DecisionMaker Evaluator

File

Overview | Metrics (adaWMGStrategy2)

Metric	Mail #0	Mail #1	Mail #2	Mail #3	Mail #4	Mail #5	Mail #6	Mail #7	Mail #8	Mail #9	Mail #10	Ma
Correct decisions count	1	1	2	3	4	5	6	7	8	9	10	11
Decision (Decision Maker)	Spam	Ham	Spam	Spam	Spam	Spam	Spam	Spam	Spam	Spam	Spam	Spa
False Negatives Count	0	1	1	1	1	1	1	1	1	1	1	1
False Positives Count	0	0	0	0	0	0	0	0	0	0	0	0
filters.bayesianato.weight	1.0	1.0	2.44141	2.44141	2.44141	2.44141	2.44141	2.44141	2.44141	2.44141	2.44141	2.4
filters.comha.weight	1.0	1.0	2.44141	2.44141	2.44141	2.44141	2.44141	2.44141	2.44141	2.44141	2.44141	2.4
filters.domainator.weight	1.0	1.0	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8
filters.earlgrey.weight	1.0	1.0	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8
filters.razor.weight	1.0	1.0	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8
spam_subjects.weight	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

Mail preview

From:

Subject:

Content:

Export (CSV) | Show selected metric(s) as chart | (Multiple selections possible)

Ready

Figure 4 – The Metrics View

For easier comparison, it is possible to display numeric data in a chart view. To do this, simply select the desired row(s) in the table (use the *Ctrl*- and *Shift* keys to select multiple rows) and press the *show selected metric(s) as chart* button. The chart is then opened in a new tab where the x-axis represents the mails and the y-axis the values of the selected metrics. For more information about what you can do in the chart view, see Section 2.2.6.
If you would like to export the data for processing with an external application, it is possible to export it into the *CSV* (*Comma Separated Values*) format. The export function can be accessed via the *Export (CSV)* button. The exported data includes the run- and mail-based information.
To display a mail in the *Mail preview* section, click on a table column header or use the buttons at the bottom of the table.

2.2.6 Comparison of multiple test runs

The *DME* offers the functionality to compare the mail-based information of several test runs. For this purpose, the values of a selected metric or mail-based property can be visualized in a chart for several test runs at once, which means that test runs can be compared on a “per metric” basis.
Proceed as follows to open the chart view:

1. Select the desired runs in the tree view by checking their corresponding checkboxes.
2. Choose a metric for comparison in the combo box at the bottom (1 in Figure 5). Note that only numeric metrics are valid. The entries in the combo box are dynamically adapted to the current selection. For example, when selecting runs from different DecisionMakers and a Metric Provider can only create the metric for one of them, it is not displayed.
3. Optionally the correct decision can also be displayed for each mail (2 in Figure 5). In the chart, a value of 0 indicates ham and a value of 1 indicates spam.
4. Press the *Compare in chart* button to display the chart.

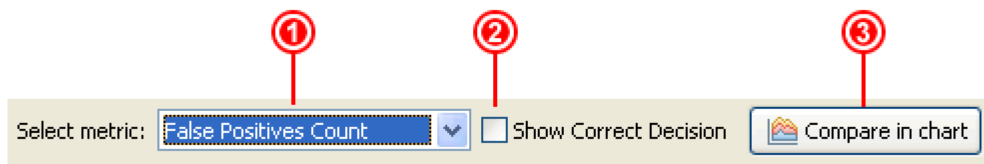


Figure 5 – Controls for test run comparison

A screenshot of the chart view is shown in Figure 6. The x-axis represents the mails and the y-axis displays the values of the metric or run-based property. Each label at the bottom corresponds to a series displayed in the chart and denotes the name of the test run with the metric being displayed in brackets.
The following operations are supported in the interactive chart view (this also applies when comparing several metrics of a single DecisionMaker as described in Section 2.2.5):

- To enlarge an area, simply drag a rectangle over it. This step can be repeated until a satisfactory magnification is reached.
- The chart can be customized by right-clicking on it and selecting *Properties...*.
- To export the chart as image in the *PNG (Portable Network Graphics)* format, press the *export as image (png)* button and specify the location where it should be saved.
- To print the chart, right-click on it and select *Print...*.

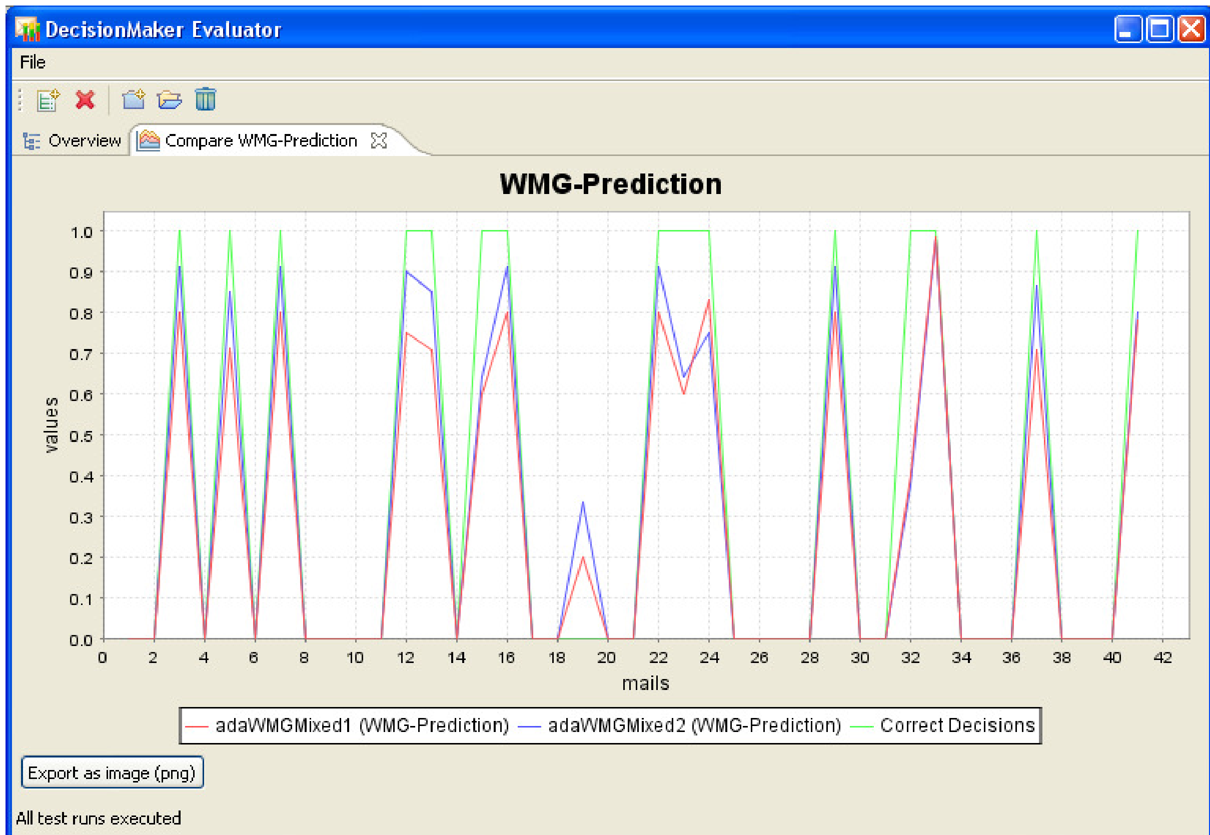


Figure 6 – The chart view showing a mail-based property for two test runs together with the correct decisions

2.2.7 Advanced topics

This section describes how the *DME* can be extended by writing own code. It is assumed that the reader is already familiar with SPAMATO.

2.2.7.1 Implementing custom Metric Providers

Each Metric Provider is represented by its own class which must implement the `IMetricProvider` interface. Metric Providers are grouped in the `metricProviders` package. There are two approaches to implement a Metric Provider: Either directly implement the `IMetricProvider` interface in your class or inherit from the `AbstractMetricProvider` class. Both ways are described in the following.

The `AbstractMetricProvider` has been created for convenient subclassing. It provides a default implementation for all the methods from `IMetricProvider` and there is only one abstract method `computeMetricValue(…)` which has to be implemented by concrete subclasses.
The `AbstractMetricProvider` has the following characteristics:

- It is able to generate the metric for all DecisionMakers
- It does not use additional provider-specific data set via `setData(…)`
- The methods `setup()` and `finish()` are empty

To implement a minimal Metric Provider by subclassing the `AbstractMetricProvider`, all you have to do is call `setName(…)`, preferably in the constructor, with the name of your Metric Provider and implement the method `computeMetricValue(…)`.

```
protected MetricValue computeMetricValue(TestResult result,
FilterProcessResult FPR)
```

It is called once for each mail in the test run's mailset with its corresponding `TestResult` and `FilterProcessResult` objects and must return the value of the metric for this mail. The return value is of type `MetricValue` which not only encapsulates the value of the metric but also supports colors. The color set via `setColor(RGB color)` is used when displaying the value. A `TestResult` object exists for every mail in the mailset and it holds several data related to a mail together. This includes the mail-based properties, the ID of the mail, and the decision made by the DecisionMaker. The correct decision for this mail can be obtained from the `FilterProcessResult` by calling `FPR.getTotalDecision()`. All this information can be used to compute the metric value.
You are free to customize the `AbstractMetricProvider` by overwriting any of the provided methods with your own implementation.

For full flexibility, the `IMetricProvider` interface can also be implemented directly. Its most important methods are described bellow. For more information have a look at the *Javadoc* and the already existing Metric Providers in the `metricProviders` package.

```
public MetricValue[] computeMetric(TestRun run)
```

The `computeMetric(…)` method is the one which actually creates the metric for the given test run. The returned `MetricValue` array should contain a `MetricValue` object for each mail in the test run's mailset, which can be accessed by calling `run.getMailSet()`. To retrieve all `TestResult` objects, simply call `run.getTestResults()`.

```
public String[] getDecisionMakerNames()
```

The `getDecisionMakerNames()` method must return the names of all DecisionMakers for which the Metric Provider can compute its metric. Return the constant `IMetricProvider.ALL_DECISION_MAKERS` if the metric can be computed for all DecisionMakers.
With the `setData(…)` and `getData(…)` methods it is possible to work with provider-specific data.

To register a new Metric Provider, you have to add a call to `addMetricProvider(IMetricProvider)`, which takes an instance of the new Metric Provider as argument, to the `init()` method of the `DecisionMakerEvaluator` class. If the Metric Provider is not registered, it cannot be used in the *DME*.

2.2.7.2 Implementing custom Property Providers

All Property Providers must implement the `IPropertyProvider` interface and are grouped in the `propertyProviders` package. Since they are bound to a specific DecisionMaker, no abstract base class, which would correspond to the `AbstractMetricProvider`, is provided. As described in the Glossary, the purpose of a Property Provider is to extract specific information, like parts of the state, from a DecisionMaker *during* the test run execution. What to extract is totally up to the developer and is highly dependent on the DecisionMaker being tested. In this sense, Property Providers are more flexible than Metric Providers. Since they are called during test run execution, the downside of them is that new Property Providers only apply to new test runs. Therefore previously executed runs do not contain the new properties. On the other hand, Metric Providers can be installed later on and generate metrics for all applicable test runs, but they do not have access to the DecisionMaker's state during test run execution. In the GUI there is no difference between metrics and mail-based properties and they are treated uniformly.

The most important methods of the `IPropertyProvider` interface are described in the following.
To indicate whether a Property Provider generates *mail-based* or *run-based* properties, simply return the appropriate value in the `isMailBased()` and `isTestRunBased()` methods. The `getDecisionMakerName()` method must return the name of the DecisionMaker for which the Property Provider was created.

```
public Map<String, String> provideProperties(DecisionMaker decisionMaker)
```

In the `provideProperties(…)` method the properties are actually created. In case of a *run-based* Property Provider, it is called once at the end of the test run. Each entry in the returned map then denotes a property (name-value pair). Therefore it is possible to return multiple properties with a single Property Provider. In case of a *mail-based* Property Provider, this method is called whenever a mail has been checked by the DecisionMaker under test. The returned map contains now only one entry (name-value pair) which represents the *mail-based* property and its current value.

To register a new Property Provider, a call to `addPropertyProvider(IPropertyProvider)`, which takes an instance of the Property Provider as argument, has to be added to the `init()` method of the `DecisionMakerEvaluator` class. Property Providers which are not registered cannot be used in the *DME*.

3 The AdaptiveSpam DecisionMaker

This chapter describes the AdaptiveSpam DecisionMaker which adaptively weights the spam filters present in SPAMATO based on their performance in the past. This knowledge is then used to make the spam/ham decision. The AdaptiveSpam DecisionMaker has been implemented as a SPAMATO plug-in and supports the WebConfig.

3.1 Concepts

In general, a DecisionMaker has access to the FilterResult objects of each filter (or sub-filter) which has checked a given mail (through the FilterProcessResult). A FilterResult encapsulates how a filter judges a mail (see Glossary for more information about FilterResults). This is what we need as input for any kind of weighting algorithm; particularly the *result* value is of interest.

Instead of implementing a fixed weighting algorithm, the AdaptiveSpam DecisionMaker uses so called *weighting strategies*. An algorithm in this context is any kind of procedure which defines how the FilterResults of a given mail are weighted and aggregated to a decision, which can be *ham*, *spam,* or *unknown*. This decision represents the final decision of the DecisionMaker and is communicated to the user. For example, if the decision is *spam*, the mail is moved to SPAMATO's spam folder. Each strategy implements a weighting algorithm and uses the infrastructure provided by the AdaptiveSpam DecisionMaker. Through this abstraction, the complexity inside the strategy can be reduced and through its flexible strategies system, the DecisionMaker is not tied to a fixed algorithm anymore. Instead strategies can be freely changed through the WebConfig and new strategies can easily be added in the future. It also offers a lightweight and easy way to quickly try out new algorithms without having to set up a full-fledged DecisionMaker for each of them. This works great in combination with the *DME*. Information on how to implement additional strategies can be found in Section 3.3.
For convenience, the AdaptiveSpam DecisionMaker provides the so called *FilterWeightIndex* to its strategies. This data structure can be used by the strategies to store an assigned weight for each filter. This information is stored persistently on the user's disk. Of course, also custom solutions are possible.
In case the AdaptiveSpam DecisionMaker made a wrong decision, it needs some kind of feedback mechanism. Since it cannot know by itself whether it made a mistake or not, this is achieved by using SPAMATO's report/revoke feature. Therefore the performance of the DecisionMaker also depends on the user. Whenever a mail has been falsely classified, it is expected that he reports or revokes the message. Through this feedback, the DecisionMaker, or more important the strategy, knows that it made a mistake and can react accordingly, for instance adjust the weights to the new situation. If a user consequentially reports and revokes messages, it is expected that the DecisionMaker's performance will increase over time because it is getting more and more adjusted to the types of mails the user receives. This is also the reason why it is called "adaptive." If the user does not report or revoke mails at all, the AdaptiveSpam DecisionMaker still works, but it uses the default weights which might not be optimal for the user. To get notified of these reports or revokes, the DecisionMaker implements the `PostReporter` and `PostRevoker` interfaces and the rest is handled by SPAMATO.

3.2 The Weighted Majority Algorithm

As a prototypic strategy, the generalized *Weighted Majority Algorithm* [4] (*WMG* for short) has been implemented. Other ideas can be found in [5]. In this section, a short overview of this algorithm and its adoption as a strategy for the AdaptiveSpam DecisionMaker is given. The *WMG* which is a so called *compound* or *master* algorithm tries to solve the following problem: Given a pool of prediction algorithms which all try to predict the same thing, how can we aggregate the individual predictions to one final prediction with a minimum error rate?

The following assumptions hold for the *WMG*:

- The predictions of the algorithms in the pool can be in the range [0,1].
- The final prediction must be binary.
- Each algorithm has an assigned weight, which is initially set to 1.

To create the final prediction, the *WMG* proceeds as follows: First it sums up all products of the algorithms predictions with their corresponding weights. To normalize this weighted sum, it is divided by the sum of all weights. If the resulting value γ is greater than ½, *WMG* predicts 1 and 0 when it is less than ½ (if it is equal to ½, either prediction is allowed).
If a mistake occurs, an *update step* is performed. During the *update step*, each weight is multiplied by some factor, which can be different depending on whether the algorithm's prediction was correct or not.
The adoption of the algorithm to a weighting strategy is straightforward: The role of the algorithms is played by the filters, the algorithms predictions are represented by the FilterResults (in particular their *result* value), and the final prediction states whether a given mail is spam ($\gamma > ½$) or ham ($\gamma \leq ½$). For the *update step* two fixed factors are used. For the filters whose decision was correct a factor greater than 1 is used, which means they get promoted and their influence increases. In contrary for the filters which were wrong, a factor less than 1 is used.

3.3 Implementing additional strategies

This section is aimed at developers and explains how additional strategies can be implemented.
The involved classes are shown in the following diagram.

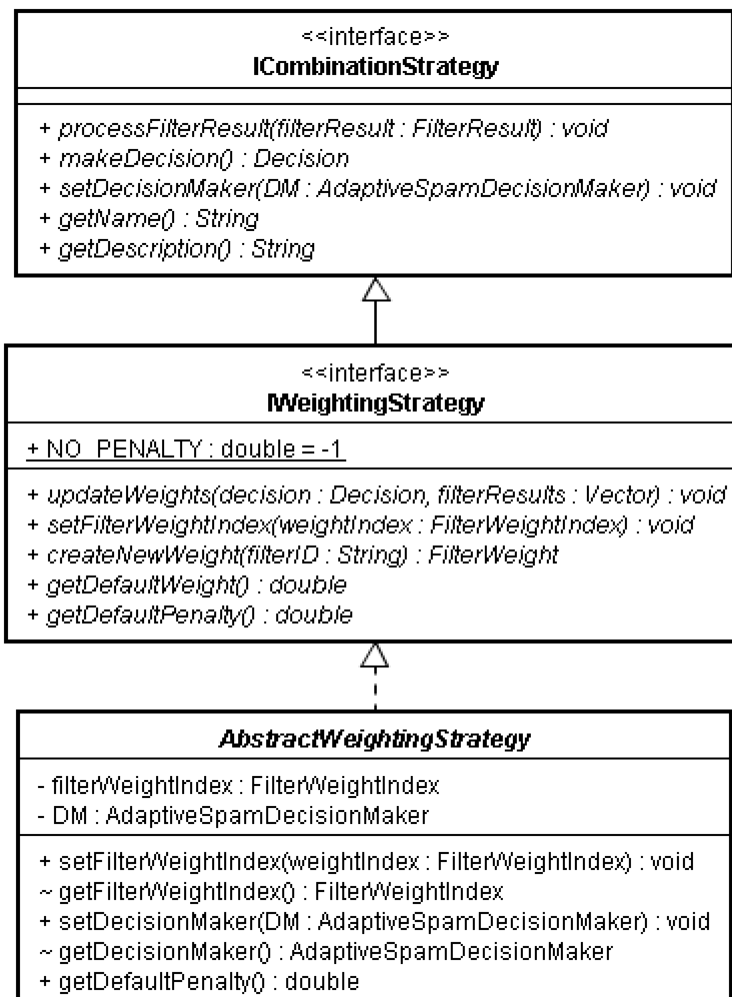


Figure 7 – Strategy classes

The `ICombinationStrategy` interface at the top of the hierarchy represents a very general interface for any kind of strategy. Its two most important methods are `processFilterResult(…)` and `makeDecision(…)`.

```
void processFilterResult(FilterResult filterResult)
```

The AdaptiveSpam DecisionMaker calls `processFilterResult(…)` with each FilterResult and thereby allows the strategy to examine the FilterResults and do some internal work, like perform computations which are needed to create the decision.

```
Decision makeDecision()
```

Once the strategy examined all FilterResults, the AdaptiveSpam DecisionMaker calls `makeDecision()` which requires the strategy to create and return its final decision.

Every strategy in the AdaptiveSpam DecisionMaker must implement the `IWeightingStrategy` interface. It extends the `ICombinationStrategy` and adds a bunch of methods which are specific to weighting strategies.

```
void updateWeights(Decision decision, Vector<FilterResult> filterResults)
```

If the strategy makes a wrong decision and the user reports or revokes the mail, the DecisionMaker calls `updateWeights(…)` on the strategy. This gives the strategy the opportunity to react to this new situation and update the weights accordingly, in order to increase its performance in the future.

```
FilterWeight createNewWeight(String filterID)
```

If the strategy requests a weight, which does not exist yet, the *FilterWeightIndex* calls `createNewWeight(…)` on the strategy. The strategy then creates and initializes a weight for the filter with the given *ID* and returns it. Therefore this method is called on the first start, where no weights exist at all and during regular operation, whenever a new filter is installed. Its purpose is to give the strategy a way of defining how new weights should be initialized.

In most cases, it might be suitable to subclass the `AbstractWeightingStrategy` instead of directly implementing the `IWeightingStrategy` interface.
Once the strategy has been implemented, it needs to be registered to the DecisionMaker. For this purpose the strategy must be added in the `setupStrategies()` method in the `AdaptiveSpamDecisionMaker` class.

4 Future Work

Instead of creating mailsets from regular mails, another approach would be to simulate mailsets. For the *DME* itself, the content of the mails does not matter since the DecisionMakers rely on the FilterProcessResults as input, and that is exactly what could be simulated. The user could set the number of spam and ham mails in the mailset and for each filter the probability that a mail is classified correctly. The *DME* could then create the fake FilterResults from this information and put them in a FilterProcessResult together with the correct decision.
Further study needs to be performed on how to create good mailsets.

Since the AdaptiveSpam DecisionMaker is only a prototypic implementation, a lot of work could be done in this area. For example, add and evaluate new strategies or optimize the current *WMG* strategy further. Ideas would be to treat false negatives and false positives differently and introduce an upper bound for the weights.

5 Conclusions

The first objective of this thesis, designing and implementing an evaluation framework for DecisionMakers, has been achieved. The framework serves its designated purpose: Comparing and analyzing different DecisionMakers by using different mailsets and supporting the programmer during development of new DecisionMakers. The tool remains extendible and can be customized through the use of Metric- and Property Providers.

The second goal, implementing an adaptive DecisionMaker, has only been reached partially. Even though the AdaptiveSpam DecisionMaker and an initial strategy have been implemented and are working, a thorough analysis is still missing. Therefore no reliable statement about its performance could be established as part of this work. There are two reasons for this. First, we simply ran out of time. During this thesis we realized, that developing the evaluation framework would take a lot more time than originally anticipated. We decided that implementing such a framework would be our top priority. Therefore the *DME* has been developed first and the remaining time was used to work on the new DecisionMaker. Second, since the new "rule-like" spam filters (as described in the Introduction) have not yet been implemented, comparing both DecisionMakers is difficult. Even if the new DecisionMaker performs the same as the old one with the current filters, which could be shown in some tests, it does not necessarily imply that they are equally good.
It was interesting to work on this thesis and to have a look behind the scenes of a complex spam filter system like SPAMATO.

6 References

[1] SpamAsssasin. `http://spamassassin.apache.org/`

[2] Eclipse SWT Project. `http://www.eclipse.org/swt/`

[3] JFreeChart. `http://www.jfree.org/jfreechart/`

[4] N. Littlestone and M. K. Warmuth. The Weighted Majority Algorithm. `http://citeseer.ist.psu.edu/littlestone92weighted.html`

[5] Shlomo Hershkop. Behavior-based E-Mail Analysis with Application to Spam Detection. `http://www1.cs.columbia.edu/~sh553/publications/final-thesis.pdf`

7 Glossary

FilterResult
After a spam filter in SPAMATO has checked a mail, it creates a FilterResult where it records how it judges the mail. This includes a *result* value between 0 and 1 through which it can express with which probability it thinks that a mail is spam. Besides these so called *REGULAR* FilterResults, it is also possible to create FilterResults of type *UNKNOWN*. This means that the filter was not able to determine at all whether a mail is spam or ham. FilterResults with type *ERROR* indicate that there was some sort of internal error during filtering. FilterResults can also be nested in case of sub-filters.

DecisionMaker
A DecisionMaker makes the final decision whether a mail is considered as spam or ham, which is also what the user perceives. Its decision is based on the FilterResults, which are aggregated to the final decision. The way this aggregation works distinguishes the different DecisionMakers from each other.

Mailset
A mailset is a collection of mails together with a correct classification for each mail which states whether it is spam or ham. This classification has to be made manually by the user.

Test run
A test run is always executed against a specific mailset and with a specific DecisionMaker. Executing a test run means that each mail in the mailset is fed as input to the DecisionMaker and the decision created by the DecisionMaker is stored in the test run object for later analysis.

Metric Provider
A Metric Provider takes a test run as input and computes a metric based on the information contained in the test run. A metric captures the evolution of a certain value over time, which means that for each mail in the test run’s mailset a metric value can be recorded. Examples for metrics are the number of correct decisions or false positives after each mail. Generally Metric Providers are applicable to all DecisionMakers. To get more information about how to implement custom Metric Providers see Chapter 2.2.7.1.

Property Provider
In contrast to Metric Providers, which are called *after* test run execution, Property Providers are used *during* test run execution. They are in general DecisionMaker-specific, because they record parts of the internal state of a DecisionMaker. There are two different types of Property Providers: *Mail-based* and *run-based* Property Providers. *Mail-based* Property Providers measure the current value of a property every time a mail has been checked by the DecisionMaker, for example the current filter weights in case of the AdaptiveSpam DecisionMaker. Therefore the development over time can be visualized later on. *Run-based* Property Providers measure only *single* values for the whole execution of a test run. For example, certain settings of a DecisionMaker could be extracted. To get more information about how to implement custom Property Providers see Chapter 2.2.7.2.

8 Appendices

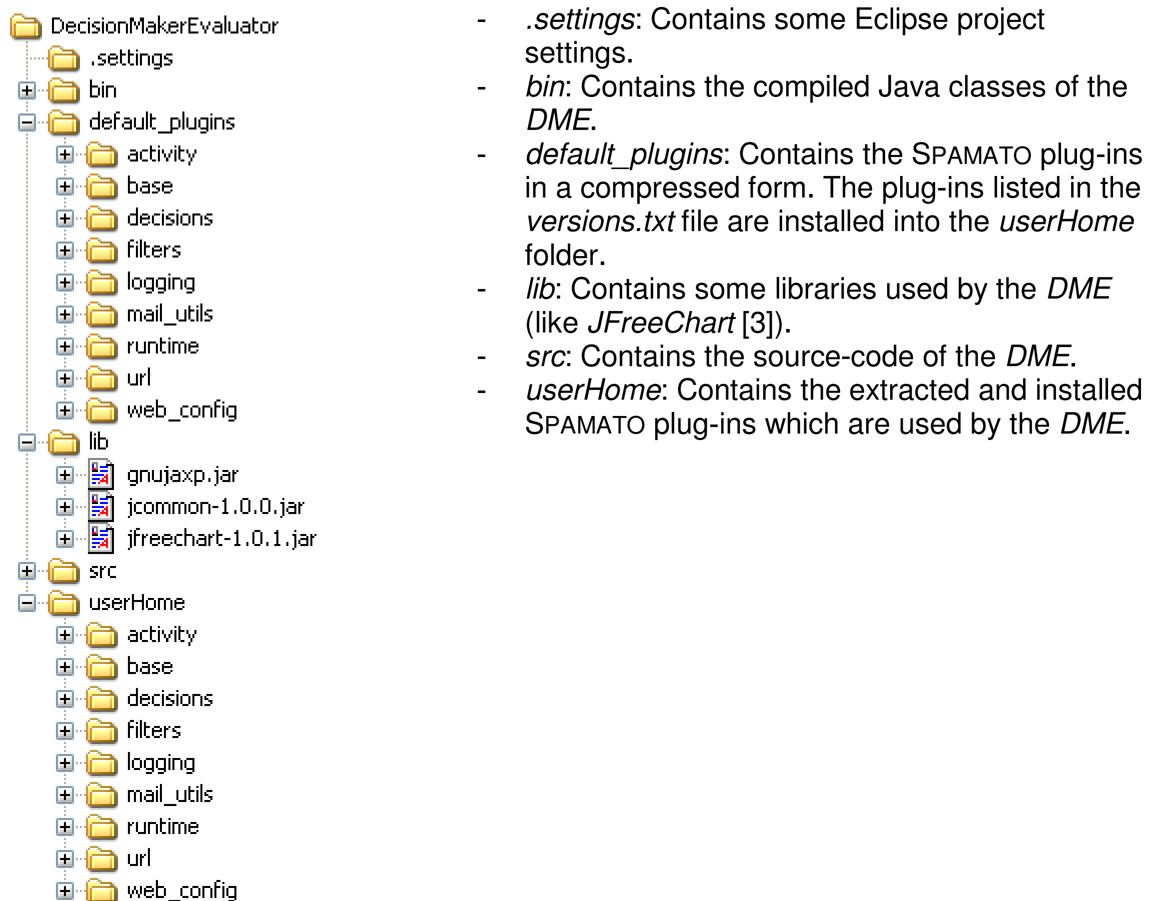
8.1 Required plug-ins

The following list contains all the plug-ins that are required for proper operation of the *DME*.

- Activity Manager
- Spamato Base
- At least one DecisionMaker
- All filters whose FilterResults are contained in at least one mailset
- Logging
- Mail Utilities
- Runtime
- URL PreProcessor
- Web Configuration

8.2 DME folder structure

The following diagram shows the folder structure used in the *DME*.



- *.settings*: Contains some Eclipse project settings.
- *bin*: Contains the compiled Java classes of the *DME*.
- *default_plugins*: Contains the SPAMATO plug-ins in a compressed form. The plug-ins listed in the *versions.txt* file are installed into the *userHome* folder.
- *lib*: Contains some libraries used by the *DME* (like *JFreeChart* [3]).
- *src*: Contains the source-code of the *DME*.
- *userHome*: Contains the extracted and installed SPAMATO plug-ins which are used by the *DME*.

Figure 8 – The *DME* folder structure