# Automatic Email Filters Generation

18.337 Final Report

Irina Zhelavskaya

December 17, 2012

## 1 Introduction

Nowadays people get lots of email that can range from the simple "hello, how are you doing" to corporative mail, mail from friends, colleagues, maybe some subscriptions, etc. Usually to manage them all people use folders, labels, tags and filters. Users can manually mark or put their messages to specific folders first couple of times, but not constantly, because foremost it is very time consuming and routine. As people are getting tired from this routine work after some time, they try to set up filters for automatic mail systematizing. Almost all mail clients have the option of filters creation. For example Gmail has very good system for that. But the problem is that their interface is comfortable for tech-savvy users, not for non-tech ones. And it may be not easy even for technical people, because they usually don't want to waste time and investigate how to create these filters. In case they do create them, these filters are usually easy and trivial, and not very complex and multi conditional, and they sometimes work not very properly and precise.

Therefore an idea to create a program that would automatically generate filters based on the information that we can get from how a user arrange his present mail in folders seems to be reasonable. Existing solutions such as mail clients that you need to install on your computer, or email categorization with Smart Folders (from Gmail) aiming to mass mailing categorization, have disadvantages and either solve the problem inefficiently or not entirely.

So the main idea of this project was to create a program that will process all user mail and use the information about folders and tags to generate filters, and after that to parallelize it in Julia in order to increase working performance of the algorithm. Since we will consider the problem when all letters are distributed in folders independently, the algorithm can be easily parallelized.

## 2 Formal statement of the problem

### 2.1 Definitions

To set up a problem first we will enter some definitions that will be used in the following sections.

We define such terms as a letter (mail), letter's attribute, a folder and a filter.

- ***Mail*** (or a ***letter***) is a vector:

$$\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k),$$

where $\mathbf{x}_i$ is a letter attribute (field), $i = \overline{1,k}$, $k$ – is a number of attributes (fields).

- ***Field*** is a vector:

$$\mathbf{x}_i = (x_{i1}, x_{i2}, \dots, x_{in_i}),$$

where $x_{ij}$ is an attribute value (unique identifier), $n_i$ – is a number of possible values of the $i$-th attribute, $j$ is a number of this attribute.

- ***Folder***

Suppose we have a set of $M$ letters (inbox) $\mathbf{X}_m$, $m = \overline{1,M}$. There also exist $S$ folders $P_1, P_2, \dots, P_S$. A folder is a subset of letters such that

$$\forall\, m\, \exists!\, s\colon \mathbf{X}_m \in P_s \Leftrightarrow P_i \cap P_j = \emptyset, i \neq j.$$

- ***Filter*** is also a vector similar to letter:

$$\mathbf{F} = (\mathbf{f}_1, \mathbf{f}_2, \dots, \mathbf{f}_k),$$

where $\mathbf{f}_i$ is a field of a filter $i = \overline{1,k}$, $k$ is a number of letter's attributes (fields).

To put it simply, filter is a set of rules that instructs to automatically deliver incoming messages to the folder of your choice.

## 2.2 Problem statement

Suppose we have $M$ letters $\mathbf{X}_m, m = \overline{1,M}$. It is known that the letters belong to $S$ folders $P_1, P_2, \dots, P_S$. We need to find such set of filters $\{\mathbf{F}_s\}, s = \overline{1,S}$, that

$$\forall \mathbf{X}_m \in P_s, \forall i\; \exists t \in \{1, \dots, n_i^m\}\colon \rightarrow\; x_{it} \in \mathbf{f}_{si},$$

and

$$\forall \mathbf{X}_m \notin P_s, \exists i\; \forall t \in \{1, \dots, n_i^m\}\colon \rightarrow\; x_{it} \notin \mathbf{f}_{si}.$$

That means that letters belonging to one specific folder will be filtered only in that folder, and other tags will not be assigned to it.

# 3 Algorithm

## 3.1 Preliminary research

We started from consideration of machine learning (ML) and neural networks (NN) algorithms and techniques as suitable tools for solving the problem of filters creation. This choice was caused by the fact that ML and NN algorithms are generally used for solving analogous problems, which are connected with rules creation based on data analysis. However this approach would require sophisticated data preprocessing. Also we would need to assess a big variety of different ML methods and algorithms that can be applicable to this problem in order to choose one that would fit best for solving this problem.

First we start testing and experimenting with different ML algorithms and techniques (such as Naïve Bayes, SVM, Random Trees, Ensemble methods, AdaBoost, etc.) in WEKA (Machine Learning Software). It appeared that filters created might be too complicated for such task (real mail is not an extremely complicated structure, and people usually put some specific and understandable logic into filters they create).

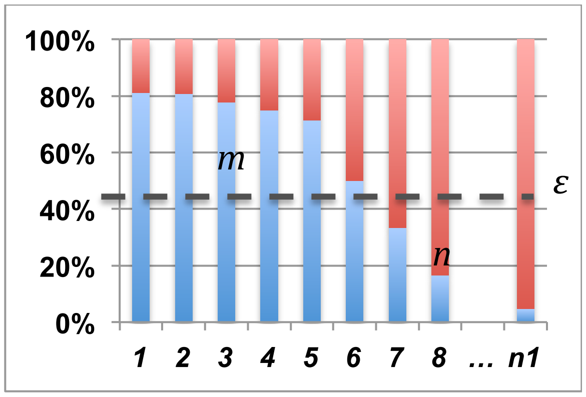
Since ML algorithms and NN may be unpredictable and to improve their performance may not be trivial, we decided to make our own algorithm basing on logic that we can get from statistical nature of mail. From our preliminary experiments we understood that it is reasonable not to try to create a small set of very complicated filters, but to create may be a bigger set of simple filters that will not affect each other. In the next section the algorithm proposed is described.

## 3.2 Algorithm

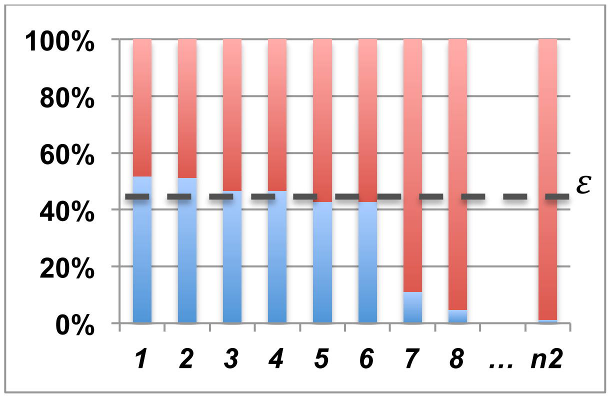
We decided to start with a simple case, when one letter can be in one folder only, which means that a letter can contain only one tag (without multiple tagging). From this it follows that folders can be considered independently from each other.

The second assumption for filters creation is the nature of the folder. We assume that letters in one folder have some feature in common, for example, one folder can contain letters from several specific senders, and letters from these senders lie mostly in this folder. Or letters can be of a similar context (that reflects in subjects). We can get the information about those common features if we calculate frequency histograms for all attributes in all letters. Basing on this information we can obtain specific patterns and create filters. Attributes are sender, recipient, and subject. At this point we do not consider such attribute as text of a letter, but it will be added in the next iterations.

The description of this algorithm is given below.



(a)



(b)

Figure 1. Frequency histograms of two attributes for folder #i: a) sender, b) subject.

Let us consider one folder, for example folder #i. The algorithm consists of the following steps:

1. Build normalized frequency histograms for each letter's attribute (for all letters in the folder).

   Example of such histograms for two attributes (sender and subject) is shown in figure 1 (these histograms are sorted as well). "Normalized" means that we take into account not only the frequency of specific attribute value in the folder in hand, but in the whole inbox, that is:

$$\Pr(folder = i|sender = j) = \frac{\Pr\big((folder = i) \cap (sender = j)\big)}{\Pr(sender = j)}$$

2. Analyze histograms: choose the "best" histograms (where m/n → **max**).
   1. If m/n > $\varepsilon$: choose this attribute value for the filter and create a simple filter,
   2. Else, take the rest attributes values (columns of the histogram) and build histograms for them for other remaining attributes.
   3. Go back to step 2.1.

Results of work depend on parameter $\varepsilon$ to be selected. It will be shown in section 5.

Listing of a couple of core functions of the algorithm is shown in Appendix A.

## 4 Data

We worked on two sets of data: real and synthetic. In the following subsections we will describe the process of getting data from real email account, and the process of generating synthetic data.

### 4.1 Real data

Real data was obtained from real email account (we used one of our own Gmail accounts). Information from account was obtained via Gmail Backup software (http://www.gmail-backup.com/), which gave us as an output a number of text files which are consistent with each letter. Each text file contains all needed information (sender, recipient, subject and etc.) but also a lot of service information which is not needed at all. That is why one of the tasks at this point was to parse each file correctly. We used MATLAB to implement the solution for this task so that the output could be used for MATLAB implementation of the algorithm. It is also very important to mention that the whole process of parsing files can be parallelized very well.

As for error evaluation, real filters from Gmail account can be exported, preprocessed as well, and we can use them to compare with generated filters.

### 4.2 Synthetic data

Naïve Bayes model was used as a model of synthetic mail. The process of generating data is as follows.

Each attribute (those are either sender, recipient, or subject) has its own model of distribution of data in folders. First we are generating different distribution models for all attributes of a letter. And then according to these randomly generated models letters (mails) for the inbox are being generated. Code for the mail generator is shown in Appendix B.

# 5 Experiments and results

## 5.1 Serial implementation

Serial version of the algorithm was implemented in MATLAB. Algorithm was tested both on real and synthetic data. Results are presented in this section.

First will be described results for the generated data, and then for the real one. For the case of generated data several parameters need to be specified:

- Number of letters,
- Number of folders,
- Number of senders,
- Number of recipients,
- Number of subjects.

It is obvious that results of algorithm's work depend on mailbox, and therefore on those parameters. For error estimation cross validation was used. It has become clear from the experiments conducted that a folder should contain a sufficient number of letters for more truthful error estimation (training set should be of realistic size to be representative, and so the test set should be). Results also depend on one more parameter: $\varepsilon$ (the meaning of this parameter is shown in Figure 1). Red curve in figure 2 represents a typical picture of error dependence on parameter $\varepsilon$. Mailbox was generated with the following parameters: number of letters was equal to 3000, number of folders to 15, number of senders to 40, number of recipients to 20, number of subjects to 200, and number of cross validations to 50. It can be seen from the graph that the higher is the value of the parameter $\varepsilon$, the more is the error. The reason for that is following. Let's consider an example when $\varepsilon$ is set to 95%. By setting it that way we assumed that we consider almost ideal case: letters are distributed in folders according to strict rules. For example, filter for folder #k with a field sender #j will be created only in the case when more than 95% of letters from sender #j lie in folder #k.

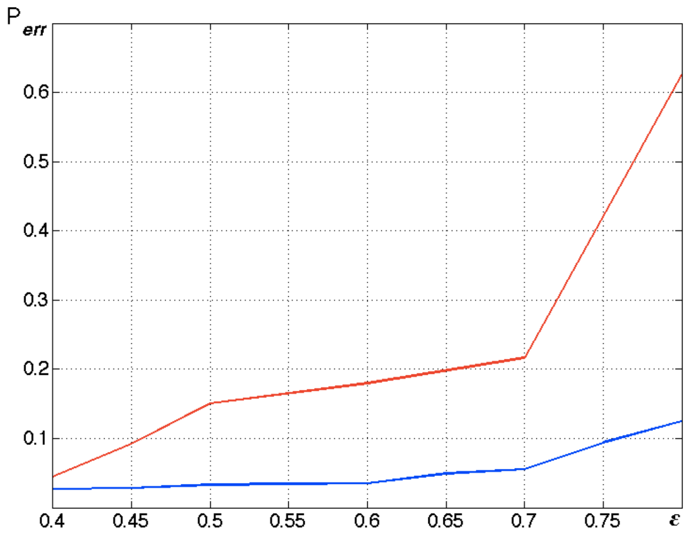


Figure 2. Error dependence on parameter $\varepsilon$: red curve stands for synthetically generated data, blue curve for the real data. Parameters for generated data: number of letters = 3000, number of folders = 15, number of senders = 40, number of recipients = 20, number of subjects = 200, and number of cross validations = 50; for real data: number of letters = 6357, number of folders = 32, number of senders = 295, number of recipients = 239, number of subjects = 2042, and number of cross validations = 50.

For the real data, only parameter $\varepsilon$ is influencing the results. The results are shown in figure 2 by a blue line. The performance is similar to the synthetically generated data. Parameters of the mailbox are: number of letters is 6357, number of folders is 32, number of senders is 295, number of recipients is 239, and number of subjects is 2042. Number of cross validations in the experiment was equal to 50.

## 5.2 Parallel implementation in Julia

Parallel version of the algorithm was partially implemented in Julia. Core functions taking the most calculation time were implemented in Julia to estimate the speedup that can be achieved with the help of parallelization. Results are presented in figure

3. The graph is built in logarithmic scale. The horizontal axis stands for the size order of a matrix A (size(A) = 10^x), for which frequency histograms are calculated. From this figure it can be seen that speedup reaches up to 6 orders magnitude on the matrix size equals to 10^6.

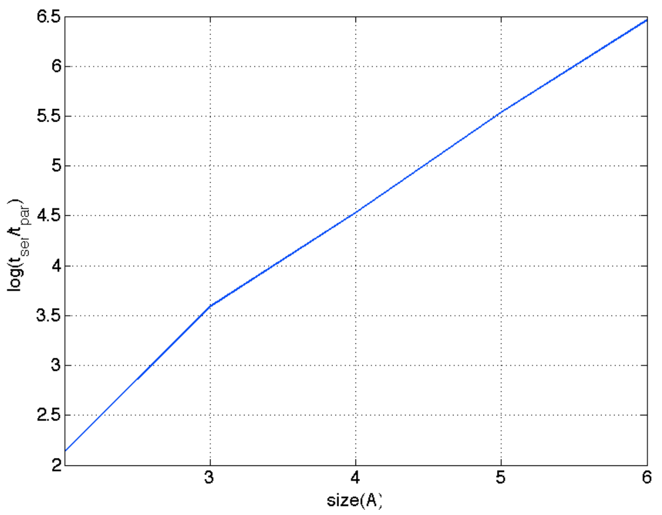


Figure 3. Plot of operation time of serial version of the algorithm to parallel one ratio against the size of the matrix, for which frequency histograms were calculated. Graph was built in logarithmic scale.

## 6 Moving forward

In this project, an algorithm for mail filters creation based on statistical information getting from mailbox was proposed. Analysis of serial and parallel version of the algorithm was done. Future development of this project will continue with the purpose of creating a plugin for Gmail that will process user's mail and use the information about letters' distribution in folders to generate filters. Future work includes improving the algorithm (add analysis of letters' text), investigating more parallelization options, and making a comparison with other algorithms.

## 7 References

[1] Combining Pattern Classifiers: Methods and Algorithms (Kuncheva, L.I.; 2004)

[2] L. Rokach, L. Naamani, A. Shmilovici, Active Learning Using Pessimistic Expectation Estimators, Control and Cybernetics, 38(1): 261-280 (2009)

[3] Lior Rokach: Ensemble-based classifiers. Artif. Intell. Rev. 33(1-2): 1-39 (2010)

[4] Kuncheva LI, Whitaker CJ (2003) Measures of diversity in classifier ensembles and their relationship with the ensemble accuracy. Machine Learning 51(2):181–207

[5] Margineantu D, Dietterich T (1997) Pruning adaptive boosting. In: Proceedings of the 14th International Conference on Machine Learning, pp 211–218

[6] Martinez-Munoz G, Suarez A (2004) Aggregation ordering in bagging. In: International Conference on Artificial Intelligence and Applications (IASTED), Acta Press, pp 258–263

[7] Martinez-Munoz G, Suarez A (2006) Pruning in ordered bagging ensembles. In: 23rd International Conference in Machine Learning (ICML-2006), ACM Press, pp 609–616

# Appendix A. Listing of the algorithm proposed

Listing 1. Main function for filters creation.

```
function [filters, errors, training_set, test_set] = main(Inbox, Num_tags, valid_rate,
eps)

% build histograms (ferquencies...) for the whole Inbox
filters = cell.empty;

errors = zeros(Num_tags, 3);

training_set = struct([]);
test_set = struct([]);

for i = 1 : Num_tags,

    filter_tag = [];
    [Set, Test_Set] = tag_filter(Inbox, i, valid_rate);     % Creating test and training
set
    if(~isempty(Set))
        training_set = [training_set; Set];
        test_set = [test_set; Test_Set];
        Universum = Inbox;

        hist_i = build_hists(Set, Universum);


        [best_hist, cons_hist] = choose_hist(hist_i, eps);

        if (~isempty(best_hist))
            filters = [filters, create_filter(best_hist, i)];
            filter_tag = [ filter_tag  create_filter(best_hist, i)];
        end,

        if (~isempty(cons_hist))
            for j = 1 : size(cons_hist, 1),
                filter = create_filter(cons_hist(j), i);
                [Set_j, Universum_j] = filtering(Set, Universum, filter);

                if(~isempty(Universum_j) && ~isempty(Set_j))
                    hist_j = build_hists(Set_j, Universum_j);
                    [best_hist_j, cons_hist_j] = choose_hist(hist_j, eps);


                    if (~isempty(best_hist_j))
                        filters = [filters, create_filter(best_hist_j, i)];
                        filter_tag = [ filter_tag  create_filter(best_hist_j, i) ];
                    end,

                    if (~isempty(cons_hist_j))
                         for k = 1 : size(cons_hist_j, 1),
                            filter_k = create_filter(cons_hist_j(k), j);
                            [Set_jk, Universum_jk] = filtering(Set_j, Universum_j,
filter_k);

                            if(~isempty(Universum_jk) && ~isempty(Set_jk))
                                hist_jk = build_hists(Set_jk, Universum_jk);
                                [best_hist_jk, ~] = choose_hist(hist_jk, eps);
                                if (~isempty(best_hist_jk))
                                    filters = [filters, create_filter(best_hist_jk, j)];
                                    filter_tag = [ filter_tag
create_filter(best_hist_jk, j) ];
                                end,
                            end


                         end
                    end
                end
            end,

        end,

        errors(i, 1) = error_count(filter_tag, Test_Set);
        errors(i, 2) = size(Test_Set, 1);
        errors(i, 3) = size(Set, 1);
    else
        errors(i, 1) = -1;
        errors(i, 2) = 0;
        errors(i, 3) = 0;
    end

end,
filters = filters';
```

Listing 2. Histograms building.

```
function [hist] = build_hists_inner(Set, Universum, field)

Univ_field = tabulate([Universum.(field)]);

Set_field = tabulate([Set.(field)]);
Set_field(Set_field(:,2) == 0, :, :) = [];

% build normilized histograms for tag in hand
Field_from_inbox_for_this_tag = Univ_field(Set_field(:,1), :);

a_send = Set_field(:, 2);
b_send = Field_from_inbox_for_this_tag(:, 2);
Field_norm_freq = a_send ./ b_send;
[~, norm_ind] = sort(Field_norm_freq, 'descend');


hist = struct.empty;
for j = 1 : size(Field_norm_freq, 1),
    hist(j).(field) = Set_field(norm_ind(j), 1);
    hist(j).frequency = Field_norm_freq(norm_ind(j), 1); % confidence
    hist(j).setcount = a_send(norm_ind(j));              % support
    hist(j).univcount = b_send(norm_ind(j));
end,
hist = hist';
```

Listing 3. Choose the best suitable histogram.

```
function [best_hist, cons_hist] = choose_hist(hist, eps)

length_hist = length(fieldnames(hist));
hist_names = fieldnames(hist);
stats = zeros(length_hist, 1);

for j = 1 : length_hist,
    frequencies.(hist_names{j}) = [hist.(hist_names{j}).frequency]';
    frequencies_j = [hist.(hist_names{j}).frequency]';
    stats(j) = nnz(frequencies_j > eps) / size(frequencies_j, 1);
end,

[~, ind] = max(stats);

best_hist = hist.(hist_names{ind})(frequencies.(hist_names{ind}) > eps);
cons_hist = hist.(hist_names{ind})(frequencies.(hist_names{ind}) <= eps);
```

## Appendix B. Listing for the mail generator

The generator was implemented in MATLAB.

### Listing 1. Mail generator.

```
function [Inbox, Tags] = inbox_generator(Num_folders, Num_letters, Num_senders,
Num_recep, Num_subj)

[Models, PModels] = generate_models_distr(Num_folders, Num_senders, Num_recep, Num_subj);


[Inbox, Tags] = generate_inbox(Num_letters, PModels, Models);
```

### Listing 2. Models generator.

```
function [Models_new, PModels_new] = generate_models_distr(Num_folders, Num_senders,
Num_recep, Num_subj)

% Each folder has specific model: distribution laws for senders,
% receipients, subject words.

% Probability density functions for models (distribution)
Models.sender = zeros(Num_senders, Num_folders);
Models.recep   = zeros(Num_recep, Num_folders);
Models.subject    = zeros(Num_subj, Num_folders);

% Cumulative distribution function
PModels.sender = zeros(Num_senders, Num_folders);
PModels.recep   = zeros(Num_recep, Num_folders);
PModels.subject    = zeros(Num_subj, Num_folders);

% After shuffling - the same structs
Models_new.sender = zeros(Num_senders, Num_folders);
Models_new.recep   = zeros(Num_recep, Num_folders);
Models_new.subject    = zeros(Num_subj, Num_folders);
PModels_new.sender = zeros(Num_senders, Num_folders);
PModels_new.recep   = zeros(Num_recep, Num_folders);
PModels_new.subject    = zeros(Num_subj, Num_folders);

mfield_names = fieldnames(Models);
length_models = length(fieldnames(Models));

for j = 1 : length_models,
    dif = zeros(1, Num_folders);
    b = 100;
    for i = 1 : size(Models.(mfield_names{j}), 1) - 1,
        Models.(mfield_names{j})(i, :) = b .* rand(1, Num_folders);
        dif = dif + Models.(mfield_names{j})(i, :);
        b = 100 - dif;
        PModels.(mfield_names{j})(i, :) = dif;
    end,
    Models.(mfield_names{j})(i + 1, :) = 100 - dif;
    PModels.(mfield_names{j})(i + 1, :) = 100;

    % Then shuffle everything a little bit, so it is more random and all that
    for i = 1 : size(Models.(mfield_names{j}), 2),
        new_ind = randperm(size(Models.(mfield_names{j}), 1));


        Models_new.(mfield_names{j})(:, i) = Models.(mfield_names{j})(new_ind', i);
    end

    dif = 0;
    for i = 1 : size(Models.(mfield_names{j}), 1),
        dif = dif + Models_new.(mfield_names{j})(i, :);
        PModels_new.(mfield_names{j})(i, :) = dif;
    end,
end,
```

Listing 3. Letters generator.

```
function [Inbox, Tags] = generate_inbox(Num_letters, PModels, Models)

mfield_names = fieldnames(PModels);

Num_models = length(fieldnames(PModels));
Num_folders = size(PModels.(mfield_names{1}), 2);

Letters_count = unidrnd(Num_letters, 1, Num_folders);
Letters_count = floor(Num_letters * Letters_count ./ sum(Letters_count));

ind = randperm(Num_folders, Num_letters - sum(Letters_count));
Letters_count(ind) = Letters_count(ind) + 1;

Inbox = struct([]);
inc = 0;

for i = 1 : Num_folders,
    for l = 1 + inc : Letters_count(i) + inc
        for j = 1 : Num_models,
            Inbox(l).tag = i;
            law = PModels.(mfield_names{j})(:,i);

            if (strcmp(mfield_names{j}, 'sender'))
                field = size(law, 1) - sum(100 * rand <= law) + 1;
                Inbox(l).(mfield_names{j}) = field;
            else
                field = 0;
                while (sum(field) <= 0)
                    field = zeros(size(law, 1), 1);
                    k_ar = 1 : size(law, 1);
                    for k = k_ar,
                        g = rand;
                        field(k) = (k == (size(law, 1) - sum(100 * g <= law) + 1));
                    end,
                end
                Inbox(l).(mfield_names{j}) = k_ar(field'>0);
            end,

        end,
    end,
    inc = l;
end,
Inbox = Inbox';


Tags = (1 : Num_folders)';
```