Spamfilter v.1.0 – documentation*

Adolf Středa, Marek Zpěváček

February 27, 2018

In further sections we shall use these keywords:
spam – an unsolicited e-mail, mostly includes advertising, scams; unfortunately this category is defined subjectively
ham – a complementary set of e-mails to spam
dictionary – a list of strings; any deviation from this definition will be explicitly stated (e.g. Python dictionaries)
vectorized e-mail – an email translated into a vector of word frequencies, the vector size and the frequency ordering is defined by a dictionary.
classification – a binary classification, a positive classification corresponds to an e-mail being classified as spam

1 Object design

The main part of the Spamfilter is consisting of three main classes, each implementing a distinct part of the data processing – classification, feature extraction, and input format parsing. These classes are completely independent, although a lesser sort of compatibility is enforced by the inheritance in concrete implementations. A high level overview is provided by a Figure 1 that illustrates the parent-child relationships. Unless we consider a user interface then these parts are independent.

1.1 Feature extraction

Feature extraction objects are defined in *DictionaryUtils.py*. There is one core function *createDictionary*, that takes 3 lists as an input – the first one contains lists of strings (each list is made by parsed e-mail split by whitespace), second one contains a list of already listed words and the third one contains corresponding frequencies – this design is intended to allow creation of a dictionary similarly to merge-sort. This approach produces a significant speed-up, as the dictionary created in a single batch is significantly smaller and thus finding the index of a word is much faster. Resulting computational overhead becomes negligible even when the batches are formed from only hundreds of emails. The core class also contains abstract function *filterDictionary* that takes 4 lists (spam/ham dictionaries and frequencies) as an input, intended output is an optimized dictionary.

1.1.1 DictionaryMI

It exposes a new function *mutualInformation* that couples every word with its mutual information with respect to label. If we denote a random variable corresponding to label as L and a random variable corresponding to word selection as W then we can define mutual information:

$$\sum_{w \in dictionary} Pr(L = spam, W = w) log \frac{Pr(L = spam, W = w)}{Pr(W = w) Pr(L = spam)}$$

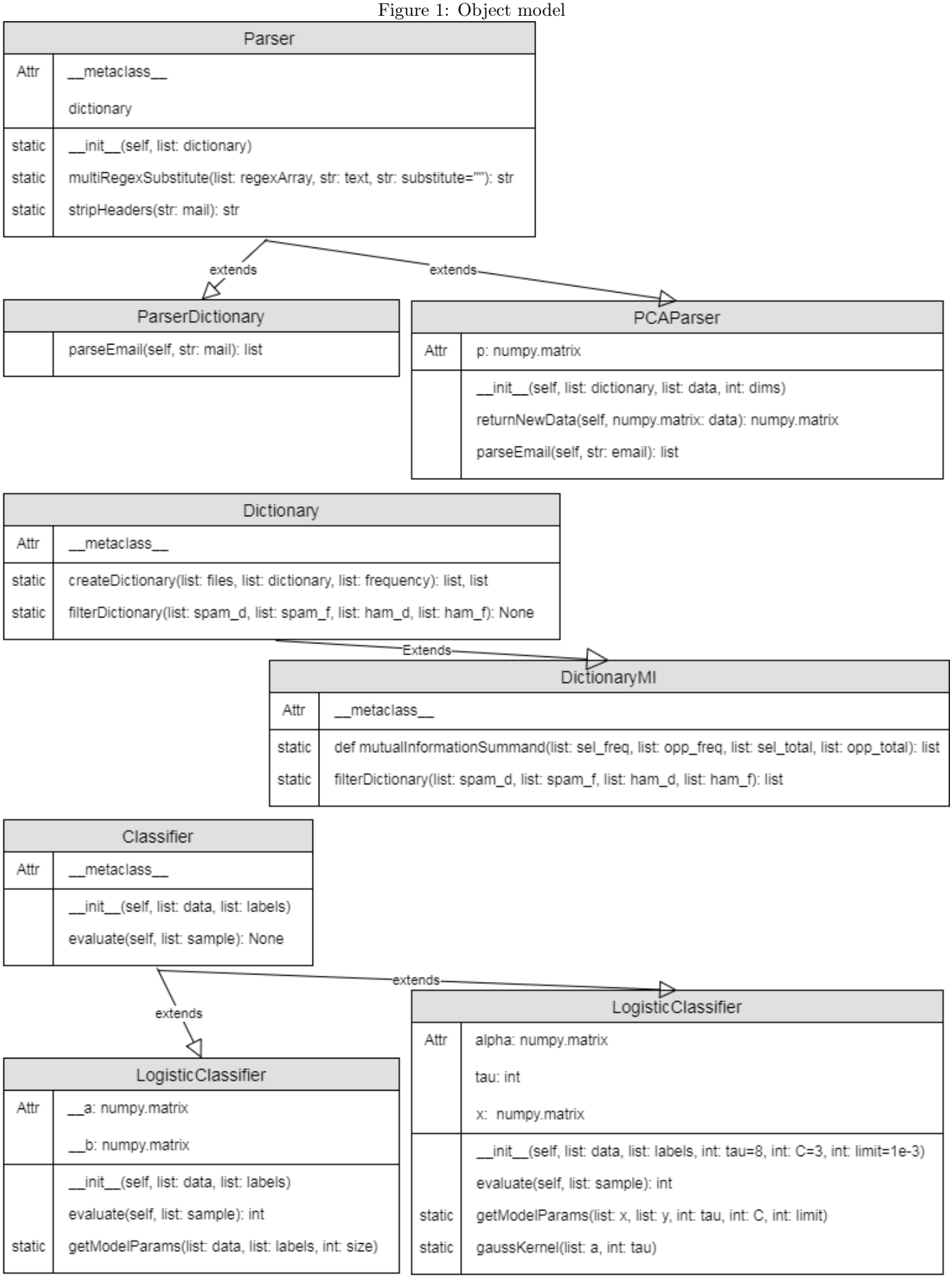
. This is the only part that is affected by the initial choice of label that corresponds to positive classification result. The choice was made in favour of spam due to the fact that one's mailbox might contain many distinct types of ham (friends, collogues, work, official communication etc.), while we suspect that spam language variability might be limited to smaller number of categories with a significant overlap between mailboxes. This class implements function *filterDictionary* in a straightforward way, it returns a dictionary sorted by the words' mutual information value.

1.2 Parser

The parser (*Parser.py*) provides a unified interface translating an e-mail into a vectorized e-mail. At the initialization a dictionary has to be provided (e.g. from *DictionaryUtils*). Currently two different approaches are supported, both inheriting two static functions – *multiRegexSubstitute* taking a list of characters to be replaced, the text and a character that should replace characters from the list; *stripHeaders* utilizes previous function to substitute common special characters and Czech special characters in a given string (e-mail). The latter function's choice of substitutions was motivated by the data available to us at MFF UK. Further improvement in terms of generality and possibly support for more language-specific character sets might be in order in the future.

*This project was supported by a grant (SFG) from the Faculty of Mathematics and Physics, Charles University

Figure 1: Object model



1.2.1 ParserDictionary

The simplest extension implementing the function *parseEmail* that takes a string (e-mail) as an input, strips headers and special characters (c.f. *stripHeaders*), and then utilises a dictionary that was given at the initialization to directly convert the e-mail string into a vector of frequencies.

1.2.2 PCAParser

The principal component analysis is a statistical method used for the dimension reduction. In the initialization phase (function *__init__*), an existing dictionary and a list of vectorized emails are provided. For this, an existing parser is required, for example *ParserDictionary* could be used. Then the input list is converted to *numpy.matrix* object and singular value decomposition (*numpy.linalg.svd* function) is applied on this matrix. That gives us a list of eigenvalues and corresponding eigenvectors sorted by eigenvalue's absolute value. Then we pick the top *dims* eigenvectors corresponding to largest eigenvalues and store them as the variable *p*.

To use this parser, new data (vectorized emails) needs to be created by *returnNewData* function. This function performs orthogonal transformation to subspace generated by saved top eigenvectors.

To parse an email, the function *parseEmail* is used. At the beginning, it does exactly the same as *parseEmail* function in *ParserDictionary* class, however it also performs orthogonal transformation into the subspace generated by the top eigenvectors.

1.3 Classifier

There are two distinct algorithms that are implemented in the classifier (*Classifier.py*). The first one is based on Support Vector Machine (SVM), which was chosen due to its popularity in general; its implementation also incorporates so called kernel trick. The second one is based on logistic regression, this time the choice was motivated by the fact that this algorithm allows to assign a confidence to its classification. Both classifiers assign positive values to e-mails classified by them as spam. While we could conversely detect ham, it would mean only a change in notation and it might probably cause a minor confusion as a definition of ham is even hazier than the definition of spam.

The classifiers utilize types and their respective operations defined by Python library *numpy*, the learning process is defined as a process of solving a convex optimisation problem defined by their respective formulas. We used Python library *cvxpy* that allowed us to define the problem formula (as opposed to *sklearn*) ourselves. *cvxpy* is mostly an extended wrapper that translates the python code into a format that a common convex optimization solver (e.g. SCS, ECOS) understands and also interprets its results in a *numpy* type. Moreover it contains several heuristics to decide which solver it should use, unfortunately in both classifiers these heuristics selected either incompatible or inefficient solver and thus the choice had to be hardcoded into classifiers.

All the classifiers should initialize the parameters during the Classifier object initialization. Moreover they should expose a function *evaluate* that takes a Classifier instance and a vectorized e-mail and outputs an integer representing the classification verdict.

1.3.1 Logistic regression

This algorithm classifies items by a logistic curve L, defined by a simple formula:

$$L(\vec{x}) = \frac{e^{A\vec{x}+\vec{b}}}{1+e^{A\vec{x}+\vec{b}}} - \frac{1}{2},\ A \in \mathbb{R}^{n\times n},\ \vec{b} \in \mathbb{R}^n,\ \vec{x} \in \mathbb{R}^n$$

. Given a vector representing the item, L outputs a value in range $(-\frac{1}{2}, \frac{1}{2})$; if the value is positive then the classification is also positive.

Learning part takes two lists as an input during the initialisation (function *__init__*) – list of vectorized emails and a list of labels with 1/0 values (zero for ham, 1 for spam). These two lists should be of the same length, similarly every vectorized email should have the same length. At the end of the part parameters __a and __b (corresponding to A and $\vec{b}$ in the formula) should be set. Due to several problems in the learning process the solver is set to *SCS*.

Classification is done through function *evaluate* that takes the classifier instance and a vectorized e-mail as an input. The output is a float in range $(-\frac{1}{2}, \frac{1}{2})$. Note that this function throws an error (*Model parameters not set!*) if the model parameters are not set. Since they are set during the initialization such error might possibly be caused by the training data format.

1.3.2 Support Vector Machine

Support vector machine takes two labelled sets of points in $\mathbb{R}^n$ as input and tries to find a hyperplane which divides these sets. In other words, given two sets S_1 and $S_2 \subset \mathbb{R}^n$, we are finding $\alpha \in \mathbb{R}^n$ and $\beta \in \mathbb{R}$ such that: $\forall x \in S_1 \colon x \cdot \alpha + \beta > 0$ and $\forall x \in S_2 \colon x \cdot \alpha + \beta < 0$. These sets are not usually linearly separable, therefore ℓ_1 regularization is used. Furthermore, we are using a kernel trick with Gaussian kernel.

The learning part is similar to the learning part in logistic regression. Two labelled lists are provided as an input and finally α is output of the learning phase of the algorithm. Usage of the Gaussian kernel has two consequences: first is

that the affine shift β could be ignored. Second: almost all the elements in α are non-zero. For performance purposes elements smaller than *limit*, default is value is 10^{-3}, are set to zero.

Classification is also similar to classification in logistic regression with notable difference that output could range whole $\mathbb{R}$, not only $(-\frac{1}{2}, \frac{1}{2})$.

2 User interface

The user interface is provided through a Python script *spamfilter.py*. The script runs in two modes: model and test. Both modes currently support *MBOX* files, lists of e-mail strings serialized through Python library *Pickle* and raw e-mail string (although any special encoding – e.g. attachment, Base64 will return useless data). The decision which method to use is decided by the file's file extension – *.mbox* (MBOX), *.p* (Pickle); any other file extension is treated as a single raw mail.

2.0.1 Training

spamfilter.py [--ham HAM] [--spam SPAM] [-o OUTPUT]
[-a {mi-logistic,mi-svm}] [-l LIMIT] [-d DICTLIMIT] [-r RAMLIMIT] model

	--ham HAM	Path to ham file
	--spam SPAM	Path to spam file
-o OUTPUT	--output OUTPUT	Output model file
-a mi-logistic,mi-svm	--algo mi-logistic,mi-svm	Classifier
-l LIMIT	--limit LIMIT	Max. amount of ham/spam
-d DICTLIMIT	--dictLimit DICTLIMIT	Max. dictionary size
-r RAMLIMIT	--RAMLIMIT	Max. RAM allocation

Limits are optional by default with default values 500 ham/spam, 250 for dictionary and unlimited RAM. First two limit the amount of data processed, RAM limit kills of the training if the limit is reached. This is due to risk of the OS hanging due to swapping or a lack of memory. OUTPUT variable is also optional, the default value is *model.p*.

2.0.2 Testing

spamfilter.py [-- HAM] [--spam SPAM] [-o OUTPUT]
[-a {mi-logistic,mi-svm}] [-l LIMIT] [-r RAMLIMIT] test

-m MAIL	--mail MAIL	Path to untagged e-mail file
-o OUTPUT	--output OUTPUT	Output e-mail file
-l LIMIT	--limit LIMIT	Max. amount of classified e-mails
-m MODEL	--model MODEL	Path to model
-r RAMLIMIT	--RAMLIMIT	Max. RAM allocation

Again limits are optional and with the same purpose. The script's output depends on the input's file extensions. All options should be described in the following two paragraphs.

If *MAIL* points to MBOX then the MBOX is either split into ham MBOX and spam MBOX (*OUTPUT* not set). Otherwise the emails in the MBOX are tagged (header *X-{svm,logistic}* with a classification score and the addition of *[SPAM]* into the e-mail's subject). Due to weird incompatibility of our department's MBOX files and the MBOX standard there is an ugly workaround: at the beginning a new MBOX is created (file-name is prepended with "._") and *>From* strings are replaced by *>From_fixed_nfkusadhgf:*. After the emails are tagged this change is reverted. Unless for whatever reason *>From_fixed_nfkusadhgf:* is used in the original e-mail, this change should not cause any problems.

If the *MAIL* points to serialized file (*Pickle*) then only a statistics (amount, amount of positive detections, percentage of positive detections) are returned. This is due to assumption that the emails are raw texts and thus there are no headers that could be tagged. If any other file extension is used then again only statistics are displayed.