Analysis of ML methods and algorithms for detecting spam e-mail messages

INNOvative SOC/INNOSOC project

Contents

Introduction 3

Experiments and results 4

Datasets 4

SpamAssassin 4

SMS Spam 5

Algorithms and methods used 6

Decision trees 6

SVM 8

Naive Bayes 9

Logistic regression 11

XGBoost 12

Settings and parameters of the selected methods 14

Decision trees 14

SVM 14

XGBoost 15

Logistic regression 16

Naive Bayes 16

Evaluation of experiments 17

Confusion matrix 17

Accuracy 18

Precision 18

Recall 18

F1-Score 19

Results 19

SpamAssassin 19

SMS Spam 19

Summary 19

Introduction

Most security threats use e-mail as an attack vector. Since the volume of traffic transmitted in this way is particularly large, it is necessary to use automated detection procedures that employ machine learning (ML) and artificial intelligence (AI) algorithms. In order to fully understand the power of ML/AI algorithms applied to spam detection, it is first necessary to clarify on which tasks the filtering of spam-type e-mail messages should be performed.

For a better understanding of the tasks performed by a spam filter, the following example can be given. Imagine that you are separating the e-mails we receive and categorizing them based on the presence or absence of specific keywords that occur in the text of the e-mails with a certain frequency. For this purpose, we could perhaps list all the messages we receive in the inbox in a table. But how should we proceed in classifying the received messages into the categories HAM (legitimate message) or SPAM?

As already mentioned, the number of occurrences of suspicious keywords in the text of e-mail messages will be searched for. Individual messages identified as spam will then be assigned a score based on the number of occurrences of the identified keywords. This score will further provide a reference for classifying subsequent e-mail messages.

A threshold value is determined that makes it possible to separate spam e-mail messages. If the calculated score exceeds the set threshold, the e-mail will automatically be classified as spam – otherwise the message will be accepted as legitimate and thus classified as ham. This threshold value (as well as the assigned scores) will be continuously determined to take into account a new series of spam messages that may be encountered in the future.

Even from the abstract description of the spam detection algorithm, some important properties that need to be kept in mind can be noticed: it is necessary to proceed with the identification of a certain number of suspicious keywords that make it possible to classify messages as potential spam e-mails, with each e-mail being assigned a score based on the number of occurrences of the identified keywords.

It is necessary to set a certain threshold value for the score assigned to individual e-mails, above which e-mails will automatically be classified as spam. It is also important to weigh correctly the significance of the keywords present in the text of e-mails, so that the probability that a message containing the given keywords is spam can be adequately represented (the keywords could in essence even be harmless, but together they are more likely to represent junk mail).

It must be taken into account that spammers are well aware of the efforts to filter unwanted messages and will therefore try their best to adopt new strategies to deceive

spam filters. This translates into a process of continuous and iterative learning, which can be implemented well using ML/AI algorithms.

From the above, it is clear that it is no coincidence that spam detection represents the first test in the introduction of ML/AI in the field of cybersecurity. The first spam detection solutions in practice employed static rules that used regular expressions to identify predefined patterns of suspicious words in the e-mail text.

These static rules quickly proved ineffective due to the ever-new deception strategies implemented by spammers to deceive anti-spam filters. It was therefore necessary to adopt a dynamic approach that would allow the spam filter to learn on the basis of the constant innovations introduced by spammers while also making use of the user's decisions in classifying their e-mails. In this way, it was possible to effectively cope with the explosive spread of the spam phenomenon.

Experiments and results

Datasets

SpamAssassin

The SpamAssassin dataset (also called the SpamAssassin Public Corpus) is a collection of e-mail messages that have been labelled as spam or legitimate messages (ham). It was created by the Apache SpamAssassin project, which is an open-source anti-spam platform. The corpus contains several thousand e-mail messages divided into the spam and ham categories. In total, the dataset contains 6,047 messages with a spam ratio of approximately 31%.

Spam messages are unsolicited commercial e-mails, phishing attempts and other unwanted communication, while ham messages are legitimate personal and professional e-mails.

The e-mails are stored as raw text files in the standard e-mail format (RFC 822), with the headers, body content and metadata preserved. The dataset contains various types of e-mails with various formatting styles, HTML content, references to attachments and various languages (although predominantly English).

In terms of content, these are real e-mails collected from actual users who donated their messages, which makes them representative of real spam and ham patterns. The dataset includes complete e-mail headers containing valuable metadata, such as sender information, routing details, timestamps and content type information.

This dataset is most often used for training and evaluating spam classification algorithms, comparing machine learning models for text classification, research in natural language processing and information retrieval, or testing feature extraction

techniques for e-mail filtering. Over the years, the dataset has been released in several versions, and the public SpamAssassin corpus is freely available for research purposes.

SMS Spam

The SMS Spam Collection is a set of SMS messages labelled as SMS messages that were collected for SMS spam research. This dataset is widely used in machine learning and natural language processing to train spam detection systems for SMS messages. There are 2 versions of this database; the most widely used for SMS is “SMS Spam Collection v.1”, which has become the standard in this field. It contains a set of SMS messages in English with 5,574 messages, labelled as ham (legitimate) or spam.

The files contain one message per line. Each line consists of two columns: v1 contains the label (ham or spam) and v2 contains the raw text. The corpus was collected from free or free-for-research sources on the internet:

- A collection of 425 SMS spam messages was manually extracted from the Grumbletext website. This is a UK forum on which mobile phone users make public claims about SMS spam messages, most of them without reporting the actual spam message received. Identifying the text of spam messages in the claims is a very difficult and time-consuming task, and it involved carefully scanning hundreds of web pages.
- A subset of 3,375 randomly selected SMS messages from the NUS SMS Corpus (NSC), which is a dataset of approximately 10,000 legitimate messages collected for research at the Department of Computer Science of the National University of Singapore. The messages come mainly from Singaporeans and mainly from students attending the university. These messages were collected from volunteers who were informed that their contributions would be made publicly available.
- 450 SMS messages were collected from Caroline Tag's doctoral thesis.
- Finally, the SMS Spam Corpus v.0.1 Big is incorporated. It contains 1,002 legitimate SMS messages and 322 spam messages.

This dataset is most often used for the following tasks:

- Binary text classification
- Feature engineering training
- Handling imbalanced data
- Natural language processing (NLP) techniques
- Model evaluation and comparison

This dataset remains popular for educational purposes, research papers and as a basis for developing spam detection algorithms.

Algorithms and methods used

Decision trees

A decision tree is a non-parametric algorithm that is used for both classification and regression tasks. It has a hierarchical tree structure consisting of a root node, branches, internal nodes and leaf nodes. As can be seen from the diagram below, a decision tree starts with a root node, which has no incoming branches.

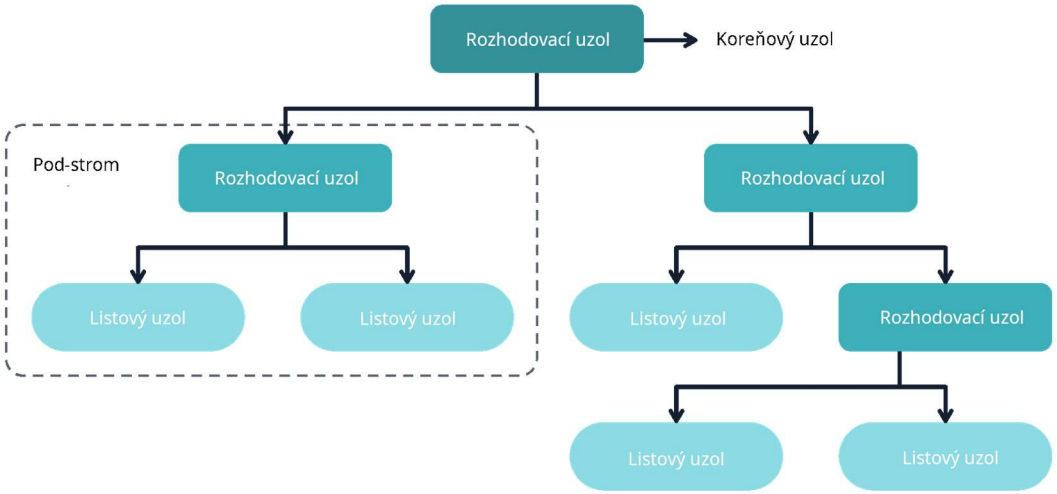


Fig. 1: Decision tree

The outgoing branches from the root node then feed into the internal nodes, also known as decision nodes. Based on the available features, both types of nodes perform evaluations in order to form homogeneous subsets, which are denoted by leaf nodes or terminal nodes. The leaf nodes represent all possible outcomes within the dataset. As an example, imagine that we are trying to assess whether or not we should go and play golf; we can use the following decision rules to make the choice:

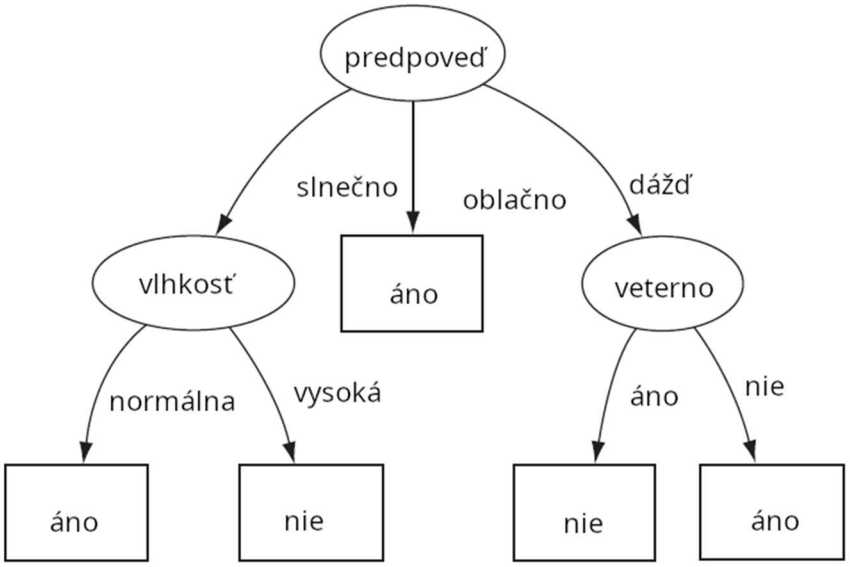


Fig. 2: Decision tree for the Golf dataset

This type of flowchart structure also creates an easily digestible representation of decision-making, which allows different groups across the entire organization to better understand why a decision was made. Decision tree learning uses a divide-and-conquer strategy by performing a greedy search to identify the optimal split points in the tree. This splitting process is then repeated in a top-down, recursive manner until all or most of the records have been classified under specific class labels. Whether all data points are classified as homogeneous sets depends to a large extent on the complexity of the decision tree. Smaller trees are more easily able to achieve pure leaf nodes – i.e. data points in a single class. However, as the tree grows, it becomes increasingly difficult to maintain this purity, and it usually results in too little data falling within a given subtree. When this happens, the phenomenon is known as data fragmentation, and it can often lead to overfitting. As a result, decision trees favour small trees, which is in line with the principle of parsimony according to "Occam's razor", meaning that "entities should not be multiplied unless necessary". In other words, decision trees should increase their complexity only when necessary, because the simplest explanation is often the best. To reduce complexity and prevent overfitting, pruning is usually used; This is a process that removes branches that split on features of low importance. The fit of the model can then be evaluated through a cross-validation process.

Another way in which decision trees can maintain their accuracy is to create an ensemble using the Random Forest algorithm. This classifier predicts more accurate results, especially if the individual trees are not correlated with one another. Among the several variants, the following most commonly used can be mentioned:

- ID3
- C4.5

- CART

SVM

The support vector machine is a classification and regression method proposed in the 1990s by V. Vapnik, which essentially determines a boundary curve separating 2 different classes. The main idea is to find a hyperplane that would optimally divide the input data in the feature space. For a given set of training data belonging to 2 classes (x_1, y_1), …,(x_l, y_l), where $x_i \in R_n$ and $y_i \in \{-1, 1\}$, we therefore try to find a hyperplane $wx + b = 0$ such that we can separate the data. Of all the hyperplanes determined by w and b, there is only one that maximizes the distance between the nearest points of these classes and the hyperplane - the optimal hyperplane, and the following holds:

$$d(w,b) = \min_{\{x_i | y_i = 1\}} \frac{wx_i + b}{|w|} - \max_{\{x_i | y_i = -1\}} \frac{wx_i + b}{|w|}$$

and the corresponding minimum and maximum values are -1 and +1, respectively

$$d(w,b) = \left| \frac{1}{\mathbf{w}} - \frac{-1}{\mathbf{w}} \right| = \frac{2}{|\mathbf{w}|} \ .$$

By minimizing the expression $|w|^2 / 2$ subject to the condition $y_i (wx_i + b) \geq 1$, we obtain the maximum distance between the 2 classes for the given pair of hyperplanes. The solution of the SVM optimization problem is given by the saddle point of the Lagrange function:

$$L(w,b,\alpha) = \frac{1}{2}\|w\|^2 - \sum_{i=1}^{l} \alpha_i \left[y_i \left(wx_i + b \right) - 1 \right] ,$$

where α_i is the Lagrange multiplier. While $L(w, b, \alpha)$ decreases with w and b, the derivatives of $L(w, b, \alpha)$ are equal to zero for all $\alpha_i \geq 0$. The solution is given by

$$\overline{w} = \sum_{i=1}^{l} \overline{\alpha}_i y_i x_i \qquad \overline{b} = -\frac{1}{2} \overline{w} \left[x_r + x_s \right] ,$$

where x_r and x_s are support vectors belonging to the class $y_r = 1$ and $y_s = -1$, respectively. The data in the original feature space are not separable by any linear hyperplane, but they may be linearly separable in a non-linearly mapped feature space that differs from the original one and usually has a higher dimension. This non-linear mapping $K(.,.)$ is performed using kernel functions:

$$f(x) = \operatorname{sgn}\left(\sum_{i=1}^{l} \overline{\alpha}_i y_i K(x_i, x) + b \right) \sum_{i=1}^{N} \alpha_i t_i K(x, x_i) + b ,$$

where yi are the target values, $\sum_{i=1}^{l} \overline{\alpha}_i y_i = 0$ and $\overline{\alpha}_i > 0$. The kernel *K*(.,.) is constructed so as to satisfy specific properties depending on the given problem, and it can be written:

$$K(x, y) = \Phi(x)\Phi(y) \ ,$$

where Φ(x) is the mapping of the input feature space into a new space with a potentially infinite dimension. There are 3 kernel functions providing non-linear mapping:

- polynomial $K(x, y) = (xy + 1)^z$ – the parameter *z* represents the degree of the polynomial
- Gaussian radial basis functions (Radial Basis Functions RBF) $K(x, y) = exp\{-[(x-y)^2 / 2\sigma^2]\}$ – where the parameter σ is the standard deviation of the Gaussian function
- MLP function $K(x, y) = tanh\ (scale(xy) - offset)$ – *scale* and *offset* are given parameters.

Naive Bayes

The naive Bayes classifier belongs to the category of probabilistic classifiers, which is based on Bayes' theorem.

In probability theory, we distinguish 3 kinds of probability for the relationship between random variables:

- Conditional probability
- Joint probability
- Marginal probability

Conditional probability is a measure of the probability of one random variable given the value of another random variable. Let $\forall X, Y \in F$ and $P(Y) \neq 0$, then

$$P(X|Y) = \frac{P(X \cap Y)}{P(Y)}.$$

$P(X|Y)$ denotes the conditional probability of *X* given the value of *Y*. For example, $P(X|Y = y)$ denotes the conditional probability of X given the value $Y = y$.

The concept of conditional distribution can be extended to measuring the probability of one random variable given several other random variables. For example, given $Y = y$ and $Z = z$, the conditional probability of $X = x$ is denoted as

$$P(X|Y, Z) = \frac{P(X \cap Y \cap Z)}{P(Y \cap Z)}.$$

Conditional probability is one of the important methods in probability theory for inference or learning under uncertainty.

Joint probability refers to the probability of two or more random variables. The joint (sometimes also called combined) probability of two random variables can be expressed as $P(X, Y)$ and $P(X, Y) = P(X \cap Y)$. From the first equation for conditional probability, it is known that the following holds for the joint distribution:

$$P(X, Y) = P(X|Y)\, P(Y).$$

According to the commutative law of probability theory, this equation can also be written as:

$$P(X, Y) = P(Y|X)\, P(X).$$

The above equations are also known as product rules. By extending the above joint distribution to n random variables, the following chain rule can be created:

$$\begin{aligned} P(X_1, \ldots, X_n) &= P(X_1)\, P(X_2|X_1)\, P(X_3|X_2, X_1) \cdots P(X_n|X_{n-1}, \ldots, X_1) \\ &= P(X_1) \prod_{i=2}^{n} P(X_i|X_{i-1}, \ldots, X_1). \end{aligned}$$

The chain rule is often used for the joint probability of several random variables, especially if there is a property of conditional independence between some of the variables.

Marginal probability refers to the probability distribution of a random variable on its own. Given the joint distribution, for the random variables X and Y we can calculate either the marginal distribution of X or of Y. To find the marginal distribution of one random variable, it is necessary to sum all the distributions of the other random variable. This can be formally expressed as:

$$P(X) = \sum_{y \in \mathrm{Val}(Y)} P(X, Y = y).$$

The above equation is also known as the sum rule.

The marginal distribution is so called because of this point of view: if we sum all the entries in a row or column of the joint distribution and place the result on the last side, it is the probability that the random variable takes this value, and this last side is called the margin (in English, margin => hence the name marginal).

Of course, this view holds only when the joint distribution involves two random variables.

The Bayes' theorem classifier based on strong independence between features is a simplification of the conditions of the theorem and has therefore acquired the attribute “naive”.

Given a data vector $\boldsymbol{x} = (x_1, \ldots, \mathbf{x}_n) \in X$, which is to be classified in the input feature space, where the output space is the set of categories $C = \{c_j \mid j = 1, \ldots, k\}$, naive Bayes classification is the conditional probability of category c_j for the given data vector $\boldsymbol{x}$, i.e. $P(c_j \mid \boldsymbol{x})$. According to Bayes' theorem, the following equation holds:

$$P(c_j|x) = \frac{P(x|c_j)\,P(c_j)}{P(x)} = \frac{P(x|c_j)\,P(c_j)}{\sum_k P(x|c_j)P(c_j)}.$$

According to the chain rule of the joint probability distribution, we obtain the following:

$$P(c_j, x) = P(c_j, x_1, \ldots, x_n) = P(c_j)\prod_{i=1}^{n} P(x_i|x_{i-1}, \ldots, , c_j).$$

$$P(c_j, x_1, \ldots, x_n) = P(c_j)\prod_{i=1}^{n} P(x_i|c_j).$$

$$P(c_j|x_1, \ldots, x_n) \propto P(c_j)\prod_{i=1}^{n} P(x_i|c_j).$$

$$P(c_j|x_1, \ldots, x_n) = \frac{1}{Z}\,P(c_j)\prod_{i=1}^{n} P(x_i|c_j),$$

Logistic regression

Logistic regression, also known as logistic regression analysis, is a statistical analysis method that predicts the corresponding binary outcomes based on the relationship between an independent variable and a dependent variable, and it can therefore be used for two-class classification. Classification is essentially a regression problem with a discrete output - the logistic regression algorithm reflects this nature.

Let X denote the input space of the classification and $C = \{0, 1\}$ denote the output space of the two-class classification. For input data $\boldsymbol{x} \in X$, the logistic regression $P(c|\boldsymbol{x})$ for predicting their category as $c \in C$ is given as follows:

$$P(c|x) = \sigma(z) = \frac{1}{1+e^{-z}} = \frac{1}{1+\exp(-z)},$$

where $\sigma(\cdot)$ denotes the logistic function, also known as the sigmoid function, which is also used as one of the activation functions in artificial neural networks.

Logistic regression has the following property, i.e. $P(c = 0|\boldsymbol{x}) = 1 - P(c = 1|\boldsymbol{x})$. This is because the probability distribution of a two-class classification should satisfy the condition $P(c = 0|\boldsymbol{x}) + P(c = 1|\boldsymbol{x}) = 1$. Also, let $z = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} - b$, from which the following is obtained:

$$P(c=1|x) = \sigma\left(w^{\mathrm{T}}x - b\right) = \frac{1}{1+\exp\left(-\left(w^{\mathrm{T}}x - b\right)\right)},$$

$$P(c=0|x) = 1 - \sigma\left(w^{\mathrm{T}}x - b\right) = 1 - \frac{1}{1+\exp\left(-\left(w^{\mathrm{T}}x - b\right)\right)}$$

$$= \frac{\exp\left(-\left(w^{\mathrm{T}}x - b\right)\right)}{1+\exp\left(-\left(w^{\mathrm{T}}x - b\right)\right)}.$$

A schematic diagram of the logistic regression classification algorithm is shown in the following figure.

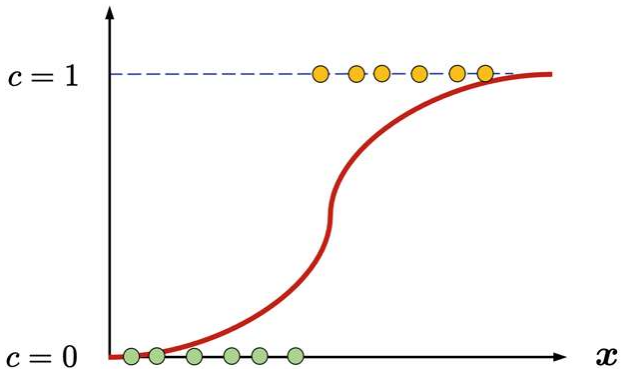


Fig. 3: Principle of the logistic regression classification algorithm

The logistic regression method described above is applicable only to two-class classification. Later, some extended versions of logistic regression appeared for multi-class classification problems, including multinomial logistic regression, ordered logistic regression and conditional logistic regression.

XGBoost

The XGBoost (Extreme Gradient Boosting) algorithm is one of the most efficient implementations of gradient-boosted decision trees and has been selected as one of the best machine learning algorithms used in Kaggle competitions. The XGBoost algorithm, specifically designed to optimize memory usage and to utilize hardware computing

power, reduces execution time while increasing efficiency compared with many machine learning algorithms and even deep learning models. The main idea of boosting is to build sub-trees from the original tree progressively, so that each subsequent tree reduces the errors (residuals) of the preceding tree. In this way, the new sub-trees will update the previous residuals in order to reduce the error of the cost function.

Consider a data set expressed as follows:

$$D = \{(x_i, y_i) \text{ where } x_i \in R_n; y_i \in R)\},$$

where n represents the dimension of the features x_i and y_i is the corresponding response or class. For the i-th element of the data set, we define the predicted value $\hat{y}_i$ as follows:

$$\hat{y}_i = \sum_{k=1}^{K} f_k(x_i), \quad f_k \in F,$$

where f_k denotes an independent tree in the space of regression trees F and $f_k(x_i)$ expresses the predicted score for the i-th sample and the k-th tree. The objective function of the XGBoost algorithm, denoted L, is given as follows:

$$\mathcal{L} = \sum_{i=1}^{n} l(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k).$$

By minimizing the objective function L, the regression tree functions f_k can be trained. The error function $l(y_i, \hat{y}_i)$ evaluates the difference between the prediction $\hat{y}_i$ and the actual value y_i. The term Ω is used to prevent the problem of overfitting by penalizing the complexity of the model as follows:

$$\Omega(f_k) = \gamma T + \frac{1}{2}\lambda \|w\|^2,$$

where T is the number of leaves in the tree, w represents the score (weight) of a leaf (e.g. the predicted value for samples that end up in this leaf), γ is a regularization factor expressing the cost of complexity, which discourages creating a tree with too many leaves, and λ is an L2 regularization term on the leaf weights, which tries to shrink them so that the model is more conservative. Direct minimization of the objective function is complex. XGBoost uses a second-order Taylor expansion to approximate it. This is a key reason for its performance, as it uses both the first and second derivatives (gradient and Hessian). Let $I_j = \{i|q(x_i) = j\}$ be the instance set of leaf j with a fixed structure $q(x)$. The optimal weights w_j of leaf j and the corresponding optimal value can be obtained by the following equations:

$$w_j^* = -\frac{g_j}{h_j + \lambda},$$

$$\mathcal{L}^* = -\frac{1}{2}\sum_{j=1}^{T}\frac{\left(\sum_{i\in I_j} g_i\right)^2}{\sum_{i\in I_j} h_i + \lambda} + \lambda T$$

where g_i and h_i are the first and second derivatives of the loss function L. The loss function L can be used as a quality score of the tree structure q. The smaller the score, the better the model. Since it is not possible to enumerate all tree structures, a *greedy* algorithm can solve the problem by starting from a single leaf and iteratively adding branches to the tree. Let I_R and I_L be the instance sets of the right and left nodes after the split. Assuming $I = I_R \cup I_L$, the loss reduction after the split is then expressed as follows:

$$\mathcal{L}_{\text{split}} = \frac{1}{2}\left[\frac{\left(\sum_{i\in I_L} g_i\right)^2}{\sum_{i\in I_L} h_i+\lambda} + \frac{\left(\sum_{i\in I_R} g_i\right)^2}{\sum_{i\in I_R} h_i+\lambda} - \frac{\left(\sum_{i\in I} g_i\right)^2}{\sum_{i\in I} h_i+\lambda}\right] - \gamma.$$

This relation is usually used in practice to evaluate split candidates. The XGBoost model uses many simple trees and the scores of leaf nodes during splitting. The first three terms of the equation represent the scores of the left, right and original leaf. In addition, the term γ represents the regularization on the additional leaf and will be used in the training process.

Settings and parameters of the selected methods

Decision trees

- criterion 'gini'
- splitter 'best'
- max_depth None
- min_samples_split 2
- min_samples_leaf 1
- min_weight_fraction_leaf 0.0
- max_features None
- random_state None
- max_leaf_nodes None
- min_impurity_decrease 0.0
- class_weight 'balanced'
- ccp_alpha 0.0
- monotonic_cst None

SVM

- C 1.0
- kernel 'linear'
- degree 3

- gamma 'scale'
- coef0 0.0
- shrinking True
- probability True
- tol 0.001
- cache_size 200
- class_weight None
- verbose False
- max_iter -1
- decision_function_shape 'ovr'
- break_ties False
- random_state None

XGBoost

- objective 'binary:logistic'
- base_score None
- booster None
- callbacks None
- colsample_bylevel None
- colsample_bynode None
- colsample_bytree None
- device None
- early_stopping_rounds None
- enable_categorical False
- eval_metric None
- feature_types None
- feature_weights None
- gamma None
- grow_policy None
- importance_type None
- interaction_constraints None
- learning_rate None
- max_bin None
- max_cat_threshold None
- max_cat_to_onehot None
- max_delta_step None
- max_depth None
- max_leaves None
- min_child_weight None

- missing nan
- monotone_constraints None
- multi_strategy None
- n_estimators None
- n_jobs None
- num_parallel_tree None
- random_state None
- reg_alpha None
- reg_lambda None
- sampling_method None
- scale_pos_weight None
- subsample None
- tree_method None
- validate_parameters None
- verbosity None

Logistic regression

- penalty 'l2'
- dual False
- tol 0.0001
- C 1.0
- fit_intercept True
- intercept_scaling 1
- class_weight None
- random_state None
- solver 'lbfgs'
- max_iter 100
- multi_class 'deprecated'
- verbose 0
- warm_start False
- n_jobs None
- l1_ratio None

Naive Bayes

- alpha 1.0
- force_alpha True
- binarize 0.0
- fit_prior True
- class_prior None

Evaluation of experiments

The following metrics were used to evaluate the experiments.

Confusion matrix

The confusion matrix is one of the most popular ways of evaluating any classification model. Although the matrix itself is not a metric, the matrix representation can be used to define various metrics, all of which become important in some specific case or scenario. A confusion matrix can be created for binary classification as well as for a classification model with multiple classes.

A confusion matrix is created by comparing the predicted class label of a data point with its actual class label. This comparison is repeated for the entire dataset (dataset), and the results of this comparison are compiled in a matrix or tabular format. This resulting matrix represents the confusion matrix (fig. below).

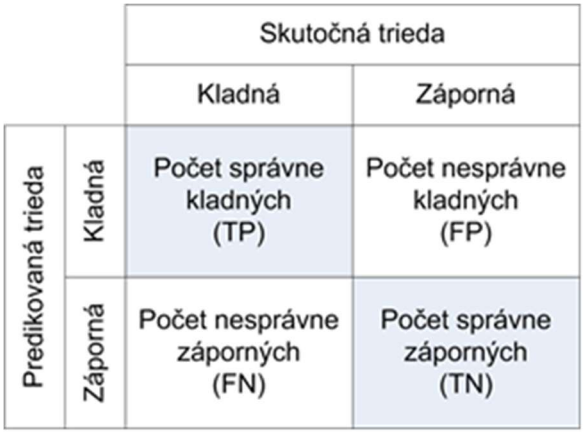


Fig. 3 Confusion matrix

In general, we usually have 2 classes – the positive class and the negative class. Based on this structure, we can clearly see four terms of importance.

• True Positive (TP) – Number of true positives: This is the sum of the total number of examples (data points from the dataset) from the positive class where the actual class label was equal to the predicted class label, i.e. the total number of cases in which we correctly predicted the positive class label using our model.

• False positive (FP) – Number of false positives: This is the number of total cases from the negative class that the model incorrectly classified by predicting them as positive. Hence the name, false positive.

• True Negative (FN) – Number of false negatives: This is the number of the total number of examples from the negative class where the actual class label was equal to the predicted class label, i.e. the total number of cases in which we correctly predicted the negative class label using our model.

• False negative (FN) – Number of false negatives: This is the number of the total number of cases from the positive class that our model incorrectly classified by predicting them as negative. Hence the name, false negative.

Accuracy

This metric is one of the most popular ways of measuring or evaluating the effectiveness of any classifier. It is defined as the overall accuracy or the proportion of correct predictions of the model. The formula for calculating it from the confusion matrix is as follows:

$$A = \frac{TP + TN}{TP + FP + TN + FN}$$

Precision

Precision (precision), also known as positive predictive value, is another metric that can be derived from the confusion matrix. This metric is defined as the number of predictions that are actually correct or relevant out of all predictions based on the positive class. The formula for this metric is as follows:

$$P = \frac{TP}{TP + FP}$$

A model with a high precision value identifies a higher proportion of the positive class compared with a model with lower precision. This metric becomes important in cases where we are more interested in finding the maximum number of positive classes, even if the overall accuracy decreases.

Recall

Recall (recall) is a model metric for identifying the percentage of relevant data points. It is defined as the number of instances of the positive class that were correctly predicted. It is also referred to as the coverage rate or sensitivity. The formula for calculating this metric is as follows:

$$R = \frac{TP}{TP + FN}$$

This metric becomes an important indicator of classifier effectiveness in scenarios in which we want to capture the largest possible number of cases of a particular class, even if this causes an increase in false detections/predictions. As an example, in the case of bank fraud, a model with a high recall value will provide us with a higher number of potential fraud cases. However, it will equally increase the triggering of alarms for most suspicious cases.

F1-Score

There are cases in which we want to achieve a balanced optimization of precision and recall. The F1-Score is a metric that is the harmonic mean of the precision and recall metrics and helps us to optimize the classifier for balanced precision and recall of prediction. The formula for calculating the F1-Score is as follows:

$$F = \frac{2.P.R}{P+R}$$

Results

SpamAssassin

Method	Accuracy	Precision	Recall	F1-score	Processing time (sec.)
Decision tree	0.9761	0. 9776	0.9774	0.9774	1.4
SVM	0.9928	0.9929	0.9928	0.9929	11.9
XGBoost	0.9928	0.9929	0.9928	0.9929	3.3
Log. Regression	0.9833	0.9833	0.9833	0.9833	1.2
Naive Bayes	0.8438	0.8624	0.8439	0.8330	1.3

SMS Spam

Method	Accuracy	Precision	Recall	F1-score	Processing time (sec.)
Decision tree	0.7585	0.7624	0.7585	0.7604	0.001
SVM	0.7706	0.7722	0.7706	0.7714	0.005
XGBoost	0.7826	0.7757	0.7826	0.7791	0.001
Log. Regression	0.7887	0.7709	0.7888	0.7795	0.001
Naive Bayes	0.7785	0.7733	0.7785	0.7759	0.001

Summary

On the basis of the experiments performed, it can be concluded that the most suitable method for detecting spam-type e-mail messages in terms of efficiency is the *XGBoost* method**.** For the "*SpamAssassin*" dataset, it achieves a spam detection success rate of 99%. This value is almost identical for both the Precision and Recall metrics, so both false positive and false negative detections of spam-type messages are very well covered. For this dataset, the *SVM* method also achieves almost identical results; however, the evaluation time in the case of the *SVM* algorithm is approximately 4x longer than for *XGBoost*. Nevertheless, it is still a process of a few seconds, so both methods can be deployed for real-time applications. For the "*SMS Spam*" dataset, the spam detection success rate dropped significantly to ~78%. The best results were achieved by the

Logistic regression method, although the difference between the *XGBoost* and *Logistic regression* methods is only tenths of a percent. In terms of the time needed for data prediction, the *SVM* algorithm again requires the longest interval. On the basis of the experiments performed, it can therefore be summarized that the most suitable method for detecting spam-type messages is the **XGBoost** algorithm.