Patterns for text classification (Part 1)

MICHAEL WEISS, Carleton University
SWARUPINI BATHULA, Carleton University
STEVEN MUEGGE, Carleton University
ALI NAZARI, Carleton University

This paper describes patterns for text classification. The patterns address problems related to creating machine learning models for a corpus of documents and describe common solutions to solve those problems. The target audience for these patterns includes developers who are familiar with basic machine learning algorithms, but do not have much experience with applying machine learning to text classification.

Categories and Subject Descriptors: D.2.11 [**Software Architectures**] Patterns; I.5.1 [**Pattern Recognition**] Models

General Terms: Design

Additional Key Words and Phrases: text classification, labeling data

ACM Reference Format:

Weiss, M. et al., 2019. Patterns for text classification (Part 1). HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 1 (October 2019), 9 pages.

1. INTRODUCTION

This paper is the first part of a pattern language for text classification. The patterns will address problems related to creating machine learning models for a **corpus** (see glossary at the end) of documents and describe common solution to solve those problems. The focus of the patterns in this paper is on **feature extraction**. Unlike **numerical** or **categorical features** often used in machine learning **models**, we need to preprocess textual data before we can use them as **features**. For example, we can extract features from a document by determining the frequency of the words in the document, or by looking for certain key phrases and words associated with sentiments.

Fig. 1 shows a map of the patterns that focus on feature extraction from textual data.[1] Arrows between the patterns indicate the order in which the patterns are typically applied.

Before we can extract features from a corpus of documents, we need to *Clean the Data*. For example, text may contain HTML tags, special characters, or URLs. In most cases, we want to remove those (HTML tags or URLs) or replace them (special characters). The next step is to separate each document into **tokens** (for example, words or phrases) using *Tokenization*. Using *Bag of Words*, we can then count the frequency of the tokens and obtain a set of potential features. Often, there will be too many features, and we need to reduce their number using a technique like *Rank*. We can also improve the output of *Tokenization* by *Normalizing* the tokens, removing *Stopwords* that are not relevant for the analysis, or by combining groups of consecutive tokens into *N-grams*.

[1] The patterns *Clean the Data*, *Normalization*, *Stopwords*, and *N-grams* will be described in a future paper.

Author's address: M. Weiss, Technology Innovation Management, Carleton University, SP 308, 1125 Colonel By Dr, Ottawa, ON K1S 5B6, Canada; email: michael_weiss@carleton.ca

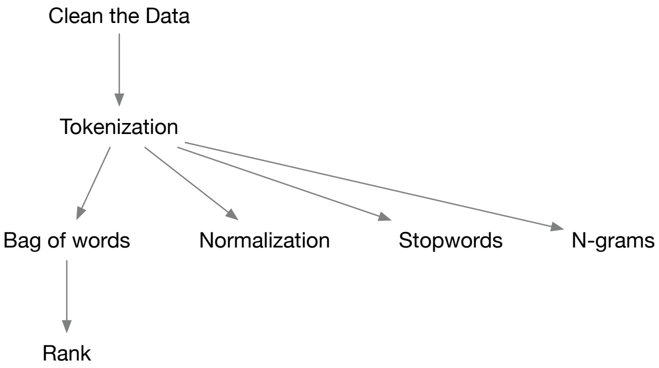


Fig. 1. Pattern map

The target audience for these patterns includes developers who are familiar with basic machine learning algorithms, but do not have much experience with applying machine learning to text classification. The "you" in the pattern description refers to those developers. The description follows the Alexandrian format: this means they consist of a context, followed by problem and forces, solution, consequences, known uses, and related patterns (visually separated by $* * *$'s).

2. EXAMPLE

To explain the use of the patterns, we will use a running example of creating a model that classifies issues in the bug tracker of an open source project by their type. Each entry in the bug tracker contains information about an issue, such as its title, its status, the date when the issue was created/modified, the developer assigned to it, a description of the problem, and a description of the solution, if the issue has already been resolved.

Many projects do not categorize issues. In order to be able to determine the likely type of an issue for such a project, we want to train a model on a set of issues from a project that does organize its issues by type. Fortunately, several large open source projects are quite diligent about labeling issues. As an example, we will be using the bug tracker of the Chromium project (`bugs.chromium.org`) to create our model.

Suppose, we want to be able to separate security-related from non-security-related issues. To identify security bugs, we can search for bugs within All issues that have the tag label=security. To obtain a list of non-security bugs, we can simply search for bugs that are not tagged as security bugs by using the search term -label=security. By scraping the issue tracker site, we can obtain a sample of security-related and non-security-related issues, which gives us labeled data to train our model. We could also limit our search to a time range and bug status.

Table I shows examples of security-related issues from the Chromium project. Table II, by comparison, shows examples of non-security-related issues.

Table I. Examples of security-related issues (just ID and summary are shown)

ID	Summary
798173	Use-of-uninitialized-value in SkMatrix::postConcat
798163	Security: privileged XSS in chrome-devtools://devtools/remote with old frontend (insufficient validation of remoteFrontendUrl)
798150	Crash in v8::internal::Invoke

Table II. Examples of non-security-related issues (just ID and summary are shown)

ID	Summary
798174	GLSL bug: s=vec2(1) , m=mat2(s,s) give wrong result
798172	Google Cloud Print should select the correct Google Drive account depending on the Gmail account
798169	Extension APIs should be also exposed under ‘browser.*’ to match the WebExtensions spec

We want the model to be able to classify a new issue into either one of those two categories based on just the summary of the issue. Of course, it is possible to include other features, for example, the name of the developer associated with the issue may be a good indicator of whether or not the issue is security-related, given the different level of security expertise developers have. However, we also want to create a model that we can apply to a different corpus of issues from another project. Thus, we do not want to use features that are project-specific.

We will also provide a working example of each pattern using the open source machine learning and data visualization platform Orange [Demšar et al. 2013], which has gained popularity among educators and practitioners[2]. However, this is just for illustration. The patterns do not in any way depend on the use of Orange. They could be just as easily created using Python and a machine learning library like Scikit-learn[3].

3. TOKENIZATION

You are creating a machine learning model for a corpus of documents. You have already cleaned the data.

* * *

Before you can create a model from a corpus of documents you need to divide each document into smaller units called tokens. How to extract these tokens depends on the language and your goals.

Usually, tokens are separated by spaces or punctuation. This is true for languages like English. In other languages (such as Japanese), you need to use a pre-trained language model to detect token boundaries.

However, sometimes it can be difficult to define what a token is, for example, the “words” in source code may contain numbers or capitalized letters.

* * *

Separate the documents into tokens based on delimiters, patterns, or language models.

In many natural languages like English or German, words are separated by spaces. In this case, we can use a space as a delimiter to split the text into tokens. Sometimes, more complex heuristics are required, for example, to extract numbers from a document or split variable names in a source code document into their components. In this case, a token can be defined by a pattern or regular expression. For example, the regular expression `\d+` matches whole numbers in the text.[4] As noted above, for languages like Japanese that do not have word boundaries, you may also need to use a pre-trained language model to tokenize the text.

As a straightforward example, consider the following document:

```
Google Cloud Print should select the correct Google Drive account
```

Splitting the text on the spaces produces this list of tokens:

```
Google, Cloud, Print, should, select, the, correct, Google, Drive, account
```

[2] https://orange.biolab.si

[3] https://scikit-learn.org

[4] To learn more about regular expressions or to experiment with specific regular expressions, there are various online resources like regular expressions 101 (https://regex101.com) with tutorials and editors for regular expressions.

A more complex example is a document that contains text, numbers and special symbols:

```
GLSL bug: s=vec2(1) , m=mat2(s,s) give wrong result
```

In this case, we want to use a pattern that extracts all text, but discards numbers and special symbols. The regular expression `[a-zA-Z]+` produces:

`GLSL`, `bug`, `s`, `vec`, `m`, `mat`, `s`, `s`, `give`, `wrong`, `result`

Sometimes, though, you want to retain special symbols. For example, for analyzing source code, symbols like + or = may carry special significance and you want to represent them by a token.

* * *

Subsequent steps of analysis have access to the significant components of the text (tokens).

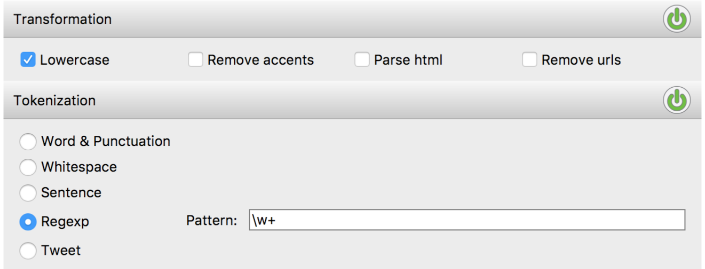
Some information is lost when converting the text into tokens (for example, punctuation or whitespace could be significant to some types of analyses).

* * *

Tokenization is a well-documented technique [Weiss et al. 2015]. A simple tokenizer can be created by using the split function provided by many string processing libraries to break the text on whitespaces. Yet, the tokens created in this simple manner will include punctuation marks.

The StandardTokenizer in Apache Solr splits text into its component words using punctuation symbols [Ingersoll et al. 2013]. OpenNLP's english.Tokenizer will also account for the grammatical roles words play [Ingersoll et al. 2013]. For example, it will split `can't` into `can` and `n't`.

In Orange (see note above), text documents can be pre-processed by applying regular expressions and various transformations (for example, changing the text to lower-case), as shown in Fig. 2.



Transformation

☑ Lowercase □ Remove accents □ Parse html □ Remove urls

Tokenization

○ Word & Punctuation
○ Whitespace
○ Sentence
● Regexp Pattern: \w+
○ Tweet

Fig. 2. Defining a tokenization pattern and transformations in Orange

* * *

Generally, tokenization is combined with *Normalization* to remove arbitrary differences between the tokens. For example, you usually want tokens to be lower-case.

Common words like "the" are often not relevant for further analysis. Use *Stopwords* to exclude such tokens, as well as other uninformative tokens from the documents.

Bag of Words is typically used next to convert a set of tokens into features.

To identify what should be included in a token (for example, is punctuation significant), validate the tokenization pattern by *Testing it on a Subset*[5] of the corpus.

[5] *Test on a Subset* will be described in the future as one of several general patterns for developing machine learning models.

4. BAG OF WORDS

You are creating a machine learning model for a corpus of documents. You already used *Tokenization* to extract the tokens from each document in the corpus.

* * *

To create features from the tokens in each document, you need to convert each document into a numerical format (a vector of numbers) that can be used as input to your model.

You cannot train a machine learning model on raw text.

Features should represent salient aspects of the document.

Features should allow the model to distinguish between documents.

Each token can appear multiple times in a document.

* * *

Measure how often (or, alternatively, just whether) each token appears in the text of each document in the corpus, and represent each document as a vector of word frequencies or occurrences.

The outcome is a table of the frequencies or, as a special case, occurrences (present or absent) of the tokens in the document. This is also referred to as a "bag" of words,[6] a collection in which each tokens can occur multiple times. Each token becomes a potential feature in the machine learning model. Its frequency is the value of the feature. Formally, the table of frequencies can be represented as a vector.

Consider the following example document:

```
This site is insecure and is harmful to visit error
```

The bag-of-words representation for this document is shown in Table III.

Table III.
Bag-of-words representation

Word	Frequency
this	1
site	1
is	2
insecure	1
and	1
harmful	1
to	1
visit	1
error	1

The basic bag-of-words approach equates the importance of each token with the frequency with which it occurs in a document. However, this only measures the local importance of a token. A more sophisticated approach would be to also account for the frequency of each token across the corpus of documents. We can also just record whether or not a token appears in the document, assigning a binary value (0/1) as its frequency.

Tokens that appear in most of the documents should not be given as much weight as tokens that are used only in only some of the documents. Such "rare" tokens are better for distinguishing documents from each other. Therefore, the frequency of the token is typically weighted with the *inverse document frequency*, which measures how relatively unique each token is in the corpus.

[6] While the bag can contain any kind of token, historically, the name "bag of words" has become common usage.

* * *

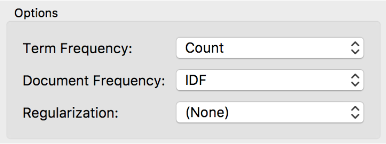
A bag of words can be considered a summary of the contents of the document.

However, neither word order nor grammar is preserved by the features.

* * *

The representation of a document as a bag of words was first used in the area of information retrieval [Weiss et al. 2015]. In this representation, each token or word is associated with its frequency in the document. The CountVectorizer in Scikit-learn[7] converts a collection of documents into a matrix of token frequencies. Each document is represented by a row in this matrix.

Fig. 3 shows how a bag-of-words model can be defined in Orange. Here, we count the term frequency in a document and weigh it with the inverse document frequency.



Options

Term Frequency: Count

Document Frequency: IDF

Regularization: (None)

Fig. 3. Defining a bag-of-words model in Orange

* * *

A bag of words creates a feature per unique token in the document. This can results in a large number of features. To reduce the number of features consider *Rank*.

Token position is lost in a bag of words. Sometimes you want to maintain it. To capture more of the context in which each token appears in the document consider *N-grams*.

N-grams is also useful when groups of tokens are semantically meaningful. For example, the words buffer and overflow often co-occur and combined form the concept buffer overflow.

5. RANK

You are creating a machine learning model for a corpus of documents. You used *Bag of Words* to extract features from the documents, but are left with a large number of features.

* * *

Simpler models are easier to understand and less likely to lead to overfitting the training data, so you want to reduce the number of features.

The more features there are, the more documents are needed to train the model.

Not all features are equally suitable for distinguishing documents from one another.

However, too few features will make the model unspecific.

* * *

Rank the features by their significance and focus on the top-ranked ones.

A common method for ranking features is to rank them by **information gain**. The greater the information gain of a feature, the better suited it is to distinguish between different classes of documents. You, therefore, want to choose the features with the greatest information gains. Other methods to rank features include statistical measures such as the Gini coefficient and chi-squared statistic.

[7] https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html

* * *

Focusing on the top-ranked features reduces the number of documents needed in your training set and reduces the computational effort it takes to train the model.

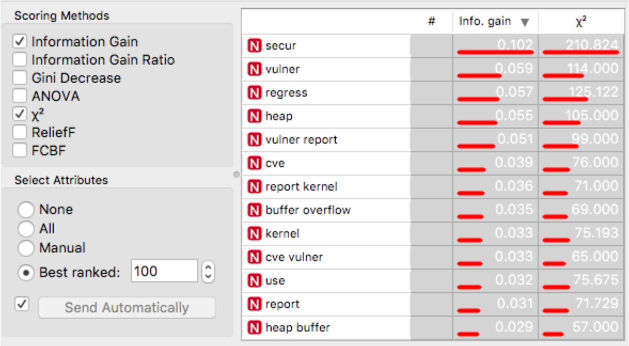
Excluding less significant features will make the model easier to understand.

To determine the right number of features, you need to experiment with different numbers of features to include and assess their impact on the quality of the model.

* * *

Ranking features according to their ability to predict the class of a document is a common technique for feature selection [Weiss et al. 2015]. The SelectKBest[8] class in Scikit-learn selects the k best features given a scoring function. Scikit-learn implements a range of scoring functions.

Fig. 4 shows how a ranking of features can be defined in Orange. Here, we rank the features by information gain and chi-squared. We can compare the impact of each scoring function and select which one we want to apply to the list of features, then choose the features we want to include in the model. Both of these measures seem to agree in this particular example, however, this is not always the case.



Scoring Methods

☑ Information Gain
☐ Information Gain Ratio
☐ Gini Decrease
☐ ANOVA
☑ χ^2
☐ ReliefF
☐ FCBF

Select Attributes

○ None
○ All
○ Manual
● Best ranked: 100

☑ Send Automatically

	#	Info. gain ▼	χ^2
N secur		0.102	210.824
N vulner		0.059	114.000
N regress		0.057	125.122
N heap		0.055	105.000
N vulner report		0.051	99.000
N cve		0.039	76.000
N report kernel		0.036	71.000
N buffer overflow		0.035	69.000
N kernel		0.033	75.193
N cve vulner		0.033	65.000
N use		0.032	75.675
N report		0.031	71.729
N heap buffer		0.029	57.000

Fig. 4. Defining a ranking of features in Orange

* * *

Eliminating features by ranking assumes that features are not interdependent: interdependent features may rank low individually, but could be significant when used in combination.

6. EXAMPLE RESOLVED

Let's apply the patterns to the example of classifying issue into security-related and non-security-related issues. The solution involves two stages (this paper focuses on the first stage):

—Identify features for training the machine learning model.

—Select a machine learning algorithm that best fits the initial corpus of issues (training data) and evaluate its performance on a corpus of issues from another project (test data).

[8] https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html

The corpus of issues contains both security-related and non-security-related issues scraped from the Chromium project website. Assume there you have collected 1,000 issues of each type and that the training data contains three fields for each issue: ID, summary, and type of issue.

The first step in identifying the features for the machine learning model is to import the corpus and *Tokenize* each issue summary. This is accomplished by the first three steps (from the left) in the Orange workflow in Fig. 5. Orange is a visual language for defining machine learning models. The nodes in an Orange workflow (such as Corpus) indicate input, output, and processing steps. Corpus loads the issues, Select Columns defines the type as the target variable to be predicted, and Preprocess Text separates the summaries into tokens.

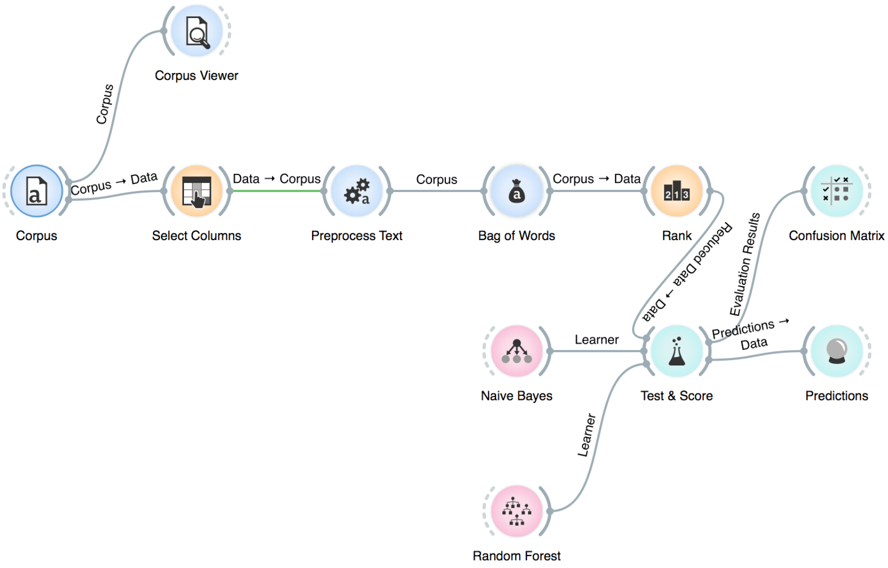


Fig. 5. Orange workflow for the example

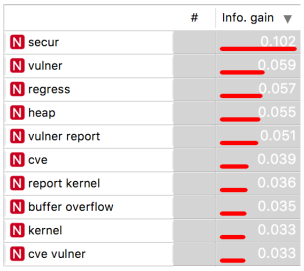
Subsequently, *Stopwords* are removed from the list of tokens that would otherwise limit the generalizability of the model. These include common English words and words like `android`, `chrome`, and `google` that would only be used frequently in describing issues for the Chromium project. Next, the tokens are *Normalized* by converting them to lower case and stemming them to common roots. To identify common combinations of tokens, we also opted to create *N-grams* of up to two tokens. All of this is accomplished as part of the Preprocess Text step.

The next step creates a *Bag of Words* from the preprocessed summaries. To emphasize distinctive tokens, we also weight their frequencies by their inverse document frequency. The features identified in this way are then *Ranked* by information gain, and we cut off after the first 100 features.

Fig. 6 shows a list of the 10 top-ranked features. As one would expect, keywords like `secur`, and `vulner`, and `cve`[9] used in a summary are indicative of security-related issues, but so are word combinations like `buffer overflow` (a common source of vulnerabilities).

These features can then be used to train and evaluate different machine learning algorithms. Here we compare Naïve Bayes (commonly used for text classification) and Random Forest.

[9] The term `cve` refers to Common Vulnerabilities and Exposures.



	#	Info. gain ▼
N secur		0.102
N vulner		0.059
N regress		0.057
N heap		0.055
N vulner report		0.051
N cve		0.039
N report kernel		0.036
N buffer overflow		0.035
N kernel		0.033
N cve vulner		0.033

Fig. 6. Ranked list of features for classifying issues

ACKNOWLEDGMENTS

I would like to express my thanks to my shepherd, Cecilia Haskins, and the participants in my writers' workshop for their generous feedback on earlier versions of this paper.

Glossary

Categorical feature. Categorical **features** have discrete values, for example, a feature sex with values male and female. They can also be created from **numerical features** by binning.

Corpus. In text analysis, a collection of documents is often referred to as a corpus.

Feature. A characteristic of the training data used to create a machine learning **model**. Features often do not exist in the raw data, but need to be extracted from it.

Feature extraction. The process of extracting **features** from the training data.

Information gain. Information gain is a statistical measure of how much information a feature provides about a class, that is, how much entropy or uncertainty it removes.

Model. In this paper, the term "model" refers to a machine learning model. A machine learning model is the outcome of training a machine learning algorithm such as Naïve Bayes.

Numerical feature. Numerical **features** are features like size or duration. **Categorical features** can be converted into numerical features through one-hot encoding.

Token. A token is the unit of interest in a text document. It is typically a word, but could be a phrase, a whole sentence, or any text that matches a specified pattern.

REFERENCES

Demšar, J., Curk, T., Erjavec, A., Č Gorup, Hočevar, T., Milutinovič, M., Možina, M., Polajnar, M., Toplak, M., Starič, A., Štajdohar, M., Umek, L., Žagar, L., Žbontar, J., Žitnik, M., and Zupan, B. 2013. Orange: Data mining toolbox in python. *Journal of Machine Learning Research 14*, 2349–2353.

Ingersoll, G. S., Morton, T. S., and Farris, A. L. 2013. *Taming Text: How to Find, Organize, and Manipulate It.* Manning.

Weiss, S. M., Indurkhya, N., and Zhang, T. 2015. *Fundamentals of Predictive Text Mining* 2 Ed. Springer.

PLoP'19, OCTOBER 7–10, Ottawa, Ontario, Canada. Copyright 2019 is held by the author(s). HILLSIDE 978-1-941652-14-5