Machine Learning with Applications 4 (2021) 100033

Contents lists available at ScienceDirect

Machine Learning with Applications

journal homepage: www.elsevier.com/locate/mlwa

Text categorization with WEKA: A survey

Donatella Merlini [a,*], Martina Rossini [b]

[a] Dipartimento di Statistica, Informatica, Applicazioni, Università degli Studi di Firenze, Italy
[b] Università degli Studi di Firenze, Italy

ARTICLE INFO

Keywords:
Classification
Clustering
Text categorization
Text Mining
WEKA

ABSTRACT

This work shows the use of WEKA, a tool that implements the most common machine learning algorithms, to perform a Text Mining analysis on a set of documents. Applying these methods requires initial steps where the text is converted into a structured format. Both the processing phase and the analysis of the transformed dataset, using classification and clustering algorithms, can be carried out entirely with this tool, in a rigorous and simple way. The work describes the construction of two classification models starting from two different sets of documents. These models are not meant to be good or realistic, but just illustrate how WEKA can be used for a Text Mining analysis.

1. Introduction

Text Mining is a term which generally refers to the automatic extraction of interesting and non-trivial information from text in an unstructured form; generally, its purpose is not to understand all or part of what is said by a particular speaker/writer, but rather extract patterns from a large number of documents. Text Mining is connected with Natural Language Processing (NLP), which includes linguistically inspired techniques, i.e., a text is typically analyzed from a lexical and syntactic point of view using a formal grammar, the resulting information is then interpreted semantically and used to extract information about what was said (Kao & Poteet, 2007). Named Entity Recognition (NER) is a standard task in NLP, consisting in searching and classifying named entities, i.e., portions of text of natural language documents that represent real world entities, such as names of people, places, data and companies (Konkol & Konopík, 2014; Nadeau & Sekine, 2007). NLP is a huge sub-field of artificial intelligence that deals with models and representations for natural language. A very common way to represent words, phrases, and documents in modern NLP involves the use of sparse vectors. These vector representations, called *word embeddings*, are typically learned from models based on neural network architectures. One of the most famous models used for learning vector representations of words is the *Word2Vec* technique (Church, 2016; Mikolov, Sutskever, Chen, Corrado, & Dean, 2013).

The applications in the field of Text Mining do not manage data from archives and databases but rather semi-structured or entirely natural language collections, composed for example of emails, blog posts, HTML files, posts in social media as Twitter and Facebook or other text documents.

Text Mining is an extremely broad field of study: some of its most common forms are topic tracking, automatic summary of one or more documents, Information Retrieval and text categorization; the latter, in particular, refers to the classification of texts written in natural language within thematic categories chosen from a predefined set. The algorithms that have the task of assigning such class labels are generally called classifiers. A recent survey on text categorization can be found in (Dhar, Mukherjee, Dash, & Roy, 2020), where an overview of some of the significant research works carried out in various languages to address the text categorization task is presented.

Categorization is a type of supervised learning — that is, the categories are known a priori for each document in the *training set*. Using an inductive process, we search for rules to distinguish each class from the others and then employ these directives to assign a category to new documents, which constitute our *test set*. Nowadays, this technique has numerous practical applications, including SMS and email spam filtering (Delany, Buckley, & Greene, 2012; Günal, S, Gülmezoğlu, & Gerek, 2006), sentiment analysis (Maks & Vossen, 2012; Singh, Singh, & Singh, 2017) and hate speech recognition (Mollas, Chrysopoulou, Karlos, & Tsoumakas, 2020). Among the most used classification algorithms in this context, we find Bayesian classifiers, decision trees, random forests, support vector machines and lazy classifiers, such as the k-nearest neighbors algorithm. There are also unsupervised learning techniques: in clustering, for instance, there are no predefined categories and the documents of the training set are not labeled in any way; we therefore try to use only the content of the texts to identify relationships between them. The groups obtained from these relationships are called *clusters* and the biggest problem of this technique is to obtain significant

The code (and data) in this article has been certified as Reproducible by Code Ocean: (https://codeocean.com/). More information on the Reproducibility Badge Initiative is available at https://www.elsevier.com/physical-sciences-and-engineering/computer-science/journals.

* Corresponding author.
E-mail addresses: donatella.merlini@unifi.it (D. Merlini), martina.rossini@stud.unifi.it (M. Rossini).

https://doi.org/10.1016/j.mlwa.2021.100033
Received 13 November 2020; Received in revised form 17 March 2021; Accepted 21 March 2021
Available online xxxx

ones without having additional information about the documents. In low-dimensional spaces, two well-known families of techniques are hierarchical and non-hierarchical clustering. The latter produce a data set partition and the former a hierarchical structure of partitions. Hierarchical clustering techniques are not feasible in high dimensional spaces, due to their computational burden. A clustering method often used for textual data is the well-known k-means algorithm (Balbi, 2010). Moreover, for many text classification problems, acquiring class labels for the documents in the training set can be extremely expensive, while gathering large quantities of unlabeled data is usually cheap. When the appropriate quantity of labeled documents is not available, the accuracy of a text classifier can be improved by augmenting its small training set with a large pool of unlabeled texts, thus combining supervised and unsupervised strategies (Nigam, Mccallum, & Thrun, 1998).

Since all these machine learning models are unable to process unstructured information as it appears initially, a typical Text Mining system contains some intermediate steps such as text preprocessing and feature extraction in which the text is brought into a more structured format. The techniques used in the preprocessing phase include, for example, the removal of the most common words of a language (such as articles and prepositions) and the reduction of terms to their basic form. The most common way to extract features from raw text data is the *Bag of Words (BoW)* encoding: documents are transformed into vectors in the high dimensional space spanned by words, which corresponds to the *vector space model* (Salton, Wong, & Yang, 1975). When a term occurs in the document, its value in the vector is non-zero. This latter quantity can be seen as a weight representing the importance of the term, i.e. how much the term contributes to explain the content of the document. A recent research explaining how to prepare a set of documents for quantitative analyses and compare the different approaches widely used to extract information automatically can be found in (Misuraca & Spano, 2020). The preprocessing and feature extraction steps will be better explored in Sections 2.1 and 2.2, respectively. Due to the high number of features that the *BoW* model usually generates, there is another phase which is extremely common in Text Mining tasks and usually precedes the actual model's application — that is, feature selection. It can be defined as the process of selecting a subset of the original features based on their importance (Shah & Patel, 2016) and it will be discussed in more detail in Section 2.3. The importance of text representation factors such as stop words removal, word stemming and weighting is illustrated in (Debole & Sebastiani, 2004; Song, Liu, & Yang, 2005; Uysal & Gunal, 2014). A complete overview of the proposed framework for the text categorization process can be found in Fig. 1.

Given the great possibilities offered by textual categorization and, more generally, by the discipline of Text Mining, over the years numerous tools and software packages that implement the most common algorithms in these fields and facilitate their application on new data have been developed; among these we find toolkits offering a visual experience like `Knime`, `Orange`, `RapidMiner` and `WEKA` but also a vast collection of specialized libraries for many different programming languages like the `Phyton` package `Scikit-Learn` or the `R` packages `Caret` and `RTextTools`. In particular, `WEKA` (Waikato Environment for Knowledge Analysis), available at https://www.cs.waikato.ac.nz/ml/weka/, is an advanced collection of machine learning algorithms and preprocessing techniques that has been designed to test different existing methodologies; it provides, among other features, several methods for transforming and preprocessing the input data and for making an attribute selection, as well as for classification, clustering and regression tasks and for a statistical evaluation of the resulting learning schemes. `WEKA` is a very user-friendly tool and it can also be easily interfaced with some of the most commonly used languages for machine learning tasks, such as `Java`, `Phyton` and `R`.

For what concerns textual data, `WEKA` provides some filters that allow to carry out the pre-computation phases and subsequently to apply the classification and clustering techniques in a quick and flexible,

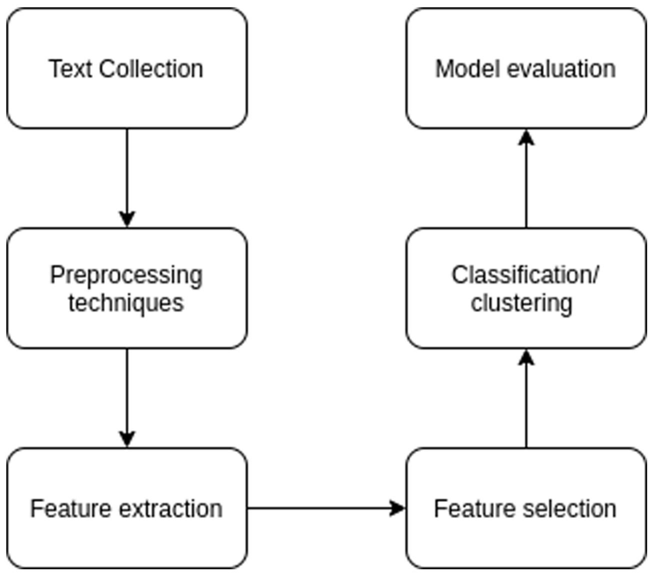


Fig. 1. Proposed framework for text categorization.

although not trivial, way. Unfortunately, the existing documentation on the subject is not very rich and finding one's way among the information available in the literature is not easy (see, however, the book (Witten, Frank, & Hall, 2011), the paper (Hall et al., 2009), the documentation at https://waikato.github.io/weka-wiki/documentation/ and the online course https://www.futurelearn.com/courses/more-data-mining-with-weka). A comparison of some classification algorithms for Arabic text categorization is illustrated in (Hmeidi et al., 2014), by using both `WEKA` and `RapidMiner`; a similar comparison, performed by `WEKA` on a benchmark dataset for text categorization, can be found in (Dan, Lihua, & Zhaoxin, 2013); a content-based citation analysis based on text categorization by `WEKA` is presented in (Taskin & Al, 2018).

`WEKA` offers several algorithms of both supervised and unsupervised types that can be applied to document datasets after the preprocessing phase. Clear descriptions of these algorithms ca be found in (Leskovec, Rajaraman, & Ullman, 2020; Tan, Steinbach, & Kumar, 2006; Witten et al., 2011); a general discussion of some of the most popular classification techniques and the corresponding `WEKA` implementations can be found in Section 4.

The goal of this paper is to illustrate the steps required to perform a text categorization analysis, using `WEKA` as the only tool; our starting point is the work described in the thesis (Rossini, 2020). We believe that, in addition to being a great analysis tool, `WEKA` is very useful in an educational context, because it allows students to become familiar with the most important machine learning algorithms and to deepen their theoretical aspects. The application of this gui tool is useful both in research and teaching and, in fact, it is used in many universities.

In Section 5, a first case study is presented concerning the problem of language identification, a relatively simple task but capable of providing important information for the construction of metadata relating to documents in international collections. As a second study, a simple classification task is illustrated in Section 6. The datasets are accessible at https://github.com/mwritescode/text-categorization-with-WEKA. The two studies allows us to describe the not trivial activities necessary to perform a text categorization analysis entirely with `WEKA`. In both cases, we used in particular the version 3.8.4 of the software.

2. Basic methodology for text categorization

In this section we describe the basic methodology for the text categorization process as depicted in Fig. 1.

2.1. Preprocessing techniques

The preprocessing methods are the first step in a Text Mining process and aim to transform the unstructured information of the text files into a structured and ordered form, which can then be interpreted by the machine learning algorithms (Song et al., 2005; Uysal & Gunal, 2014). Some of these techniques can also reduce the eventual noise present in a collection and the space necessary for storing it; it is in fact estimated that about 20%–30% of the total words in a document are stopwords, that is, terms that can be eliminated as they are repetitive and have no semantic value. In this section we simply summarize the most common practices used in this area, but a complete discussion can be found starting from (Gupta & Lehal, 2009; Kannan & Gurusamy, 2014; Manning, Raghavan, & Schütze, 2008).

2.1.1. Tokenization

Tokenization is the process of breaking text into words, phrases, symbols or other significant elements, called tokens. These terms can be single items (*1-gram*) or sets of them (*n-gram*); the items can be phonemes, syllables, letters, words or even sentences. Formally, a token is defined as a sequence of characters grouped together as they are considered a semantic unit useful for processing. The biggest problem when applying this technique is deciding which are the correct tokens to use; for example, one can think that it is enough to separate the words in correspondence of spaces and tabs and eliminate the punctuation marks but this, in reality, is not enough. Separating terms at whitespace also divides characters that should be considered a single token (i.e., city names like Los Angeles). Another problem is due to characters that, depending on the context, may or may not be token delimiters: for example, the period used as the end of a sentence is to be considered a delimiter but the same cannot be said for the period used to separate the part integer from the decimal in numbers. Finally, it must also be taken into account that the tokenization problem is language-specific. Considering the English language, for example, we can see that the hyphen is used for several purposes, including: to divide vowels into words, to join surnames and to indicate grouping of words. It is easy to understand that the first case should be treated as a single token and the last as separate tokens; in the second case, however, it is less clear how to behave. Similarly, consider the problem of apostrophes, used in the English language both for contractions and to indicate possession: it is clear that, if we simply chose all non-alphanumeric characters as delimiters, good results would not be obtained. Other languages pose different problems. The difficulties described here are just some of those that must be faced during the tokenization process, thus, it is often chosen to develop ad hoc tools based on the problem to be solved.

`WEKA` makes available a `tokenizer` that allows to choose among various tokenization techniques, based on words or n-grams. We will see this in Section 3.

2.1.2. Stemming and lemmatization

Stemming is a preprocessing technique that deals with reducing a word to its basic form, called *stem*. In this context it is not important that the chosen stem is a real word, it is enough that the morphological variants of the same term – which in most cases have similar semantic interpretations – are all mapped to the same stem. In classification problems, this procedure allows to reduce the number of distinct terms in the text and to increase the frequency of some of them, something that can improve the performance of the entire system. However, attention must be paid to two very common types of errors: over-stemming and under-stemming. There is a case of over-stemming when two words with different roots are mapped to the same stem; in this case we have a false positive. Instead, there is a case of under-stemming when two words that should be reduced to the same root are mapped to different stems; this error is also called a false negative. Stemming algorithms can be classified into three categories: truncation methods, statistical methods and mixed methods, as illustrated in (Jivani, 2011).

Lemmatization is a preprocessing technique similar to stemming; the goal is to reduce the morphological variant of a word to its *lemma*. The fundamental difference between the two methods lies in the fact that, in order to obtain the lemma of a term, it is reduced to its basic form after having analyzed its context and part-of-speech in the sentence in which it appears; instead, to get the stem of a word, some syntactic rules must be applied. As we have already seen, with stemming it is not necessary that the found base form be a real word. With lemmatization, however, what you get is the linguistically correct root of a term. Generally this technique is difficult to implement as it requires a vocabulary to refer to and a morphological analysis of the words. Furthermore, it is much more difficult to create a lemmatizer for a new language than a stemmer, as it requires a very thorough understanding of the language structure.

The `WEKA StringToWordVector` filter gives the possibility to choose among various stemmer algorithms.

2.1.3. Stopwords removal

The term *stopwords* refers to words that occur very often in the text but which do not contribute in any way to defining the context and content of the document; they are commonly articles, prepositions and pronouns. Generally we choose to remove these terms, not only to reduce the space required for storing the tokens, but also to improve the performance of the system, as they are traditionally considered useless for most classification tasks. The classic technique for removing stopwords involves relying on a pre-filled list, containing all those terms not considered semantically relevant for a given language. This approach for stopwords identification is sometimes called *static*. A number of stopword lists adopted as standard in many research works are available for many languages (Kaur & Buttar, 2018; Ladani & Desai, 2020).

As discussed in (Debole & Sebastiani, 2004; Kaur & Buttar, 2018; Lam & Ho, 1998; Misuraca & Spano, 2020), there are also several approaches to create a list of stopwords based on the weights assigned to the words. This can be seen as a *dynamic* approach, where the stopwords are identified on the go and not fixed apriorly (Ladani & Desai, 2020) and the words, or features, are selected in terms of their weights. Various term weighting techniques – among which *term frequency* and *inverse document frequency* feature prominently – will be discussed in Section 2.2 when dealing with feature extraction methods. For now, suffice to say that taking advantage of these measures, it is possible to choose as stopwords to eliminate, for example, all those words with a very high term frequency and/or with a low inverse document frequency. In the first case, in fact, we are dealing with terms that appear too often within the same document, probably providing minimal useful information about its content; in the second case, instead, the words appear in most of the texts of the dataset and therefore are not useful for categorization. Finally, case-folding can be useful, bringing all characters in the collection to lowercase. Note that these approaches to stopwords removal can also be seen as a very primitive form of feature selection (see Section 2.3 for more information).

2.2. Feature extraction

As already observed in the Introduction, the vector space model (Salton et al., 1975) is commonly used to represent a document d_i as a vector $(w_{i1}, w_{i2}, \dots, w_{in})$ in a n-dimensional vector space spanned by the terms belonging to the vocabulary. When a term j occurs in the document, its value w_{ij} in the vector is non-zero and this quantity can be seen as a weight representing the importance of the term in the document d_i. Several ways of computing these term-weights have been proposed in the literature, for example *binary weights*, *raw frequency weights* (tf), *normalized frequency weights*, *inverse document frequency* (idf) and *term frequency-inverse document frequency weights* ($tf - idf$). Binary weights just consider the presence or the absence of a term j in a document d_i by assigning to w_{ij} a value of 1 or 0, respectively. However, this binary representation is not efficient in some applications, and in these cases more sophisticated weighting schemes are needed. Since terms that frequently appear in the single documents usually have a good discrimination capacity, a measure of the term occurrence in each text is often considered. Raw frequency weights are calculated as the number of occurrences of a term in a document and correspond to the absolute frequency tf_{ij}, providing a way to estimate how well this term describes the content of the text under consideration. Normalized frequency weights incorporate $1/\max_{l=1\dots n} w_{il}$ as normalization factor of the raw frequencies, thus dividing by the highest number of occurrences observed in the document d_i. However, if terms with high frequency are not concentrated in a set of few documents but occur in the whole collection, then their predictive power lessens. Thus, also a measure of the presence of each term in the whole collection must be used. The inverse document frequency idf, indicates the inverse of the frequency of a certain term in all the documents of the dataset. It is an important indicator as it is possible to assume that a term which appears in all, or in many, documents to be classified is not particularly useful for distinguishing the texts of a category from those of another. If N is the total number of documents in the collection and n_j the proportion of documents in which the term j appears, idf_j can be obtained as

$$idf_j = \log \frac{N}{n_j}.$$

In $tf-idf$ schemes, the raw frequencies or the normalized frequencies are multiplied by idf. When the collections are composed of documents of very different lengths, the longer ones usually are associated with larger term sets and, thus, are favored in the classification. Therefore, often the $tf-idf$ weights are normalized by the Euclidean vector length of the document (Salton & Buckley, 1987), to avoid biases introduced by unequal document lengths, where the length is represented as the total number of tokens used in a document:

$$w_{ij} = \frac{tf_{ij} \log \frac{N}{n_j}}{\sqrt{\sum_{l=1}^{n} \left(tf_{il} \log \frac{N}{n_l} \right)^2}}.$$

Among other variants, the *logarithmic* tf, computed as $\log(1+tf)$, is another standard weight used in literature (Debole & Sebastiani, 2004).

As we will describe in Section 3, in WEKA there is the StringToWordVector filter to transform a set of documents into a set of numeric vectors, with a stopwordsHandler facility that allow us to choose among various stopwords lists or to specify a new one. There is also the possibility to set the minimum term frequency with the parameter minTermFreq, by using raw tf, logarithmic tf and $tf-idf$ weights, eventually normalized via the parameter normalizeDocLength.

Finally, note that the transformation of documents into vectors with the *BoW* approach does not consider the meaning, context or order in which the terms appear, but only if they are present in the document under analysis or not. Therefore, it is easy to understand that the model considers two documents with similar representation vectors as similar also in content, even if this is not always correct. However, several practical experiments have shown that more sophisticated approaches, which consider meaning and word order, can lead to worse performance. A review of syntactic similarity measures can be found in (Gali, Mariescu-Istodor, Hostettler, & Fränti, 2019), while semantic similarity measures are discussed in (Corley & Mihalcea, 2005).

2.3. Feature selection

The feature space of a text categorization problem usually has got a very high dimensionality and that is not always solved just by applying stopwords removal. The feature selection process lets us select a subset of our original feature set, eliminating the attributes that are considered to have limited predictive power. There are many different approaches to this task: wrapper methods, for instance, perform a search over all the possible subsets of the original feature set, evaluating the performance of a classifier over each one (Forman, 2003). However, they proved to be impractical for large scale problems and thus, are not widely used in text classification, where filter methods are favored: indeed, they are independent from the classifier and have a lower computational cost (Uysal & Gunal, 2014). Filter methods simply assign a score to each term in the initial feature space, according to a particular feature selection metric, and then select the k best attributes according to it. Filters that are commonly applied prior to using the feature selection metric include elimination of rare words and overly common words, as already discussed in Section 2.2; moreover, the common practice of stemming or lemmatization, merging various word forms such as plurals and verb conjugations into one distinct term, also reduces the number of features to be considered. Some commonly used metrics for score computation are document frequency, that measures in how many documents a word appears, information gain, F-measure and the common statistical test chi-squared. A complete analysis of the various measures and of their performances can be found in (Forman, 2003; Yiming & Pedersen, 1997). Feature selection include the extraction of new features which combine lower level features, that is words, into higher-level orthogonal dimensions. Feature extraction based on principal component analysis is discussed for example in (Lhazmir, El Moudden, & Kobbane, 2017; Uğuz, 2011), where orthogonal dimensions in the vector space of documents are found.

In WEKA, the Select attributes panel of the Explorer interface allows us to perform some of the previous tasks.

3. Preprocessing in WEKA

The acronym WEKA stands for Waikato Environment for Knowledge Analysis and indicates a software developed in Java by the University of Waikato in New Zealand and released under the GNU General Public license. This software is made up of a collection of algorithms for machine learning and tools for pre-processing and transforming data, including methods for discretization and sampling. A general description of the application can be found in (Witten et al., 2011).

When starting the software, the GUI chooser is displayed (Fig. 2), a window that allows the user to select one of the five WEKA interfaces for use - Explorer, Knowledge Flow, Experimenter, Workbench and Simple CLI; through this window it is also possible to access various other tools, such as the Package Manager and the visualization toolkits.

Since WEKA has constantly evolved over time, with the addition of new algorithms and features, it soon became necessary to remove some features from the main package so as not to unnecessarily complicate the initial experience; these features were then included in additional packages that can be installed and removed through the Package Manager. In addition to the official packages, reviewed by the WEKA team, there are also unofficial packages, that the user can choose to install by clicking on File/URL in the unofficial panel and then entering the address in the corresponding dialog box: one of these, for example, is graphviz-treevisualize, the package for the visualization of decision trees which was used to produce Fig. 3.

The main interface is the Explorer, which allows you to import data in various formats, visualize and pre-process it and finally analyze the data using classification, clustering and association rules mining algorithms. Together with the Experimenter, this is the interface that has been mainly used for this work.

The Explorer interface consists of six different panels, which can be accessed from the bar at the top left; each of them corresponds to a different Data Mining task supported by WEKA. In particular, these panels have the following tasks:

Waikato Environment for Knowledge Analysis
Version 3.8.4
(c) 1999 - 2019
The University of Waikato
Hamilton, New Zealand

Fig. 2. WEKA: GUI Chooser.

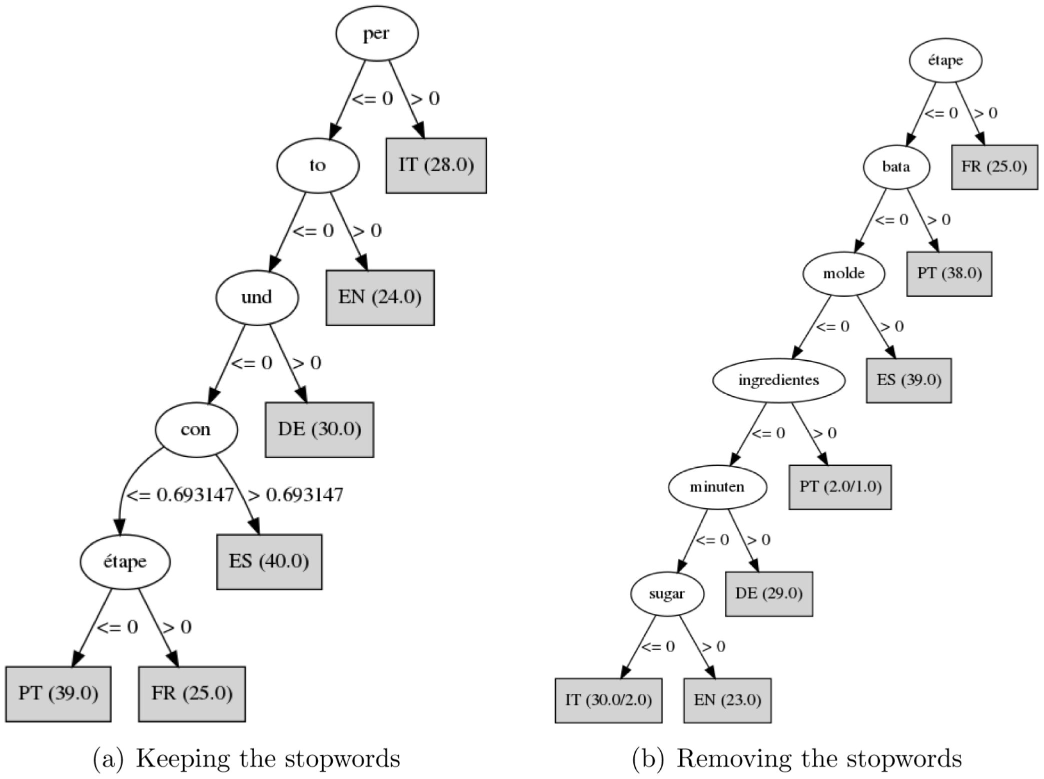


Fig. 3. Decision trees generated by J48 for the language identification problem.

- `Preprocess`: allows you to choose the datasets, view their attributes and modify them in various ways;
- `Classify`: allows you to apply machine learning schemes and evaluate their performance;
- `Cluster`: used to identify significant groups (clusters) in a dataset;
- `Associate`: allows you to find association rules for the data in the collection and evaluate them;
- `Select attributes`: allows you to select the most relevant features of the dataset;
- `Visualize`: allows you to view various two-dimensional graphs relating to the dataset in question and to interact with them.

Along the top of the `Preprocess` interface there are buttons for opening files, databases or urls. As for the first option, WEKA's native data type is ARFF, but other formats are also supported, including CSV, JSON and XRFF; as we will show below, there is also an option to acquire the contents of a folder of `.txt` files: each document becomes an instance of the dataset, with a string type attribute that collects its contents and with a class label equal to the name of the subfolder it belongs to. Table 1 illustrates a sample ARFF file containing a dataset consisting of a set of texts to which a class has been associated.

WEKA offers some tools particularly useful for text classification purposes:

Table 1
A sample ARFF file with text data.

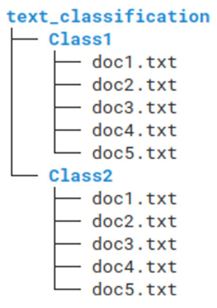
```
@relation text_data
@attribute text string
@attribute class {class1,class2}
@data
'text1', class1
'text2', class1
'text3', class2
'text4', class2
```

- the TextDirectoryLoader, to load each .txt file of a folder as an instance;
- the StringToWordVector filter, to transform each text of the dataset into a feature vector.

Assuming that each document in the training set is a separate .txt file and that these files are all located in the text_classification folder, in order for this to be imported into WEKA via the TextDirectoryLoader its structure must reflect the one shown in Fig. 4. Note that each subfolder groups the documents belonging to a category and that, in fact, the name of this folder will become the class label for them in WEKA. Once the files that make up the collection have been properly organized, they can be imported into WEKA: from the Preprocess panel of the Explorer, select Open file, look for the text_classification folder and click Open. At this point an error message is displayed as WEKA is not able to independently determine the selected file type; by clicking OK you access another window: from here, using the Choose button you can choose the TextDirectoryLoader as the data acquisition mode (Fig. 5). This mode allows you to specify the encoding with which to read the files; moreover, through the outputFilename option, you can choose whether to save the file names in an additional attribute or to ignore them. Once this procedure has been completed, the dataset has been correctly imported into WEKA; it consists of only two attributes: text, of type string, in which the contents of each file are stored and @@class@@, which is the class attribute and has as values the names of the subfolders. Note that WEKA inserts the @ symbol in the name of this attribute to prevent ambiguous situations: an English language document could in fact contain the word class inside it, which would become a feature of the dataset during a word-vector conversion, with the same name as the class attribute. Manual conversion to ARFF files is better suited for small files, while using the TextDirectoryLoader is better for long and numerous texts. We point out that WEKA does not admit newline in string-type attributes in an ARFF file, so in the case of large files these must first be processed in order to remove all newlines, the TextDirectoryLoader takes care of them automatically instead. Note that manual conversion to ARFF file can be problematic even for short sentences because you have to be careful of characters like quotes, which WEKA uses to delimit strings. Finally, a dataset with cases to predict needs to have the same structure that the dataset used to learn the model. The difference is that the value of the class attribute is ? for all instances.

It is now necessary to convert the data from natural language to a more structured format; this is possible by applying the StringToWordVector filter, which implements many of the preprocessing algorithms described in Section 2.1. Once this filter has been selected, clicking on the box next to the Choose button opens a screen that allows you to configure its parameters; the possible options are many and are described in detail in Table 2.

Once these parameters have been set correctly, the filter is applied using the Apply button in the Preprocess panel; the new dataset can be inspected using the Edit button and we can also delete some attributes using the Remove button, if we believe they are not very significant, or apply other filters. The dataset thus obtained is ready to be processed by a machine learning algorithm.



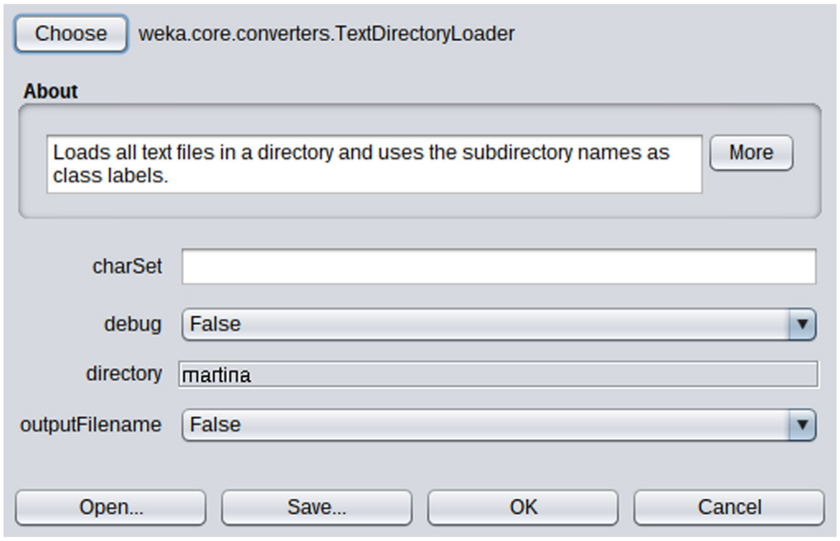
```
text_classification
├── Class1
│   ├── doc1.txt
│   ├── doc2.txt
│   ├── doc3.txt
│   ├── doc4.txt
│   └── doc5.txt
└── Class2
    ├── doc1.txt
    ├── doc2.txt
    ├── doc3.txt
    ├── doc4.txt
    └── doc5.txt
```

Fig. 4. Example of a text directory structure.

A more accurate investigation of the behavior of the various learning schemes typically requires running different classifiers on different collections and with different initial parameters; although it is possible to carry out this experiment using Expolorer, this is complex and repetitive. To cope with this problem, the Experimenter allows the user to set up large-scale experiments, run them and then analyze the results, automating the experimental process. This interface consists of three panels, Setup, Run and Analyse, the first of which is shown in Fig. 6.

To start a new experiment, use the New key at the top right; the other two buttons on the same line are used to save a configuration and to open a previously stored one. The box immediately below them is then used to specify the path and the format of the file in which to save the results, so that they can eventually become the subject of new Data Mining activities. In addition, the Experiment Type box allows you to specify whether to use cross-validation or hold-out as the validation and to define the type of task to be performed (classification or regression); Iteration Control instead allows you to choose the number of repetitions of the experiment. Using the boxes immediately below it is then possible to specify the datasets on which to perform the experiment and the algorithms to be used for it. We note explicitly that it is not possible to read a directory as it is in Explorer with the TextDirectoryLoader. To modify the parameters of one of the methods already entered, select it in the list and then click on Edit selected. Finally, the Experimenter also has an advanced mode, which can be accessed by selecting Advanced from the drop-down menu located near the top of the page; this option increases the number of parameters that can be changed to control the experiment, allowing for example to generate learning curves, to experiment using clustering algorithms and to divide the work across multiple machines. To start an experiment it is then necessary to go to the Run panel, which contains only the Start and Stop keys and a log box, where any error is communicated; once the execution is complete, to analyze the results, go to the Analyse tab (Fig. 7).

To view the data produced by the experiment you just performed, use the Experiment button at the top right; otherwise it is also possible to load a file that contains the results of a previous test. By then clicking on Perform test in the Actions box, you can compare the performance of the first classifier in the list with those of the others; the results of this comparison are shown as a table in the space labeled with Test output. Note that by default the test is performed on the error rate, but it is possible to change this metric by clicking on the Comparison field: the values obtained by each method for the chosen measure are then displayed in the corresponding boxes of the table; the symbol possibly located next to them indicates if the result is statistically better (v) or worse (*) than that obtained by the reference classifier. You can also use the Basic Test menu to change the reference method for the analysis; in addition to the other classifiers on which the experiment was carried out, this menu also allows you to choose Summary and Ranking: the first option compares all the learning schemes with each other and returns a matrix whose cells contain the number of datasets on which a model was better than the


Choose weka.core.converters.TextDirectoryLoader

About

Loads all text files in a directory and uses the subdirectory names as class labels.

charSet	
debug	False
directory	martina
outputFilename	False

Fig. 5. WEKA: `TextDirectoryLoader`'s setup.

Table 2
`StringToWordVector`'s parameters.

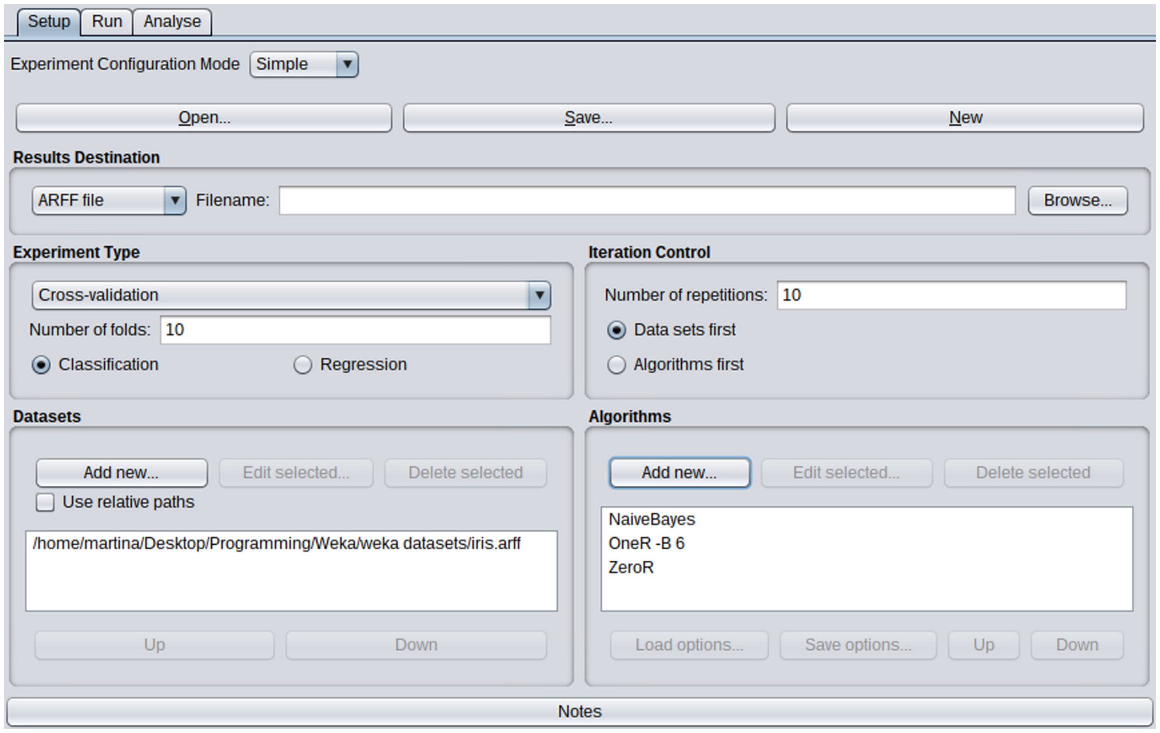
Parameter	Purpose
`IDFTransform` and `TFTransform`	allow you to decide whether the term frequency should be transformed into $f_{ij}\log(\frac{N}{n_i})$ and $\log(1+f_{ij})$, respectively, where f_{ij} is the frequency of word i in document j, N is the number of documents and n_i the number of documents with word i (default values `False`)
`attributeIndices`	allows you to specify the range of attributes to which to apply the filter; `first` and `last` are also valid values to indicate the first and last attribute of the dataset without having to specify the index
`attributeNamePrefix`	allows you to choose a common prefix to add to the name of each feature; this is useful to avoid any instance of naming ambiguity
`dictionaryFileToSaveTo`	allows you to specify a file in which to save the vocabulary produced
`doNotOperateOnPerClassBasis`	when set to `True`, the maximum number of words saved and the minimum term frequency are not calculated based on the class but on the entire dataset (default value `False`)
`lowerCaseTokens`	allows you to convert all words into lowercase characters before processing the text (default value `False`)
`minTermFreq`	allows you to specify the minimum value allowed for the term frequency (default value 1)
`normalizeDocLenght`	allows you to decide whether to normalize the frequencies of the words or not (default value `No normalization`)
`outputWordCounts`	when set to `True`, the algorithm uses the number of times the words appear in the text instead of limiting itself to the Boolean value to indicate their presence, 1, or absence, 0 (default value `False`)
`stemmer`	allows you to choose a stemming algorithm to use
`stopwordsHandler`	allows you to choose how to manage stopwords; if nothing is specified, they are not removed
`tokenizer`	allows you to choose the algorithm for tokenization
`wordsToKeep`	allows you to decide how many words to keep, per class if the option `doNotOperateOnPerClassBasis` is set to `false`, otherwise in total.

others, while the second sorts the algorithms according to the number of collections on which they performed better and then prints this list on the screen. Finally, note that the `Analyse` panel output can be saved to a file using the `Save output` button.

4. Supervised and unsupervised classification

Text categorization can be formally defined as the task in which a Boolean value must be associated with the pair $(d_j, c_i) \in D \times C$, where D is a collection of documents and C is a predefined set of categories. If d_j is considered to belong to category c_i, the pair (d_j, c_i) must be assigned the value *true*, otherwise the value $false$. More precisely, what needs to be done is to approximate the unknown target function $\phi : D \times C \to \{true, false\}$ with the function $\bar{\phi} : D \times C \to \{true, false\}$, called a classifier. This approximation must be obtained by exploiting only the information contained in the documents of the collection, without having access to any type of metadata. As already mentioned, categorization is a type of *supervised learning*, where the categories are known a priori for each document in the *training set*; an inductive process generate rules that allow the various classes to be distinguished and then these rules are used to assign the category to which the new documents, which constitute the *test set*, belong.

The different strategies through which test and training sets can be selected from a single text collection are collectively known as *validation schemes*. The simplest of these uses the entire textual collection both to train the model and to test it; obviously, it records overly optimistic performances. A more sensible scheme is called *hold-out evaluation* and involves dividing the initial dataset into two parts, with the training set usually being about twice the size of the test set. Although this method is simple to apply, it has some important disadvantages: the performance evaluation is in fact carried out only on a small part of the data and it may happen that the instances contained in the test set are too simple or too difficult to classify compared to those of the rest of the collection, thus distorting the results obtained. Also, depending on how the split is done, some instances that are critical to build the model may not be part of the training set. To try to



Experiment Configuration Mode: Simple

Results Destination: ARFF file, Filename:

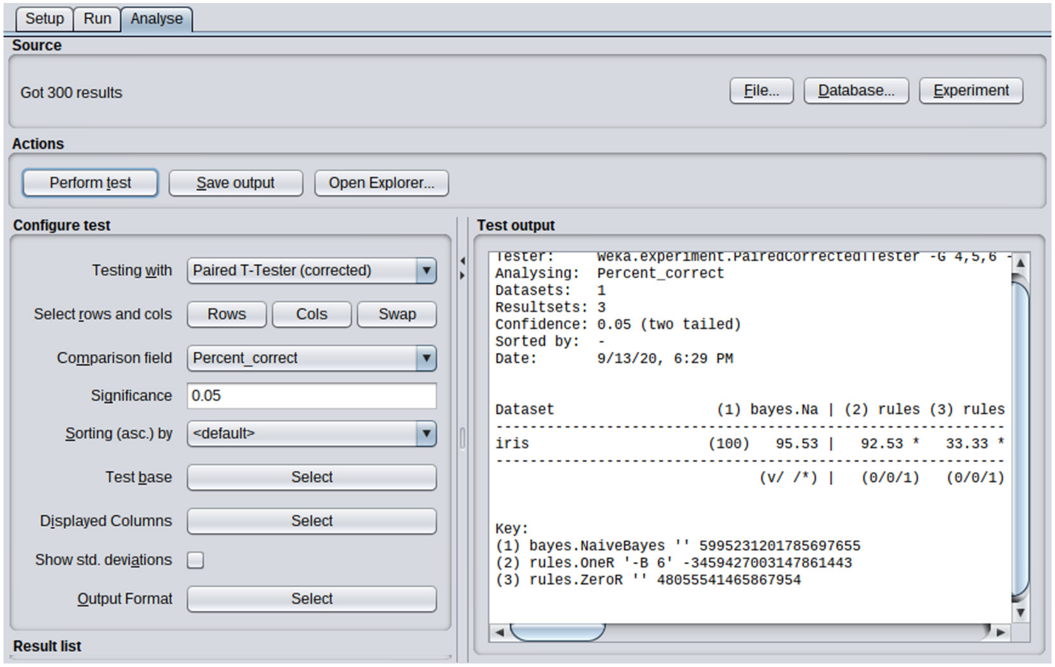
Experiment Type: Cross-validation
Number of folds: 10
(● Classification) Regression

Iteration Control
Number of repetitions: 10
(● Data sets first) Algorithms first

Datasets
☐ Use relative paths
/home/martina/Desktop/Programming/Weka/weka datasets/iris.arff

Algorithms
NaiveBayes
OneR -B 6
ZeroR

Fig. 6. WEKA: Experimenter in Setup panel.



Source: Got 300 results

Configure test
Testing with: Paired T-Tester (corrected)
Comparison field: Percent_correct
Significance: 0.05
Sorting (asc.) by: <default>
Show std. deviations: ☐

Test output

```
Tester:     weka.experiment.PairedCorrectedTTester -G 4,5,6 -
Analysing:  Percent_correct
Datasets:   1
Resultsets: 3
Confidence: 0.05 (two tailed)
Sorted by:  -
Date:       9/13/20, 6:29 PM


Dataset                   (1) bayes.Na | (2) rules (3) rules
------------------------------------------------------------
iris                     (100)   95.53 |   92.53 *   33.33 *
------------------------------------------------------------
                               (v/ /*) |   (0/0/1)   (0/0/1)


Key:
(1) bayes.NaiveBayes '' 5995231201785697655
(2) rules.OneR '-B 6' -3459427003147861443
(3) rules.ZeroR '' 48055541465867954
```

Fig. 7. WEKA: Experimenter in the Analyse panel.

solve these problems, a more systematic approach has been developed, known as k-fold cross-validation, which consists in dividing the data into k equal parts and classifying each of them using a model trained on the remaining $k - 1$ (Aggarwal, 2014).

Given a training set, supervised classification procedures build the rules for assigning a new document (object) to a category; on the other hand, *unsupervised classification* algorithms do not need any background knowledge and search data to be partitioned in groups (clusters). Such techniques do not use predefined categories and the documents of the training set are not labeled in any way: only the content of the documents is used to identify relationships between them. The groups obtained from these relations are called clusters and the biggest problem of this technique is to obtain significant ones without having additional information about the objects.

In the following paragraphs we explore several well known supervised and unsupervised algorithms which are often applied to text classification problems, highlighting their particular implementation in WEKA. Clear descriptions of the algorithms described below and many other, including classification methods based on neural networks, can be found in (Leskovec et al., 2020; Tan et al., 2006; Witten et al., 2011).

Less conventional methods based on fuzzy logic and deep learning are discussed in (Dhar et al., 2020). Note that, in addition to the algorithms we will present, WEKA offers several other classification and clustering models. Finally, we illustrate some of the most common metrics to evaluate the performance of the algorithms.

4.1. Bayesian methods

Among the supervised classification methods we find *Bayesian methods*, probabilistic models based on the application of the *Bayes theorem* which are known, in particular, for their simplicity and efficiency. The Bayes Theorem provides a simple formula for calculating the *posterior probability* which, in the case of categorization, can be interpreted as the probability that document d_j belongs to class c_i given the vector of its features x. In the field of text categorization, reference is often made to *naive* Bayesian classifiers, methods that assume a conditional independence between the characteristics (features) of the dataset. However, this hypothesis does not reflect reality; frequently, in fact, the occurrences of two or more terms in a document are interrelated — just think, for example, of the words *machine* and *learning* in texts dealing with machine learning. Despite this, the naive hypothesis allows you to estimate the parameters for each feature independently of the rest, greatly simplifying the construction of the model without excessively penalizing its performance. In particular, the *Bernoulli naive Bayes* model assumes that all features are binary; therefore, the kth element of the representation vector $x = (x_1, x_2, \dots)$ can only assume two values, 1 if the term x_k appears in the document and 0 otherwise. The *multinomial naive Bayes* model instead considers the frequency of words in documents. Therefore, within the representation vector x of a generic document, the weight x_k represents the number of times that this term appears in the text; more precisely, often this is measured using the term frequency. These models have been successfully used in a number of practical experiments throughout the years, including classification of app review topics (Maalej & Nabil, 2015) and spam detection (Zhang & Li, 2008); WEKA has several Bayesian classifiers, in our experiments we used in particular the `NaiveBayesMultinomial` classifier illustrated in (Mccallum & Nigam, 1998).

4.2. Decision tree models

Another important classification technique is that based on the construction of a *decision tree*, that is, a simple direct tree with root, in which each internal node corresponds to a partitioning rule and each leaf corresponds to the predicted class. In order for an instance to be classified, it must cross the tree starting from its root: once it reaches an internal node, the data is directed to one of the child nodes following the corresponding splitting rule and the path continues like this until a leaf is reached. Being as easy to interpret as they are to use, models of this type are among the most intuitive tools for text classification. Once all the possible splitting rules have been determined, it is necessary to define a metric to evaluate their goodness. Since the ideal test is the one that creates pure partitions, with instances that all belong to the same class, the rule that minimizes the value of impurity after the split is considered to be the best. Among the best known impurity functions we find *entropy* and *information gain*. The process of building a decision tree continues as long as the impurity function continues to report some improvement after the split. This greedy strategy, however, risks causing *overfitting*, a phenomenon in which increasing the complexity of the model causes a decrease in the error rate on the training set and, at the same time, an increase in the same value on the test set. To reduce the size of the tree obtained and, at the same time, limit overfitting, a technique called pruning is therefore used, which involves replacing some sub-trees with leaf nodes or with simpler sub-trees, maintaining an acceptable accuracy in the classification. Decision trees are very common classification algorithms, which are also extremely easy to interpret; they have thus been used in many different domain, including text classification (Maalej & Nabil, 2015; Pawar & Gawande, 2012). Among the best known algorithms for the construction of decision trees we find the C4.5 algorithm which uses the normalized information gain, the gain ratio, as a partitioning criterion, ends the construction of the tree when the number of instances to divide falls below a certain threshold and uses pruning techniques. In Sections 5 and 6, we used J48, which is the WEKA implementation of C4.5 algorithm (Quinlan, 1993).

4.3. Instance based learning

Most classification methods first build a global model from the data in the training set and then use it to categorize specific instances of the test set; this two-step modality is also known as *eager learning* as the models are built before it is known which examples they will have to classify. In *instance-based learning*, however, given the entire training set and known the instance to be classified, a local model is built based on the most relevant examples for it. This mode is also known as *lazy learning*, as most of the processing is not done in advance, but only in the classification phase, when the instance of the test set becomes known. The best known instance-based learning algorithm is the *k-nearest neighbors*, which uses the k closest neighbors to the instance i to be classified to build a local model for it; in the simplest case, this model trivially consists in choosing as a class for i the one assumed by the majority of its neighbors. However, this approach may not be appropriate for unbalanced datasets; in these cases, in fact, the rare class may not be sufficiently present among the closest neighbors of i, even if i is, indeed, part of it. It is therefore advisable to assign weights to the instances to reflect the class distribution in the collection before moving on to determine the most frequent category. One of the most important traits of this classifier is that it can be used for almost any type of data, as long as it is possible to define a distance function for its objects. Despite this advantage, the k-nearest neighbors is not without problems; in particular, computational efficiency represents a great criticality, as obtaining the closest k neighbors can have a linear cost in the size of the dataset. To cope with this problem it may be useful to build indexes and other data structures capable of facilitating the process of searching for the neighbors. Typically, the distance function adopted by the k-nearest neighbors for numeric attributes is the Euclidean norm.

The k-nearest neighbors classification is a widely used techniques for text classification (Bijalwan, Kumar, Kumari, & Pascual, 2014; Pawar & Gawande, 2012; Trstenjak, Mikac, & Donko, 2014). In WEKA, the implementation of the algorithm as illustrated in (Aha, Kibler, & Albert, 1991) is called IBk.

4.4. Support vector machines

Support vector machines or SVMs were introduced by (Vapnik & Chervonenkis, 1964) but, despite the solid theoretical background, they were initially underestimated by the scientific community, as they were deemed unsuitable for practical applications. This conviction was however disproved when the SVMs demonstrated excellent results on reference datasets in fields such as computer vision, text categorization and digit recognition. Nowadays, these models are able to obtain results comparable to those of Neural Networks and other statistical methods on the most common benchmark problems. A crucial notion for understanding the way these machines operate is that of a *linearly separable collection*. Suppose you have a two-dimensional dataset with only two possible class labels; it is said to be *linearly separable* if there is a line that clearly divides the two classes from each other. Note that this concept can be easily generalized to three-dimensional collections, in which a dividing plane is sought instead of a straight line, but also to n-dimensional datasets, in which it is necessary to find a hyperplane that separates the two classes. The purpose of SVMs is to orient this hyperplane so that it is as far as possible from the closest examples of each class. The use of SVMs for learning text classifiers can be found for example in (Ji et al., 2012; Joachims, 1998; Luo, 2021); WEKA implements the SMO (Sequential Minimal Optimization) algorithm for training a support vector classifier; a more precise description of the algorithm can be found in (Platt, 1998).

4.5. Random forest

There are also techniques known as *ensemble methods* which aggregate the predictions of different classifiers with the aim of improving the accuracy of the classification. An ensemble method constructs a set of base classifiers from training data and perform classification by taking a vote on the predictions made by each base classifier. In particular, the class can be obtained by taking a majority vote on the individual predictions or by weighting each prediction with the accuracy of the base classifier. Among these methods, *Random forest* is a class of ensemble methods designed specifically for decision trees: multiple trees are generated from a set of vectors sampled independently and with the same distribution from the original training data set and then combined to obtain the final classification. Examples of the method's application in text classification can be found in (Gaikwad & Halkarnikar, 2014; Usman, Ayub, Shafique, & Malik, 2016); WEKA makes available the implementation RandomForest of the algorithm described in (Breiman, 2001).

4.6. Clustering

As already observed, clustering is an unsupervised classification technique. In particular, a *partitional clustering* is simply a division of the data into non-overlapping subsets such that each data object is in exactly one subset. If we permit clusters to have subclusters, then we obtain a *hierarchical clustering*, which is a set of nested clusters that are organized as a tree.

In Data Mining and Text Mining, a simple partitional algorithm which is often considered the parent of other proposed models is k-*means* (MacQueen, 1967), which partitions a set of n points lying in a d-dimensional space into k non-overlapping groups. First, k initial centroids are chosen. Each object is then assigned to the closest centroid and each collection of objects assigned to a centroid is a cluster; distances can be calculated with different measures depending on the type of objects. The centroids are then updated based on the objects assigned to the cluster. The assignment and update steps are repeated until centroids do not change. The k-means algorithm has a number of variations, for example, different ways of choosing the initial centroids. The success of the algorithm is due to its simplicity and speed, although it often converges to a local solution. The WEKA implementation SimpleKMeans allows also to use the variation of the algorithm described in (Arthur & Vassilvitskii, 2007).

The most common approach for hierarchical clustering is *agglomerative*, i.e., we start with the objects as individual clusters and, at each step, merge the closest pair of clusters. This requires defining a notion of cluster proximity, which is the key operation. Among the various methods proposed, the *single link* and *complete link* strategies correspond to define cluster proximity as the proximity between the closest and the longest two points that are in different clusters, respectively; the *average link* strategy, instead, defines cluster proximity to be the average pairwise proximities of all pairs of points from different clusters. In WEKA the implementation HierarchicalClusterer admits various link strategies, including the ones described. Applications of clustering algorithms for text classification can be found in Aly and Kelleny (2014), Fung, Wang, and Ester (2005), Singh, Tiwari, and Garg (2011).

Table 3
Predicted versus true classes.

		Predicted	
		Pos	Neg
Actual	Pos	TP	FN
	Neg	FP	TN

4.7. Evaluation metrics for classification

In the following case studies we used different metrics to evaluate the models' performance: in particular, accuracy, precision, recall, F-measure, AUC and Kappa statistic. Accuracy can be simply defined as the number of correctly classified instances over all the instances in the test set, while Kappa statistic is a measure that can compensate for classification that may be due to chance or random effects. Both scale really well from binary to multi-class classification problems.

Given a binary classification task, we can call the first class label the *positive class* and the other the *negative class*. Then the predictions of a classifier can be divided into True Positives, False Positives, True Negatives and False Negatives, according to Table 3.

Precision can then be defined as the rate of the true positives amongst all the elements classified as positive, while recall can be thought of as the rate of the true positives over all the elements that actually are of positive class. F-measure is simply the harmonic mean of precision and recall.

$$precision = \frac{TP}{TP+FP} \qquad recall = \frac{TP}{TP+FN}$$

Note that an immediate extension of these metrics to the multi-class problem is given by simply computing the score individually for each class label and then returning a weighted average over all classes. In the Experimenter such a behavior is obtained through Weighted_avg_IR_precision, Weighted_avg_IR_recall and Weighted_avg_IR_F_measure. Lastly, AUC stands for *Area under the ROC Curve*, where a ROC curve is an evaluation metric particularly suited for those classifiers that return probability estimations for each class instead of a single value. The curve is obtained by choosing different probability thresholds and plotting the True Positive rate against the False Positive rate in a two dimensional graph. The area between this curve and the x axis is the AUC. This score can be computed for each class label separately and then averaged to get an overall value; in WEKA's Experimenter a weighted average of this metric over all the classes can be computed using Weighted_avg_area_under_ROC. A more detailed explanation of each of the mentioned metrics can be found in (Aggarwal, 2014).

5. A first case study: language identification

Language identification is a simple but popular application of text classification techniques, which can help generate metadata for documents in international collections. Moreover, many *Natural Language Processing* systems need to know the language of the documents they work with: using a model trained on English data to classify French documents can lead to a dramatic decrease in performance. Thus, language identification is often used as a preliminary step in this kind of applications.

At the moment there are two state of the art approaches which aim to solve this problem using machine learning techniques. The first one looks for the most common words of each language in the training set and compares them to the most common words in the document it is trying to classify. The second approach does something similar but using common n-grams instead of common words, where an n-gram is a portion of a bigger string that is exactly n characters long (Cavnar & Trenkle, 1994; Witten, 2004). Usually white spaces are added at the beginning and at the end of each word in order to distinguish if an n-gram occurs at the extremities of the word or in the middle of it.

Table 4
Some of the instances in our dataset, with the respective classes.

Recipe	Class
Preheat oven to 350 degrees F (175 degrees C). Grease and flour 2 - 8 inch round pans. In a small bowl, whisk together flour, baking soda and salt; set aside. In a large bowl …	EN
Lavate i mandarini e grattugiate la buccia. Spremete i mandarini fino ad ottenere 100 ml di succo. Sbucciate i rimanenti frutti e tagliateli a fettine. In un'ampia ciotola montate …	IT
Étape 1 : Préchauffer le four Th.6 (180 °C). Étape 2 : Dans un saladier, mettre : le sucre, les oeufs, le beurre ramolli, la levure, le lait, le rhum, la farine, remuer. Étape 3 …	FR
Eier trennen, Eiweiße mit 1 Prise Salz steif schlagen. Den Boden einer Springform mit Backpapier auslegen. Nüsse im Food Processor oder Mixer fein mahlen. Datteln …	DE

Previous experiments (Cavnar & Trenkle, 1994; Grefenstette, 1995) showed that the common words approach is faster than the *n*-grams one. This is probably due to the fact that, in any given sentence, there are usually a lot less words than there are *n*-grams. However, the *n*-grams method proved to be extremely resilient to errors, thus being the most suitable approach to work with data that comes from noisy sources, like tweets or optical character recognition (OCR). Moreover, this approach also performs a kind of *stemming*, since *n*-grams coming from similar words (i.e., ‘advance’, ‘advancing’, ‘advanced’) are naturally related. Note that this form of stemming is language independent, something that is not true for the traditional techniques.

The dataset

The dataset we used for this case study was constructed starting from cooking blogs in various languages, specifically English, French, Spanish, German, Italian and Portuguese. Using the Python library `recipe-scrapers` we extracted between 25 and 40 cake recipes for each language, for a total of 186 recipes, and saved each one in a text file. Thus, our collection initially was not in `ARFF` format, but could still be imported in `WEKA` using the `TextDirectoryLoader` described above. Note that each recipe is a short document, with less than 800 words and a mean number of terms of 173. Table 4 shows an extract from this first collection.

Note that the aim of this paper is not that of building a state of the art model for the language identification problem, but rather that of showing how such a problem can be solved using `WEKA`. Still, we would like our resulting model to be at least realistic, which is why we used another dataset as a test set to verify that it can actually recognize the languages in various kinds of texts and not only in cake recipes. This second collection was built starting from the LeipzigCorporaCollection (Goldhahn, Eckart, & Quasthoff, 2012); more precisely, for each language, we downloaded the 2014/2015 newscrawl and then extracted the first 50 sentences. This produced a toy dataset of small random phrases with a mean number of words of about 25. Both datasets are available at https://github.com/mwritescode/text-categorization-with-WEKA.

The framework for classification

We choose to solve this language identification problem using two separate approaches, which differ one from the other mainly in the construction of the feature vectors. Indeed, in the first approach, explored in Section 5.1, we extracted bi- and tri-grams from our document collection, computed their frequency of occurrence and used them as the terms for our vectorized representation. In the second approach instead we extracted a set of words from our collection, computed their frequency and only used the more frequent ones as the elements of our feature vectors. The only other preprocessing step we applied in this experiment is stopwords removal: for the common words approach, in fact, we explored two different scenarios, with or without the stopwords. We choose not to use a `stemmer`, as `WEKA` only provides monolingual implementations of this tool and those would obviously not perform well on our dataset. Moreover, as we stated before, the *n*-grams approach natively performs some kind of simple stemming. We also choose not to employ any form of normalization for the document lengths as, in this case, the collection is composed of texts of similar length. A clear explanation of the steps we followed during the experiments can be found in Fig. 8.

The data, now in a structured form, can be used to train different classification models; in particular we used `J48`, `NaiveBayesMultinomial`, `RandomForest`, `IBk` with $k = 1, 3$ and 5 and `SMO`. For this last algorithm we choose a linear kernel as it has been proved that many text classification problems are linearly separable (Song et al., 2005); every other model, for the sake of simplicity, has been used with the default parameters provided by `WEKA`. Also note that we used the `Experimenter` interface to set up 10 rounds of 10 fold cross-validation for each of these algorithms, in order to get a good idea of how they would perform on new data. The different metrics we used to evaluate the models' performance are those explained in Section 4.7 – that is, accuracy, precision, recall, F-measure, AUC and Kappa statistic. Using them, we were able to select the model with the best performance on our initial dataset, which we then tested also on the Leipzig set.

The framework for clustering

As we observed in the Introduction and in Section 4, in the context of Text Mining, in addition to classification techniques, unsupervised learning methods such as clustering can also be used: in this case there are not predefined categories and the documents are not labeled with a class attribute. The goal is to automatically identify groups of documents with common characteristics. For this clustering experiment we choose not to use any form of stemming and to repeat the experiment two times, both keeping and removing the stopwords. Moreover, a vectorized representation of the input text has been built starting from the single words extracted from the documents in our collection. Unlike the solution adopted for the classification tasks, in this case we used simple Boolean values for the weights and normalized the documents to unit length. The reason for this will be clear in a second. `WEKA` offers several partitioning, hierarchical and density-based clustering algorithms and most of these methods require the computation of the distance between pairs of records in the dataset for the composition of the groups. Depending on the type of records, it may be convenient to use one distance measurement rather than another: the Euclidean distance, for example, is convenient when there are instances with non-binary and non-asymmetric numeric attributes while the Jaccard distance is well suited for asymmetric binary attributes where 0-0 matches are not of interest. When we have word vectors associated with documents, often containing relatively few non-zero attributes, it is convenient to use the cosine distance that, like Jaccard, does not depend on the number of 0-0 matches but is able to handle non binary vectors (see, e.g., Tan et al., 2006). However, in `WEKA`, among the parameters of the various clustering algorithms, the cosine distance is not in the list. The normalization step is needed because using the cosine distance is equivalent to using the Euclidean distance after normalizing the instances so that they all have the same unit length. A clear explanation of how the experiment was structured can be found in Fig. 9.

The clustering schemes we choose are *k*-means and agglomerative hierarchical clustering, two simple and easy to understand and interpret methods. Note that, since we maintained information regarding the real class labels of our documents, we choose to perform a *classes to clusters evaluation* to estimate our models' accuracy. This process computes the majority class for each cluster *i* found by our models using the supervised labels, then it assigns that class to all the documents in *i* and computes the accuracy.

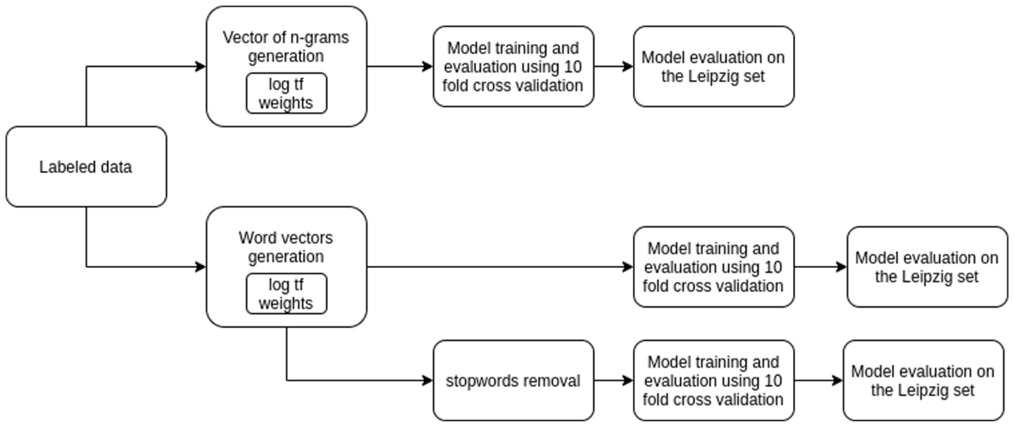

Vector of n-grams generation
log tf weights
Model training and evaluation using 10 fold cross validation
Model evaluation on the Leipzig set
Labeled data
Word vectors generation
log tf weights
Model training and evaluation using 10 fold cross validation
Model evaluation on the Leipzig set
stopwords removal
Model training and evaluation using 10 fold cross validation
Model evaluation on the Leipzig set

Fig. 8. Language identification experiment framework.

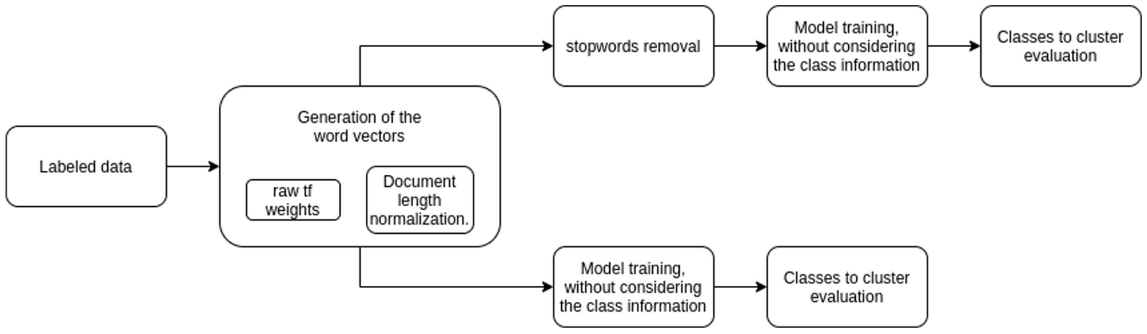

stopwords removal
Model training, without considering the class information
Classes to cluster evaluation
Labeled data
Generation of the word vectors
raw tf weights
Document length normalization.
Model training, without considering the class information
Classes to cluster evaluation

Fig. 9. Language identification clustering experiment.

5.1. The n-gram approach

To implement this approach in WEKA we first need to extract the single *n*-grams from the various documents. This can be done using the tool `CharacterNGramTokenizer`, which is accessible through the settings of the filter `StringToWordVector`. In particular, since we are only interested in getting bi- and tri-grams we set the tokenizer's parameters accordingly (`NGramMaxSize` = 3 and `NGramMinSize` = 2). Regarding the other configuration options for `StringToWordVector`, to avoid repetitions of the same *n*-gram with mixed character cases, we set `lowerCaseTokens` to `True`. We also choose to keep a maximum of 5000 sub-strings, as to limit the number of *n*-grams memorized by the model and to not overwhelm the machine we run our tests on (`wordsToKeep = 5000` and `doNotOperateOnPerClassBasis = True`). Lastly, since we are looking for the most common *n*-grams, we choose to measure their logarithmic frequency and not only their presence or absence in a document; thus we set both `TFTransform` and `outputWordCounts` to `True`.

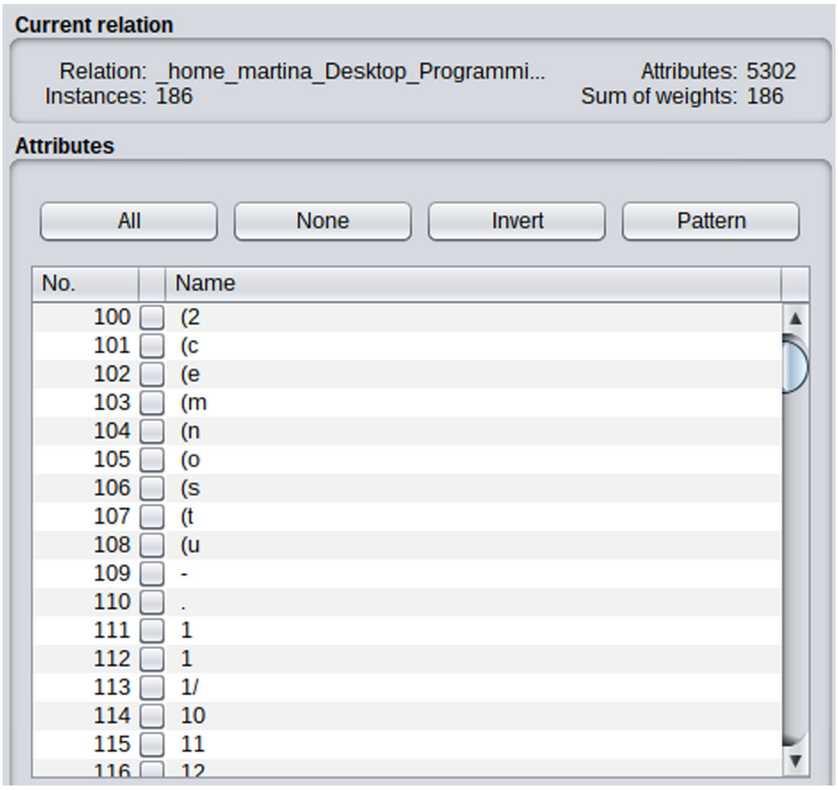
Once `StringToWordVector`'s parameters are set to satisfaction, we can run the filter and peruse the results. Using the above configuration, we get a new dataset with 186 instances and 5302 attributes. Note that we actually ended up with more *n*-grams than we intended to (recall that we set `wordsToKeep` to 5000). That is due to the fact that WEKA does not break ties: thus, if there are some sub-strings which are as common as the least common *n*-gram, those are kept too.

As shown in Fig. 10, however, our new dataset contains some 'strange' tokens (numbers, punctuation signs, parenthesis), which we want to remove as we do not think they are useful for the classification task. WEKA's unsupervised filter `RemoveByName` does exactly this: it takes a regular expression which is used to match attribute names and it also lets us specify (through the parameter `invertSelection`) if the matched attributes need to be removed or kept. Executing this filter with the regex `^[-’a-zA-ZÀ-ÖØ-öø-ÿ\s]+$` and `invertSelection = True` we can remove all attributes whose name contains characters other than white spaces, apostrophes, hyphens or letters. Please note that we choose to keep hyphenation marks since they are commonly used in some languages (English included) for compound words; similarly the apostrophes were kept because they are used to indicate the omission of letters or numbers.

The final dataset (shown in Fig. 11) contains 4323 attributes; in the picture, each row represents a document, each column represents an *n*-gram and each cell represents the term frequency of a certain sub-string in a given document.

Please note that this dataset can only really be used to explore the data we are working with and get a more complete idea of the available information. Indeed, if we apply the `StringToWordVector` filter in the `Preprocess` panel as we have just done, the features are computed starting from every document in the collection. However when, at classification time, we choose either 10 fold cross-validation or hold-out as our validation schema, some of the documents are used as a test set. Thus, the model's features are chosen starting from both training and test set: this is a kind of bias and may result in an excessively optimistic evaluation. Moreover, if we want to use a completely new dataset as a test set, we cannot apply the `StringToWordVector` filter independently to train and test set. Indeed, this would result in two datasets with different attributes and WEKA would not start the classification process, instead showing an error. Thus, what we need is a way to build the features starting from the training set and then reuse them to vectorize the test documents.

To solve both the previous problems, we can apply the filter at classification time. In WEKA, this can be done using a particular classifier called `FilteredClassifier`. This algorithm is part of the *meta learners* natively implemented in WEKA and takes as parameters both the filter to apply during the categorization and the algorithm used to perform it. Note that we can only pass a single filter to this learner, but

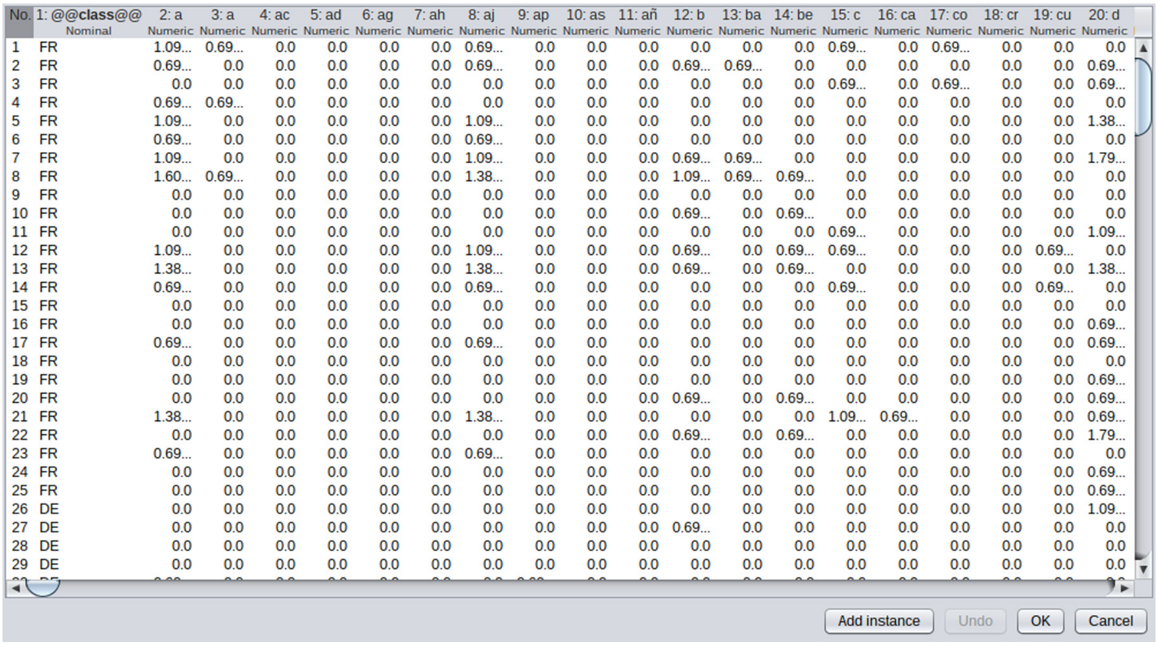

Current relation

Relation: _home_martina_Desktop_Programmi... Attributes: 5302
Instances: 186 Sum of weights: 186

Attributes

No.	Name
100	(2
101	(c
102	(e
103	(m
104	(n
105	(o
106	(s
107	(t
108	(u
109	-
110	.
111	1
112	1
113	1/
114	10
115	11
116	12

Fig. 10. Dataset attributes after applying `StringToWordVector`.



No.	1: @@class@@	2: a	3: a	4: ac	5: ad	6: ag	7: ah	8: aj	9: ap	10: as	11: añ	12: b	13: ba	14: be	15: c	16: ca	17: co	18: cr	19: cu	20: d
	Nominal	Numeric	Numeric	Numeric	Numeric	Numeric	Numeric	Numeric	Numeric	Numeric	Numeric	Numeric	Numeric	Numeric	Numeric	Numeric	Numeric	Numeric	Numeric	Numeric
1	FR	1.09...	0.69...	0.0	0.0	0.0	0.0	0.69...	0.0	0.0	0.0	0.0	0.0	0.0	0.69...	0.0	0.69...	0.0	0.0	0.0
2	FR	0.69...	0.0	0.0	0.0	0.0	0.0	0.69...	0.0	0.0	0.0	0.69...	0.69...	0.0	0.0	0.0	0.0	0.0	0.0	0.69...
3	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.69...	0.0	0.69...	0.0	0.0	0.69...
4	FR	0.69...	0.69...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
5	FR	1.09...	0.0	0.0	0.0	0.0	0.0	1.09...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.38...
6	FR	0.69...	0.0	0.0	0.0	0.0	0.0	0.69...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
7	FR	1.09...	0.0	0.0	0.0	0.0	0.0	1.09...	0.0	0.0	0.0	0.69...	0.69...	0.0	0.0	0.0	0.0	0.0	0.0	1.79...
8	FR	1.60...	0.69...	0.0	0.0	0.0	0.0	1.38...	0.0	0.0	0.0	1.09...	0.69...	0.69...	0.0	0.0	0.0	0.0	0.0	0.0
9	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
10	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.69...	0.0	0.69...	0.0	0.0	0.0	0.0	0.0	0.0
11	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.69...	0.0	0.0	0.0	0.0	1.09...
12	FR	1.09...	0.0	0.0	0.0	0.0	0.0	1.09...	0.0	0.0	0.0	0.69...	0.0	0.69...	0.69...	0.0	0.0	0.0	0.69...	0.0
13	FR	1.38...	0.0	0.0	0.0	0.0	0.0	1.38...	0.0	0.0	0.0	0.69...	0.0	0.69...	0.0	0.0	0.0	0.0	0.0	1.38...
14	FR	0.69...	0.0	0.0	0.0	0.0	0.0	0.69...	0.0	0.0	0.0	0.0	0.0	0.0	0.69...	0.0	0.0	0.0	0.69...	0.0
15	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
16	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.69...
17	FR	0.69...	0.0	0.0	0.0	0.0	0.0	0.69...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.69...
18	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
19	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.69...
20	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.69...	0.0	0.69...	0.0	0.0	0.0	0.0	0.0	0.69...
21	FR	1.38...	0.0	0.0	0.0	0.0	0.0	1.38...	0.0	0.0	0.0	0.0	0.0	0.0	1.09...	0.69...	0.0	0.0	0.0	0.69...
22	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.69...	0.0	0.69...	0.0	0.0	0.0	0.0	0.0	1.79...
23	FR	0.69...	0.0	0.0	0.0	0.0	0.0	0.69...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
24	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.69...
25	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.69...
26	DE	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.09...
27	DE	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.69...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
28	DE	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
29	DE	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Fig. 11. Final dataset with each *n*-gram frequency.

we need to apply both `StringToWordVector` and `RemoveByName`, in this order. The solution for this problem comes again from `WEKA` itself, which provides a particular filter called `MultiFilter`, that simply takes a list of filters and applies them sequentially.

Table 5 shows the results achieved by each of the tested algorithms, according to some of the most common evaluation metrics. Immediately we notice that all our models get excellent recall and precision values, with `NaiveBayesMultinomial` (`NBM` in the results' table), `Random Forest` (in the table `RF`) and `SMO` that get the maximum possible value for both metrics. Such results imply that the tested classifiers do not produce many false positive nor many false negatives: thus, they are extremely capable of separating the instances of each class label from the others. As we would expect from such good values in recall and precision, all our models generally perform much better than a random classifier, as testified by the high kappa statistic and AUC. The `Experimenter`'s significance test results with respect to `NBM` and with a significance level of 0.05 are shown in Table 5 as specific symbols next the scores: a (*) means that the

Table 5
Results of the *n*-grams experiment.

Classifiers	Evaluation metrics					
	% Correct	Precision	Recall	F-measure	K Statistic	AUC
NBM	100.00	1.00	1.00	1.00	1.00	1.00
J48	95.44 (*)	0.97 (*)	0.95 (*)	0.95 (*)	0.94 (*)	0.97 (*)
IB1	96.24 (*)	0.97 (*)	0.96 (*)	0.96 (*)	0.95 (*)	0.98 (*)
IB3	95.97 (*)	0.97 (*)	0.96 (*)	0.96 (*)	0.95 (*)	0.99
IB5	96.62 (*)	0.98 (*)	0.97 (*)	0.97 (*)	0.96 (*)	0.99
SMO	100.00	1.00	1.00	1.00	1.00	1.00
RF	100.00	1.00	1.00	1.00	1.00	1.00

Table 6
Results of the common words experiment.

Classifiers	Evaluation metrics					
	% Correct	Precision	Recall	F-measure	K Statistic	AUC
NBM	100.00	1.00	1.00	1.00	1.00	1.00
J48	93.92 (*)	0.96 (*)	0.94 (*)	0.94 (*)	0.93 (*)	0.96 (*)
IB1	55.63 (*)	0.90 (*)	0.56 (*)	0.67 (*)	0.48 (*)	0.74 (*)
IB3	49.86 (*)	0.91 (*)	0.50 (*)	0.66 (*)	0.41 (*)	0.82 (*)
IB5	50.65 (*)	0.89 (*)	0.51 (*)	0.60 (*)	0.42 (*)	0.86 (*)
SMO	99.89	1.00	1.00	1.00	1.00	1.00
RF	100.00	1.00	1.00	1.00	1.00	1.00

score is significantly worse than that of Naive Bayes on this problem, while a (v) means that the score is significantly better. If there is no symbol, then there is no statistical difference between the current score and that of Naive Bayes. Note that the `NaiveBayesMultinomial` algorithm was chosen as the test base because of its simplicity and extremely good results on our problem. As we can see, `NBM`, `SMO` and `RF` get significantly better scores than every other classifier for each classification metric apart form the area under the ROC curve, where k-nearest neighbors with $k = 3$ and $k = 5$ both get 0.99, which is still less than the 1.00 scored by the three best classifiers, but not statistically so. Since Naive Bayes is faster, simpler and performs as well as the `SMO` and `RF`, it is probably the classifier we would choose if we were designing a real system. Lastly, to verify that said `NBM` model can really identify the language of a generic document, we test it on the Leipzig set using the `Explorer`, and get a 99.667% of correctly classified instances, an F-measure score of 0.997 and very high values for both precision and recall (about 0.997).

5.2. The common words approach

Again, the first thing we need to do is to extract the single words from our document collection. In `WEKA`, this can be done using the `WordTokenizer`, a tool which is accessible through the settings of the filter `StringToWordVector`. More precisely, this tokenizer takes a string of characters called *delimiters* according to which it splits the texts into words; in our case, since we want to remove punctuation and quotation marks but also parenthesis and numbers, the string is the following:

```
\r\n\t.,;:"’()?!-¿¡+*&#$%\/=<>[]_‘@ 1234567890
```

As concerning the other parameters of the `StringToWordVector` filter, we decided to limit the number of words per language to 2700 (`wordsToKeep = 2700` and `doNotOperateOnPerClassBasis = False`). Note that this value was not randomly chosen: indeed, a previous work showed that each language has between 900 and 2700 most common words (Grefenstette, 1995). We also set `lowercaseTokens = True` as not to have mixed character cases. Since we want to extract the most common words, we also choose to compute each term’s overall frequency in a document instead of only its presence or absence, thus setting `TFTransform` and `outputWordCounts` to `True`. Note that without the logarithmic transformation applied by the `TFTransform` option, we would have gotten decidedly worse results. Finally, as already anticipated, we choose to repeat this experiment two times, first keeping and then removing the stopwords. Please note that since the most common words in a language are typically articles, prepositions and pronouns – that is, the same terms which are usually considered stopwords – we expect the second model to have a much worse performance than the first one. Moreover, there is a real possibility that this second model will only be able to correctly recognize the language in documents which contain cake recipes, thus resulting almost useless. Therefore, the model is not meant to be good or realistic, but just to illustrate how `WEKA` can be used to execute the stopwords removal preprocessing step, even on a multilingual dataset.

5.2.1. Classification without stopwords removal

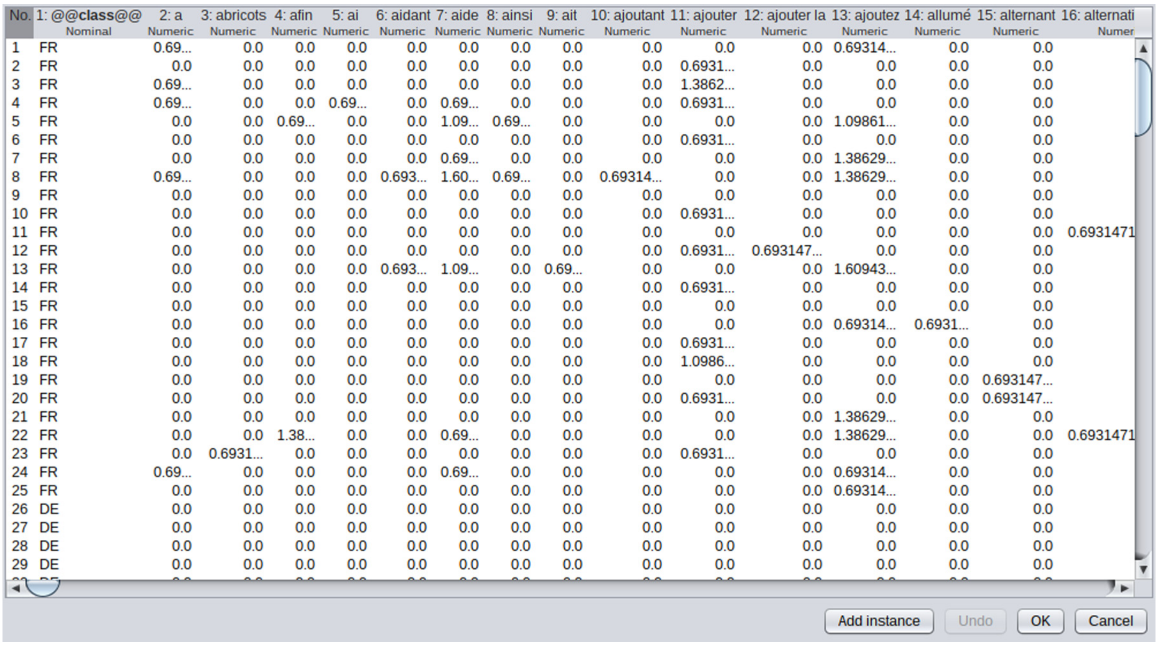
After applying `StringToWordVector` with the parameters we just specified, we get a dataset composed of 186 instances and 4785 attributes. Notice that the number of attributes is relatively small: we set the maximum number of terms per language to 2700, so we expected something along the lines of 16200 (6 times 2700) attributes; but we got only about a fourth of that. The reason for this can be found in how we built the training set: we used at most 40 texts per language and they all talk about cakes, thus they use a lot of the same words. That makes the overall vocabulary size for our dataset of only about 5000 terms.

Examining this new collection using the `Edit` option in the `Explorer`’s `Preprocess` panel, we get the table shown in Fig. 12. Here, every row represents a document and every column represents a word; the cells contain the frequency of each term in each document.

As we already saw for the *n*-grams approach, using this dataset for the classification could cause a bias; once again we need to use `WEKA`’s `FilteredClassifier`.

Table 6 shows the results obtained in the `Experimenter` with respect to the same evaluation metrics described in Section 4.7. Note that the lazy learners (`IB1`, `IB3` and `IB5`) seem to have a lot of difficulties with this task, while `NaiveBayesMultinomial`, `RandomForest` and `SMO` remain the more effective classifiers. In fact, all the k-nearest neighbors models get a low accuracy score (ranging from about 55% to 49.86%) and also a small recall value. Precision is however always around 0.90. This implies that the three algorithms in question produce many false negatives but only a small amount of false positives, therefore – when predicting the class label for a new instance i of class C_i – the models tend misclassify i as being of class $\neg C_i$; the opposite error is less frequent. Note that the F-measure, being the harmonic mean between precision and recall, is heavily influenced by the classifiers’ low recall score and is thus quite small itself. Finally, the low kappa statistic that these algorithms achieve means that they perform little better than a random classifier. With respect to `IBk`, `J48` gets significantly better results; however, the manual inspection of the computed scores and the ensuing analysis of the `Experimenter`’s significance test tell us that its performance is still sensibly lower than that of the other three models. Finally, the results of `WEKA`’s statistical test with significance level of 0.05 and `NaiveBayesMultinomial` as the test base are shown in Table 6 as (*) and (v) symbols next to the various scores. As we can see, the performances of `SMO` and `RF` on this task are not statistically different from those of the Bayesian model. Our algorithm of choice for the final model is `NBM`, as it is simpler and faster than the other two, but still gets exceptionally good results. Since we know that our initial collection only contains a very small part of all the words present in each language, it is very important to verify that our model can really identify the language of a generic document. Again, this is done using the `Explorer` to test the Naive Bayes model on the Leipzig set, resulting in a 96% of correctly classified instances and an F-measure score of 0.96, with high precision and high recall (both of about 0.96).

Consider for a moment a situation in which the previous test is not as successful. In such a case we might want to investigate the



No.	1: @@class@@	2: a	3: abricots	4: afin	5: ai	6: aidant	7: aide	8: ainsi	9: ait	10: ajoutant	11: ajouter	12: ajouter la	13: ajoutez	14: allumé	15: alternant	16: alternati
	Nominal	Numeric	Numeric	Numeric	Numeric	Numeric	Numeric	Numeric	Numeric	Numeric	Numeric	Numeric	Numeric	Numeric	Numeric	Numer
1	FR	0.69...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.69314...	0.0	0.0	
2	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.6931...	0.0	0.0	0.0	0.0	
3	FR	0.69...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.3862...	0.0	0.0	0.0	0.0	
4	FR	0.69...	0.0	0.0	0.69...	0.0	0.69...	0.0	0.0	0.0	0.6931...	0.0	0.0	0.0	0.0	
5	FR	0.0	0.0	0.69...	0.0	0.0	1.09...	0.69...	0.0	0.0	0.0	0.0	1.09861...	0.0	0.0	
6	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.6931...	0.0	0.0	0.0	0.0	
7	FR	0.0	0.0	0.0	0.0	0.0	0.69...	0.0	0.0	0.0	0.0	0.0	1.38629...	0.0	0.0	
8	FR	0.69...	0.0	0.0	0.0	0.693...	1.60...	0.69...	0.0	0.69314...	0.0	0.0	1.38629...	0.0	0.0	
9	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
10	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.6931...	0.0	0.0	0.0	0.0	
11	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.6931471
12	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.6931...	0.693147...	0.0	0.0	0.0	
13	FR	0.0	0.0	0.0	0.0	0.693...	1.09...	0.0	0.69...	0.0	0.0	0.0	1.60943...	0.0	0.0	
14	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.6931...	0.0	0.0	0.0	0.0	
15	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
16	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.69314...	0.6931...	0.0	
17	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.6931...	0.0	0.0	0.0	0.0	
18	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0986...	0.0	0.0	0.0	0.0	
19	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.693147...	
20	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.6931...	0.0	0.0	0.0	0.693147...	
21	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.38629...	0.0	0.0	
22	FR	0.0	0.0	1.38...	0.0	0.0	0.69...	0.0	0.0	0.0	0.0	0.0	1.38629...	0.0	0.0	0.6931471
23	FR	0.0	0.6931...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.6931...	0.0	0.0	0.0	0.0	
24	FR	0.69...	0.0	0.0	0.0	0.0	0.69...	0.0	0.0	0.0	0.0	0.0	0.69314...	0.0	0.0	
25	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.69314...	0.0	0.0	
26	DE	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
27	DE	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
28	DE	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
29	DE	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	

Fig. 12. Final dataset with each word frequency.

possibility of getting better results on the Leipzig test using any of the other classifiers. Of course, we could check such a thing from the `Explorer`, manually executing each algorithm after having set its parameters. But that would soon become tedious, especially if we need to compare a large number of different algorithms with different configuration settings. Moreover, the `Explorer` does not really let us compare each model's results, not with the same effectiveness of the `Experimenter`. Unfortunately, setting up an external test set is an option which is only available in said interface's `Advanced mode`, where we would need to choose the training set and then select `ExplicitTestsetResultProducer` as the `Result Generator`. In the generator's configuration options we should then choose the directory and the name of the test set. Finally, in the `Generator properties`, we select `Enabled` and choose the classification algorithms through `SplitEvaluator`. Note that the `Experimenter` in `Advanced mode` is more complicated to setup; in this regard, it could be useful to know that this interface preserves every parameter you have already set when you switch from `Simple` to `Advanced`; thus you could first set up the dataset(s) and the classifier(s) in `Simple` mode, which is far easier to use.

5.2.2. *Classification with stopwords removal*

For this second model, we first need to create a list of stopwords, containing the most common words for each of the languages in our training set. We can download six different lists of stopwords (one for each language) from here https://sites.google.com/site/kevinbouge/stopwords-lists and then simply concatenate them together in a single file. To use this list of stopwords in `WEKA` we need to set the parameter `stopwordsHandler` of `StringToWordVector` to `WordsFromFile`. Then we just set its `stopwords` parameter so that it points to our file. Note that, to work properly, this tool needs the file to have one word per line.

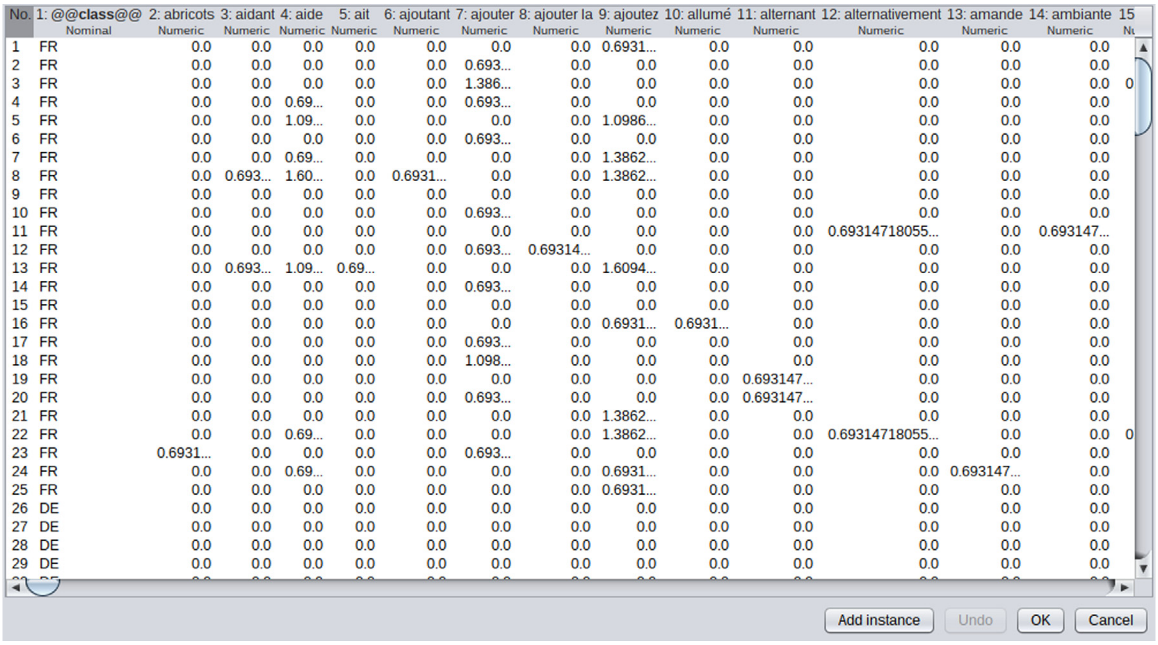
We can now apply the filter to our data, making sure to keep the same configuration used in the paragraph above, a part from the `stopwordsHandler`. The new dataset we end up with is composed of 186 instances and 4084 attributes and, as we can see in Fig. 13, is very similar to the one obtained for the previous model, with the exception of some words like *a*, *afin* and *ainsi*. Note that they are all french prepositions or adverbs.

Table 7
Results of the common words experiment, once the stopwords were removed.

Classifiers	Evaluation metrics					
	% Correct	Precision	Recall	F-measure	K Statistic	AUC
NBM	100.00	1.00	1.00	1.00	1.00	1.00
J48	94.67 (*)	0.96 (*)	0.95 (*)	0.95 (*)	0.94 (*)	0.98 (*)
IB1	44.35 (*)	NaN	0.44 (*)	NaN	0.35 (*)	0.68 (*)
IB3	40.51 (*)	NaN	0.41 (*)	NaN	0.30 (*)	0.76 (*)
IB5	46.23 (*)	NaN	0.46 (*)	NaN	0.37 (*)	0.81 (*)
SMO	99.25	0.99	0.99	0.99	0.99	1.00
RF	100.00	1.00	1.00	1.00	1.00	1.00

As we know, this dataset can only be used to better understand our data: to make sure our classification is not biased, the `StringToWordVector` filter needs to be applied directly during the categorization using the meta learner `FilteredClassifier`. The results of such and experiment are shown in Table 7.

The first thing we notice is that there are some NaN values, which are scores that the `Experimenter` could not compute. Recall that, as we stated in Section 4.7, the precision is computed as the rate of the true positives over everything that has been classified as positive and that the results we show in the table are the weighted average of this value computed for each class label. Thus, the NaN precision values obtained by the lazy learners (`IB1`, `IB3`, `IB5`) are probably due to the fact that, for at least one class label C_i both the true and the false positive rate were zero. This made the precision impossible to compute for label C_i and, therefore, the `Experimenter` could not return the weighted average. Seeing that the F-measure is computed as the harmonic mean between precision and recall, this also explains the NaN values obtained by the `IBk` models for this score. Note that our intuition can be confirmed by using the `Explorer` to run 10-fold cross-validation on each of the three models and looking at the TP and FP values that this interface's output panel shows for each of the class labels. Executing this check for `IB1`, for example, shows that it cannot properly classify the Spanish recipes, getting $TP = 0.0$ and $FP = 0.0$ for this class. Regarding the other algorithms' performance, the results of `WEKA`'s statistical test with significance level of 0.05 with respect to the Naive Bayes model as the test base are again shown in the table as (*) or (v) next to the computed scores. As we can see, `J48`



No.	1: @@class@@ Nominal	2: abricots Numeric	3: aidant Numeric	4: aide Numeric	5: ait Numeric	6: ajoutant Numeric	7: ajouter Numeric	8: ajouter la Numeric	9: ajoutez Numeric	10: allumé Numeric	11: alternant Numeric	12: alternativement Numeric	13: amande Numeric	14: ambiante Numeric	15 Nu
1	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.6931...	0.0	0.0	0.0	0.0	0.0	
2	FR	0.0	0.0	0.0	0.0	0.0	0.693...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
3	FR	0.0	0.0	0.0	0.0	0.0	1.386...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
4	FR	0.0	0.0	0.69...	0.0	0.0	0.693...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
5	FR	0.0	0.0	1.09...	0.0	0.0	0.0	0.0	1.0986...	0.0	0.0	0.0	0.0	0.0	
6	FR	0.0	0.0	0.0	0.0	0.0	0.693...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
7	FR	0.0	0.0	0.69...	0.0	0.0	0.0	0.0	1.3862...	0.0	0.0	0.0	0.0	0.0	
8	FR	0.0	0.693...	1.60...	0.0	0.6931...	0.0	0.0	1.3862...	0.0	0.0	0.0	0.0	0.0	
9	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
10	FR	0.0	0.0	0.0	0.0	0.0	0.693...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
11	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.69314718055...	0.0	0.693147...	
12	FR	0.0	0.0	0.0	0.0	0.0	0.693...	0.69314...	0.0	0.0	0.0	0.0	0.0	0.0	
13	FR	0.0	0.693...	1.09...	0.69...	0.0	0.0	0.0	1.6094...	0.0	0.0	0.0	0.0	0.0	
14	FR	0.0	0.0	0.0	0.0	0.0	0.693...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
15	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
16	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.6931...	0.6931...	0.0	0.0	0.0	0.0	
17	FR	0.0	0.0	0.0	0.0	0.0	0.693...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
18	FR	0.0	0.0	0.0	0.0	0.0	1.098...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
19	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.693147...	0.0	0.0	0.0	
20	FR	0.0	0.0	0.0	0.0	0.0	0.693...	0.0	0.0	0.0	0.693147...	0.0	0.0	0.0	
21	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.3862...	0.0	0.0	0.0	0.0	0.0	
22	FR	0.0	0.0	0.69...	0.0	0.0	0.0	0.0	1.3862...	0.0	0.0	0.69314718055...	0.0	0.0	0
23	FR	0.6931...	0.0	0.0	0.0	0.0	0.693...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
24	FR	0.0	0.0	0.69...	0.0	0.0	0.0	0.0	0.6931...	0.0	0.0	0.0	0.693147...	0.0	
25	FR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.6931...	0.0	0.0	0.0	0.0	0.0	
26	DE	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
27	DE	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
28	DE	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
29	DE	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	

Fig. 13. Final dataset with each word frequency.

gets significantly lower results than NBM for each of the six evaluation metrics considered, while the results of SMO and RF do not show any statistically relevant difference from them. Once again, amongst these three models, our choice falls on the Naive Bayes algorithm, which is less computationally expensive, simpler and still gets optimal results. We would now like to be sure that it can get similar accuracy even on data that does not relate to cake recipes. However, when we test the model on the Leipzig set, we get only 36% correctly classified instances and an F-measure of about 0.34 with low precision score (0.52) and even lower recall value (0.36). That is to be expected, since two documents which talk about completely different things will likely not have many words in common, a part from articles, adverbs, prepositions and pronouns, that are exactly the stopwords we removed.

5.2.3. *Clustering*

As we already explained, among the various distance metrics WEKA offers for its clustering algorithms, the cosine similarity is not present. However, this measure is particularly well suited for high dimensional sparse data such as our document vectors and its behavior can be obtained by using the Euclidean distance on word vectors that have been normalized to unit length. This can be achieved by setting the normalizeDocLength parameter to Normalize all data in the StringToWordVector filter. For what concerns the other parameters, we used the same already employed for the classification (wordsToKeep = 2700, doNotOperateOnPerClassBasis = False, lowercaseTokens = True, outputWordCounts= True and \r\n\t.,;:"’()?!-¿¡+*&#$%\/=<>[]_‘@ 1234567890 as tokenizer), except for the TFTransorm parameter which we preferred to set to False, thus using raw tf weights. Moreover, in a first experiment we used stopwordsHandler=Null and then, in a second one, stopwordsHandler=WordsFromFile, specifying the same file used in Section 5.2.2.

After this preprocessing phase, starting from the Cluster panel of Explorer, we used the k-means clustering algorithm and various types of hierarchical clustering, using Classes to clusters evaluation as cluster mode, in order to have the number of instances incorrectly classified. For what concerns k-means, we set numClusters = 6 and left the other default parameters after setting the dontNormalize option relative to the Euclidean distance to True, since we already made a vector normalization to unit length. As for hierarchical clustering, we used various link type strategies, numClusters = 6 and dontNormalize = True for the Eucledian distance.

Without stopwords removal, the k-means algorithm was able to correctly recognize the language of all documents, that is, the algorithm clustered incorrectly 0% instances. The same happened for hierarchical clustering by using different link strategies, such as single, average and complete. With stopwords removal, k-means had worse performance than before. Since the algorithm depends on the choice of initial centroids, we tried different seed values for the random generation of centroids and different initialization methods (seed and InizializationMethod parameters) and finally obtained the best result with k-means++ as initialization method, corresponding to a number of incorrectly instances equal to 15%. With hierarchical clustering and average link strategy, we obtained again a 0% of incorrectly clustered instances; with single and complete strategies we obtained an error percentage of 21% and 1%, respectively. For instance, Fig. 14 represents the dendogram which corresponds to the execution of the algorithm with the average link strategy and with stopwords removal, which gave an error of 0%; the figure clearly shows the 6 groups, each corresponding to a different language.

We underline that to obtain the whole dendogram, the hierarchical clustering algorithm must also be run with numClusters = 1; moreover, if we want the leaves of the tree to be labeled, for example with the reference to the language of each document, an attribute of type string containing the label must be added to the data set. To this purpose, you can use the Copy filter to copy the class attribute and then convert it to a string with the NominalToString filter. Finally, note that these experiments were done by using Explorer since Experimenter has little functionalities for clustering.

6. A second case study: recipe type classification

For this second case study we choose to keep dealing with recipe instructions. Thus, we propose a simple classification task which aims to recognize what kind of dish a certain recipe is about. Specifically, we used the Python library recipe-scrapers to extract almost 50 recipes each for the following categories: cakes, salads, pasta and noodles and stews. Note that, in this case, the recipes are all written in English and they all have less than 500 words. Table 8 shows an extract from our dataset, available at https://github.com/mwritescode/text-categorization-with-WEKA.

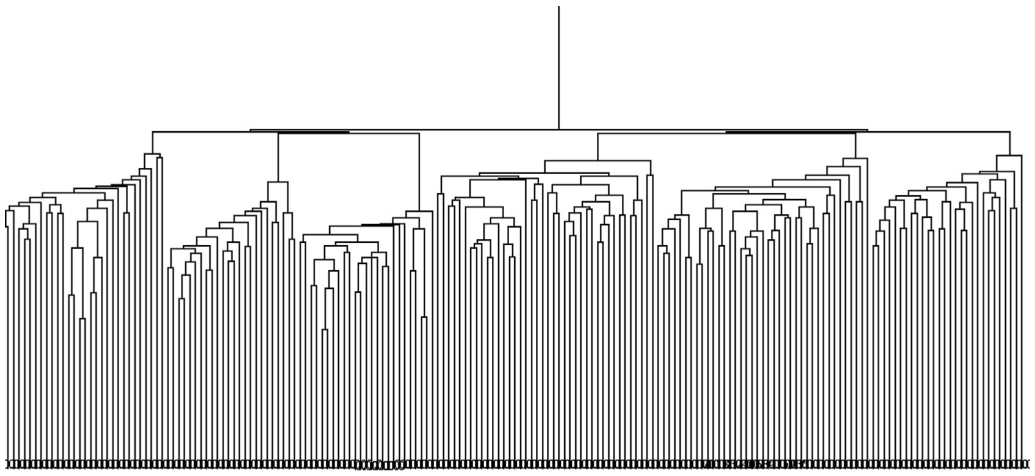


Fig. 14. The dendogram corresponding to average link strategy with stopwords removal.

Table 8
Some of the instances in our second dataset, with the respective classes.

Recipe	Class
Beat egg whites until they form stiff peaks, and then add cream of tartar, vanilla extract, and almond extract. Sift together flour, sugar, and salt. Repeat five times. Gently …	Cakes
Preheat the oven to 350 degrees F (175 degrees C). Lightly grease a 2-quart baking dish. In a large pot of salted water, lightly boil the macaroni for about 5 min until half-cooked. Whisk the egg and milk together in a large cup. Add …	Pasta and Noodles
In a medium bowl, combine avocados, onion, bell pepper, tomato, cilantro and lime juice. Gently toss until evenly coated. Season with salt and pepper.	Salads
Bring a large pot of water to a boil; add potatoes and carrot. Return mixture to a boil and add eggs; cook until potatoes are tender, 20 to 30 min. Drain and slightly cool mixture. Chop potatoes and carrot; peel and chop eggs. Mix …	Salads
Preheat oven to 350 degrees F (175 degrees C). Lightly grease and flour one 9 × 13 inch cake pan. Beat vegetable oil and eggs until foamy. Add the sugar, flour, ground cinnamon …	Cakes
Preheat oven to 250 degrees F (120 degrees C). Stir beef, potatoes, tomato-vegetable juice cocktail, carrots, celery, onion, tapioca, sugar, salt, and pepper together in a roasting pan; cover. Bake in preheated oven until beef and potatoes are tender, about 5 h.	Stews

6.1. The framework

For this second experiment we choose to adopt a *Bag of Words* approach and use Boolean weights for our feature vectors, as the vocabulary size is really small. We also choose to remove the stopwords and – since the texts in this collection are monolingual – to make use of stemming as to reduce morphological variations of the same term to a single base form. For some text categorization tasks, this can greatly increase a classifier performance; in this case, however, we are focusing on showing how this step can be executed in `WEKA`, rather than on how much it will or will not improve the classification. Lastly, we choose not to normalize the document lengths to one as all the texts in our collection have similar sizes. A complete overview of how this experiment was structured can be found in Fig. 15.

The models that we trained on the preprocessed dataset are exactly those already described in Section 5 and, again, we evaluated them using ten rounds of 10 fold cross-validation each.

6.2. Setup and experiment

Like always when we are dealing with textual data, we first need to convert it in a more structured format. As we have already seen, `WEKA` lets us extract features from raw text and apply all of the preprocessing steps we illustrated in Section 2.1, using the filter `StringToWordVector`. Thus, once again, the first thing we need to do is to find a configuration for this filter which is appropriate for the task at hand.

We are going to use the `WordTokenizer` to create word vectors starting form the entire documents; since we aim at removing punctuation marks, parenthesis, numbers and quotation marks, but we want to keep hyphens and apostrophes, the string of delimiters we choose to use is:

```
\r\n\t.,;:"'()?!-¿¡+*&#$%\/=<>[]_`@ 1234567890.
```

We would also like to remove articles, prepositions and pronouns, which are hardly useful in any categorization task a part from language identification. Note that `WEKA` has got an integrated stopword list for the English language, which is called `Rainbow`. Regarding the stemming procedure, `WEKA` supports both the Lovins and the Porter stemmer, but we will concentrate on the second one, as it is the most widely used. To employ it in our system we need to set `stemmer = SnwoballStemmer` in the configuration options for `StringToWordVector`. Note that *Snowball* is the name given by Porter himself to a framework, which he designed, for the implementation of stemming algorithms in other languages. The `SnowballStemmer` class in `WEKA` only works if we first install the relative package (called `snowball-stemmers`) using the `Package manager`. After the installation is complete, if we access the configuration options of `SnowballStemmer` we now see a parameter that lets us specify which language we need to stem: the complete list of the languages supported can be found here https://weka.sourceforge.io/doc.dev/weka/core/stemmers/SnowballStemmer.html. Note that there are two options for the English language, `porter` and `english`; experimenting on our dataset we found out that using the second one results in better performances. Thus, to summarize, when we are choosing the parameters for `StringToWordVector` we need to set `stemmer = SnowballStemmer` and then specify the option `english`.

As regarding the other parameters for `StringToWordVector` we choose to keep a maximum of 1000 words per class (`wordsToKeep = 1000` and `doNotOperateOnPerClassBasis = False`). We also choose to set `lowercaseTokens = True` as not to have any mixed character cases and to set `TFTransorm` and `OutputWordCount` parameters are set to `False`, thus using binary term weights.

If we try to apply the filter with this configuration we get only 851 attributes, which is decidedly less than the 4000 we expected; as we saw for the common words approach of the language identification problem, this is due to the small number of documents in the collection we started from and to the very limited range of different words that

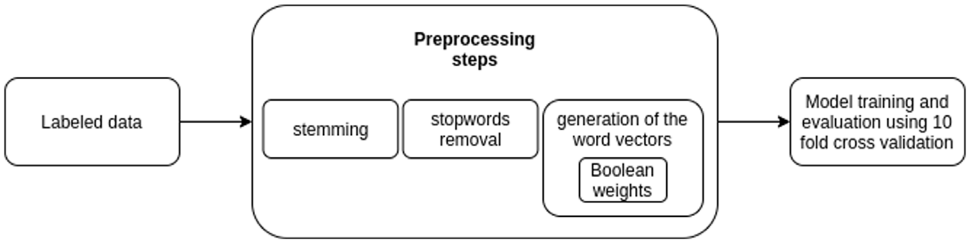

Labeled data
Preprocessing steps
stemming
stopwords removal
generation of the word vectors
Boolean weights
Model training and evaluation using 10 fold cross validation

Fig. 15. Recipe type classification experiment framework.

Table 9
Results of the dishes classification experiment.

Classifiers	Evaluation metrics					
	% Correct	Precision	Recall	F-measure	K Statistic	AUC
NBM	95.28	0.96	0.95	0.95	0.94	1.00
J48	84.61 (*)	0.86 (*)	0.85 (*)	0.84 (*)	0.79 (*)	0.92 (*)
IB1	85.65 (*)	0.88 (*)	0.86 (*)	0.86 (*)	0.81 (*)	0.93 (*)
IB3	89.58 (*)	0.91 (*)	0.90 (*)	0.89 (*)	0.86 (*)	0.98 (*)
IB5	88.95 (*)	0.91 (*)	0.89 (*)	0.89 (*)	0.85 (*)	0.98 (*)
SMO	93.89	0.95	0.94	0.94	0.92	0.98 (*)
RF	92.47	0.94	0.92	0.92	0.90	0.99

are commonly used in recipe instructions. We now use the `Experimenter` together with WEKA's `FilteredClassifier` to apply the filter at run time, while training and evaluating five classifiers on our training set. The results obtained by each algorithm according to the six evaluation metrics described in Section 4.7 are shown in Table 9. As we can see, all the models have a moderately high performance: precision and recall values never drop below 0.85, meaning that not a lot of classification errors are made — neither in terms of false positives nor in terms of false negatives. At the same time, the high AUC scores and a kappa statistic that only drops below 0.80 for the decision tree, indicate a capacity of correctly classifying instances well above that of a random classifier. As we have stated before, the `Experimenter` is also able to output the results of a statistical significance test, for which we can choose both the base algorithm and the significance level. We choose `NBM` as the test base, seeing as it is a simple and computationally efficient model that, despite the simplistic assumptions on which it is based, often yields very good results in text classification tasks; we also set the significance level to 0.05. The results of such a test are shown in Table 9 by means of the symbols (*) and (v) next to the various scores. As previously mentioned, (*) means that the score is significantly lower than that of Naive Bayes, while (v) means that it is significantly better. This statistical test lets us assert that both `J48` and the lazy learners yield significantly worse results with respect to Naive Bayes, while `SMO` and `RF`'s performances differ from those of `NBM` by an amount that is – in general – not statistically relevant. Note however that the `AUC` score of `SMO` does appear to be significantly lower with respect to Naive Bayes' `AUC`. Taking these considerations into account we can say that `NBM`, `SMO` and `RF` are the best models, amongst those that we have tested, for our task. Note that the Naive Bayes multinomial algorithm is extremely simple and fast to compute but gets results which are at least as good as the other two models, if not better.

7. Conclusions

In order to highlight the potential of `WEKA` to carry out a text categorization task, in this work two case studies have been examined and analyzed from different points of view. The first case study is presented in Section 5 and concerns the problem of language identification. The dataset has been processed using different approaches. In Section 5.1, in particular, *n*-grams are extracted from documents and neither stemmer algorithms nor a list of stopwords have been used. In Section 5.2, it was decided to use an approach based on common words and to use the word tokenization algorithm made available by `WEKA`. Also in this case a stemmer was not used, since `WEKA` only provides monolingual implementations of this preprocessing step, however the experiment was conducted in two different ways, that is with and without stopwords. In all cases, the vector of terms found in the preprocessing phase was then analyzed with some of the most common supervised classification algorithms made available by `WEKA`. The classification models were evaluated with a 10 fold cross-validation technique and also using a separate test set. Finally, in Section 5.2.3, with reference to the approach with common words, unsupervised learning algorithms, in particular partitional and hierarchical clustering, were also used; this required a normalization of the instances so that they all had the same unit length.

The second case study is presented in Section 6, and corresponds to a simple classification task. In this case we used a tokenization algorithm, a list of stopwords that `WEKA` makes available for the English language and also a stemmer algorithm. The term vector analysis was then carried out with classification methods.

With these examples, we have described various scenarios for the different preprocessing steps summarized in Section 2 and have shown how it is possible to build supervised learning models for text categorization, using `WEKA` as the only tool.

CRediT authorship contribution statement

Donatella Merlini: Supervision, Conceptualization, Methodology, Writing - review & editing. **Martina Rossini:** Conceptualization, Methodology, Software, Writing - review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The authors of this paper wish to thank the anonymous referees for their useful comments and suggestions.

References

Aggarwal, C. C. (2014). *Data classification: Algorithms and applications*. Chapman and Hall/CRC.

Aha, D., Kibler, D., & Albert, M. (1991). Instance-based learning algorithms. *Machine Learning*, *6*, 37–66. http://dx.doi.org/10.1007/BF00153759.

Aly, W., & Kelleny, H. A. (2014). Adaptation of cuckoo search for documents clustering. *International Journal of Computer Applications*, *86*(1), 4–10.

Arthur, D., & Vassilvitskii, S. (2007). k-means++: the advantages of carefull seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms* (pp. 1027–1035).

Balbi, S. (2010). Beyond the curse of multidimensionality: high dimensional clustering in text mining. *Italian Journal of Applied Statistics*, *22*, 53–63.

Bijalwan, V., Kumar, V., Kumari, P., & Pascual, J. (2014). KNN based machine learning approach for text and document mining. *International Journal of Database Theory and Application*, *7*(1), 61–70.

Breiman, L. (2001). Random forests. *Machine Learning*, *45*(1), 5–32, https://link.springer.com/article/10.1023/A:1010933404324.

Cavnar, W. B., & Trenkle, J. M. (1994). N-gram-based text categorization. In *Proceedings of 3rd annual symposium on document analysis and information retrieval*.

Church, K. W. (2016). Word2vec. *Natural Language Engineering*, *23*(1), 155–162. http://dx.doi.org/10.1017/S1351324916000334.

Corley, C., & Mihalcea, R. (2005). Measuring the semantic similarity of texts. In *Proceedings of the ACL workshop on empirical modeling of semantic equivalence and entailment* (pp. 13–18).

Dan, L., Lihua, L., & Zhaoxin, Z. (2013). Research of text categorization on WEKA. In *Proceedings of the 2013 Third International Conference on Intelligent System Design and Engineering Applications* (pp. 1129–1131). http://dx.doi.org/10.1109/ISDEA.2012.266.

Debole, F., & Sebastiani, F. (2004). Supervised term weighting for automated text categorization. In *Studies in fuzziness and soft computing*: *Vol. 138*, (pp. 81–97). Berlin, Heidelberg: Springer, http://dx.doi.org/10.1007/978-3-540-45219-5_7.

Delany, S. J., Buckley, M., & Greene, D. (2012). SMS spam filtering: Methods and data. *Expert Systems with Applications*, *39*(10), 9899–9908. http://dx.doi.org/10.1016/j.eswa.2012.02.053.

Dhar, A., Mukherjee, H., Dash, N. S., & Roy, K. (2020). Text categorization: past and present. *Artificial Intelligence Review*, http://dx.doi.org/10.1007/s10462-020-09919-1.

Forman, G. (2003). An extensive empirical study of feature selection metrics for text classification. *Journal of Machine Learning Research*, *3*, 1289–1305.

Fung, B., Wang, M., & Ester, K. (2005). Hierarchical document clustering. In J. Wang (Ed.), *Encyclopedia of data warehousing and mining* (pp. 555–559). London: IGI Global, http://dx.doi.org/10.4018/9781591405573.ch105.

Gaikwad, B. U., & Halkarnikar, P. P. (2014). Random forest technique for E-mail classification. *International Journal of Scientific & Engineering Research*, *5*(3), 145–152.

Gali, N., Mariescu-Istodor, R., Hostettler, D., & Fränti, P. (2019). Framework for syntactic string similarity measures. *Expert Systems with Applications*, *129*, 169–185. http://dx.doi.org/10.1016/j.eswa.2019.03.048.

Goldhahn, D., Eckart, T., & Quasthoff, U. (2012). Building large monolingual dictionaries at the Leipzig Corpora Collection: from 100 to 200 languages. In *Proceedings of the 8th international conference on language resources and evaluation* (pp. 759–765). Istanbul, Turkey.

Grefenstette, G. (1995). Comparing two language identification schemes. In *Proceedings of the 3rd international conference on statistical analysis of textual data* (pp. 263–268). Rome, Italy.

Günal, S., S, S. E., Gülmezoğlu, M. B., & Gerek, O. N. (2006). On feature extraction for spam E-mail detection. In B. Gunsel, A. K. Jain, A. M. Tekalp, & B. Sankur (Eds.), *Multimedia content representation, classification and security, vol. 4105* (pp. 635–642). Berlin, Heidelberg: Springer, http://dx.doi.org/10.1007/11848035_84.

Gupta, V., & Lehal, G. S. (2009). A survey of text mining techniques and applications. *Journal of Emerging Technologies in Web Intelligence*, *1*(1), 60–76.

Hall, M., E., F., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I. H. (2009). The WEKA data mining software: An update. *ACM SIGKDD Explorations Newsletter*, *11*, 10–18. http://dx.doi.org/10.1145/1656274.1656278.

Hmeidi, I., Al-Ayyoub, M., Abdulla, N. A., Almodawar, A. A., Abooraig, R., & Mahyoub, N. A. (2014). Automatic Arabic text categorization: A comprehensive comparative study. *Journal of Information Science*, *41*(1), http://dx.doi.org/10.1177/0165551514558172.

Ji, L., Cheng, X., Kang, L., Li, D., Li, D., & Wang, Y. (2012). A SVM-based text classification system for knowledge organization method of crop cultivation. *IFIP Advances in Information and Communication Technology*, *368*(AICT PART 1), 318–324. http://dx.doi.org/10.1007/978-3-642-27281-3_38.

Jivani, A. G. (2011). A comparative study of stemming algorithms. *International Journal of Computer Technology and Application*, *2*(6), 1930–1938.

Joachims, T. (1998). Text categorization with support vector machines: learning with many relevant features. In *Proceedings of the European conference on machine learning* (pp. 137–142). http://dx.doi.org/10.1007/BFb0026683.

Kannan, S., & Gurusamy, V. (2014). Preprocessing techniques for text mining. *International Journal of Computer Science & Communication Networks*, *5*(1), 7–16.

Kao, A., & Poteet, S. R. (2007). Overview. In A. Kao, & S. R. Poteet (Eds.), *Natural language processing and text mining*. London: Springer, http://dx.doi.org/10.1007/978-1-84628-754-1_1.

Kaur, J., & Buttar, P. K. (2018). A systematic review on stopword removal algorithms. *International Journal on Future Revolution in Computer Science & Communication Engineering*, *4*(4), 207–210.

Konkol, M., & Konopík, M. (2014). Named entity recognition for highly inflectional languages: Effects of various lemmatization and stemming approaches. In P. Sojka, A. Horák, I. Kopeček, & K. Pala (Eds.), *Text, speech and dialogue. TSD 2014, vol. 8655* (pp. 267–274). Springer, http://dx.doi.org/10.1007/978-3-319-10816-2_33.

Ladani, D. J., & Desai, P. N. (2020). Stopword identification and removal techniques on TC and IR applications: A survey. In *6th international conference on advanced computing and communication systems* (pp. 466–472). http://dx.doi.org/10.1109/ICACCS48705.2020.9074166.

Lam, W., & Ho, C. (1998). Using a generalized instance set for automatic text categorization. In *Proceedings of the 21st annual international ACM SIGIR conference on research and development in information retrieval* (pp. 18–89). http://dx.doi.org/10.1145/290941.290961.

Leskovec, J., Rajaraman, A., & Ullman, J. D. (2020). *Mining of massive datasets* (3rd ed.). Cambridge University Press.

Lhazmir, S., El Moudden, I., & Kobbane, A. (2017). Feature extraction based on principal component analysis for text categorization. In *2017 international conference on performance evaluation and modeling in wired and wireless networks (PEMWN)* (pp. 1–6). http://dx.doi.org/10.23919/PEMWN.2017.8308030.

Luo, X. (2021). Efficient english text classification using selected machine learning techniques. *Alexandria Engineering Journal*, *60*(3), 3401–3409. http://dx.doi.org/10.1016/j.aej.2021.02.009.

Maalej, W., & Nabil, H. (2015). Bug report, feature request, or simply praise? On automatically classifying app reviews. In *2015 IEEE 23rd international requirements engineering conference, RE 2015 - Proceedings* (pp. 116–125). http://dx.doi.org/10.1109/RE.2015.7320414.

MacQueen, J. B. (1967). Some methods for classification and analysis of multivariate observations. In *Proceedings of 5-th berkeley symposium on mathematical statistics and probability* (pp. 281–297). University of California Press.

Maks, I., & Vossen, P. (2012). A lexicon model for deep sentiment analysis and opinion mining applications. *Decision Support Systems*, *53*(4), 680–688. http://dx.doi.org/10.1016/j.dss.2012.05.025.

Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to information retrieval*. Cambridge University Press.

Mccallum, A., & Nigam, K. (1998). A comparison of event models for naive bayes text classification. In: *Learning for text categorization: Papers from the 1998 AAAI workshop* (pp. 41–48).

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. *Advances in Neural Information Processing Systems*, 3111–3119.

Misuraca, M., & Spano, M. (2020). Unsupervised analytic strategies to explore large document collections. In D. F. Iezzi, D. Mayaffre, & M. Misuraca (Eds.), *Text Analytics. JADT 2018, vol. 8655* (pp. 17–28). Springer, http://dx.doi.org/10.1007/978-3-030-52680-1_2.

Mollas, I., Chrysopoulou, Z., Karlos, S., & Tsoumakas, G. (2020). ETHOS: an online hate speech detection dataset. https://arxiv.org/abs/2006.08328.

Nadeau, D., & Sekine, S. (2007). A survey of named entity recognition and classification. *Lingvisticae Investigationes*, *30*, http://dx.doi.org/10.1075/li.30.1.03nad.

Nigam, K., Mccallum, A., & Thrun, T. (1998). Learning to classify text from labeled and unlabeled documents. In *Proceedings of the 15th national/ 10th conference on Artificial intelligence/ Innovative applications of artificial intelligence* (pp. 792–799).

Pawar, P. Y., & Gawande, S. H. (2012). A comparative study on different types of approaches to text categorization. *International Journal of Machine Learning and Computing*, *2*(4), 423–426.

Platt, J. (1998). Fast training of support vector machines using sequential minimal optimization. In B. Schoelkopf, C. Burges, & A. Smola (Eds.), *Advances in kernel methods - Support vector learning*. MIT Press, https://www.microsoft.com/en-us/research/publication/fast-training-of-support-vector-machines-using-sequential-minimal-optimization/.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. San Mateo, CA: Morgan Kaufmann Publishers.

Rossini, M. (2020). *Analisi di comuni tecniche di text classification con WEKA. Tesi di laurea in Informatica (relatore D. Merlini)*. Italia: Università di Firenze.

Salton, G., & Buckley, C. (1987). Term-weighting approaches in automatic text retrieval. *Information Processing and Management*, *24*(5), 513–523.

Salton, G., Wong, A., & Yang, C. S. (1975). A vector space model for automatic indexing. *Communications of the ACM*, *18*(11), 613–620.

Shah, F. P., & Patel, V. (2016). A review on feature selection and feature extraction for text classification. In *Proceedings of the 2016 IEEE International conference on wireless communications, signal processing and networking* (pp. 2264–2268). http://dx.doi.org/10.1109/WiSPNET.2016.7566545.

Singh, J., Singh, G., & Singh, R. (2017). Optimization of sentiment analysis using machine learning classifiers. *Human-centric Computing and Information Sciences*, *7*(1), http://dx.doi.org/10.1186/s13673-017-0116-3.

Singh, V. K., Tiwari, N., & Garg, S. (2011). Document clustering using K-means, heuristic K-means and fuzzy C-means. In *International conference on computational intelligence and communication networks* (pp. 297–301). http://dx.doi.org/10.1109/CICN.2011.62.

Song, F., Liu, S., & Yang, J. (2005). A comparative study on text representation schemes in text categorization. *Pattern Analysis and Applications*, *8*(1–2), 199–209. http://dx.doi.org/10.1007/s10044-005-0256-3.

Tan, P. N., Steinbach, M., & Kumar, V. (2006). *Introduction to data mining*. Addison-Wesley.

Taskin, Z., & Al, U. (2018). A content-based citation analysis study based on text categorization. *Scientometrics*, *114*, 335–357. http://dx.doi.org/10.1007/s11192-017-2560-2.

Trstenjak, B., Mikac, S., & Donko, D. (2014). KNN With TF-IDF based framework for text categorization. *Procedia Engineering*, *69*, 1356–1364. http://dx.doi.org/10.1016/j.proeng.2014.03.129.

Usman, M., Ayub, S., Shafique, Z., & Malik, K. (2016). Urdu text classification using majority voting. *International Journal of Advanced Computer Science and Applications*, *7*(8), 1–10.

Uğuz, H. (2011). A two-stage feature selection method for text categorization by using information gain, principal component analysis and genetic algorithm. *Knowledge-Based Systems*, *24*(7), 1024–1032. http://dx.doi.org/10.1016/j.knosys.2011.04.014.

Uysal, A. K., & Gunal, S. (2014). The impact of preprocessing on text classification. *Information Processing and Management*, *50*(1), 104–112. http://dx.doi.org/10.1016/j.ipm.2013.08.006.

Vapnik, V., & Chervonenkis, A. (1964). A note on one class of perceptrons. *Automation and Remote Control*, *25*.

Witten, I. H. (2004). Text mining. In M. P. Singh (Ed.), *The practical handbook of internet computing*. Chapman and Hall/CRC.

Witten, I. H., Frank, E., & Hall, M. A. (2011). *Data mining: Practical machine learning tools and techniques* (3rd ed.). Morgan Kaufmann.

Yiming, Y., & Pedersen, J. (1997). A comparative study on feature selection in text categorization. In *Proceedings of the Fourteenth international conference on machine learning* (pp. 412–420).

Zhang, H., & Li, D. (2008). Naïve Bayes text classifier. In *2007 IEEE international conference on granular computing* (p. 708). http://dx.doi.org/10.1109/grc.2007.40.

Further reading

https://corpora.uni-leipzig.de/en. (Accessed 8 February 2021).

https://github.com/mwritescode/text-categorization-with-WEKA. (Accessed 8 February 2021).

https://sites.google.com/site/kevinbouge/stopwords-lists. (Accessed 8 February 2021).

https://waikato.github.io/weka-wiki/documentation/. (Accessed 8 February 2021).

https://weka.sourceforge.io/doc.dev/weka/core/stemmers/SnowballStemmer.html. (Accessed 8 February 2021).

https://www.cs.waikato.ac.nz/ml/weka/. (Accessed 8 February 2021).

https://www.futurelearn.com/courses/more-data-mining-with-weka. (Accessed 8 February 2021).