The University of Warwick

Assignment 8
Data Mining
CS909

Alberto Prospero

May 18, 2014

Instructors: Rob Procter Maria Liakata

1 Objective

The objective of the project has been to concretely analyze and correctly solve a specifically provided text mining problem. In particular, starting from a selected corpus of documents (the *Reuters21578* database), the set of relevant features was determined, and the connected classification and clustering tasks were consequently performed. Also, the considered documents, originally provided in a SGML format, were first evaluated and parsed, and the corresponding text elements has been properly preprocessed and examined.
In this report, the methods and the procedures which the solutions was found with are explained and motivated, and the possible sources of errors and/or difficulties which emerged during the work are pointed out. Moreover, the code utilized during the implementation is generally described and, in some cirmustances, reported.

2 Explanation of the python code

In order to concretely generate and obtain a practical solution for the text mining problem, the Python programming language was selected. It is a flexible tool which typically allows the user to perform complex operations by utilizing a low number of code rows. This important advantage, combined with a simple syntax, results in a substantial saving of time and efforts needed to implement a general procedure. Also, alongside several standard modules, Python can count on many external libraries that allow to carry out almost any type of computational operation, making it a complete and versatile programming language.
In particular, within the project the three external Python components *nltk*, *gensim* and *scikit* were used. The former is a rich open-source module composed by several algorithms and tools oriented to the analysis of human language data, and it allows to perform both basic and advanced operations related to the cleaning and preprocessing phase of a generic document. The second library is a free Python vector space and topic modeling toolkit, which in turn utilizes the NumPy, SciPy components in order to improve the attained results. The third module instead is a widely spread data mining and machine learning software suite, featuring various classification, regression and clustering algorithms including support vector machines, logistic regression, naive Bayes, random forests, gradient boosting, k-means and DBSCAN.
For the practical implementation of the overall solution therefore, the following Python modules, were created:

- *main.py* is the main program that has to be executed by the user. The retrieving, preprocessing, classification and clustering operations are set up in this file and executed by the corresponding external components.

- *SGMLReutersParser.py* allows to parse all the documents contained in the *Reuters21578* dataset by analyzing the corresponding tags and outputting a final Python dictionary containing the whole information retrieved.

- *TextPreProcessing.py* is the module containing all the methods and the algorithms aimed to carry out the text preprocessing phase. It utilizes the *nltk* library to practically and correctly analyze the document words.

- *Select.py* allows to perform the selection on the desired words within each of the documents. By utilizing the *nltk* library, it generates frequency distributions for the elements of the corpus and basing on the user preferences applies proper filters.

- *LdaTopicsModels.py* garantuees a concrete and complete interface with the *gensim* library. In particular, it allows to determine the proper topic model to be associated with the considered document, and to generate the tf-idf representation starting from the provided corpus and the given text.

- *confusion_matrix.py* allows to generate and store the confusion matrix plots.

Moreover, alongside with these external modules, different initial variables are set up at the very beginning phases of the execution of the main program. They allow to completely define the overall evolution of the Python code, tuning and constructing the different program parameters according to the user preferences. The following list specifically describes these components, though their actual meaning might be clear only after having read the subsequent sections of the report:

- *data_path* indicates the local folder containing the router files.

- *keep_first* determines whether the most recurrent or the less frequent topic word (with respect to the whole dataset) has to be maintained when there is more than one topic for an article. True means keeping the first one.

- *min_frequency* specifies the minimum frequency that a word must have (with respect to the overall training frequency list) in order to be maintained. This parameter is not actually utilized during the definitive version of the program.

- *first_words* specifies the number of most recurrent words to be maintained within the global training frequency list during the selection phase.

- *class_attribute* contains the name of the class attribute within the router dataset.

- *text_bag* indicates the feature representation which has to be utilized for creating the bag of words. The accepted values are: *none*, *binary_weights*, *term_frequency*, *tf*, *tfidf*.

- *selected_topics* specifies the topics over which performing the classification.

- *selected_keys* determines the parameters that have to be kept after the data are retrieved.

3 Retrieve the data

Before starting to consider and analyze any data mining technique, or any preprocessing task for a general set of instances, it is always necessary to have a solid method or a stable process that, when required, effectively allows to retrieve the data in the most reliable, accurate and fastest manner. In fact, especially when the regarded database continuously changes over time, it is important, in order to obtain detailed and precise predictions, that the most recent information is constantly identified and properly stored within the dataset. Also, when the elements that are taken into account are not explicitly provided, the correct generation of such a procedure becomes essential for allowing generic data mining strategies or even simple statistical evaluations to be applied over the desired data. Therefore, in both these cases a proper downloading or retrieving process has to be generated.

As it concerns the considered corpus, namely the *Reuters21578* dataset, it is important to notice that it is presented in a quite bad formatted pattern. All the 21578 documents contained in it in fact are encoded in the SGML web language, characterized by a sequence of tag delimiting the different parameters of interest. The author, the date, the texts and all the other significative variables are delimited by special strings having distinct specific meanings.

Hence, in order to concretely obtain a proper set of information and elements from the provided database, the generation of a practical method that allowed to deal with these kind of formats had to be accomplished. Consequently, the *SGMLReutersParser.py* Python module was created.

This component, initialized and called within the *main.py* file, allows to parse all the documents contained in the selected dataset by analyzing the corresponding tags and outputting a final dictionary containing the whole set of information retrieved. In particular, by utilizing the inner Python *sgmllib* library, it is able to detect and determine when a special character is encountered, subsequently selecting and formatting all the following text that it is incorporated in it. In fact, for each variable that has to be identified, two parameters are initialized during the creation of this object, and then respectively utilized to estabilish if the corresponding tag is hit and to store the desired values.

For example, in order to select the text contained within the SGML tag "text", the two variables `self.is_text = 0` and `self.text = ""` are generated during the initialization of the *SGMLReutersParser* module. When the correct tag is hit during the scan of one document, the function *start_text()* is called from the *sgmllib.SGMLParser* class, enforced as super class of the considered module. It is important to notice that if another tag is encountered during the browsing of the document, the corresponding *start_tagName()* is consequently invoked.

At this point, inside the *start_text()* routine, the variable `self.is_text` is set equal to one, to indicate that the special character related to the text has been selected. Immediately after this procedure is executed, the *handle_data()* method, that allows to detect and identify the string contained within a generic tag, is invoked in a similar manner (again it is invoked by the superclass). Within this function, the text tag is recognized by using the `self.is_text` variable, which at this time is the only boolean parameter equal to one, and the data string is hence stored inside the `self.text` element. Finally, after the *handle_data()* routine is executed, the *end_text()* function is called in a similar manner. Therein, the `self.is_text` variable is set equal to zero again.

Moreover, it is important to observe that when a single document is ended the method *end_reuters()* is invoked. Inside this routine, different splitting operations are accomplished for some of the retrieved strings (for example for the string author the "By " characters are erased when they are encountered in order to obtain a more fashionable format for this variable) and the Python dic-

tionary containing all the elements recovered during the scan of the document is generated. This is therefore inserted in the large list `documents`, before being re-initialized together with all the other boolean and text parameters within the function *_reset()* invoked afterwards.
At the end of all this process hence, the `documents` parameter contains a sequence of distinct dictionaries each corresponding to the results attained from a single article. It is returned to the main file, where it will be further analyzed, examined and processed during the following phases.
One final observation related to this initial section is that, according to the accomplished implementation of the *SGMLReutersParser* class, several parameters related to every document of the *Reuters21578* dataset are selected and identified, instead of limiting the research to the only basic features. In fact, within the realized module, not only the text of each article is retrieved and stored into the corresponding dictionary, but also the authors, the titles and even the dates are detected and collected. The reason is that I wanted to give the dataset as much information as possible for possibly subsequently utilizing the largest part of it, if deemed necessary. I have supposed that all the elements that can be obtained and recovered from the database had to be preserved, in order not to exclude data that could have been interesting and effective during the following classification and clustering tasks. This approach seems also quite justifiable and explainable from an implementation point of view. In fact, up to a statistical analysis or a proper data mining algorithm is not applied over the recovered element, it is not possible to determine the exact set of worth and significative characteristics. Therefore, in these situations, the best procedure surely is storing the whole available information in a well-constructed parameter, leaving the selection and choice of the relevant feature to the next phases.

4 First selections

After the retrieving of the data was performed, some initial selections were accomplished. In particular, the first choice carried within the Python code aimed to mantain only the features contained in the *selected_keys* list. In fact, although (as explained in the previous section) all the possible article information have been recovered and stored, only the actual texts were considered for each document. The reason of this is that even if the other collected attributes such as the *author* and *data* might perhaps be quite explicative and interesting (for example the classifier could match a particular author with a specific topic), the efforts were concentrated on the words of the documents, surely containing the most amount of information.
Secondly, only the articles belonging to the training or test instances and characterized by at least one of the ten provided topics were selected. In fact, all the models were built by utilizing the default splitting parameter (the LEWISSPLIT) and, as indicated in the assignment, the evaluation was concentrated on the most common classes. Therefore, in the case a the document contained no one of the selected elements, it was deleted.
Finally, when more than one class label for a single item characterized a document, the value which was globally more or less frequent was chosen (this setting is ruled by boolean parameter *keep_first*). The reason of this is that

this substitution allowed to deal with the class labels properly. By replacing in fact, when required, the list of values with the mode (or vice versa the less frequent variable) of the class distribution, all the articles were represented by one unique topic and the classification could take place.
However, it is important to remark that when the mode is selected, then only 9 topics are considered (instead of 10), since the parameter 'corn' always appears in the reuters dataset matched with other topics having a greater distribution (this does not occur if the less frequent value is considered, in which case the number of topic is equal to 10). Moreover, limiting at the test instances, the topic 'wheat' disappears, because again there are not instances.

5 Text Preprocessing

In order to properly implement any generic classification and clustering data mining operation over a considered text corpus, it is essential that the documents taken into account are well formatted and constructed. In fact, if this is not the case, it is not usually possible to determine a proper and correct set of common features and attributes that may be utilized during the following significative steps. Some of the regarded texts could contain unnecessary words or noisy elements that, if not handled properly, could affect the final global output producing considerable bias and errors.
Therefore, within the concrete implementation of the global problem, the *TextPreProcessing* module was created in order to deal with this kind of issue. This, by utilizing some of the algorithms and routines contained in the *nltk* library, allows to perform distinct and effective operations over a generic text string, transforming a given input component into a structured and precise ensemble of words that will be utilized during the following phases. In particular, within the Python code, the module class is invoked in the *main* file by using the command
`text_preproc = TextPreProcessing.TextPreProcessing(doc['text'])`,
which generates a new *TextPreProcessing* object by providing all the text related to one article as a string format. Then, in the *main* file, a sequence of routines aimed to respectively accomplish different and significant operations is called for each instance in the set of documents that it is created. In fact, this cleaning process has to be carried out for each of documents that are contained in the *Reuters* corpus, without distinguishing between the training and the test articles.
Specifically, the following function are called:

- *text_preproc.spell_check()* trasforms some english abbreviations in well written text, by adjusting and expanding the common English contractions. To this end, the internal python library *re* has been utilized.
  Moreover, a list of the main abbreviations can be retrieved online at the page http://en.wikipedia.org/wiki/Contraction. However, for evident reasons, the modification was carried out only when there was one single original full form starting from the contracted element.

- *text_preproc.tokenize()* generates tokens by utilizing a well-constructed regex instruction. In particular, the row
  `pattern = r'(?x) ([A-Z]\.)+ | [^\w\s]+ | \d+[\.,]\d+ |\w+ '` has

been utilized. It not only allows to identify the alphanumeric or numeric words, but it recognizes the terms formed by sequences of letters and dots, such as U.S.A or U.K.

- *text_preproc.POS_tagging()* allows to perform the POS tagging task according to the corresponding method of the *nltk* package. The call to this routine has to be carried out before the remaining text functions. In fact, it needs to have the exact and original sequence of terms and punctuations in order to obtain the best results. However, compared to the other text preprocessing functions, it surely requires the largest computational time to be executed for each document.

- *text_preproc.entity_names()* performs the entity names recognition by utilizing the corresponding component of the *nltk* library. In order to work properly, it requires a list of tuples each representing a specific word and the related tag. However, because a large number of wrong classifications was empirically observed, and not significative classification or clustering improvements was noticed by implementing this technique, it is not invoked during the last, definitive version of the program.

- *text_preproc.lowercase()* lowers all the words, transforming only the upper case elements. This method is called before the lemmatization, since this last process is based on a dictionary comparison, which is composed only by lower characters.

- *text_preproc.lemmatization()* carries out the lemmatization of the words, trying to match each term with its basic form according to a complete dictionary. In order to properly identify the exact semantic significance of a word, the *get_wordnet_pos()* function is utilized. In particular, it uses the POS tagging for constructing and obtaining a single character representing the lessical meaning of the term. Moreover, although this function does not require the same overall time as the POS tagging routine in order to be executed, it still demands a certain amount of computational cost to run.

- *text_preproc.remove_punctuation()* removes the punctuation from the considered elements.

- *text_preproc.remove_numbers()* removes all the elements containing one or more digits.

- *text_preproc.remove_stopwords()* deletes all the english stop words according to an internal list of elements provided by the *nltk* library and some forbidden words that have been arbitrarly chosen (for example the word 'reuters' that comes to the end of each database).

One important observation which concerns the implemented routines is that most of them are able to handle more than one input type. In fact, several of these generated functions can deal with both provided strings and simple lists, or lists and sequences having a tuple at each element. This property turned out to be really significative within the overall program, since it surely produced a relevant improvement of the portability and scalibility of the code. In fact, thanks to this characteristic, the execution order of these methods could

be easily changed and modified according more or less with any reasonable user preference. For example, the lowering process could be moved before the lemmatization or after the numeric elements have been deleted, if deemed necessary. Or the punctuation could be moved before performing the POS tagging process over the selected elements. In such cases, the only operations to perform consist in properly modifying the corresponding lines within the Python code. However, it is quite evident that not only the possible changes are allowed. In fact, not all the routines can be executed basing on an arbitrarly or random order.
A second relevant remark regarding the methods described above is that each of them, along with processing a default resulting element, it is able to accept a provided user input. In particular, besides considering the parameter deriving from all the functions called before, it is possible to invoke every regarded method by passing a desired string, list, of tuple list. When this occurs, the called routine selects and processes the user input, neglecting the default component. Instead, if this is not the case, an opposite choice is carried out. By invoking for example the function *lowercase()* without giving any parameter (namely calling *text_preproc.lowercase()*) then the element resulting from all the methods invoked before of it (i.e. *text_preproc.spell_check()*, *text_preproc.tokenize()* and *text_preproc.POS_tagging()* in the code) is taken into account and consequently modified. However, when an input parameter is provided, the routine selects and processes it, leaving unchanged the default component.
Therefore, this characteristic not only grants an easy an practical way to test the implemented functions, but at the same time it allows to utilize them in other informatic contexts or different external frameworks.
Finally, it is necessary to remark that this phase surely requested the highest time in order to be correctly and completely executed. As pointed out above in fact, the functions *text_preproc.POS_tagging()* and *text_preproc.lemmatization()* are particularly demanding in term of computational cost, and then this phase could take a lot of time in order to be accomplished (about one hour on my laptop). For this reason, within the Python code, the library *cPickle* was utilized. This, allows to easily and quickly save a generic element in a file, retrieving it when necessary. Therefore, before starting to compute this phase, the program estabilish if this operation (computed with the selected value of the *keep_first* parameter) was already carried out by searching the corresponding file. If it is found, the list containing the document dictionaries is recovered and subsequently used.

6 Feature selection

Once that the texts related to all the documents were analyzed and processed, the selection of the relevant features took place. In particular, within the Python code, all the words connected to the training samples were first collected in a common list and subsequently ordered with respect to their frequency. Then, only the most 150 recurrent ones were kept (hence `first\_words` = 150 was fixed) . This choice was due to the fact that this number guaranteed some of best performances in terms of classification. Nevertheless, it is important to remark that it was not effectively possible to test the program with the largest number of features. In fact, starting from a value of 180 overall words, the computational and memory costs required to execute the code were actually unsustainable and

unsuitable for the computer, and hence no tests were possible in these cases.
However, from an implementation point of view, the *Select* class was utilized in order to produce the right set of training features. It, along with different practical methods which allow to construct the frequency distributions related to the element provided, contains practical and specific functions that grant to perform the selection of the terms either by picking the n most recurrent words discarting the remainings (the *select_n_frequent_words_ ** routines are suited to this purpose) or by choosing those terms which are more frequent than a index specified by the user (the *select_upto_n_frequent_words_ ** methods were constructed to this purpose).
Hence, for concretely select the most frequent terms within training instances, the following row was generated:

```
all_words = sel.select_n_frequent_words_all(first_words)
```

Furthermore, after this command was provided, a cycle for skimming all the documents according to the generated *all_words* list was created. Actually, because in the following feature engineering phase all the representation are computed by starting from the *all_words* list, this process is not necessary and therefore it surely could be deleted. However, along with decreasing the total computational cost required during the next process, this cycle provides some useful basic functions to perform, if required, other types of selections and therefore, in the final version, it has been mantained.

7 Feature engineering

In order to deal with a generic text mining problem, different feature representations can be computed and applied. In particular, analyzing the word recurrences or evaluating their frequencies with respect to the remaining documents, distinct measures can be retrieved and identified starting from the texts taken into account. In fact, after the text preprocessing phase has be carried out, one can generally utilize one of the attributes summed up and described in the following list, finally choosing that one which guarantees the best performances:

- *Binary weight* $B(t, d)$ For each word t that has been selected during the previous phases (therefore the text preprocessing and the feature selection), it is estabilished if the considered document d has that precise term within its text. If this is the case, a positive binary value is recorded (so $B(t, d) = 1/True$), otherwise a negative parameter is stored (so $B(t, d) = 0/False$). In this way, each document is characterized by a sequence of positive or negative elements that represent it. This measure is practically the most applied during text classification tasks.

- *Term frequency (TF)* $TF(t, d)$ For each word t that has been selected during the previous phases, the number of time n that term appears in the considered document d is evaluated and then recorded (so $TF(t, d) = n$). In this particular representation therefore, each document is represented by a sequence of integer each indicating how many times the corresponding word occurs within the text. This measure is surely more precise that the previous one, since it conveys more information. However, it tends to favour the longer elements, characterized by an higher number of terms.

- *tf* $tf(t,d)$ This feature representation weights the *term frequency* measure $TF(t,d)$ by dividing it for the maximum recurrence of the selected word in any document that is taken into account. So, defined as $M = max\{TF(t,d) : d \in D\}$ where D is the ensemble of regarded documents (in our case, all the training dataset), then $tf(t,d) = TF(t,d)/M$.
  By utilizing this representation, the bias towards longer documents, characterizing the $TF(t,d)$ measure, is prevented.

- *Document frequency (DF)* $df(t,D)$ For each word t, the number of items in the collection containing this term is recorded. Namely $df(t,D) = |\{d \in D : t \in D\}|$

- *idf* $idf(t,d)$ This feature representation rescales the df measure, first by dividing the total number of documents for DF and then computing the logarithm on the result. Namely, $idf = \log \frac{D}{df(t,D)}$.

- *tf-idf* This feature representation is defined as the product $tf * idf$. It expresses how much a term is for a particular document, and can be very useful in representing a document as a bag of word. However, it can only be computed when it is known all the documents in the collection. Also for this reason, although it is effectively implemented as one possible feature representation and practically tested in the following classification phase, it was not selected as definitive measure.

From an implementation point of view, all the described feature representations except for the DF and IDF measures were generated within the Python code. As it regards the first measure, although it is quite simple to be computed, it is not informative with respect to the document taken into account. By definition in fact, it counts the number of items in the collection containing the considered term, and therefore it does not depends at all on the regarded item. Instead, as it concerns the second neglected representation IDF, it can be easily recovered by utilizing the inverted expression $idf = tfidf/tf$ if necessary, though this explicit computation has not been carried out within the program. Moreover the tf measure, though it has been implemented, was not utilized, because computationally too expensive.
Anyway, all the reamining representations can be easily computed within the Python code, by setting the desired value of the parameter `text_bag`, defined at the top of the *main.py* script. In particular, the accepted values are *none, binary_weight, TF, tf, tfidf*.
Moreover, in order to determine the selected measures, the class *select*, containing the *nltk* functions necessary to construct the term frequencies, was utilized extensively.
Also, all the representations except for the $tfidf$ have been constructed following the definitions explained above for each element in the training and in text set (always starting by an ensemble of words defined and retrieved from the training samples in both the cases, as explained in the previous section), and the exact code instructions can be visualized in the middle part of the *main.py* file and (hopefully) understood by reading the large and extensive provided comments. As it concerns the $tfidf$ measure instead, it has been computed by utilizing the *gensim* library and the Python class *LdaTopicModels* expressly created to guarantee a concrete interface with the basic routines contained in this module.

However, the use of this external component is not the only method to compute this measure. Another possibility is to generate the IDF measure basing on the definition provided above and then computing this representation accordingly, or utilize different libraries, such as *Scikit*, that allows to calculate this parameter in an easy and practical manner.

8 Topic Models

Another important feature which can be retrieved starting from the text contained in each document is the topic model. Every article in the database can be associated, through a particular statistical process, with a sequence of semantically related concepts or words, which in turn define a specific theme or subject for the item taken into account. This procedure, when effective, can be really important and essential in solving generic text mining problems. In fact, not only it allows to obtain another relevant feature necessary in terms of classification or analysis over the considered documents, but at the same time it turns out to be significant in several practical situations, when for example unstructured collections of texts have to be automatically connected to precise arguments.

For these reasons, the topic model technique was taken into account and implemented within the Python code. To this end, the class *LdaTopicModel* was created. This, initialized and utilized within the *main.py* file, grants a proper interface with the *gensim* library, which in turn is formed by different routines aimed to re-write the data in a useful and common vector space and consequently build distinct models over them. Specifically, two of the most utilized models allows to determine the $tfidf$ measure over the selected instances and to identify the subject of one specific document by utilizying the *Latent Dirichlet Allocation*, which is the simplest method that grants to construct the topic probability distributions related to a picked article. In this regard, the *gensim* component accepts an integer number indicating how many topics have to be globally generated for the provided corpus (this number can be specified by setting the value of `number_topics`). For the implemented program, this number was set equal to ten (or nine), according to the number of class labels that were generated from the reuter dataset.

However, after the model is built, for each provided article a given distribution between all the overall themes based on the precise characteristics of the document text is decided. By providing for example the first element contained in the reuters dataset, the following vector is returned:

$[(9, 0.072003890208168941), (8, 0.32551040462684844), (4, 0.59541299900861366)]$

As it possible to observe, in this case the model returned three possible topics for the given item, represented by the numbers 9,8,4. Different probabilities were associated to them, while implicit zero likelihoods were instead assigned to the remaining theme (described by the remaining seven numbers 1,2,3,5,6,7,10).

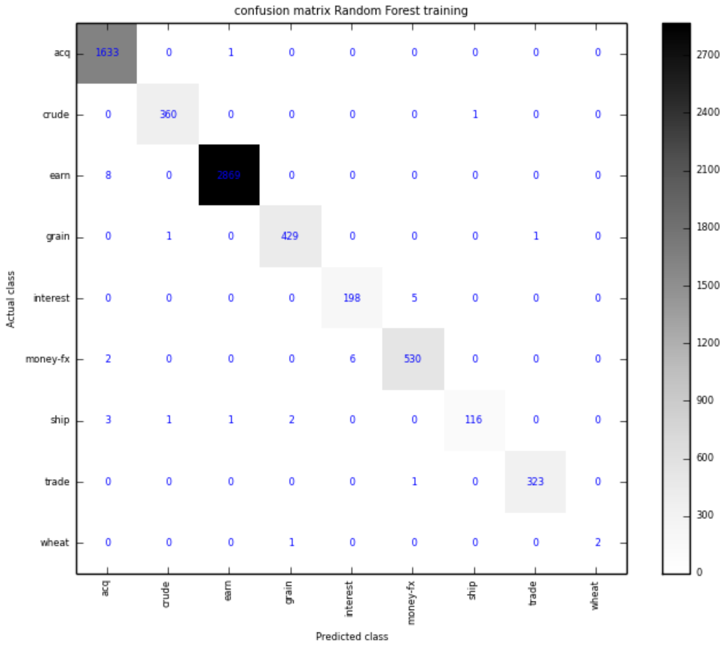
At this point, it is important to point out that the basic starting corpus was, once again, provided by all the words in the training dataset. In fact, in order not to introduce any bias in the test instances, the model was trained and constructed starting only from these known instances (as already did within the feature engineering phase).

Moreover, this section can be easily avoided by setting the variable `topic_model`, located at the beginning of the *main.py* file, equal to false. Although this choice does not particularly speed up the program, it causes, in several cirmunstances, a quite significative decrement of the global performances of some classification learners, which reduce their capability of predicting the correct final class label (to be convinced about that, it is useful to compare the measurements contained in the text files when the topic model was utilized with those obtained when this feature was not taken into account).
However, neither using the topic model by its own is a good choice in terms of classification. As it possible to observe for example in the text file *training_measures_150_none_True_True.txt* or *test_measures_150_none_True_True.txt* (obtained by setting `topic_model = True` and `text_bag = 'none'`), poor performances were attained when this single variable was provided alone.

9 Classification

Once that the feature selection, feature engineering and topic model phases were properly accomplished, the classification over the training and test instances was performed. In particular, the *scikit* library was utilized. This module, along with to guarantee a practical and simple technique to convert the list of dictionaries created into a numpy array which can be analyzed and processed, allows to concretely generate the different classification models and test them over the selected data.
By using this library therefore, the desired learners were created. Specifically, the *Naive Bayes* (gaussian when discrete feature representations were considered, binomial in the other case), *Random Forest*, *SVM* (with linear kernel), *One layer perceptron* were chosen. This decision does not come from a particularly strong motivation. Actually, when a data mining problem is taken into account and then analyzed, some picked algorithms can be tested according to precise personal (and objective) hypotheses. However in some situations, these considerations could be wrong or inaccurate. Sometimes the selected learners could produce unexpected results, without confirming the considered assumptions. Instead, other and not previously regarded methods might, in those situations, be more efficient and effective. For this reason it is important, when facing a data mining problem, not to be stuck too much with a technique or a strategy, but try different methods that might provide better results.
As it concerns the considered problem therefore, the most common and utilized methods to solve a generic text mining problem were basically utilized. Obviously, this choice does not guarantees an optimization of the performances, since it could evidently exist another method (or a combination of different techniques) performing much better with respect to the algorithms taken into account. However, this selection allows to start from a well consolidated base of classification learners which have been proved to be quite effective and stable in dealing problems of this kind. Moreover, it is important to point out that they globally offer wide alternative ways of carrying out the classification task, based of different approaches and distinct concepts coming, in some circumstances, from various and not completely related mathematical branches. In fact, while *SVM* and the *One layer perceptron* are linear classifiers, basically built from

the geometrical concept of distance, *Naive Bayes* and *Decision trees* arise from other data mining suppositions and ideas. Therefore, when considered all together, these methods globally ensure quite diversified approaches, that hardly can be considered independent or related each other. This characteristic can be obviously really helpful and useful to evaluate or at least have an idea about the overall performances which can be obtain for the regarded problem.



	acq	crude	earn	grain	interest	money-fx	ship	trade	wheat
acq	1633	0	1	0	0	0	0	0	0
crude	0	360	0	0	0	0	1	0	0
earn	8	0	2869	0	0	0	0	0	0
grain	0	1	0	429	0	0	0	1	0
interest	0	0	0	0	198	5	0	0	0
money-fx	2	0	0	0	6	530	0	0	0
ship	3	1	1	2	0	0	116	0	0
trade	0	0	0	0	0	1	0	323	0
wheat	0	0	0	1	0	0	0	0	2

Figure 1: Confusion matrix related to the random forest algorithm for the training instances.

Regarding the tuning of the parameters, this process was only necessary for the *SVM* algorithm. Unlike the other models in fact, the most important parameter deciding the penalization has to be provided in this case, and it can significatively change the output of the algorithm. In this regard, after that the best combination of initial feature was selected (it will be exposed in a second), different tests were run in order to compute this variable (the different output obtained are in the file containin a 'SVM' at the end part of their name). As a result a value of $C = 3.0$ was selected.

Once the classification strategies were selected and implemented, different tests were run over the training instances in order to select the best learner algorithm. In particular, all the measures that have been collected over the training set are reported during the execution of the program and some of them are subsequently written in a proper text file. In turn, each of these file is named according to the particular sequence of (some) initial variables that have been created at the top of the *main.py* script. Specifically, every text element begins with *'training_measures'*, and the following five elements respectively represent the total number of words that have been maintained from the training set, the type of feature representation that has been chosen, the absence/pres-

ence of the topic model and the selection of the first or last word when more than one class label characterized an article. So, for example, the text file: *training_measures_100_binary_weight_True_True* contains all the measures related to the classification over the training instances with 100 total training words selected, using a binary_weight feature, the topic model and the mode value of the word distribution when different labels characterize a document. In a similar way, the text file related to the test classification corresponding to the best feature combination determined can be read (the name of which starts with *'test_measures'*).
Each of these text elements contain therefore almost all the measures that are first retrieved and then printed for every learner during one single execution of the *main.py* file. While the *accuracy*, *recall*, *precision*, *micro-precision*, *micro-recall*, *macro-precision*, *macro-recall* are recovered and displayed during the execution of the *main.py* script, the *accuracy*, *micro-precision*, *micro-recall*, *macro-precision*, *macro-recall* (therefore all of them except for the *recall* and *precision*) are reported in the corresponding files. The reason of this is that because the number of class labels is fair high, namely 9 or 10, the vectors generated from these measures are not easy to compare each other, and hence they risk of distracting preventing from acquiring a global view of the metrics.
Instead, it is more useful in this case have a look to the micro and macro averaged measures, which sum up in one number the information contained in the vectors. As it regards the difference between them, while *macro* precision or recall calculate metrics for each label, and find their mean, the *micro* measures

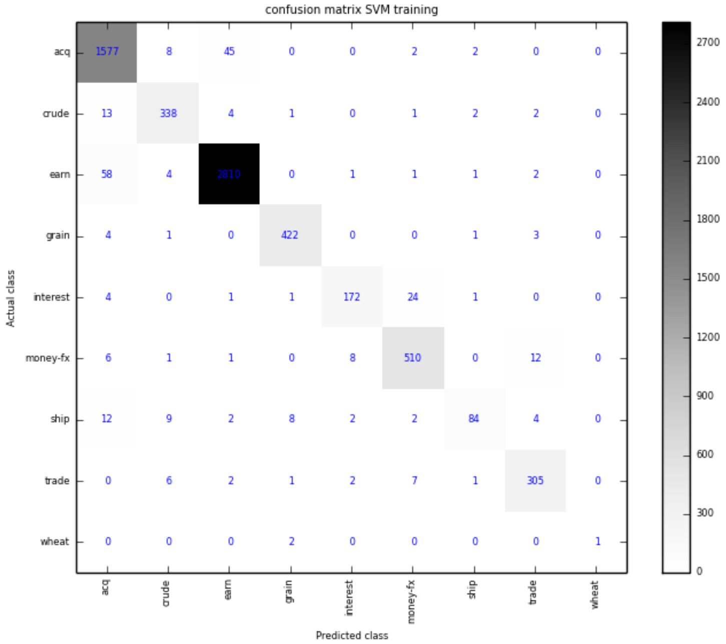


Figure 2: Confusion matrix related to the SVM algorithm (with linear kernel) for the training instances.

compute metrics globally by counting the total true positives, false negatives and false positives. Hence, macro-average methods can be used when one want to know how the system performs overall across the sets of data, whereas micro-average can be a useful measure when the dataset varies in size.
By comparing all the results obtained and stored in the corresponding text file, it is possible to notice that when the words of text are considered and therefore a particular feature representation is constructed, almost all the methods perform very well. In particular, the majority of the measures are more or less above the 88% and hence very good results are obtained. However, an obvious and evident decrease of performance can be noticed by evaluating the 9 class labels learners (deriving from setting the `keep_first` parameter equal to true) or 10 class labels instances (deriving from setting the value of `keep_first` parameter equal to false). In order to be convinced, compare for example the text file *training_ measures_ 150_ binary_ weight_ True_ False* with *training_ measures_ 150_ binary_ weight_ True_ True*. This difference does not seem to particularly affect the random forest algorithm. Another remark is that the SVM and the Naive Bayes methods seems not work very well when the 'tfidf' measure is utilized. Instead, the *Random Forest* algorithm appears to reach amazing results over the training data in all the situations. Nevertheless, considering that according to this algorithm the decision trees generated from time to time are not pruned, this is probably due to an overfitting problem.
In conclusion, the best feature representation appears to be, the *binary_ weight*. For this component in fact all the algorithms achieve the best (or almost the best) performances, and the measures obtained are for this reason, more stable. Once that this feature representation is selected, it is possible to notice that the best performances are obtained when 150 training words are picked, the topic model are provided and 9 class labels are utilized. In such a case, the measures obtained from all the four learners (excluding recall and precision) are reported in the following table. Moreover, also to provide a more detailed information with respect the two best performing algorithms, the plots 1 and 2, representing the confusion matrices related to the Random forest and SVM, are represented. As it concerns the final choice of the algorithm to test over the data, the more reasonable choice seems to be given either by the SVM or by the random forest (the perceptrion performs quite well though), keeping however in mind that for this last algorithm an overfitting process could be occured. Considering both these techniques over the test instances, it can be discovered that they both perform quite well. The requested values of *accuracy*, *precision* and *recall* (and for all the micro-macro measures) for these two methods (and indeed for all the remaining ones) are displayed during the execution and stored in the text file *test_ measures_ 150_ binary_ weight_ True_ True*. Moreover, in the table 2,

Learner	**Accuracy**	**Micro-Precision**	**Macro-Precision**	**Micro-Recall**	**Macro-Recall**
Naive Bayes	0.8831	0.8831	0.7065	0.8831	0.7391
Random Forest	0.9947	0.9947	0.9918	0.9947	0.9507
SVM	0.9576	0.9576	0.9465	0.9576	0.8455
Perceptron	0.9274	0.9274	0.7926	0.9274	0.8297

Table 1: Classification measures obtained over the training data with 150 total considered words, binary weight, topic models and 9 class labels.

some of these metrics are reported.

As it possible to observe, the random forest algorithm and the SVM seem to be the most effective learner (for the random forest this could however means that the words characterizing the test documents are similar to those of the training set), while the other methods guarantee good predictions overall. In particular, the Naive Bayes method perform better on the test instances than over the training ones. A global view of the attained results is also shown for the SVM and Random Forest strategies in the graphs 3 and 4, representing the corresponding attained confusion matrices.

Learner	**Accuracy**	**Micro-Precision**	**Macro-Precision**	**Micro-Recall**	**Macro-Recall**
Naive Bayes	0.8975	0.8975	0.7772	0.8975	0.7749
Random Forest	0.9113	0.9113	0.8433	0.9113	0.7704
SVM	0.9215	0.9215	0.7259	0.9215	0.7075
Perceptron	0.8999	0.8999	0.7002	0.8999	0.6859

Table 2: Classification measures obtained over the test data with 150 total considered words, binary weight, topic models and 9 class labels.

Finally, it is important to remark that other feature constructions or selections could be analyzed and taken into account, and they could possibly provide better results. Although in fact a quite wide set of different parameter combinations were considered and implemented, these do not cover all the potential variable

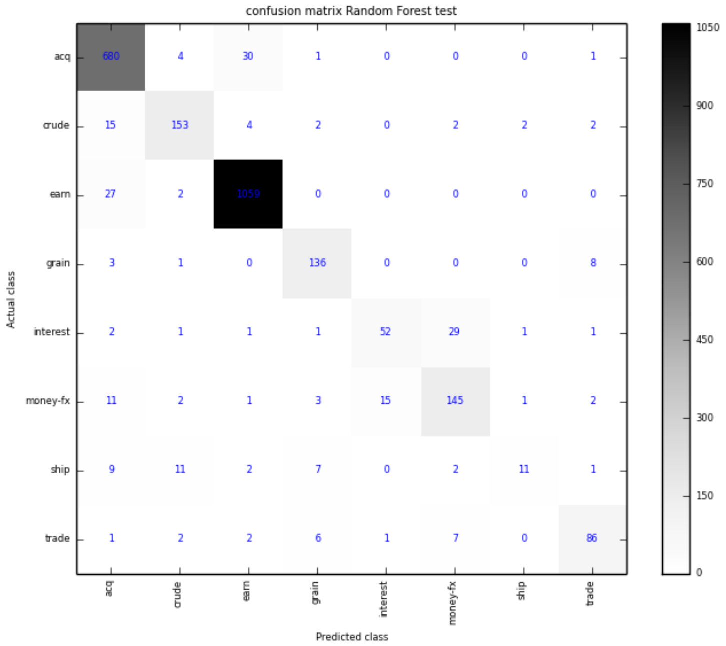


Figure 3: Confusion matrix related to the random forest algorithm for the test instances. Note that for the test instances the actual number of topics is reduced to 8 (no instances of the wheat label)

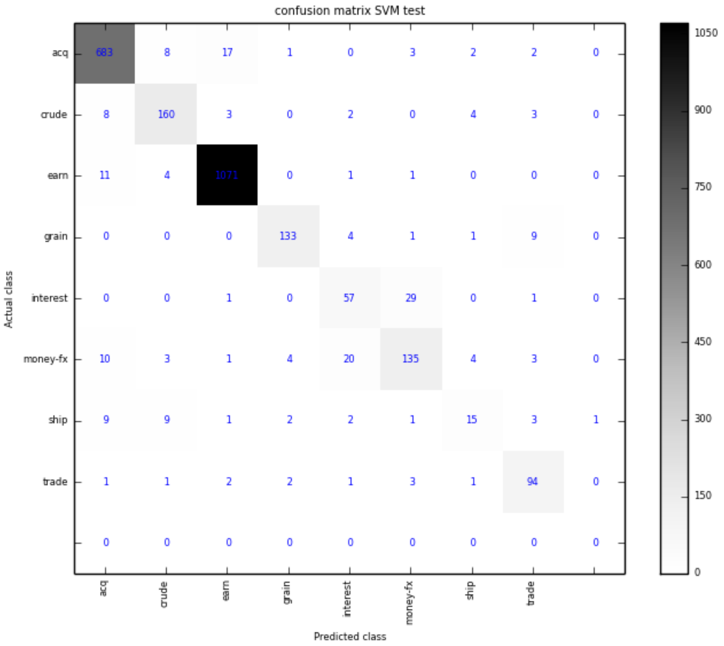


Figure 4: Confusion matrix related to the SVM algorithm (with linear kernel) for the training instances. Note that for the test instances the actual number of topics is reduced to 8 (no instances of the wheat label)

associations that can be accomplished. For example in fact, the `first_words` component, indicating the number of words that are considered, could be easily modified from the values of 100 or 150. But then, because it is basically a continuous parameter, this process would generate an outstanding number of combinations, which could not be tested for computational problems. In fact, although one possible solution could be printing distinct plots tracking the evolution of the results obtained, they, in order to be complete, should report each training measure with respect to each possible combination. But then, the total computational cost (and time) would become surely high and definitely prohibitive. In conclusion, it can not be guaranteed that the selected features and variables produce the best results, but surely, as empirically can be observed, they generate more than acceptable performances, which, considering the difficulty and the complexity of the classification task, can be definitely regarded as a good outcome.

10 Clustering

The last section of the project dealt with the clusterization of the dataset. Starting from the best performing features introduced in the previous section, the selected documents were grouped and arranged in different clusters, each ideally corresponding to a particular prediction topic. In particular, a set of 9 class labels (`keep_first = True`) was selected (however, all the arguments

that follow can perfectly and easily be applied for 10 class labels by setting `keep_first = False`) and 150 training words, 9 topic models and a binary feature representation were chosen.
Moreover, in order to carry out this operation, the *scikit* library was utilized once again. This module in fact, offers a quite wide range of techniques and methods that allow to properly tackle and face the clusterization problem (a complete list of such strategies can be retrieved at the link http://scikit-learn.org/stable/modules/clustering.html). In particular, among all the possible implementable routines, the following three algorithms have been selected:

- *K-Means*, which is maybe the most utilized algorithm, allows to produce a number of clusters arbitrarly estabilished by the user. In order to that, it first computes the centroids randomly and then iterativelly adjusts their values according to the data provided. However, the default *scikit* implementation selects initial cluster centers in a smart way to speed up convergence, instead of choosing them randomly.
  The number of clusters to be generated was enforced equal to 9, basing on the number of different class labels which globally characterized the reuter dataset in this case.

- *Ward*, which performs a clustering based on the Ward algorithm. This method represents a particular implementation of the larger hierarchical approach family, which are a group of clustering algorithms quite utilized in dealing with real problems. Specifically, nested clusters are iterativelly built by merging them successively, after that, at the very beginning, each of the instances is associated with a single cluster element. This hierarchy of clusters is represented as a tree (or dendrogram), in which the root is the unique cluster that gathers all the samples, while the leaves are the clusters connected, as said before, with only one sample. Again, the number of cluster provided to the algorithm (namely the number of distinct groups required) was equal to 9. Also this method (as expected), compared to the other two techniques, requires the highest time in order to be executed.

- *DBSCAN* this algorithm views clusters as areas of high density separated by areas of low density. Due to this rather generic view, clusters found by DBSCAN can be any shape, as opposed to k-means and hierarchy methods which assume that clusters are spherical shaped. The central component to the DBSCAN is the concept of core samples, which are samples that are in areas of high density. A cluster is therefore a set of core samples, each close to each other (measured by some distance measure) and a set of non-core samples that are close to a core sample (but are not themselves core samples). Unlike the other implemented methods, it does not accept an enforced number of clusters. Moreover, it is not particularly sensitive to outliers and noise data, though it strongly depends on the input variables which are provided to it. In this regards, different choices have been tested for the ϵ, denoting the distance within the neighbours are searched, and $minThreshold$, indicating the minimum number of neighbours elements above which the considered point is defined as a core object. However, for all the trials no significative improvements were obtained, and therefore the default selections have been left.

The reason of these methods lies, once again, that these algorithms are some of the best known and applied clusterization techniques. Moreover, as in the case of the classification phase, they offer a quite wide range of different starting assumptions and ways of cluster construction, globally ensuring a certain level of independence and uncorrelation.
Once that these three methods were selected and implemented within the Python program, an evaluation over the clusters obtained was accomplished. Such an evaluation over the performance of a clustering algorithm is not as trivial as counting the number of errors or the precision and recall of a supervised classification algorithm though. Actually they can be used, but generally other methods are utilized, especially when there an effective knowledge of the ground truth set of classes (which, fortunately, in our case are provided by the topic labels). The measures taken into account have to satisfy the separations of the data according to these ground truth elements or answer to some specific assumptions such that members belong to the same class are more similar that members of different classes according to some similarity metrics.
For these reasons, the following measures were taken into account:

- *Homogenity*, which measures how much each cluster contains only members of a single class.

- *Completeness*, which measures how much all members of a given class are assigned to the same cluster.

- *V-measure*, which is the harmonic mean between the measures of homogenity and completeness.

- *Silhouette coefficient*, which is a measure that can be computed starting from the model itself, hence even when the ground truth labels are not known. In particular, for each instance x_i in the dataset, this value is computed according to:

  $$s(x_i) = \frac{b(x_i) - a(x_i)}{\max(a(x_i), b(x_i))} \tag{1}$$

  where $a(x_i)$ is the average distance between the element x_i and all the remaining components of the same class and $b(x_i)$ is the average distance between the instance and all other points in the next nearest cluster. In the scikit implementation, alongside with this effective definition, an useful measure, summing up the global silhouette behaviour is defined. In particular, this silhouette score is bounded between -1 for incorrect clustering and +1 for highly dense clustering. Scores around zero instead, indicate overlapping clusters.

While the first three measures are quite intuitive elements that offer a fair easy interpretation and that can be quickly computed, the last one requires the most number of operations in order to be performed. In this regard, normally lunching the computation of this measure within the Python code produced a memory error, and therefore, for this reason, a quite reasonable sample of elements (8000) were selected in order to compute it.
Moreover, it is important to remark that all the four values offer bounded scores, and basically they do not make any assumption on the cluster structure.

Finally, with respect to the first three element, the average scores were retrieved instead of vectors (by the given definition in fact, *Homogenity*, *Completeness*, *V-measure* all depend on the class over which they are recovered), in order to have more compact and unique values. This concept applies also for the last measure, for which the mean silhoeutte coefficient over some of the samples, as said before was utilized. A graph representing the value of this measure for each of the instances could not be generated and reported, since the corresponding *scikit* function did not accept sample elements as input (also for this reason the Silhouette coefficient was utilized).
The results are therefore presented in the table 3. Higher the value of a measure better are the performances of the clusterization related to it (except for the Silhouette coefficient, as explanied previously).

Algorithm	**Homogenity**	**Completeness**	**V-measure**	**Silhouette mean**
K-means	0.469	0.349	0.400	0.014
DBSCAN	0.018	0.205	0.033	-0.301
Ward	0.541	0.418	0.472	0.021

Table 3: Clustering measures obtained over all the dataset with 150 total considered words, binary weight, topic models and 9 class labels.

As it possible to observe, the classification phase surely guarantees better results, even if the clusters obtained manage to capture the majority of the instances grouping them in the correct subsets. However, the elements obtained are not so good as expected. Moreover, for the relative poor level of performances, it is not possible to associate each cluster to a specific topic model. Looking at the *homogenity* parameter in fact, on average each cluster contains at most (for the Ward technique) a 54.1% members of a single class, and hence a straight and useful correspondence between the actual class labels and the clusters can not be retrieved. This concept is also sustained and supported by the *completeness parameter*, and therefore also by the harmonic mean between the two variables, namely the *V-measure*. Unfortunately, with respect to these three measures, it is not possible to investigate more in detail. In fact, the scikit package does not allow to report the corresponding results for each of the clusters generated, as instead it can be done for the *recall* and *precision* metrics.

11 Github

All the code and the files can be found at the github url:
https://github.com/colibri17/TextCategorization