

Adding Sentiment Analysis support to the NLTK Python Platform

Pierpaolo Pantone



Master of Science
Artificial Intelligence
School of Informatics
University of Edinburgh
2015

Abstract

This MSc thesis discusses the implementation of Sentiment Analysis support within the *Natural Language ToolKit*. We review several approaches and corpora for Sentiment Analysis and discuss the feasibility of their incorporation into *NLTK*, considering parameters such as the maturity of the approach, the availability of related resources and their licensing terms and conditions. We then describe the steps required for their inclusion in the platform, and the implementation of a lightweight structure that allows users to employ and compare the discussed approaches. Finally, we evaluate our work in the context of open source practices, using automated testing tools and asking for feedback by the *NLTK* community.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Professor Ewan Klein for accepting me for this project, for his immense helpfulness and for his active collaboration in every stage of my work. Working under his supervision has been stimulating and challenging, and I am very grateful for this experience.

I thank my friends for supporting me during the whole year and for sharing so many sleepless nights with me in Appleton Tower and in the university library. I thank Sarah, David, Nora and Anna for their presence and for being such interesting persons and good friends. I am also deeply grateful to my Italian friends, who always encouraged me, for believing so much in me.

I want to thank my friend Angela for her clever humour and for her dedication to everything she does. Her strength has been a continuous inspiration to me, and her directness helped me to turn problems into challenges.

I thank my sister Alida for being so unpredictable and for cheering me up during difficult moments.

Finally, I want to thank my parents, simply for everything. Their unconditional support has been more than I could ever ask, and it has always been.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Pierpaolo Pantone)

Table of Contents

1	Introduction	1
1.1	Objectives	2
1.2	Sentiment Analysis	3
1.3	<i>NLTK</i> and <i>GitHub</i>	5
2	Main approaches to Sentiment Analysis	9
2.1	Machine Learning	10
2.1.1	Naïve Bayes	10
2.1.2	Maximum Entropy	11
2.1.3	Support Vector Machines	12
2.1.4	Unsupervised learning	13
2.2	Lexicon-based Sentiment Analysis	14
2.2.1	Lexicon generation	15
2.3	Aspect-based Sentiment Analysis	16
3	Lexicons and corpora	19
3.1	Lexicons	20
3.1.1	Opinion Lexicon	20
3.1.2	<i>MPQA</i> Subjectivity Lexicon	22
3.1.3	Harvard General Inquirer	23
3.1.4	Linguistic Inquiry and Word Counts (LIWC)	23
3.1.5	SentiWordNet	23
3.1.6	Vader Lexicon	24
3.2	Document-level corpora	24
3.2.1	Polarity Dataset	25
3.2.2	Experience Project	25
3.2.3	Product and Political Debate Corpora	25

3.2.4	Sentiment140	26
3.2.5	STS-Gold	27
3.2.6	Sanders Analytics Twitter Sentiment Corpus	28
3.2.7	TASS 2013 General Corpus for Spanish	28
3.2.8	SemEval-2014	29
3.3	Sentence-level corpora	30
3.3.1	Pros and Cons	30
3.3.2	Sentence Polarity Dataset	30
3.3.3	Subjectivity Dataset	31
3.4	Aspect/feature-level corpora	32
3.4.1	Customer Review Datasets	32
3.4.2	Comparative Sentence dataset	33
4	Preprocessing and feature extraction	35
4.1	Whitespace tokenization	37
4.2	Penn Treebank-style tokenization	37
4.3	Punctuation	38
4.4	Length reduction	39
4.5	Letter case	39
4.6	Negation scope	39
4.7	Other relevant scopes	41
4.8	Emoticons	41
4.9	Hashtags and handles	42
4.10	Stopwords	43
4.11	POS tags	43
4.12	Dependency structure	44
4.13	Stemming	44
4.14	Word Frequency	45
4.15	Unigrams and bigrams	45
4.16	Final remarks	46
5	Design and implementation	47
5.1	Corpus readers	47
5.1.1	ProductReviewsCorpusReader	48
5.1.2	ComparativeSentencesCorpusReader	51
5.1.3	CategorizedSentencesCorpusReader	56

5.1.4	OpinionLexiconCorpusReader	57
5.1.5	ProsConsCorpusReader	58
5.2	SentimentAnalyzer	58
5.2.1	Feature extractors	59
5.2.2	Other methods	62
5.3	Demos and helpers	63
5.3.1	Preprocessing and feature extraction	63
5.3.2	Demos	65
5.3.3	Other functionalities	66
5.4	Documentation and Testing	68
5.4.1	Docstrings and Sphinx	68
5.4.2	Doctests and Tox	69
5.5	PEP-8 and Pylint	70
5.6	Comparisons	71
6	Evaluation and discussion	77
6.1	Community feedback	77
6.2	Experience with <i>NLTK</i>	78
6.3	Future work	79
7	Conclusions	81
	Bibliography	83

Chapter 1

Introduction

The unstoppable expansion of the Web has led, in the past years, to a huge increase of stored and exchanged data. Recent estimates evaluated that this *digital universe* will consist, by 2020, of about 44 zettabytes¹ of data; while a single number might not be sufficient to concretely convey the idea of a similar quantity, it has been calculated that 42 zettabytes would be enough to store *all words ever spoken by human beings* (Lieberman, 2003; EMC, 2014; Gantz and Reinsel, 2012; Roe, 2012). This massive amount of information can be considered part of the widespread interest in data analysis and Machine Learning research.

The overwhelming quantity of information that is conveyed through the Web, mostly using protocols such as HTTP, TCP, UDP, POP and FTP, represents a significant source of knowledge in a wide range of fields. Data Analysis techniques can be used, for example, for various tasks in Information Security, Computer Graphics, Information Retrieval and, in general, in those settings that require a large amount of data to be examined and evaluated in the smallest amount of time.

While part of this data consists of structured information, whose syntax is explicitly declared and whose content can be consistently² stored in databases and other organized collection systems, most of the Web is conversely populated by unstructured information. This kind of data does not have a univocal and unambiguous structure, and cannot be therefore directly employed by automated systems unless a specific interpretation strategy is provided.

As noted in Liu (2010), until recently most of the interest about unstructured data has focused on Text Mining, Information Retrieval and topic classification tasks. In

¹1 zettabyte is 1 quadrillion megabytes.

²In reality, *consistency* should not always be assumed as granted. The entire field of Data Integration and Exchange focuses on consistency proofs and techniques to fix data inconsistencies.

particular, textual data provided by Web users represents one of the most useful and interesting sources of unstructured information that can be currently found. Under this light, the entire field of Natural Language Processing evolves together with the increase of computational power and the discovery of new techniques which are aimed to interpret such textual information.

However, in the past ten years the interest in textual information classification has extended to opinionated text, whose availability has grown thanks to the World Wide Web and, in particular, to social networking platforms such as Facebook and Twitter. One of the most fascinating applications of NLP techniques is in fact nowadays related to Sentiment Analysis, also known as Opinion Mining, that we will introduce in section 1.2.

1.1 Objectives

Our project aims to provide support for Sentiment Analysis (see section 1.2) within the *Natural Language ToolKit*³, a widely used Python library for Natural Language Processing that will be discussed in section 1.3. This task can be further split in two sub-tasks regarding *review* and *implementation* details.

Review. In order to decide which kind of support should be incorporated into *NLTK*, a review of the range of existing approaches and corpora for Sentiment Analysis is required. This review should provide considerations in order to decide which elements will receive support during the *implementation* phase. Useful parameters would take into consideration aspects such as the maturity of proposed approaches, the availability of related resources, licensing terms and issues, and specific implementation requirements that could eventually hinder the feasibility of implementing the specific approach within *NLTK*.

Implementation. Following the considerations made during the *review* phase, a working implementation of the proposed elements should be provided. This task takes into account the following requirements:

- the implementation should provide working functionalities especially designed for teaching and research purposes. Code should therefore be readable, documented and easy to use;

³<http://www.nltk.org>

- new classes and components should be implemented to incorporate suitable resources (i.e. corpora and lexicons) for Sentiment Analysis tasks within *NLTK*;
- new implemented functionalities should be able to represent a base for further development of new approaches and experiments;
- the implementation should provide a sort of framework to be employed as a standard way for combining different approaches (e.g. classifiers, corpora, tokenizers, preprocessing functions and feature extractors);
- new functionalities should make use of existing *NLTK* libraries and components whenever they are available;
- code should follow open source best practices and *NLTK* community guidelines (see section 1.3).

We now proceed to provide a brief description of *Sentiment Analysis*, defining its main characteristics and scopes.

1.2 Sentiment Analysis

While knowledge produced by automated systems is usually (and quite simplistically) considered as univocal and necessarily consistent, user-generated content reproduces those human dynamics that lead to inconsistencies and hardly-verifiable information. In other words, humans express *opinions*.

In general, what is needed to extract meaning from an unstructured source of information is an interpretation or model. Humans unrelentingly face the challenge of interpreting each other's messages, not only understanding factual contents of a communication, but also (and sometimes more importantly) discriminating between factual expressions and subjective opinions. However, tracing the boundary between a fact and an opinion is not always straightforward, even for humans.

Sentiment Analysis can be defined as the task of computationally extracting information about sentiments and opinions from text. From a practical point of view, what makes Sentiment Analysis interesting is the possibility to *automatically* derive information about user preferences and tastes from a huge unstructured dataset, which is the Web. Companies currently employ Sentiment Analysis techniques to understand user opinions about their products without having to rely on surveys and other expensive and time-consuming procedures. This is part of the reason why user-generated

contents like tweets or Facebook statuses are so precious: opinions can shine a light on upcoming trends, provide feedback about existing products and, in general, shape the next business move of a successful company. However, opinions have to be correctly interpreted, and a good interpretation requires assigning some meaning to those symbols that we can extract from documents. Working with text, the choice of the symbols that we want to consider can already tell something about the meaning that we attribute them; for example, adding a negation suffix to unigrams that fall in the scope of a negative sentence means that we consider them as bearers of a particular meaning, which differentiates them from non-marked unigrams. We will further discuss these considerations in chapter 2.

What needs to be taken into consideration when discussing Sentiment Analysis is that this area is almost as complex as the concept of *opinion*. Sentiment Analysis tasks can therefore have different objectives. First of all, we need to distinguish an opinionated text from a non-subjective text; this is what is usually defined as *subjectivity classification*, and it is often employed as the first step in hierarchical classification approaches. The second level of the hierarchy consists in *sentiment classification* (or *polarity classification*). This allows us to establish whether the text that we considered as subjective or opinionated in the first step is expressing, for example, a positive or a negative opinion about *some topic*.

This is exactly where things start to become more complicated: knowing that a tweet or a product review is providing a positive opinion about something is often not enough. In fact, we most probably want to know in much detail as possible what that *something* is, if a product, a movie, a political party, etc. And once again, we might now face another challenge: given that, for example, our user-generated document is describing a specific model of camera, we can ask ourselves which exact feature of that camera (e.g. zoom, viewfinder, LCD screen) is being positively or negatively described by the user who is reviewing it. This is what *aspect-based Sentiment Analysis* is about, and it often subsequently leads to *opinion summarisation* tasks, which consist in the generation of an output that summarises thousands of user-generated opinions in a compact meaningful representation (Jindal and Liu, 2006b,a; Hu and Liu, 2004a).

Note that, even during the initial stages of the analysis, many complications can arise. As Liu (2011) efficiently shows, the concepts of *sentiment*, *subjectivity* and *emotion* are not equivalent. Sentiments, in fact, can also be expressed by factual statements (e.g. “*The fire destroyed all my work*”, “*The bridge collapsed ten days after opening*”), and this is enough to recognise that they are neither equivalent nor in-

cluded in the concept of subjectivity. On the other hand, emotions are only expressed in the realm of subjectivity, but they do not always entail specific opinions (e.g. “*I was surprised to see what was happening*”). Moreover, we can express sentiments and opinions without necessarily be guided by any particular emotion⁴ (e.g. “*The company leaders are all experienced workers*”, “*The style of this book is concise*”). All these distinctions have to be taken into consideration if we desire to trace more precise boundaries for Sentiment Analysis.

Even once we move inside these limits, a multitude of different techniques and approaches are available. If we assume that the first step of our analysis involves simple raw text (unstructured data), we will necessarily have to face tasks such as data retrieval, preprocessing, feature and entity extraction, etc. This is the reason why Sentiment Analysis can benefit from research in a wide range of related fields; nonetheless, its peculiarities also require us to be careful about the initial assumptions we make, as we will show in the next chapters.

In order to provide a better structure for the concepts we depicted so far, three levels of detail can be used to describe Sentiment Analysis:

- sentence-level
- document-level
- feature/aspect-level

Our review and implementation takes into consideration each of these levels, obtaining and providing optimal corpora that are currently used for Sentiment Analysis in a multitude of research papers and concrete implementations. We will describe the incorporation of these corpora into *NLTK* in detail in chapter 5.

1.3 NLTK and GitHub

The *Natural Language ToolKit* is an open source project mainly created for research and teaching purposes, whose community-driven approach continuously produces new prompts to experiment and incorporate both mature and cutting-edge Natural Language Processing techniques. The framework is therefore often employed by students and researchers to build prototypes for NLP tasks; this specific kind of use requires *NLTK*

⁴As much as it might seem a Stoicistic position, most respectable opinions in many fields are still considered those that are not conditioned by any strong and potentially misleading passion.

code to be clear, modular and open, and these aspects, along with the self-contained nature of the platform, have often been praised by the academic community (Madnani and Dorr, 2008).

Python, the language that has been chosen to develop *NLTK*, meets the requirements of clarity and conciseness that currently make the entire platform one of the most used NLP tools available. However, the language of choice is not enough to guarantee that the code will be readable and understandable, especially for teaching purposes. For this reason, *NLTK* project leaders provide specific guidelines for developers who want to contribute to the project (NLTK, 2015a,b,c; Bird et al., 2009).

Contributions to *NLTK* are managed through *GitHub*, a repository hosting service for distributed version control and code management. *GitHub* provides statistics and functionalities such as Issues, Wikis and Pull Requests that ease the entire distributed development process, making it easily reviewable by both contributors and project supervisors. In our case, *GitHub* is used at the same time to propose new functionalities, review them and eventually submit them to the main project branch once they have been discussed by the community. With this strategy in mind, the project is almost entirely developed on a separate branch called `sentiment`, so that its final version can be proposed for a single submission request on the official main branch.

In order to facilitate the reading of the next chapters, we present a list of the main terms that are commonly used throughout *GitHub*, taking the cue from the glossary provided by the company (GitHub, 2014):

Branch A parallel version of the original project (which usually resides on the `master` branch⁵). Developers generally implement new features on separate branches and then merge these changes into `master` when their code is considered complete.

Commit A change made to one or more files. Each commit is saved with its unique hash identifier, so that it is possible to track single commits and create a history of all changes made during the project development.

Fork A copy of the original project repository that can be modified without affecting the original code. Forks are linked to their source, allowing developers to interact with the main repository code (e.g. proposing changes or fetching latest updates).

⁵The main *NLTK* branch is called `develop`.

Issue Bugs, problems or suggestions can be reported as Issues on the repository pages. This is one of the most valuable components of *GitHub* and open source projects, since it allows a deeper involvement of the community in the development of the project.

Pull Request When working on a distributed environment (as an open source project) it is not advisable to grant every developer the right to directly merge changes from his branch into the `master` branch. Developers can therefore submit their proposed changes through a Pull Request, which can then be revised by a project collaborator who will accept or reject it.

Our approach consists in creating a fork of the original *NLTK* project, implementing our code on a specific branch called `sentiment` (occasionally interacting with a second branch named `twitter`), and periodically using Pull Requests to propose our changes to the `sentiment` branch of the original repository. This strategy will also allow us to gather feedback from the *NLTK* community, as we will discuss in chapter 6.

Chapter 2

Main approaches to Sentiment Analysis

Popular Sentiment Analysis surveys usually describe existing approaches using the three-levels structure that we introduced in chapter 1. We focus our review on the techniques that are mostly employed to implement them, rather than on the types of instances they analyze.

The most interesting results in the field of Sentiment Analysis are mainly obtained employing Machine Learning techniques or lexicon-based analyses. Machine Learning can be described as the process of automatically inferring patterns and structures from data, ideally providing as few as possible domain-specific instructions to the machine that has to accomplish the task. As we will see, it is still not possible to simply ask a machine to guess a pattern without providing it with some guidelines that reflect our assumptions: this extreme flexibility is something that still has to be reached, especially if we take performance issues into consideration. However, a reasonable balance between this idea and very specific step-by-step instructions can still be reached thanks to Machine Learning and statistical methods.

On the other hand, at the price of a lower flexibility, alternative approaches can guarantee reasonable results and good performance using specific analytic procedures. Specifically, lexicon-based approaches parse the data they are fed with (e.g. a natural language sentence) and output a final result comparing the words they encounter with a finite lexicon of opinion words. A score or polarity can then be assigned to each relevant word, and the output will be obtained using some specific techniques to combine all individual word scores. Since these methods are mostly based on look-up tables and simple procedures, their speed performance can significantly outperform that of

most Machine Learning approaches. However, lexicons are often too limited, difficult to obtain, and in general tightly dependent on their specific domain.

It is important to note that Machine Learning and lexicon-based methods should not be considered as mutually incompatible. These techniques, in fact, can also be combined, for example using a lexicon to extract specific word features that will then be fed to a statistical classifier, as shown in Das and Chen (2001).

In order to outline the main differences between these two types of approaches we will provide a review of several Machine Learning and lexicon-based techniques, especially focusing on those methods that appear to be more compatible with the *NLTK* platform. Concluding, we will also present a different approach aimed to provide more subtle distinctions in Sentiment Analysis classifications based on aspect and feature analysis.

2.1 Machine Learning

Machine Learning predictors are widely employed and examined throughout the entire Sentiment Analysis literature. Some of these classifiers, namely *Naïve Bayes*, *Maximum Entropy* and *Support Vector Machines*, have produced the best results in subjectivity and polarity classification so far, and can be therefore rightfully considered as the most interesting subjects for our review.

2.1.1 Naïve Bayes

Naïve Bayes is probably the most commonly employed generative classifier in Natural Language Processing. Its advantage is mainly given by its simplicity, which can however also be considered as its main disadvantage. In *Naïve Bayes*, in fact, all feature values extracted from each instance we examine (during training or testing phase) are simply multiplied together. The underlying assumption behind this strategy is that each feature produces a contribution for the final classification result and that its contribution is equally important as the others. In other terms, we are considering every token we extract from our text as being equally important with respect to every other extracted token. It is important to highlight, however, that we are only considering those tokens that have already been *extracted* from the data, such as the most frequent unigrams or bigrams. In Machine Learning, all other words and symbols that were part of the original raw text are simply not considered anymore, since our instance is now

a feature vector that *represents* the original text.

As we described, the main assumption of *Naïve Bayes* classifiers (which is what makes them *naïve*) is that every feature or attribute of an instance is considered *independent* from all other features, given the class. This is what allows us to multiply all members of a feature vector. We therefore have the following formula:

$$P(y|\mathbf{x}) = \frac{P(\mathbf{x}|y) P(y)}{\sum_{y'} P(\mathbf{x}|y') P(y')} \quad (2.1)$$

where y is the class (e.g. *positive* or *negative*) and \mathbf{x} is our instance, defined by a feature vector $\{x_1, x_2, \dots, x_n\}$, with n being the number of features. We then have that $P(y|\mathbf{x})$, the Bayesian probability of a class given an instance, is equal to the likelihood $P(\mathbf{x}|y)$ of that instance being seen under that specific class, multiplied for the prior probability of the class $P(y)$; the result is then normalized so that the final probabilities sum up to 1.

Intuitively, the independence assumption seems inappropriate for natural language. Sentences and documents are almost never made of completely independent words and they assume a certain degree of syntactic, semantic, lexical and even stylistic coherence. This peculiarity of the language plainly appears when discussing, for example, about cognitive research, garden paths and incremental parsing (Hale, 2001; Bever, 1970). Extracting n-gram features from text is a way to mitigate this issue trying to amend inaccuracies caused by the independence assumption. Nonetheless, Sentiment Analysis embraces very peculiar areas of our language, where a single word can change the entire polarity of a document. We do not therefore necessarily expect to find a very strict correspondence between opinionated text and coherence.

Naïve Bayes classifiers are usually employed in text classification tasks together with features based on word frequency counts. However, previous research shows that word frequency is not a key feature for Sentiment Analysis tasks, especially when working with short texts as sentences or Twitter tweets. We will discuss more about this aspects in chapter 4.

2.1.2 Maximum Entropy

Maximum Entropy, also known as *logistic regression* (or *multinomial logistic regression* when working with multiple classes), is a generalization of basic *logistic regression* methods, in which we basically employ a discriminative logistic regression approach to perform classification tasks.

With *Maximum Entropy* we can specify complex features and train our model so that it will assign a different real-valued weight to each of them. In this way we avoid problems caused by independence assumptions and we can confer a specific importance to each individual feature. Assuming we want to achieve a multi-class classification, the final outcome of the classification is then given by the formula:

$$P(y = k|\mathbf{x}) = \frac{\exp(\mathbf{w}_k^T \mathbf{x})}{\sum_{j=1}^C \exp(\mathbf{w}_j^T \mathbf{x})} \quad (2.2)$$

where the following holds:

- k is the specific class for which we are computing our prediction;
- \mathbf{x} is the feature vector that represents our instance;
- \mathbf{w} is the vector of all weights;
- $\exp(x)$ is the exponential function e^x ;
- $\mathbf{w}_k^T \mathbf{x}$ is the dot product of \mathbf{w}^T and \mathbf{x} (for class k);
- $\sum_{j=1}^C$ represents an iteration over all feature vectors for all classes (relatively to our specific instance). So $\sum_{j=1}^C \exp(\mathbf{w}_j^T \mathbf{x})$ is our normalizer.

In order to choose the best weight vector \mathbf{w} during the training phase of our classification (in a supervised learning environment) we choose the weights that make our predictions closer to the actual label of the instance. This means that we need to find parameters that minimize the error function of our classifier maximizing the log of the likelihood, and we often do it using a numerical optimization method such as *gradient descent*. While “descending” the error surface we learn better weights, and this operation can be repeated for a predefined number of iterations until we reach acceptable results.

2.1.3 Support Vector Machines

While *Naïve Bayes* and *Maximum Entropy* are characterized by their probabilistic nature, a third category of classifiers relies on the idea of *maximum margin*. Given that two (or more) classes of instances are linearly separable by some decision boundary,

it is still possible in theory that different weight vectors can define boundaries that, although less precise, achieve the same final classification. Every hyperplane that can be placed in between two classes of instances can then produce a linear separation. However, it is still possible to find a single weight vector that maximizes the margin between the hyperplane and the nearest instance of each of the two classes. In this case, we do not only find one of the possible hyperplanes, but we find the one that maximizes the margin from both classes we want to separate.

In mathematical terms, we can compute the weights that define the maximum margin optimizing the following problem:

$$\begin{aligned} \min_w ||\mathbf{w}||^2 \\ \text{such that } y_i(\mathbf{w}^T \mathbf{x}_i + w_0) \geq +1 \text{ for all } i \end{aligned} \quad (2.3)$$

which leads to the following:

$$\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i \quad (2.4)$$

where $\alpha_i = 0$ for instances that are not support vectors¹.

As Joachims (1998) shows, many text categorization problems are linearly separable, and SVM performance on this kind of problems is very competitive. Our implementation consequently takes SVM into consideration, providing demos and methods that can incorporate it and combine it with other *NLTK* functionalities.

2.1.4 Unsupervised learning

A specific mention has to be reserved to the unsupervised learning method proposed in Turney (2002). In this case, *Pointwise Mutual Information* is used to compute the distance between each phrase of a document and a very limited set of seed words (i.e. “*excellent*” and “*poor*”); resulting values are subtracted to obtain the phrase polarity score, and finally all phrase scores are averaged to derive the overall document polarity.

However, in order to compute PMI scores, Turney’s approach heavily relies on Web search results returned by a specific query engine.² As noticed in Hu and Liu (2004a), this aspect causes his work to be computationally inefficient; moreover, since phrase scores are not reported in the paper and the Web search engine was shut down in 2013, new results are not easily comparable.

¹We call *support vectors* those class instances that contribute to trace the final hyperplane.

²We refer to the *AltaVista* Web search engine, which was acquired by *Yahoo!* in 2003 and shut down ten years later.

2.2 Lexicon-based Sentiment Analysis

As we described, Machine Learning approaches employ statistical methods to predict subjectivity and polarity classes for texts. Lexicon-based methods, on the contrary, analytically derive text polarity based on a set of words, each of which is annotated with specific information that contributes to extract the final orientation of the text.

Basic dictionaries split opinionated words into two sets of *positive* and *negative* word types, while more complex datasets also include finer-grained classes. Numeric scores can also be reported for each word, and the final *semantic orientation*³ is computed by average, summing up all values and dividing the result by the number of opinionated tokens in the instance that is being analyzed (Taboada et al., 2011).

One of the main advantages of lexicon-based approaches over Machine Learning techniques is that their performance can be consistent among different domains. Classifiers trained on data that belong to some specific domain, on the contrary, often suffer from a dramatic performance drop when they are tested on different domains.

Moreover, when dealing with negation and other specific scopes, statistical approaches require more data in order to counterbalance the growth in the number of features (see chapter 4). Using a lexicon, instead, it is possible to specifically handle *contextual valence shifters* like *truly*, *rather*, *not* etc., attributing to each of them a specific score or percentage value that will be combined with the score of the following tokens (Kennedy and Inkpen, 2006; Taboada et al., 2011); an expression like “*truly interesting*” will then be considered stronger than “*rather interesting*”, because of the intensifier valence that is being included in the computation. Negation shifters are mostly used to flip score values changing their sign (*switch negation* approach): supposing that *good* has a score of +3, *not good* would have a score of −3. Other techniques like *Vader* (Hutto and Gilbert, 2014) and *SO-CAL* (Taboada et al., 2011) use a softer approach (*shift negation*), highly modifying negated tokens valence without completely inverting their score.⁴

³*Semantic orientation* is a measure that includes information about word *orientation* (e.g. *positive/negative*) and opinion *strength* or *potency* (Taboada et al., 2011).

⁴Specifically, *Vader* approach gives a value of −0.74 to negative valence shifters, multiplying this score for the token value specified in the lexicon. *SO-CAL*, instead, modifies scores moving their value by a fixed amount (4) towards the opposite polarity.

2.2.1 Lexicon generation

As described in Liu (2010), lexicons can be *manually* populated collecting and labeling polar words, or *automatically* generated exploiting linguistic and structural cues. Automatic generation can in turn be implemented extracting opinion patterns from large corpora, or using dictionaries such as WordNet (Miller, 1995).

Corpus-based approaches aim to detect word orientations by looking for syntactic and co-occurrence patterns. Hatzivassiloglou and McKeown (1997) assume the concept of *sentiment consistency* to describe constraints that hold between pairs of co-occurring adjectives, subsequently using connectives to infer their orientation; starting from a seed set of adjectives whose polarity is already known, we can infer other adjective polarities thanks to the connectives that are used to link them when they co-occur. For example, supposing that the word “*tasty*” belongs to our seed of positive words, from the sentence

“Food was *tasty* but *expensive*”

we can infer, thanks to the connective “*but*”, that the adjective “*expensive*” has a negative orientation. Conversely, if we have the sentence

“Food was *tasty* and *cheap*”

we can derive that “*cheap*” has to be classified among the positive words. It is possible to notice that when a domain-specific corpus is employed, this strategy allows us to create lexicons specialised on that specific domain.

Dictionary-based methods also move from a small set of seed words whose polarity is already known. Large dictionaries such as WordNet are subsequently employed to search for their synonyms and antonyms that will be added to the seed set with their respective inferred label (Hu and Liu, 2004a). This process, which is iteratively repeated until no more words can be added, has the downside of not distinguishing between different senses of the same word (e.g. “*small*” in “*The room is too small*” and “*This laptop is light and small*”) and therefore cannot be used to create specialised domain-based lexicons. However, this approach can in general be employed to generate lexicons with a greater number of words if compared to common *corpus-based* approaches.

2.3 Aspect-based Sentiment Analysis

We conclude our review by discussing a third approach to Sentiment Analysis which employs the techniques we previously explained to extract more complex and structured sentiment and subjectivity information from texts.

The main assumption behind many of the approaches we previously described for sentence-level classification is that each sentence is mostly considered as bearer of only one opinion. This generalisation becomes even stronger with *document-level* analysis, since each document can be entirely classified as being positive or negative, regardless of its internal structure and complexity. Many studies try to deconstruct this assumption, producing a deeper analysis of the different parts that constitute an opinionated text.

In particular, Liu (2010) provides a formal definition of the concept of *opinion*⁵ in order to identify the steps that can be performed to achieve more detailed analyses. An opinion can then be defined as a quintuple

$$(e_j, a_{jk}, so_{ijkl}, h_i, t_l)$$

where the following holds:

- e_j is the entity discussed by the opinion;
- a_{jk} is an aspect or feature of the entity (and the opinion is focusing on it);
- h_i is the *holder* (or source) of the opinion (e.g. a customer writing a review on *Amazon.com*);
- t_l is the time when h_i produced his opinion;
- so_{ijkl} is the sentiment orientation of the opinion given by h_i at time t_l regarding the aspect a_{jk} of entity e_j ;

Even if we simplify the definition assuming we already have information about time and opinion holders, it is still clear how this approach requires more information and structure than *document* and *sentence-level* classification methods. Assuming we have the following sentence in a digital camera review:

“4x zoom is nice”

⁵We are discussing about *direct opinions*, which are different from *comparative opinions* that concern more than one object.

we can infer that the *entity* is the specific camera that is being reviewed, the *aspect* is the zoom, and the *sentiment orientation* is represented by a positive (and not too strong) opinion⁶.

Supervised classification approaches can be used to extract entity aspects/features from annotated textual data, applying patterns that combine sequences of wildcards (to match different features) followed by their POS tag (Liu, 2007). Specifically, Liu (2010) reports the following explanatory steps:

1. Sentence: “Included memory is stingy”

2. POS-tagging: <{included, VB}{memory, NN}{is, VB}{stingy, JJ}>

3. Human-labeling: <{included, VB}{\$feature, NN}{is, VB}{stingy, JJ}>

The third step just replaced the original feature (memory) with the \$feature wildcard. This allows us to extract *label sequential rules* as shown by the following example:

<{easy, JJ}{to}{*, VB}> → <{easy, JJ}{to}{\$feature, VB}>

Extracting features from text is therefore straightforward at this point, since we only need to store the word that is captured by the \$feature wildcard whenever a rule pattern is encountered.

However, these rules may not guarantee good results for texts that are not constructed using a pros-cons structure⁷ (*free format* reviews); in these cases, POS tags are used to identify frequent nouns and to sift the data looking for potential feature candidates. Each sentence that contains some aspect about an entity can be subsequently classified using Machine Learning or lexicon-based methods for *sentence-level* analysis, as we described in the previous sections (Hu and Liu, 2004b; Ding et al., 2008).

⁶Note that this structure can also be applied to other domains such as movie reviews, politics, music etc., and not just to product features.

⁷In this type of structure, *pros* and *cons* are explicitly stated and distinguished. E.g.: “*Pros: bright screen - Cons: short battery life*”. As we will describe in Chapters 3 and 5, we add a suitable corpus of pros and cons for this specific task into *NLTK*.

Chapter 3

Lexicons and corpora

As we described in chapter 2, several Sentiment Analysis approaches make use of algorithms that have already been employed in fields related to text classification and Natural Language Processing tasks. For this reason, many classifier implementations (like *Naïve Bayes*, *Maximum Entropy* and *SVM*) have already been included in *NLTK* during the past years.

However, the platform currently only provides a single dataset specifically built for Sentiment Analysis, which is the *Polarity Dataset* by Lee and Pang (described in section 3.2.1). We find that a significant number of discussions and entries in blogs and *Question & Answer* websites refer to this specific data, but this has the downside of producing a lot of redundant publications that often focus on the same kind of *document-level* analysis. We therefore believe that the main issue we need to address is represented by the lack within *NLTK* of a consistent and richer set of corpora that can be employed for Sentiment Analysis.

Several sources provide lists of available lexicons and corpora for Sentiment Analysis. However, most of these references are currently outdated or incompatible with regards to the *open* nature of the platform. We will therefore review a significant number of datasets using metrics aimed to evaluate their compatibility with academic research and the *Natural Language ToolKit*. Our analysis will take into account the following parameters:

- maturity of the research that makes use of the data;
- licensing and Terms of Service;
- data structure;
- quality of the data.

All the above parameters will contribute to a final evaluation of the feasibility of incor-

porating the specific dataset within *NLTK*.

In order to provide context for each corpus in relation to the approaches discussed in chapter 2, we organize our survey into four main sections for *lexicons*, *document-level*, *sentence-level* and *aspect/feature-level* corpora (see table 3.1).

	Name	Instances	Domain	License	Format	NLTK
<i>Lexicons</i>	Opinion Lex.	6,789	Word Polarity	-	Text	✓
	MPQA Subj. Lex.	8,222	Word Polarity	GNU GPL	Text	📎
	Harvard GI	-	Various	-	Various	📎
	LIWC	4,500	Various	Proprietary	Various	✗
	SentiWordNet	117,659 synsets	Word Polarity	CC BY-SA 3.0	Text	✓
	Vader	7,517	Word Polarity	Free	Text	📎
<i>Document-level</i>	Polarity Dataset	2,000	Product Reviews	-	Text	✓
	Experience Project	-	User messages	📎	HTML	✗
	Product Debate	-	Debates	-	Text	📎
	Political Debate	-	Debates	-	Text	📎
	Sentiment140	1,600,000	Tweets	Twitter ToS	CSV	✗
	STS-Gold	2,034	Tweets	Twitter ToS	CSV	✓
	Sanders Analytics	5,513	Tweets	Twitter ToS	CSV	✓
	TASS 2013	68,000	Tweets	Twitter ToS	XML	✓
	SemEval-2014	17,290	Tweets + SMS	📎	📎	✗
<i>Sentence</i>	Pros & Cons	45,875	Product Reviews	-	Text	✓
	Sentence Polarity	10,662	Movie reviews	-	Text	✓
	Subjectivity Dataset	10,000	Movie reviews	-	Text	✓
<i>Aspect</i>	Customer Review	313	Product Reviews	-	Text	✓
	Additional Review	325	Product Reviews	-	Text	✓
	Comparative Sentence	7,998	Product Reviews	-	Text	✓

Table 3.1: Reviewed datasets. Licenses refer to the original terms of each corpus before being eventually included within *NLTK*.

The “*NLTK*” column states whether the dataset is already supported by the platform.

“-”: not defined; ✓: suitable; ✗: not suitable; 📎: see description.

3.1 Lexicons

We start our survey with a list of several lexicons that can be employed for Sentiment Analysis.

3.1.1 Opinion Lexicon

Authors: Minqing Hu and Bing Liu.

Year: 2004.

References: Liu (2010, 2004).

License: Not defined.

The *Opinion Lexicon* (also known as *Sentiment Lexicon*) is probably one of the most known lexicons for English language publicly available at the present day. Its popularity is motivated by the relevance of the publications produced by its authors, which focused on tasks such as extraction and summarization of customer reviews based on aspect/feature-level analysis. The success of this lexicon is also due to the passionate didactic and educational activity of Liu, author of one of the most complete Sentiment Analysis handbooks (Liu, 2010), whose personal website directly provides lexicons and datasets used in many of his papers (Liu, 2004).

The *Opinion Lexicon* is structured as a list of 6,789 words split in two files for *positive* and *negative* utterances. The peculiar aspect of this dataset is that it has not been automatically derived from a single computation, but it has been built and expanded during several years of research, along with the publication of new research papers. The lexicon not only includes words in their canonical form, but it also lists idiomatic and misspelled forms that usually appear in social-networking contexts. These peculiarities have made the *Opinion Lexicon* popular especially for micro-blogger Sentiment Analysis tasks.

Since the original dataset does not contain any specific license statement, we directly contacted the author and agreed on the terms reported below, which we will apply for each corpus that we will incorporate:

- Any data published by *NLTK* will use a *CC-BY 4.0* license (see details below). In order to respect this kind of license, any use of the corpus should credit the author(s) as its creator.
- The corpus will be integrated with a *README* file that will ask users who eventually use the corpus to cite the relevant publications, also providing a link to the author's website.

Figure 3.1: Licensing terms

Having an explicit license permission is highly desirable, especially when contributing to open source code, in order to avoid copyright restrictions that may cause problems at a later stage. *NLTK* benefits from the *Apache License* (Apache, 2004),

which does not require users to use the same license terms in case they want to modify or redistribute the licensed product; *Apache License* is compatible with Creative Commons licenses like *CC-BY 4.0*, which allow users to modify and share the material, under the restriction of giving credit to the authors and indicate if changes were made to the original content (CreativeCommons, 2013).

3.1.2 MPQA Subjectivity Lexicon

Authors: Theresa Wilson.

Year: 2005.

References: Riloff and Wiebe (2003); Wilson et al. (2005, 2009).

License: GNU General Public License.

Theresa Wilson's *Subjectivity Lexicon* can be probably considered as the main missed opportunity with regards to our project. Wilson's works are widely recognised to be some of the most relevant contributions to Sentiment Analysis, and her lexicon could represent a valuable instrument for research and teaching purposes (Wilson et al., 2005; Riloff and Wiebe, 2003).

The *Subjectivity Lexicon* distinguishes between *strong* and *weak* subjectivity clues, respectively indicating expressions and features that can be considered subjective in most context or only under certain circumstances. Moreover, the dataset contains indicators such as length, word token, POS tag and prior polarity, together with a flag that specifies if the clue is supposed to match all unstemmed variants of the word. It is worth noting that neutral words are also included in the lexicon, because they can be considered as valuable indicators of the presence of an opinion.

Wilson's dataset is publicly available on the MPQA website under a *GNU General Public License*. However, this kind of license does not explicitly clarify whether a Natural Language Processing system can be trained on a GPL-licensed corpus without being constrained to be GPL as well. We therefore tried to contact *MPQA* to obtain clarifications about this ambiguity, but we unfortunately did not receive any answer yet. For this reason we are not able to incorporate the *Subjectivity Lexicon* into *NLTK* at the present moment.

3.1.3 Harvard General Inquirer

Authors: Philip Stone.

Year: 2005.

References: Stone et al. (1966); Harvard and MIT (2000).

License: See description.

The *Harvard General Inquirer* is a system built for content analysis of texts. Its lexicon provides an extremely broad set of categories for each word, including orientation (*positive* and *negative* valence categories), specific categories (e.g. *pleasure*, *pain*, *arousal*), institutional field categories (e.g. *legal*, *military*, *religious*) and many others, for a total of 182 tag classes.

Unfortunately, the licensing terms of this lexicon do not seem to be compatible with *NLTK*. As stated on the project website, the categories that constitute the *General Inquirer* may be copyrighted at one point or another. We believe that, while it could be in theory possible to contact lexicon maintainers to discuss about an explicit agreement, time constraints and implementation complexity issues do not allow us to undertake such a task for our specific project.

3.1.4 Linguistic Inquiry and Word Counts (LIWC)

Authors: Roger J. Booth, Martha E. Francis, James W. Pennebaker.

Year: 2001.

References: Pennebaker Conglomerates (2007); Pennebaker et al. (2001, 2007).

License: Proprietary.

The *Linguistic Inquiry and Word Counts* is a software suite built for Natural Language Processing. Its lexicon provides regular expressions for more than 70 categories (or *language dimensions*, as reported on the website). Some of these dimensions (like *sadness*, *affect*, *negative emotion*) are related to sentiment and subjectivity analysis and can be therefore employed for Sentiment Analysis classification tasks.

However, *LIWC* lexicon database is proprietary and license is not free. We do not therefore consider the corpus as being suitable for *NLTK* integration.

3.1.5 SentiWordNet

Authors: Stefano Baccianella, Andrea Esuli and Fabrizio Sebastiani.

Year: 2006.

References: Esuli and Sebastiani (2006); Baccianella et al. (2010).

License: Creative Commons Attribution ShareAlike 3.0.

SentiWordNet is a corpus which assigns positivity, negativity, and objectivity scores to each *WordNet* synset (see chapter 2). This corpus, freely available for research purposes since its original release in 2006, is already integrated in *NLTK* under a *Creative Commons Attribution ShareAlike 3.0 Unported* license. The terms of the license are similar to those defined by *CC-BY 4.0*, with an additional restriction that requires to distribute derivative contributions under the same license as the original.

3.1.6 Vader Lexicon

Authors: C.J. Hutto and Eric Gilbert.

Year: 2014.

References: Hutto and Gilbert (2014); Hutto (2014b,a).

License: Free to download and use.

We briefly introduced *Vader* (*Valence Aware Dictionary and sEntiment Reasoner*) in chapter 2, with regards to its approach to *contextual valence shifters* and negation handling. This model is based on a set of five general rules that aim to capture common sentiment regularities in natural language, specifically for Sentiment Analysis purposes.

Vader is based on a gold-standard lexicon which also includes idiomatic expressions in micro-blogging domains, such as emoticons and popular acronyms. Each entry in the lexicon is annotated with its sentiment intensity score (*valence*), which can range from *Extremely Negative* (−4) to *Extremely Positive* (4).

This lexicon, which is mostly based on human-annotation efforts, is free to download and use. Its incorporation within *NLTK* is currently a work in progress, and it is expected to be achieved soon.¹

3.2 Document-level corpora

We now review several corpora that can be employed for *document-level* Sentiment Analysis tasks. We also include in this list tweet corpora, considering each tweet as a

¹We had the chance to give our contribution to improve *NLTK* support for *Vader*. Further development is expected to be realized shortly.

small document rather than a single sentence.

3.2.1 Polarity Dataset

Authors: Lillian Lee and Bo Pang.

Year: 2004.

References: Pang and Lee (2004).

License: Creative Commons Attribution 4.0 International (CC-BY 4.0).

The *Polarity Dataset* contains 2000 movie reviews from *imdb.com*, equally split in two sets of *positive* and *negative* reviews. Each sentence is stored on a different line, so that it is also possible to employ the same data for *sentence-level* analysis tasks.

The *Polarity Dataset* is already included in the *Natural Language ToolKit* with the name of `movie_reviews`.

3.2.2 Experience Project

Authors: Armen Berjikly.

Year: 2007.

References: Project (2007).

License: See description.

Experience Project is a social web project in which users can share their experiences and obtain community feedback and comments about them. Thanks to the website structure, built around a bulk of more than 24 categories, the data gathered from this source can represent a valuable source of opinionated data for Sentiment Analysis purposes.

Since the data is not directly distributed in a structured format, it would be necessary to implement specific web extraction functions in order to collect user-generated contents with their respective labels. Additionally, *Experience Project* contents are declared to be *property of Experience Project or its content suppliers* (Project, 2007), and their inclusion into *NLTK* does not therefore seem to be possible at the moment.

3.2.3 Product and Political Debate Corpora

Authors: Swapna Somasundaran and Janyce Wiebe.

Year: 2009/2010.

References: Somasundaran and Wiebe (2009, 2010).

License: Not defined.

The *Product Debate* and *Political Debate* corpora, distributed by *MPQA*, are two valuable sources for *document-level* Sentiment Analysis. The former dataset contains user-generated texts extracted from product discussion forums. Each file provides information about the debate stance that a single user is supporting, together with the textual content for the specific post.

The *Political Debate Corpus*, published one year after the *Product Debate Corpus*, contains ideological and political debate stances extracted from several political discussion forums. The dataset is structured as a set of folders, each of which identifies subsets of topic-related debates. Each user post is represented by three separate lines indicating the debate topic (see table 3.2), the textual content of the stance, and the position that the post is actually supporting.

Topic	Debates	Posts
Healthcare	16	336
Existence of God	7	486
Gun Rights	18	566
Gay Rights	15	1186
Abortion	13	618
Creationism	15	729

Table 3.2: Topics, debates and stances statistics in the *Political Debate Corpus*

We believe that these datasets would be extremely interesting for our purposes. However, their inclusion in the *NLTK* platform would be highly time-consuming; since other areas of Sentiment Analysis are still not covered by *NLTK* datasets, we prefer to give higher priority to other corpora.

Moreover, as we discussed with regards to the *MPQA Subjectivity Lexicon*, at the present moment we have not been able to contact *MPQA* to agree about suitable licensing terms for their corpora.

3.2.4 Sentiment140

Authors: Alec Go, Richa Bhayani, and Lei Huang.

Year: 2009.

References: Go et al. (2009); Alec Go and Huang (2009a,b).

License: See description.

As discussed in chapter 1, micro-blogging platforms are nowadays considered as one of the most relevant and interesting sources of data for Opinion Mining tasks. Following this assumption, we believe that it would be significantly valuable to add a corpus of tweets to *NLTK*, specifically designed for Sentiment Analysis purposes.

Sentiment140 is a tweet corpus composed by 1,600,000 Twitter messages, divided in two CSV files for *training* and *test* purposes. Each message is stored on a different line, with fields reporting values for *polarity label*, *user id*, *date*, *query*, *user name* and *text content*.

While the size of this dataset represents a positive aspect for its use with Machine Learning approaches, the high number of tweets that are contained in the corpus also explains our choice of not including it into *NLTK*. Due to Twitter Terms of Service restrictions, in fact, it is only possible to use automated methods to distribute *de-hydrated* tweets (i.e. only message IDs or user IDs). Non-automated methods (e.g. downloadable CSV files, like in *Sentiment140*) can be employed to only distribute up to 50,000 tweet messages (Twitter, 2015).

3.2.5 STS-Gold

Authors: Hassan Saif, Miriam Fernandez, Yulan He and Harith Alani.

Year: 2013.

References: Saif et al. (2013).

License: Not defined.

The *STS-Gold Sentiment Corpus* is a tweet corpus composed of 2034 hand-labeled tweets. Differently from the *Sentiment140* corpus, this dataset only provides fields for tweet IDs, labels (*positive/negative*) and texts.

Additionally, the corpus includes a set of 58 named entities extracted from the tweets and annotated with five different sentiment labels depending on the tweet in which they occur (*negative*, *positive*, *neutral*, *mixed* and *other*).

Finally, a third set contains aggregate information about each of the 58 named entities, that is the number of tweets in which the entity is annotated with each different label.

entity	negative	positive	neutral	mixed	other
vegas	0	3	10	1	0
twitter	2	1	14	2	0
cancer	13	0	0	0	0
...

Table 3.3: Named entity aggregated information extracted from the *STS-Gold* corpus

We believe that this dataset contains interesting information, especially for what concerns named entities and their aggregate information. However, we believe that the limited number of instances can represent a downside in this case, and we therefore prefer to give higher priority to other datasets.

3.2.6 Sanders Analytics Twitter Sentiment Corpus

Authors: Niek Sanders.

Year: 2011.

References: Sanders (2011).

License: “*The sentiment classifications themselves are provided free of charge and without restrictions. They may be used for commercial products. They may be redistributed. They may be modified*” (README file).

Sanders Analytics Twitter Sentiment Corpus provides 5513 hand-labeled tweets. In order to bypass Twitter Terms of Service restrictions, the dataset is published with a Python script that has to be used to download the original instances (Sanders, 2011). The same type of function is also supported within *NLTK* thanks to the `expand_tweetids()` method included in the Twitter client module.

The corpus splits tweets into four main topics (*Apple*, *Google*, *Microsoft*, *Twitter*), classifying each instance as *positive*, *negative*, *neutral* or *irrelevant* (e.g. foreign tweets).

We think that this corpus can be taken into consideration for a future expansion of the *NLTK* set of twitter-related corpora.

3.2.7 TASS 2013 General Corpus for Spanish

Authors: Taller de Análisis de Sentimientos en la SEPLN (Tass).

Year: 2013.

References: Villena Román et al. (2013); TASS (2013).

License: Not defined.

TASS 2013 General corpus contains the IDs of 68,000 Spanish tweets in XML format. These tweets, written by famous Spanish personalities, are labeled using semi-supervised techniques with five different polarity levels (*strong positive, positive, neutral, negative, strong negative* and *no sentiment*). Entities and topics are also provided.

The data distributed within this corpus might represent an interesting addition for Sentiment Analysis support in *NLTK*. However, we think that at the present stage support for English *document-level* analysis should be provided first.

3.2.8 SemEval-2014

Authors: SemEval 2014.

Year: 2014.

References: SemEval (2014).

License: “*You MUST NOT re-distribute the tweets, the annotations or the corpus obtained*” (README file).

SemEval is a competition aimed to promote and evaluate NLP systems, especially for semantic analysis tasks. The 2014 (Task 9) competition was composed of two subtasks for *Contextual Polarity* and *Message Polarity* disambiguation. In order to participate to the challenge, a dataset of tweets and SMS messages was distributed by *SemEval*. Specifically, as reported on the competition website (SemEval, 2014), the corpus was made of the following subsets:

Set	Instances	Type
training	9,728	Twitter messages
development	1,654	Twitter messages
development-test #1	3,814	Twitter messages
development-test #2	2,094	SMS messages

Table 3.4: SemEval 2014 dataset composition

As clearly stated in the documentation, the corpus is *not-redistributable*, and cannot be therefore incorporated within the *Natural Language ToolKit*.

Concluding our tweet corpora review, we believe that it would be ideally possible to include all of the above micro-blogging messages as lists of (*tweet id, label*), subsequently using *NLTK* Twitter client functionalities for *re-hydrating* them obtaining full messages.

3.3 Sentence-level corpora

We now provide a review of different corpora specifically built for *sentence-level* Sentiment Analysis.

3.3.1 Pros and Cons

Authors: Junsheng Cheng, Murthy Ganapathibhotla, Mingqing Hu and Bing Liu.

Year: 2005.

References: Liu et al. (2005); Ganapathibhotla and Liu (2008).

License: Not defined.

The *Pros and Cons* dataset is employed to identify opinionated words and to extract product features in *aspect/feature-based* analysis tasks (Liu et al., 2005; Ganapathibhotla and Liu, 2008). Since this field seems to be narrower compared with other more general Sentiment Analysis approaches, we have no information about the spread of this dataset before its incorporation in *NLTK*. We believe it would be interesting, for research purposes, to keep trace of all the downloads of this kind of corpora, especially when they are distributed only by a limited number of sources; it would then be possible to conduct a census of different research approaches based on the number of downloads and publications in which they are involved.

Using the *Pros and Cons* corpus produced accurate results for studies that involved studying opinions in comparative sentences. We think that, narrow though this field may be at the moment, providing *NLTK* with support for this kind of analysis is valuable for both academic research and teaching purposes, and will facilitate further studies about the topic.

3.3.2 Sentence Polarity Dataset

Authors: Lillian Lee and Bo Pang.

Year: 2005.

References: Pang and Lee (2005).

License: Not defined.

The *Natural Language ToolKit* platform already provides partial support for analysis on movie reviews data, thanks to the precious contribution given by Pang and Lee's *Polarity Dataset*. While this dataset is especially suitable for *document-level* Sentiment Analysis, the *Sentence Polarity Dataset 1.0* by the same authors provides 10,662 snippets² which can be employed for *sentence-level* analysis tasks (Pang and Lee, 2005). All instances have been extracted from the website *RottenTomatoes.com*, and they have been labeled as *positive* or *negative* depending on their source review label.

Adding this data to *NLTK* would therefore mean adding specific support for an entire area of Sentiment Analysis research that was previously not considered by the platform.

3.3.3 Subjectivity Dataset

Authors: Lillian Lee and Bo Pang.

Year: 2005.

References: Pang and Lee (2004).

License: Not defined.

The *Subjectivity Dataset 1.0* represents another relevant contribution for *sentence-level* analysis. While the *Sentence Polarity Dataset* can be employed for *sentiment classification*, this corpus provides support for *subjectivity classification* instead.

The dataset is made of 10,000 sentences extracted from *RottenTomatoes.com* and *imdb.com*, respectively labeled as *subjective* or *objective*. As the authors state, this labeling strategy might yield unaccurate labels for those entries that contain subjective sentences together with objective plot summaries (Pang and Lee, 2004); however, most labels are still accurate, and the data can be considered as a valuable source for opinion mining experiments.

²Some of the dataset rows are composed by more than one sentence.

3.4 Aspect/feature-level corpora

3.4.1 Customer Review Datasets

Authors: Mingqing Hu and Bing Liu.

Year: 2004.

References: Hu and Liu (2004a,b); Ding et al. (2008).

License: Not defined.

One of the most important additions we make to *NLTK* is represented by the *Customer Review Dataset* and the *Additional Review Dataset* (Hu and Liu, 2004a,b; Ding et al., 2008). These datasets contain customer product reviews extracted from *Amazon.com* and annotated by the authors. Each line of the corpora represents a tokenized sentence of a specific review.

- Customer Review Dataset (5 products)
 - Digital camera: Canon G3
 - Digital camera: Nikon coolpix 4300
 - Cellular phone: Nokia 6610
 - Mp3 player: Creative Labs Nomad Jukebox Zen Xtra 40GB
 - Dvd player: Apex AD2600 Progressive-scan DVD player
- Additional Review Dataset (9 products)
 - Digital camera: Canon PowerShot SD500
 - Digital camera: Canon S100
 - Diaper champ
 - Router: Hitachi
 - Router: Linksys
 - iPod
 - Mp3 player: MicroMP3
 - Cellular phone: Nokia 6600
 - Antivirus software: Norton

Figure 3.2: Annotated products from *Customer Review* and *Additional Review* datasets

Annotations can be used to extract the following information from the data:

- title of the review;
- product feature discussed in the sentence;

- polarity of the opinion about the feature (positive or negative);
- strength of the opinion, ranging from 1 (weakest) to 3 (strongest);
- notes about features and comparisons:
 - feature does not appear in the sentence;
 - feature does not appear in the sentence and pronoun resolution is needed;
 - suggestion;
 - comparison with a competing product from a different brand;
 - comparison with a competing product from the same brand.

This corpus can be considered as the main nucleus for *aspect/feature-level* Sentiment Analysis. We think that it is therefore necessary to take into consideration its incorporation in the *NLTK* framework.

We will discuss about implementation and qualitative details and show how we retrieved specific information from the datasets in chapter 5.

3.4.2 Comparative Sentence dataset

Authors: Nitin Jindal and Bing Liu.

Year: 2006.

References: Jindal and Liu (2006a,b); Ganapathibhotla and Liu (2008).

License: Not defined.

Identifying comparisons can be thought as being quite a different task from common polarity classification problems: comparisons are not necessarily subjective, and their linguistic structure is in general more complex than that of a simple opinion. In fact, comparisons express some kind of relation between two or more entities, often focusing on some specific aspect of the compared objects. However, identifying comparisons can be extremely helpful in environments that also employ Sentiment Analysis techniques, such as product reviews analysis and summarisation (Jindal and Liu, 2006a,b; Ganapathibhotla and Liu, 2008). For this reason we decide to incorporate the *Comparative Sentence Dataset* into *NLTK*.

Chapter 4

Preprocessing and feature extraction

As we described in chapter 1, Sentiment Analysis is closely bound to Information Retrieval and Natural Language Processing tasks. However, treating an opinion mining process using the same structures and techniques employed in other text classification tasks usually produces unexpected and disappointing results. Working with text, we need to use features that are only based on words. In order to achieve good results, however, we need to make assumptions about the words that will yield more benefits to our work; this is what is usually described as *feature selection* in most Natural Language and Machine Learning approaches, and the difference with other text classification tasks mostly resides in this process.

The words we care about in Sentiment Analysis are different from those that we would usually employ in other tasks as topic classification or named entity recognition; in our specific environment, for example, adverbs and adjectives are more interesting and useful than nouns and verbs, since they usually convey more information about sentiments and subjectivity.

Moreover, what we are describing as *word* should be better called *token*; what this term generalization is supposed to convey is the fact that, especially in Sentiment Analysis, the data components that can help our task are not only restricted to regular words. In fact, when interpreting the sentiment of a text, we are usually influenced by other symbols: punctuation marks, uppercase letters and also groups of characters that form so-called *emoticons*.

With this in mind, we can now analyze several feature extraction strategies that have been used and evaluated in previous research. The first step to perform in order to extract features from raw text consists in splitting strings into smaller constituents (our *tokens*). Tokenizers are often employed to break long documents into paragraphs,

paragraphs into sentences, sentences into individual words and symbols and sometimes, depending on the task, even words can be further split into characters. In our work, we will mostly assume that a *token* refers to an instance found at word-level, which also includes punctuation symbols, emoticons, and eventually n-grams.

Tokens obtained as result of a tokenization process can later be filtered so that the system will only keep the most meaningful ones with regards to the specific application, or they can be preprocessed in several ways depending on the domain we are working into. Different kinds of tokenization algorithms can then be employed to prepare the text for feature extraction.

Feat	Explanation	NLTK
Whitespace tokenization	Split text at spaces, tabs, or newlines	✓
Penn Treebank tokenization	More sophisticated tokenization strategy (e.g. for punctuation and abbreviations)	✓
Emoticons	Consider sequences of symbols denoting emoticons as individual tokens	✓
Hashtags & handles	Consider micro-blogging hashtags and mentions as individual tokens (e.g. “#niceday”, “@mark”)	✓
Length reduction	Replace repeated character sequences of length 3 or greater with sequences of length 3. E.g. “happyyyy” → “happy”.	✓
Preserve case	Do not lower the case of entirely capitalized words (e.g. “NEVER”)	✗
Negation handling	Append _NEG suffix to words appearing in the scope of a negation	✓
Speech handling	Append _QUOTE suffix to words appearing in the scope of a direct or indirect speech	✓
POS	Tag each word/token with its Part Of Speech tag (e.g. “like_VBP”)	✓
Twitter-POS	Tag each word/token with its POS tag, using models specifically trained for micro-blogging domains (e.g. “:-D_E”, for a smiling emoticon tagged with an E tag)	✗
Dependency	Extract dependency graphs from texts (see detail below)	✗
Stemming	Reduce words to their stem (e.g. “loves”) → “love”	✓
Frequency	Employ the number of token occurrences as feature values	✓
n-grams	Extract a sequence of n items from text	✓

Table 4.1: Common features and techniques employed in Sentiment Analysis, and their *NLTK* support. We implemented *Negation handling* support within *NLTK* during the project implementation. *Dependency parsing* within *NLTK* has only partial support at the present moment.

4.1 Whitespace tokenization

Whitespace tokenizers simply split text whenever a space, tab, or newline is encountered. E.g.:

Sentence: “Whose woods these are I think I know.”

Tokens: ['Whose', 'woods', 'these', 'are', 'I', 'think', 'I', 'know.']

While this might seem a reasonable way to proceed at first, it is possible to notice how this kind of approach will return an unsuitable output for Sentiment Analysis. For example, the last word “*know*” has been returned together with the final period of the sentence. Following Potts (2011b) we now present tweet examples, which can be useful to clarify several tokenization issues. Assume we have the following text:

“yeaaaaah yippppy!!! my acct verified rqst has succeed got a blue tick mark on my fb profile :) in 15 days”

A basic whitespace tokenizer would output the following tokens:

```
['yeaaaaah', 'yippppy!!!', 'my', 'acct', 'verified', 'rqst', 'has', 'succeed', 'got', 'a', 'blue', 'tick', 'mark', 'on', 'my', 'fb', 'profile', ':)', 'in', '15', 'days']
```

Our objective is to find regularities in the data that can help us classifying the text. Tokens like “*yippppy!!!*” do not represent a real help, because they would be considered different from other forms like “*yippy*”, “*yippy!*”, “*yippppppppppy*” etc, thus not contributing (for example) to the final frequency count of the basic form “*yippy*”.

4.2 Penn Treebank-style tokenization

In order to overcome these issues we can use more sophisticated tokenizers. An interesting example is represented by the Treebank tokenizer, employed to tokenize text as in the Penn Treebank (Marcus et al., 1993). The Treebank tokenizer splits contractions like *won't*, *he'll*, *she's* to tokens like ['wo', 'n't'], ['he', 'll'], ['she', 's']; punctuation symbols are otherwise mostly considered as individual tokens. Our outcome becomes:

```
['yeaaaaah', 'yippppy', '!', '!', '!', 'my', 'acct', 'verified', 'rqst',
  'has', 'succeed', 'got', 'a', 'blue', 'tick', 'mark', 'on', 'my',
  'fb', 'profile', ':', ')', 'in', '15', 'days']
```

As we will see, this produces better tokens with respect to our final analysis. However, even this more sensible approach does not take into consideration many linguistic and syntactic aspects that could lead to an improved performance. The token “yippppy” has not been modified, the three exclamation marks are considered separately, abbreviations like “acct” and “rqst” have not been expanded and, above all, the emoticon “:)” has been split in the two tokens “:” and “)”.

4.3 Punctuation

Punctuation can often provide meaningful information about text polarity. Consider these two sentences:

“I am having fun.”

“I am having fun!!!”

Even if the sentences are made by the same words, it is quite evident that the sequence of exclamation marks confers a more positive angle to the second text; on the contrary, the single period at the end of the first sentence might even confer a somewhat sarcastic tone to the message, flipping its polarity¹. In this particular case we can reasonably argue that having two distinguished tokens “fun!!!” and “fun.” could result in a more efficient classification. This is not always the case. Anyway, even if we consider this assumption as correct, we can still combine the advantages of individual tokens (“fun”) and richer tokens (“fun!!!”): this can be done extracting bigram features instead of simple unigram features, as we will discuss later.

While some approaches prefer to consider sequences of multiple punctuation marks as a single token (e.g. “!!!” or “!?!?”) possibly normalizing their length (Hutto and Gilbert, 2014), other studies tend to split them into separate tokens, believing that their influence on the final polarity is in fact cumulative (Potts, 2011b); the latter choice can be especially important when considering frequency counts instead of Boolean features to flag the presence or absence of tokens.

¹Whether sarcasm has to be considered as a negative feeling is actually something debatable; studies about its classification in micro-blogging platforms have been recently conducted (González-Ibáñez et al., 2011)

4.4 Length reduction

In 4.1 we found the word “*yeaaaaah*”. While this word (as it is spelled here) is not present in dictionaries, it is still clear that the message it wants to convey is meant to be stronger than the one that a simple “*yeah*” would carry. The same can be applied to expressions like “*booooooring*” or “*happyyyyyy*”, and we can affirm that word lengthening is in general employed to highlight strong feelings and opinions. In order to reduce the dimensionality of our feature vectors and make the feature space less skewed, we can preprocess these tokens reducing them to a more compact form. For example, we can just keep three repeated characters and discard the rest. This would produce a single token for all words that contain sequences of three or more repeated characters, therefore reducing the total number of extracted features.

4.5 Letter case

One of the basic rules in the so-called *netiquette*, the set of guidelines aimed to ease social coexistence over the Internet, strictly forbids users to write messages using only capital letters. Typing one or more words in uppercase is in general considered rude, and should therefore be avoided to respect those who will read our messages (Robb, 2014). Uppercase words are also used (as we have seen with word lengthening) to intensify the polarity strength of an utterance. As much as it might be irritating or rude (and actually for this very reason) an uppercase word can thus provide useful information for Sentiment Analysis.

4.6 Negation scope

Another aspect that differentiates Sentiment Analysis from traditional topic classification and Information Retrieval tasks is the role of negation (Councill et al., 2010; Potts, 2011a). Supposing we have a document like:

“Chairs, lamps and couches were very coloured, but I do not like that style anyway.”

using simple word tokens we can affirm that the topic is related to interior design. Nonetheless, a sentence like “*I do not like that style anyway*” might mistakenly be classified as *positive*, since it contains a strong positive verb (*like*). Using bigrams we can already partially compensate for this, directly obtaining a negative token (*not*

like). However, we can further improve our performance specifically handling negation, which is mostly done marking all words that are comprised between a negation and a strong punctuation mark.

It has to be considered that negations are not always introduced by a simple “not” token. In fact, particles, phrases and adverbs like “never”, “no one”, “yet to” and suffixes like “’nt” can easily introduce a negative sentence or word. Following Potts (2011b), we will consider a wide range of these negativity-bearer tokens and consequently handle their scope.

Our Sentiment Analysis module (described in chapter 5) provides a `mark_negation()` function that appends a `_NEG` suffix to all words that appear in the scope of a negation, following Das and Chen (2001, 2007). For example, the above sentence would become:

“Chairs, lamps and couches were very coloured, but I do not like_NEG that_NEG style_NEG anyway_NEG.”

In this way the number of unigram features extracted from our dataset necessarily grows, because we generate tokens for both `like` and `like_NEG`. Moreover, we just created some uninformative tokens like `that_NEG`, which do not seem to yield any useful information about sentiment polarity; hopefully, these redundant features will be discarded at a later stage, for example filtering out less informative features based on their frequency or using other specific metrics like *TF-IDF* (Manning et al., 2008).

Negation taxonomies often include more complex forms of negation, some of which can be classified as *denials*, *rejections*, *imperatives*, *questions*, *supports* and *repetitions* (Councill et al., 2010). In order to better capture these complex facets, other approaches contemplate the use of more advanced techniques like semantic parsing or POS-tagging to better identify the scopes of negation (Kennedy and Inkpen, 2006; Na et al., 2004). However, especially in domains such as micro-blogging networks where texts are short and direct, this effort can be unnecessary and computationally expensive.

As a final remark, there are some very specific cases in which even a double positive could yield a negative meaning (LinguisticsSE, 2011). In these cases, handling bi-grams could be a simple yet effective solution to handle ironic expressions like “*yeah, right*”.

4.7 Other relevant scopes

Even if negation probably represents the most relevant sentiment valence shifter, there exist other types of scopes that are likely to improve classification performance, if correctly handled.

For example, words that appear in *indirect speech* sentences might deserve a specific scope. In a sentence like:

“She said the movie was really awesome, but I hated it.”

it is possible to mark a distinct scope starting from verbs that introduce speech (e.g. “said”), trying to find the best strategy to take nested structures into consideration. This would ideally produce an output like the following:

“She said the_QUOTE movie_QUOTE was_QUOTE really_QUOTE awesome_QUOTE, but I hated it.”

Finding this kind of pattern can be considerably challenging, especially when dealing with user-generated content from social networks. This kind of data, in fact, can contain several grammatical and syntactic mistakes or idiomatic phrases that hinder the process of extracting a structure from the text. Things can be considerably easier for *direct speech*, which is mostly introduced by quotation marks and thus generally “easier” to detect².

4.8 Emoticons

As described in chapter 1, a conspicuous part of the interest in Sentiment Analysis arose with the evolution of social networks and micro-blogging platforms like Facebook and Twitter. This kind of application represents the natural environment for visual sentiment clues, which are often expressed as raw text in form of *emoticons* (fig. 4.1).

Most of the approaches in Sentiment Analysis use emoticons as features together with regular words. Sometimes these clusters of symbols are also used *before* the feature extraction stage, to create new corpora whose labels are established only based on the presence of happy or sad emoticons. An example of this procedure is supplied by the *Sentiment140* corpus, that we discussed in chapter 3.

As Read (2005) showed, most Sentiment Analysis approaches and results are topic-dependent, domain-dependent and even temporally-dependent. Emoticons can im-

²As this case shows, however, quotation mark do not always introduce direct speech scope.

```

HAPPY:
    :-), :) , ;), :o), :], :3, :c), :>, =], 8), =), :}, :^), :-D, :D, 8-D,
    8D, x-D, xD, X-D, XD, ==D, =D, ==3, =3, :-)), :-), :), :*, :^*, >:P,
    :-P, :P, X-P, x-p, xp, XP, :-p, :p, =p, :-b, :b, >:), >:), >:-), <3
SAD:
    :L, :-/, >:/, :S, >:[, :@, :-(, :[, :-||, =L, :<, :-[, :-<, =\, =/,
    >:(, :(, >.<, :-(, :(, :\, :-c, :c, :{, >:\, ;(

```

Figure 4.1: Common emoticons

prove performance of Machine Learning classifiers in such complex tasks, and this makes them a valuable resource for both polarity and subjectivity classification.

4.9 Hashtags and handles

Other domain-specific features that can be extracted from social network entries (like tweets) are represented by tags and user handles. In Twitter and Facebook platforms, topics are often introduced using *hashtags*, words (or sequences of words without whitespaces) preceded by the hash symbol “#”; users, instead, are often mentioned prepending a “@” symbol in front of their nickname. This syntactical information can be useful to extract a more informative structure from raw text, or even to exclude those parts of the message that we consider as non-informative.

“Aww the poor thing :(Hope it’s okay and in good health. luckily it has been freed from those Rocks #orcalove <https://t.co/tOSZMOafv6>”

“@_Kimimi A great enough reason to listen to one epic soundtrack. :D”

Two examples of tweets

The above example also highlights the importance of creating specific tokenization rules for URLs. Tokenizing the sequence <https://t.co/tOSZMOafv6> using a Treebank-style tokenizer would in fact yield:

```
[ 'https', ':', '//t.co/tOSZMOafv6' ].
```

Splitting the link into different parts would not be useful towards our sentiment classification task, and it is therefore something that we usually want to avoid.

4.10 Stopwords

In many applications word features are added based on their frequency counts. While this may produce good results in general, especially from a performance point of view, it has often been shown that removing some of the most frequent words can generally improve classification results with Machine Learning methods. These common words are usually called *stopwords*, and their removal is based on the assumption that tokens that appear too frequently in all documents are likely to be too generic and thus not very informative.

While sifting out stopwords can improve results in many text classification tasks, this is not always the case for Sentiment Analysis. Additionally, it is important to consider that this kind of dimensionality reduction has to be conducted only during the feature extraction process (e.g. when extracting unigram features) and not during text preprocessing. This is important because a-prioristically removing all stopwords would also mean removing tokens like “no”, “nor”, “not”, “against” and many others that, even if not considered as features, could yield important information about the structure of our data (for example helping identifying negation scopes).

4.11 POS tags

Suppose we have the following sentences:

“I like this movie.”

“It was like I always dreamt.”

It is easy to notice that the word “like” is employed in a different way in the two sentences. This difference is linguistically explained by the role that the word is assuming, which is usually defined as its *Part Of Speech*. POS tags are symbols used to denote these categories, and it is possible to use them, for example, to highlight the differences between the two uses of “like” in our example above.

Assuming the tags employed in the *Penn Treebank*, it would then be useful to differentiate between the two tokens “like_VBP” and “like_IN”³. However, in some domains, specific POS tagging procedures are recommended in order to achieve significant improvements. An example in this direction is provided by the *TOKENIZER* tagger

³VBP: Verb, non-3rd person singular present. IN: Preposition or subordinating conjunction (University of Pennsylvania, 2003).

and parser implemented by *ARK* (Gimpel et al., 2011); this experiment aims to overcome issues related to the fact that POS taggers are usually trained on corpora which are not linked to micro-blogging domains, and thus produce a worse performance on social network data.

Note that POS-tagging is also extremely helpful in *aspect/feature-based* Sentiment Analysis, since it simplifies the task of extracting item features⁴ from raw text (Hu and Liu, 2004a). Finally, POS-tagging has also been employed to extract subsets of sentiment-bearer words (e.g. adjectives, verbs and adverbs) and use them as word features (Na et al., 2004), yielding better accuracy performances on *document-level* analysis tasks.

4.12 Dependency structure

Using a dependency model, every word in a text is represented as a node of a tree structure, while grammatical relations between words are represented by the edges of the graph. This configuration allows us to highlight and investigate connections between lexical items, getting us closer to a complete semantic representation of the data. Since dependency structures directly connect words, their use produces good results especially with languages that have flexible word order, like German or Chinese (Klein and Manning, 2004).

In Wilson et al. (2005) dependency structures are used to identify subjectivity modifier features, distinguishing between *strong subjectivity* and *weak subjectivity* clues. If a word has a parent which is included among the lexicon subjectivity clues, and if the word and the parent share a relationship with an object, adjective, modifier or verbal modifier, then the feature is activated. In this case, activating the feature means that we include a “*modified by polarity*” entry in our feature vector, setting its value to be equal to the word parent polarity (which is extracted from the lexicon).

4.13 Stemming

In those cases where a more compact feature space would produce better results, words can be reduced to shorter tokens using a *stemmer*. Words like “*loves*”, “*loved*” and “*love*” could be for example reduced to the same string “*love*”; while this operation

⁴In this case we use the word *feature* to indicate specific aspects of a product or item, such as *viewfinder*, *zoom*, *screen*, etc. See chapter 1 for more information.

reduces the sparseness of the data, it decreases at the same time the specificity of each token. In the given example, we can see that we lost the suffixes that could have helped us to discriminate between the *lover* and the *loved*, which is somewhat ungraceful especially for Sentiment Analysis.

4.14 Word Frequency

Traditional text classification and retrieval techniques often employ word frequencies as feature values. This is often done when a *bag-of-words* model is employed, and it is assumed to represent a sensible approach in many traditional classification tasks. However, it has been shown that Sentiment Analysis classifications do not gain any particular benefit from word frequency features, especially in domains where short texts like sentences or tweets are analyzed (Pang et al., 2002).

4.15 Unigrams and bigrams

While discussing punctuation and negation, we mentioned the role of bigrams and unigrams. These two concepts can actually be thought as a more abstract layer of our feature extraction task; in fact, they can be applied on top of many of the features that we previously discussed. For example, given the sentence:

“I did not like the show.”

simply using unigram word features would not give enough consideration to the “*not*” token in front of “*like*”. Especially if we decide to remove features that consist of stop-words, we could obtain a wrong classification which would give great importance to “*like*” as a positive orientation word. Using bigrams, instead, we would also consider the feature for “*not like*”, thus differentiating it from the simple “*like*” token we had before.

In our experience, however, we noticed how unigram features yield better performances than bigrams (especially on micro-blogging datasets), confirming results from Pang et al. (2002) and Yessenov and Misailovic (2009). Other studies showed that sometimes bigrams can instead perform better than unigrams (Dave et al., 2003; Perkins, 2010a), so this is still a controversial topic in Sentiment Analysis⁵. Our hypothesis is that these categories produce very different results on different domains,

⁵Some more advanced techniques obtain relevant results employing bigrams made by a valence shifter followed by a word token (Kennedy and Inkpen, 2006).

and that adding bigram features to a small dataset can yield worse results because of the increased sparseness.

4.16 Final remarks

Before proceeding to the description of our implementation, a final note about feature extraction is required. As we said, it has to be considered that each distinction we make about word tokens can consistently increase data sparseness. Especially when working with small datasets, distinguishing between expressions like *“happy”*, *“happyyyy”*, *“HAPPY”* and *“happy_NEG”* can lead to inconsistencies, overfitting and less predictable classifications. This is one of the reasons why we do not always extract all of the possible features from text. Another reason is related to system performance: creating a feature vector of hundreds of thousands features for each document can consistently slow down our application and even compromise its usefulness, especially in live environments such as websites and web services.

Chapter 5

Design and implementation

Incorporating new features into an open source platform requires a different approach if compared to common coding activity. Open source contributions, in fact, are often supposed to be shared and discussed among a distributed and various community before they can be accepted and incorporated in the main project. This aspect subsequently involves the necessity of following shared guidelines and practices that are explicitly or implicitly established in the community, ranging from naming conventions and package structure to license agreements and documentation production.

As we briefly discussed in chapter 1, the *Natural Language ToolKit* provides developers with useful information, publishing guides, wikis and books about conventions and specific implementation practices. Moreover, doubts about implementations and contributions can also be discussed using the *NLTK-dev* forum hosted by *Google Groups* (NLTK, 2008).

We now discuss about our specific design choices and implementation details, in order to provide more information about our final contribution and its usage.

5.1 Corpus readers

Adding a new dataset into *NLTK* is not as simple as placing new files in the project folders. The toolkit is equipped with several classes that are meant to parse and output the information contained in different data structures, and to allow inter-operability with other processing steps supported by the platform. These classes mostly inherit their functionalities and attributes from a base class called `CorpusReader`, whose role is that of supplying more specific readers with basic methods than can be applied to common parsing procedures.

Suppose we want to use a lexicon contained in a textual file in which each word is stored on a separate line. Our corpus reader (called `WordListCorpusReader` in this specific case) will be responsible of opening the text file (handling encodings, exceptions and different Python file-management strategies), reading each line, and eventually returning the list of distinct words that are contained in the lexicon. These operations can be quite straightforward to implement when using a specific Python version, but the task can get considerably complex when we need to ensure a backward compatibility with five different Python major releases.

Moreover, the simple lexicon example we provided does not apply to complex datasets. In fact, many corpora employ complex structures that cannot simply be returned line-by-line. However, we deliberately choose not to alter the original structure of any of the corpora that we plan to incorporate within *NLTK*.¹ For this reason, we believe that a deeper analysis of our design and implementation decisions for the creation of new corpus readers is required.

5.1.1 ProductReviewsCorpusReader

```
*****
* Annotated by: Mingqing Hu and Bing Liu, 2004.
*      Department of Computer Science
*      University of Illinois at Chicago
*
* Product name: Nikon coolpix 4300
* Review Source: amazon.com
*
* See Readme.txt to find the meaning of each symbol.
*****

[t]the best 4mp compact digital available
camera[+2]##this camera is perfect for an enthusiastic amateur photogra-
pher .
picture[+3], macro[+3]##the pictures are razor-sharp , even in macro .
size[+2][u]##it is small enough to fit easily in a coat pocket or purse
```

Figure 5.1: *Customer Review Dataset* extract

The *Customer Review* and *Additional Review* datasets are structured in several files, each one containing reviews related to a single product. Reviews are further split

¹Minor changes are only applied in order to convert contents to UTF-8 encoding and end-of-line characters to Unix syntax.

into sentences, each of which occupies a single row of the file. Each sentence can additionally contain information about product features, opinion strength and polarity, and several types of other annotation that we list in table 5.1.

Tag	Meaning
[t]	The title of the review: Each [t] tag starts a review.
xxxx[+ -n]	xxxx is a product feature.
[+n]	Positive opinion, n is the opinion strength: 3 strongest, and 1 weakest.
[-n]	Negative opinion
##	Start of each sentence. Each line is a sentence.
[u]	Feature not appeared in the sentence.
[p]	Feature not appeared in the sentence. Pronoun resolution is needed.
[s]	Suggestion or recommendation.
[cc]	Comparison with a competing product from a different brand.
[cs]	Comparison with a competing product from the same brand.

Table 5.1: Meaning of tags (from *Customer Review Dataset* README file)

A specific corpus reader called `ProductReviewsCorpusReader`, which inherits from `CorpusReader`, is employed to read our corpus. Since we need to deal with a hierarchical structure, we design our implementation using an explicit object-oriented approach, constructing instances that belong to the new classes `Review` and `ReviewLine`.

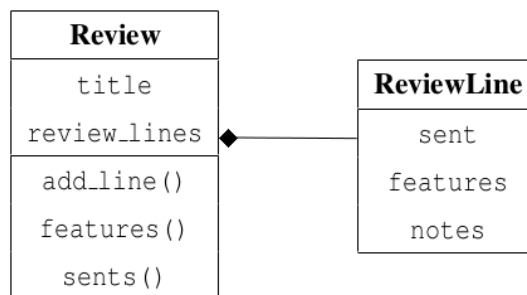


Figure 5.2: `Review` and `ReviewLine` classes. A `Review` can have many `ReviewLines`.

In order to overcome efficiency issues caused by reading large datasets, we wrap them around a `CorpusView` object (whose main class is `StreamBackedCorpusView`), which can read files as streams and output results as blocks of information; this strategy allows us to read large datasets block after block, saving memory resources and optimizing performance.

```
TITLE = re.compile(r'^\[t\](.*)$')
FEATURES = re.compile(r'((?:\w+\s)+)?\w+\[([?:\+|\-)\d]\]')
NOTES = re.compile(r'\[(?!t)(p|u|s|cc|cs)\]')
SENT = re.compile(r'##(.*)$')
```

Listing 5.1: Product Reviews regular expressions

As we said, Reviews are considered our main information blocks. The `reviews()` method outputs a list of reviews concatenating results obtained by the `CorpusView`. In turn, the `CorpusView` object obtains its results from a more specific method called `_read_review_block()`.

We decide to read review blocks using regular expressions. Since a single review always starts with a `[t]` tag, we create a new `Review` object each time we encounter one of these symbols, subsequently extracting the title and storing it in the `Review` object.

We then employ several regular expressions to analyse the next lines. If a new `[t]` symbol is met, our review is returned causing the `CorpusView` to start a new iteration with a new `Review` object. Otherwise, anything else is considered part of the current review: the raw sentence, along with its features and notes, is extracted and added to a new `ReviewLine` object (one for each line). When a new review is met, all previously stored review lines are added to the previous review and a new iteration starts. This strategy also allows us to skip readme lines like the ones reported in figure 5.1, since they will not be considered part of any review.

The `sents()` and `_read_sent_block()` methods are simply based on the outputs of `_read_review_block()`: since each returned review is built as a `Review` object, we can store and return every review sentence with the `sents()` method implemented in the `Review` class:

```
def _read_sent_block(self, stream):
    sents = []
    for review in self._read_review_block(stream):
        sents.extend([sent for sent in review.sents()])
    return sents
```

We also provide `features()` and `words()` methods that respectively output all features and words present in the dataset. While the former method parses each line and directly returns the outputs of the feature regular expression match, the latter checks if

the line contains a sentence and eventually returns it as a tokenized list of strings.

Details and issues

Some issues arose during the implementation and testing of our corpus reader, showing minor anomalies and inconsistencies in some of the files of the two datasets. Specifically:

ipod.txt does not contain any [t] placeholders. It is therefore impossible for the `ProductReviewsCorpusReader` to distinguish between different reviews belonging to this product.

Canon PowerShot SD500.txt only contains a single [t] placeholder, even if the text contained in the file does not entirely belong to just a single review.

Nokia 6610.txt contains a `***[t]` placeholder on line 118. This placeholder does not seem to have a particular meaning, and it is not attested in the *readme* file:

```
***[t]great phone , but no warranty !
```

In a private email conversation with Bing Liu, the author of the dataset reported that this kind of data is supposed to be mostly employed in *aspect/feature-based* Sentiment Analysis, for which *sentence-level* annotation is sufficient. However, we are willing to eventually fix this issue even for *document-level* classification in the future, collaborating with Liu and his former students to incorporate missing review boundaries.

5.1.2 ComparativeSentencesCorpusReader

```
*****
Amazon review    digital camera: Nikon coolpix 4300
*****
<cs-3>
  the best 4mp compact digital available
</cs-3>
3_4 mp compact digital (best)
```

Figure 5.3: *Comparative Sentence Dataset* extract

The *Comparative Sentence Dataset* contains reviews from several sources, whose sentences are annotated using specific XML tags (see table 5.2).

Tag	Meaning
<cs-1>...</cs-1>	Non-equal gradable
<cs-2>...</cs-2>	Equative
<cs-3>...</cs-3>	Superlative
<cs-4>...</cs-4>	Non-gradable

Table 5.2: Tags meaning (from *Comparative Sentence Dataset* readme file)

Non-equal gradable comparisons are expressions which describe aspects of an object as being, for example, *better* or *worse* than other objects features. *Equative* comparisons, conversely, consider two features as being on the same level. *Superlative* relations place one object in a better or worse position with regards to all other objects. Finally, *Non-gradable* expressions do not provide an explicit degree of comparison between items (e.g. similarity, enumeration of different features, etc.).

As we did for the *Customer Review* and *Additional Review* datasets, we decide to implement a new corpus reader called `ComparativeSentencesCorpusReader`, based on a specific class named `Comparison`. We do so because, as we discussed in chapter 3, comparative opinions often have a complex structure that we believe would be better represented by a specific class rather than a plain string or list.

Comparison
text
comp_type
entity_1
entity_2
feature
keyword

Following the same design choices that we made while implementing the `ProductReviewsCorpusReader`, we structure our reader on a set of regular expressions that will help us discriminating between normal sentences, comparisons and individual comparison components:

```
STARS = re.compile(r'^\*+$')
COMPARISON = re.compile(r'<cs-[1234]>')
CLOSE_COMPARISON = re.compile(r'</cs-[1234]>')
GRAD_COMPARISON = re.compile(r'<cs-[123]>')
```

```

NON_GRAD_COMPARISON = re.compile(r'<cs-4>')
ENTITIES_FEATS = re.compile(r"(\d)_((?:[\.\w\s/-](?!d-))+)")
KEYWORD = re.compile(r'\((?!.*\)(.*)\)$')

```

Listing 5.2: *Comparative Sentence* regular expressions

In this case the main block of our corpus reader is represented by the `_read_comparison_block()` method. However, this method is noticeably more complex than the `_read_review_block()` method implemented before. This is due to the more flexible structure of the dataset, which can present a variable number of lines containing features and keywords, depending on the number of comparisons that appear in each sentence. For example, parsing the sentence in figure 5.3 we would only extract features from the single line below the `</cs-3>` tag. But when we parse a block like the following, our approach needs to change:

```

<cs-1><cs-2><cs-2>
looks sort of like picasa software ( google it if you dont know ) in
the interface and is as easy to install and operate as g2 's , but
more intuitive .
</cs-1></cs-2></cs-2>
2_g2 3_intuitive (more)
2_picasa software 3_looks (like)
2_g2 3_install 3_operate (as easy)

```

Figure 5.4: Multiple comparisons (*Comparative Sentence Dataset*)

In this case we need to extract features from all the three lines below the closing tags. Each line related to gradable comparative sentences is expressed using the following syntax:

```
<type>_<entity_or_feature> * (keywords)
```

where *type* is a number from 1 to 4 (see table 5.2) followed by a variable number of entities or product features and by a conclusive keyword or phrase expressing the comparative relation.

Our method reads each line until one or more comparison tags are found, and splits them into two sets of *gradable* and *non-gradable* comparison tags. The sentence on the next line is then stored (and eventually tokenized) while the following line, containing closing tags, is skipped.

For each *gradable* comparison we then advance of one line, creating a new `Comparison` instance and populating it with features, entities and keywords. Since *non-gradable* comparisons do not require additional information, we simply store each of them in a new `Comparison` instance without parsing any additional line. All these comparisons are finally added to a list called `comparison_bundle`, which is in turn returned by the method. This means that, in fact, the blocks we are extracting are not always composed of individual comparisons, but they can be made of several comparisons included in the same sentence. We believe that this strategy does not reduce performance speed while it highly simplifies code.

From the example above, we would then obtain a list of three comparisons:

comparison_1		comparison_2		comparison_3	
text	[...]	text	[...]	text	[...]
comp_type	1	comp_type	2	comp_type	2
entity_1	-	entity_1	-	entity_1	-
entity_2	g2	entity_2	<i>picasa software</i>	entity_2	g2
feature	intuitive	feature	<i>looks</i>	feature	<i>install, operate</i>
keyword	more	keyword	<i>like</i>	keyword	<i>as easy</i>

All the comparisons above will have the same text value, which is a list of tokens obtained using `WhitespaceTokenizer` on the line “*looks sort of like picasa software [...]*”.

The `ComparativeSentencesCorpusReader` also provides several other methods to return keywords, sentences and words, no differently from what we described when we discussed the `ProductReviewsCorpusReader` class.

Details and issues

After committing our final changes we identified a minor issue occurring in the corpus. In fact, one of the dataset lines contains a badly formatted closing tag:

```
<cs-2>
nice camera but does n't take as sharp as pictures i thought they
would , and again .
/cs-2>
```

This typo should only affect the `sents()` method, which returns all lines that do not match any regular expression regarding comparisons, entities, features and *closing comparisons*.

Concerned by this anomaly, we then further analyze the corpus. Using the following regular expression we start looking for gradable comparisons that are not followed (as they should) by their feature/entity line, which usually starts with a digit:

```
</cs-[4]>\s*\n[^\s\d]
```

Many matches do not represent anomalies, and they simply identify comparisons that do not need entities or features:

```
<cs-3>
the best of everything
</cs-3>
(best)
```

Other matches, on the contrary, are likely to be considered as anomalies:

```
<cs-1>
Should I get the AMD Athlon ? Is the Intel Celeron a good choice ? Is the
    Pentium III the processor that I should get ? Well each of those are
    valid questions .
</cs-1>
But which Processor is best for you ? (good choice)
```

In this case, the sentence enclosed between comparison tags is not related to the line below, which is supposed to contain its entities, features and keywords.

In two other cases a very long text (that we shorten) is enclosed in comparison brackets, and it is not followed by any meaningful relation line:

```
<cs-3>
Prior to Mourinhos arrival or a bit before that , [...]
</cs-3>
()
-----
<cs-3>
    they need atleast 2 solid ball winners ( parker ? , [...]
</cs-3>

Dri Digi - Sydney , Australia
```

In the following cases, instead, it seems like annotations have been written on the wrong lines:

```

<cs-2>
Also , Coca-Cola , like Pepsi , signed counter trade agreements
with Poland .
</cs-2>
Both trade their concentrate for Polish beer . (like)
1_Both 3_trade their concentrate for Polish beer
<cs-2>
All of this has helped Coca-Cola to close in on Pepsi 's lead in
Poland .
</cs-2>
1_Coca-Cola 2_Pepsi (close in)
</cs-2>
Conclusion on Eastern Europe :

```

Here, “*Both trade their concentrate for polish beer .*” should have been enclosed between `<cs-2></cs-2>` tags; the conclusive “*(like)*” seems instead to be related to the previous sentence, for which we do not have any line expressing features and entities. The following sentence seems to be correctly annotated, but a second closing `</cs-2>` tag unexpectedly appears on the next line.

Other anomalies can be found on the following lines:

```

<cs-2>
All three units share the same speakers, even with the different
sound cards, I couldn't tell any difference when playing MP3s or
DVDs .
</cs-2>
Heat sensor used to take heat measurements (All)
-----
<cs-1><cs-2>
While the D70s won this battle, it's amazing just how well the
399 FinePix F10 did against it
</cs-1></cs-2>
The good news is that if you use noise reduction software like
Noise Ninja you can make these photos look even better. (against)
-----

```

We think that these inconsistencies, which are probably related to automated labeling procedures, do not significantly affect classification results, since their quantity is negligible if compared to the dataset dimensions.

5.1.3 CategorizedSentencesCorpusReader

Basic *NLTK* corpus readers already provide structures to parse lexicons and datasets in which each word is stored on a separate line. However, a simple `WordListCorpus-`

Reader would not provide the same functionalities and method names that we need if we aim to parse corpora made of sentences.

We therefore decide to implement a reader called `CategorizedSentencesCorpusReader` which, in contrast to the readers we previously presented, can be used as a general-purpose tool to parse corpora not necessarily related to Sentiment Analysis. In fact, we employ the `CategorizedSentencesCorpusReader` to parse both the *Sentence Polarity Dataset* and the *Subjectivity Dataset*, which are composed of categorized sentences, each one arranged on a different line of the file. In this case, we do not need any specific object to manage the blocks that will be read by the `StreamBackedCorpusView` stream.

The main peculiarity of this reader is that it not only inherits from `CorpusReader`, but also from the more specific `CategorizedCorpusReader`. This allows us to specify different categories depending on the names of the files that compose the corpus, using regular expressions (with the `cat_pattern` parameter) or more explicit mappings (using the `cat_map` parameter) when instantiating corpora. In particular, for our two datasets we have:

```

sentence_polarity = LazyCorpusLoader('sentence_polarity',
    CategorizedSentencesCorpusReader, r'rt-polarity\.(neg|pos)', cat_pattern
    =r'rt-polarity\.(neg|pos)', encoding='utf-8')
subjectivity = LazyCorpusLoader('subjectivity',
    CategorizedSentencesCorpusReader, r'(quote.tok.gt9|plot.tok.gt9)\.5000',
    cat_map={'quote.tok.gt9.5000':['subj'], 'plot.tok.gt9.5000':['obj']}},
    encoding='latin-1')

```

Listing 5.3: `CategorizedSentencesCorpusReader` instantiation

A `LazyCorpusLoader` (in listing 5.3) is a proxy object used by *NLTK* to temporarily stand for an actual corpus, until some of the reader functions are actually invoked. This expedient is used to reduce loading times due to the import of a corpus.

5.1.4 OpinionLexiconCorpusReader

The `OpinionLexiconCorpusReader` that we employ to read the *Opinion Lexicon* is a very basic subclass of `WordListCorpusReader`. The need for its implementation is due to the presence of a long preamble at the beginning of each file, which makes it impossible to directly use the parent class. Following Perkins (2014) we substitute

the usual `StreamBackedCorpusView` with a customized `IgnoreReadmeCorpusView` which inherits from it; when initialised, members of this class will set the initial position of the stream to the first blank line of the file, thus skipping the unwanted preamble. A sensible improvement for this approach will be discussed in chapter 6.

Together with the usual `words()` method we also provide the shortcuts `positive()` and `negative()`, that can be used to substitute `words('positive-words.txt')` and `words('negative-words.txt')`.

5.1.5 ProsConsCorpusReader

```
<Pros>Easy to use, economical!</Pros>
<Pros>Digital is where it's at...down with developing film!</Pros>
```

Figure 5.5: *Pros and Cons* extract

The `Pros` and `Cons` dataset is built using a peculiar syntax. Each sentence is stored on a separate line, but is also enclosed by specific tags and preceded by several white spaces. We therefore implement a new `ProsConsCorpusReader` class, which uses the following regular expression to extract plain sentences:

```
^(?!\\n)\\s*<(Pros|Cons)>(.*?)</(?:Pros|Cons)>
```

We can safely discard tags at this stage because, as previously described, the inheritance from `CategorizedCorpusReader` allows us to infer them from file names.

5.2 SentimentAnalyzer

Introducing new corpora represents the most important contribution we can provide for additional Sentiment Analysis support in *NLTK*. However, in order to provide support for teaching purposes and to test whether our corpora can be efficiently employed, we decide to also implement a specific module for Machine Learning approaches.

The module, included in the `sentiment` package, is based on a class that we call `SentimentAnalyzer`. Instances of this class can be considered as a sort of wrapper around existing *NLTK* functionalities, organised in a way that simplifies common feature extraction and classification tasks; we believe that our approach results in a clearer structure for teaching purposes, without losing in flexibility and modularity.

SentimentAnalyzer
<div>feat_extractors</div> <div>classifier</div>
<div>add_feat_extractor()</div> <div>all_words()</div> <div>apply_features()</div> <div>bigram_collocation_feats()</div> <div>classify()</div> <div>evaluate()</div> <div>extract_features()</div> <div>train()</div> <div>unigram_word_feats()</div>

Table 5.3: SentimentAnalyzer class representation

5.2.1 Feature extractors

The most important characteristic of `SentimentAnalyzer` is the feature extraction structure. *NLTK* already provides an `apply_features()` method which takes a function and a list of instances as parameters, subsequently applying that function to the list to create feature vectors representations of all the instances. The difference with a basic `map()` operation is that feature sets are now constructed lazily using `LazyMap` objects, which results in a significant reduction of memory usage.

NLTK represents feature vectors using dictionaries. Supposing we want to extract gender features from a set of labeled names (Bird et al., 2009), we could create a specific function (e.g. `gender_features`) that extracts the last letter from our instances:

```
>>> gender_features('Shrek')
{'last_letter': 'k'}
```

If we then decide to feed these feature vectors to a classifier, together with their relative label, we can use `apply_features()`:

```
>>> train_set = apply_features(gender_features, labeled_names)
>>> train_set
[({'last_letter': 'n'}, 'male'), ({'last_letter': 'e'}, 'male'), ...]
```

The train set is now a list of instances in form of (*dictionary*, *label*) tuples, where the dictionary represents the feature vector. One of the problems with this strategy is that adding new features (e.g. *word length*) to each instance is not straightforward. In fact,

this would require creating a new set of features and merge it with the previous one. We would subsequently have to iterate over each tuple, get its feature dictionary, and merge it with the dictionary of the corresponding tuple in the other set. Alternatively, we would have to change our `gender_features` function and make it more complex, so that it could also directly extract the new features from our instance². As we will show, `SentimentAnalyzer` highly simplifies this task, and it allows us to use multiple functions (that we will call *feature extractors*) to extract features in an intuitive way:

```
>>> analyzer = SentimentAnalyzer()
>>> analyzer.add_feat_extractor(gender_features)
>>> analyzer.add_feat_extractor(length_features)
>>> training_set = analyzer.apply_features(labeled_names)
>>> training_set
[({'length': 5, 'last_letter': 'r'}, 'male'), ({'length': 5, 'last_letter':
'n'}, 'male'), ...]
```

Listing 5.4: Multiple feature extraction using `SentimentAnalyzer`

We believe that this design choice (that we will later describe more in detail) highly simplifies the feature extraction process, making it more modular and scalable. Multiple feature extractors can in fact now be combined in order to experiment different configurations, thus facilitating comparisons and experiments during research and teaching sessions.

Another problem with the standard strategy is that `apply_features()` does not allow to specify additional parameters for feature extraction functions; this means that if we want to feed the feature extractor with, for example, a set of specific unigrams that should be extracted from an instance, we would need to store them in a non-local variable *before* applying the extraction function (see listing 5.5).

```
best_unigrams = get_best_unigrams(training_data)
def extract_unigrams(tokens):
    return dict([(unigram, True) for unigram in tokens if unigram in
best_unigrams])
train_set = apply_features(extract_unigrams, training_data)
```

Listing 5.5: Extracting selected unigram features using non-local variables

²This is, in fact, the strategy described in Bird et al. (2009)

We think that this can sometimes become a cumbersome operation³ and we therefore decide to handle the process in a different way. Using an `add_feat_extractor()` method (already shown in listing 5.4) we populate a dictionary of feature extractors, whose values are filled with their specific arguments; these functions will then be applied to the supplied instance using the `extract_features()` method. It is *this* method that will be finally passed to the original `apply_feature()` function (see listings 5.6 and 5.7).

```
def add_feat_extractor(self, function, **kwargs):
    self.feats_extractors[function].append(kwargs)

def extract_features(self, document):
    all_features = {}
    for extractor in self.feats_extractors:
        for param_set in self.feats_extractors[extractor]:
            feats = extractor(document, **param_set)
            all_features.update(feats)
    return all_features
```

Listing 5.6: `add_feature_extractor()` and `extract_features()` methods

```
analyzer = SentimentAnalyzer()
all_words = analyzer.all_words(training_docs)
unigram_feats = analyzer.unigram_word_feats(all_words, min_freq=4)
analyzer.add_feat_extractor(extract_unigram_feats, unigrams=unigram_feats)
```

Listing 5.7: Adding unigram features directly providing a set of unigrams. `all_words()` retrieves all tokens from a list of instances, `unigram_word_feats()` returns the most significant unigrams, and `extract_unigram_feats()` is the extractor that populates the feature dictionary.

We also thought about a different strategy, that consisted in applying decorators on extractor functions to directly add them to the `SentimentAnalyzer` extractors list. However, this would have reduced code flexibility, since users would have been required to manually add and remove decorator prefixes from functions⁴; the operation would have been especially difficult when using internal *NLTK* functions, since ap-

³We hypothesise that this may be one of the reason behind the following comment appearing just above `apply_features()`: “*or.. just have users use LazyMap directly?*”

⁴Another problem would have been represented by the hypothetical need of creating different `SentimentAnalyzer` instances with different feature extractors. Fortunately, we abandoned the idea of this approach before we had to deal with similar problems.

plying these changes requires re-installing the entire platform. Our current implementation, on the contrary, allows users to write their own feature extractors or to apply existing ones, and simply add them to the `SentimentAnalyzer` instance they are using.

Note that the original `apply_features()` function also has another downside. As we said, this helper will apply a function to each token of the instance that we feed to it. However, if this instance is just a single sentence, the function will consider each letter (if the sentence is a raw string) or each word (for tokenized sentences) as tokens, therefore mistakenly applying the extractor to each of them, and not to the entire instance. We can partially solve the problem wrapping the instance in a list (e.g. `[tokenized_sentence]`). But another issue is then represented by the strategy that `apply_features()` uses to check whether the instances that we pass to it are labeled or not:

```
if labeled is None:
    labeled = toks and isinstance(toks[0], (tuple, list))
```

However, in our case we have that

```
>>> isinstance([tokenized_sent][0], (tuple, list))
True
```

So the function will consider our sentence as if it had a label, which is wrong. The *list* here is just a list of word tokens, and not a list made by a string and its label. We can in any case overcome this issue explicitly specifying that we are not passing labeled instances using the `labeled=False` parameter:

```
instance_feats = apply_features([instance], labeled=False)
```

5.2.2 Other methods

Our `SentimentAnalyzer` class is supposed to comprise an entire classification procedure. For this reason, we also provide training and evaluation methods that produce and save output results. The `train()` method can store the trained classifier in an external file, so that it can be later used for new classification tasks and comparisons. The `evaluation()` method can be called using several parameters in order to specify which score we need to obtain from our classification; combining existing *NLTK* functions in a new structure, we implement support for *accuracy*, *precision*, *recall* and

F-Measure. Results are returned in a dictionary that can be subsequently fed to the `output_markdown()` function, situated in the `util.py` module that we are now going to describe.

5.3 Demos and helpers

As we stated in chapter 1, our project is especially aimed to provide Sentiment Analysis support for teaching purposes. We believe that it is therefore valuable to supply a substantial set of illustrative examples that can help users to apply Sentiment Analysis with different settings on various corpora.

5.3.1 Preprocessing and feature extraction

The `util.py` module contains feature extractors and preprocessing functions that implement some of the concepts we described in chapter 4.

Handling negation. The `mark_negation()` function can be applied to mark negation scopes adding a `_NEG` suffix to the instance tokens that belong to them. Since negation handling is often only employed with unigrams, we provide a `shallow` parameter that allows us to decide whether we want to modify original documents in place, or we need to keep their original version to later extract other features (e.g. bigrams).

The `double_neg_flip` parameter can be used to specify whether a double negation should be treated as an affirmation (deactivating again the negation scope) or not.

Unigrams and bigrams. Feature extractors for unigrams and bigrams are also provided. Since our extraction implementation allows us to specify additional argument values for them, we can for example specify a `handle_negation` parameter that will tell our `extract_unigram_feats()` function to mark negation scope in each document before extracting its unigram features. Moreover, we could also manually create a list of tokens that we want to consider as unigram or bigram word features, so that they will be the only ones that will populate our feature space⁵.

⁵Some papers and websites filter their features counting frequencies on both train and test data (Pang et al., 2002; Perkins, 2010a). In our examples we only consider training data, but our implementation potentially allows users to follow their preferred strategy.

Tweets preprocessing and parsing. As discussed in chapter 3, due to Twitter Terms of Service restrictions it is not possible to provide *NLTK* with a tweet corpus like *Sentiment140*, which exceeds the limits imposed by the micro-blogging platform (Twitter, 2015). We therefore decide to follow the same approach employed to build *Sentiment140* (that we describe below) to create a new, smaller dataset made of 10,000 tweets in *JSON* format, split into two equal-sized classes of *positive* and *negative* instances.

In order to construct the corpus we gathered data using Twitter APIs, and we subsequently preprocess tweets as follows:

- remove all retweets (to avoid redundant tweets);
- remove tweets with “:P” emoticons⁶;
- remove tweets which contain both sad and happy emoticons;
- strip off all emoticons from tweets;
- remove duplicate tweets.

We also manually remove some duplicate tweets by a particular user, which always contain the same text but end with a different number⁷. Considering the limited dimensions of the dataset, incorporating these tweets in the corpus would have caused problems with classifiers results.

In order to guarantee a higher flexibility, we implement a `json2csv_preprocess()` method which takes a *JSON* corpus as input and produces a *CSV* file of preprocessed tweets. This is also done to allow users to create their own corpus using *NLTK* `twitter` package, bypassing Twitter Terms of Service restrictions and applying the same preprocessing steps that we described above. Each step is considered optional and can be skipped using its related parameter, so that it is also possible to derive different corpora from the same original *JSON* file (e.g. with and without emoticons, as we did to compare *Vader* performances in fig. 5.14).

Finally, the `parse_tweets_set()` function is used to parse the *CSV* file returning a list of *(text, label)* tuples. At this stage of the process, our design choices contemplate the possibility of choosing specific word and sentence tokenizers, that can respectively be passed to the function using the `word_tokenizer` and `sent_tokenizer` arguments.

⁶Following *Sentiment140* documentation, these emoticons can sometimes be ambiguous.

⁷The digits at the end of the message represented coordinates that pointed to various geographic locations.

This is specifically done to allow an easier combination of different approaches, especially for teaching and research purposes.

5.3.2 Demos

Within the *NLTK* platform it is commonly recommended to create explanatory usage examples writing specific documentation and tests. In this case, however, we find it better to also explicitly implement demo functions that can be called and modified by users to reproduce and tweak our experiments. This choice entails a certain amount of redundancy in our code, since many operations are repeated in different demos; nonetheless, and *only* in this specific case, we consciously proceed in this direction, believing that demonstrations are meant to be as explicit as possible, while an excessive modularity can lead to obscurity and confusion.

Machine learning-based demos. Demo methods based on Machine Learning approaches share the same structure:

- retrieve the desired amount of instances from a specific corpus;
- split instances depending on their label (e.g. *pos/neg*);
- split train and test instances;
- specify feature extractors;
- apply all feature extractors to obtain final training and test feature sets;
- train and evaluate a specific classifier (e.g. *Naïve Bayes*, *Maximum Entropy*, *SVM*);
- output results (optionally creating a markdown file).

Every demo accepts the following parameters:

trainer	train method of a specific classifier
n_instances	Number of total instances to use for training and testing
output	Output file where results have to be written

Table 5.4: Common parameters for demo functions

The `demo_subjectivity()` function also accepts a Boolean `save_analyzer` argument: setting it to `True` users will obtain a *pickle*⁸ file containing the `SentimentAnalyzer` object together with its trained classifier and feature extractors. This file can

⁸*Pickle* is a data format and module for Python object serialization.

subsequently be used by `demo_sent_subjectivity()` to perform a single-instance classification task.

Lexicon-based. In order to compare *Vader* approach with other types of classification, we implement a method called `demo_vader_tweets()`. Since this is not a Machine Learning approach, we can directly classify all instances without having to split them into train and test sets, using the pre-existing `SentimentIntensityAnalyzer` class to predict their labels. Results are then evaluated and returned in the same format we used for statistical approaches.

Vader is also employed in `demo_vader_instance()` to classify single instances⁹. It has to be noted that *Vader* implementation only accepts raw non-tokenized strings, since a basic tokenization is internally executed by the `SentiText` class in `vader.py`.

Finally, in `demo_liu_hu_lexicon()` we implement an extremely naïve baseline that classifies instances simply counting the number of positive (+1) and negative (−1) word occurrences, with the final score sign identifying text polarity. This function accepts a Boolean `plot` parameter that can be used to generate a visual representation of the results.

5.3.3 Other functionalities

Other minor features are available in the `util.py` module.

Logging metadata. Together with a basic function to store classifiers and objects into output files, we also implement an helper function to save results using markdown syntax. The `output_markdown()` function accepts a variable number of arguments and arranges them into properly formatted bullet-point lists.

Profiling. A light decorator is provided to inspect and output timing performances. Results will be formatted depending on the execution time: functions executed in less than 10 seconds will output results with a scale of three decimal places, while long-execution functions will be presented in a more readable format:

```
[TIMER] evaluate(): 0h 2m 50s
```

⁹Specifically, the final polarity label is expressed by the `compound` variable returned by the `SentimentIntensityAnalyzer`. Even if *Vader* documentation does not explicitly mention it, we have been able to contact its author which confirmed our hypothesis.

```

12/08/2015, 22:35

• Classifier: NaiveBayesClassifier
• Dataset: movie_reviews
• Feats:
  ◦ extract_unigram_feats
  ◦ extract_bigram_feats
• Instances: 200
• Results:
  ◦ Accuracy: 0.625
  ◦ Precision [neg]: 0.6190476190476191
  ◦ Precision [pos]: 0.631578947368421
• Tokenizer: WordPunctTokenizer

```

Figure 5.6: Markdown output example

Visualization. A private `_show_plot()` function can be used to generate graphs for `demo_liu_hu_lexicon()` results. Since graph drawing functionalities require the `matplotlib` module, which is not a required *NLTK* dependency, we wrap the relative import statement in a `try/catch` block; if the module is not installed, an *ImportError* exception will be shown notifying users about the additional requirement.

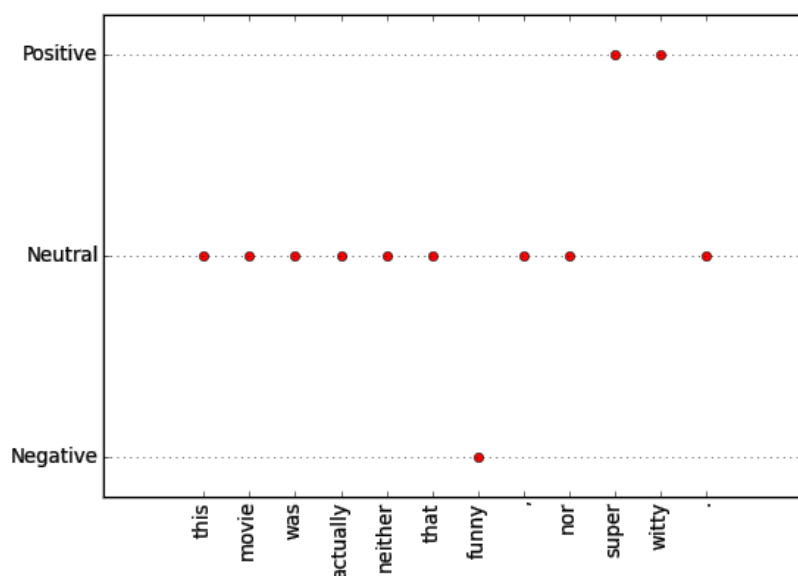


Figure 5.7: Visualization example for “*This movie was actually neither that funny, nor super witty.*”. Thanks to this visualization we also discover that the word *funny* is included by the *Opinion Lexicon* in the list of negative words.

Other general-purpose functions are implemented to split instances into train and test sets, respectively corresponding to $\frac{8}{10}$ and $\frac{2}{10}$ of the total number of instances. A *seed* for the random function is also provided, so that results obtained experimenting on the same dataset can be compared.

5.4 Documentation and Testing

Even before open source became a well-known reality, several studies showed the benefits generated by good commenting practices with regards to code readability and maintainability (de Souza et al., 2005; Woodfield et al., 1981). Web expansion, together with open source philosophy and collaborative work practices, further amplified the need for meticulous and up-to-date comments to explain code fragments.

Moreover, using Python, documentation lines can also be enriched in order to provide support for automatic testing procedures. These functionalities have been refined during the years, so that it can now automatically be checked if a specific modification made to the code is eventually causing other parts of the code to fail.

5.4.1 Docstrings and Sphinx

NLTK promotes and supports commenting practices especially through *docstrings*. *Docstrings* are string literals that can be used to comment portions of code, producing documentation that, differently from usual comments, can also be used and inspected at runtime. We decide to follow *NLTK* guidelines to accurately describe each class, module and method we implement. Additionally, we document method parameters using *Sphinx* markup syntax, as in the example below:

```
def unigram_word_feats(self, words, top_n=None, min_freq=0):  
    """  
    Return most common top_n word features.  
    :param words: a list of words/tokens.  
    :param top_n: number of best words/tokens to use, sorted by frequency.  
    :rtype: list(str)  
    :return: A list of `top_n` words/tokens (with no duplicates) sorted by  
             frequency.  
    """
```

Specific keywords (:param, :rtype, :return) are employed to describe parameters, types and outputs of each method, providing useful support for users and developers that need to understand and modify existing features.

5.4.2 Doctests and Tox

```
>>> product_reviews_1.features('Canon.G3.txt')
[('canon powershot g3', '+3'), ('use', '+2'), ...]
```

Listing 5.8: An example of doctest lines

Doctests are an intuitive yet powerful tool to perform tests on existing code, or even *test driven development* on non-yet-existing code. The `doctest` module is used to identify and execute Python interactive commands in *docstrings* (like the first line of the snippet above) subsequently checking if their output corresponds to the one provided by the *doctests* (on the second line).

Doctests benefits can be noticeably increased using *Tox*. This tool can be used to automatically manage several virtual environments and perform test procedures code on all of them. Consequently we do not need to include and execute any specific `doctest` call inside modules, but we can simply run `tox` from command line and let the program execute all the tests we wrote.

NLTK also employs *Jenkins*, a *Continuous Integration* server, that automatically verifies if a recently integrated commit is causing test failures in the toolkit. Using this instrument we have been able to quickly identify and fix some of the problems caused by the integration of new Twitter functionalities, creating a new Pull Request that has been promptly merged on the main framework branch.

Backward compatibility

The *Natural Language ToolKit* currently supports several Python versions, from Python 2.6 to Python 3.4. Transitions from one version to another revise and adapt several aspects of the programming language, often to ensure better performances and improve code readability. However, maintaining a full backward compatibility can sometimes be a challenging task.

In order to make sure that all our implementations are compatible with *NLTK* requirements, we use the *virtualenv* tool to create multiple environments in which we frequently test our code, making sure that all new additions do not cause any exception or unexpected behaviour. This strategy is useful to debug common issues like the ones caused by the following lines of code:

```
with io.open(filename, 'rb') as csv_file:
with io.open(filename, 'rt') as csv_file:
```

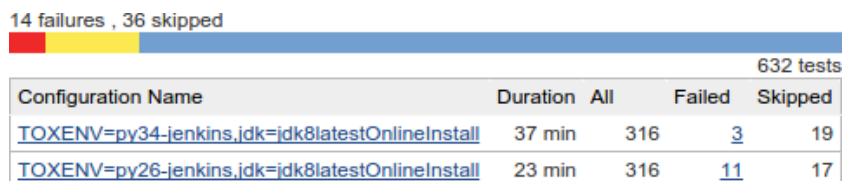
The first line works with Python 2.7, but raises an exception under Python 3.4:

iterator should return strings, not bytes (did you open the file in text mode?)

Conversely, the second line fixes Python 3.4 problems, breaking Python 2.7 functionalities:

```
'ascii' codec can't encode character u'\xa3' in position 93: ordinal not in range(128)
```

To solve this problem we therefore choose a different strategy depending on the Python version.



14 failures , 36 skipped

632 tests

Configuration Name	Duration	All	Failed	Skipped
TOXENV=py34-jenkins.jdk=jdk8latestOnlineInstall	37 min	316	3	19
TOXENV=py26-jenkins.jdk=jdk8latestOnlineInstall	23 min	316	11	17

Figure 5.8: Continuous Integration server reports

Even with regards to backward compatibility tests, *Tox* and *Continuous Integration* prove to be extremely valuable instruments. For example, thanks to their use, an issue regarding Python 3.2 (a not so common version of the language) has been found in the `TweetTokenizer` class module:

```
return u'^' if remove_illegal else match.group(0)
SyntaxError: invalid syntax
```

The error is caused by the fact that unicode literals support has been removed in Python 3.2 and reintroduced since Python 3.3. Collaborating with `TweetTokenizer` contributors we could easily fix the issue.

5.5 PEP-8 and Pylint

Contributions to open source projects are not only required to preserve existing functionalities, but also to be consistent with the prevailing project coding style. *NLTK* supervisors vet all contributors Pull Requests to make sure that they comply with commonly accepted Python guidelines. These guidelines are officially provided in a *Python Enhancement Proposal* document called *PEP-8*, written, among the others, by Python author Guido Van Rossum himself¹⁰.

In order to check that our contributions follow *PEP-8* guidelines, we use *Pylint*. This tool is a source code analyzer for Python, which can be invoked from command

¹⁰Van Rossum's active and tireless involvement in Python development made him gain the title of *Benevolent Dictator For Life* among the coding community.

line to sift files and output a list of warnings when some of the guidelines are not respected.

Using *Pylint* we correct several issues concerning unused imports, missing *doc-strings* and wrong indentation. However, other warnings produced by the tool are not supposed to be taken into consideration. For example, the following code

```
for words, sentiment in documents:
    all_words.extend(words)
```

will produce the following warning:

```
W: 36,19: Unused variable 'sentiment' (unused-variable)
```

These kinds of warnings are therefore often suppressed in *NLTK* using comment lines such as

```
# pylint: disable=W0612
```

where `W0612` represents the code for the specific warning we aim to disable.

Concluding, we believe that *Pylint* can provide useful indications, even if its results are generally meant to be further inspected and not uncritically accepted.

5.6 Comparisons

We conclude our discussion about design and implementation providing examples of comparisons that we have been able to conduct using our framework.

Classifiers. Simply feeding our demo functions with different classifiers, we can obtain results from *Naïve Bayes*, *Maximum Entropy* and *SVM* performances on several datasets.¹¹

```
naive_bayes = NaiveBayesClassifier.train
demo_tweets(naive_bayes, n_instances=1000, output='results.md')
```

Listing 5.9: Example of basic classification using Naïve Bayes with our demo function

We report visual comparisons between these three algorithms on our tweet corpus and on the `movie_reviews` dataset. We also report their scores for accuracy, precision, recall and F-Measure on *positive* and *negative* class labels.

¹¹*Maximum Entropy* classifiers have been trained using 10 iterations and GIS algorithm (Darroch and Ratcliff, 1972). The number of instances includes both training and test data, which respectively correspond to $\frac{8}{10}$ and $\frac{2}{10}$ of the total.

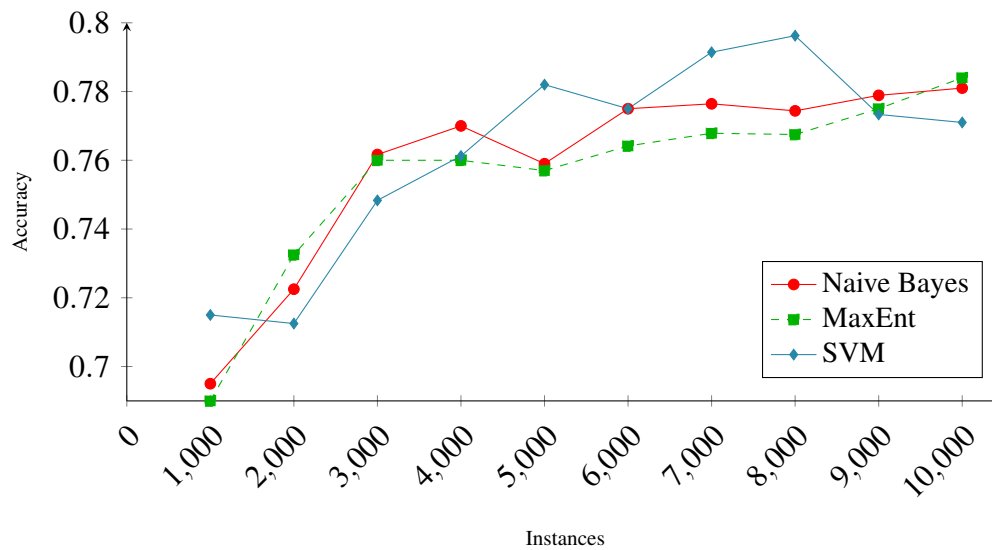


Figure 5.9: Naïve Bayes, MaxEnt and SVM accuracy on our tweet dataset using 1,000 most frequent unigrams.

Classifier	Accuracy	Precision		Recall		F-Measure	
		pos	neg	pos	neg	pos	neg
Naïve Bayes	0.781	0.8061	0.7597	0.822	0.74	0.7716	0.7896
Maximum Entropy	0.784	0.8028	0.7674	0.753	0.815	0.7771	0.7905
SVM	0.771	0.7705	0.7715	0.772	0.77	0.7712	0.7708

Table 5.5: Accuracy, precision and F-Measure scores for classifiers on 10,000 instances (tweet dataset).

Best scores on the full tweet dataset are obtained by *Maximum Entropy* and *Naïve Bayes* classifiers. However, as shown in fig. 5.9, *SVM* can yield highly competitive results on a lower number of instances.

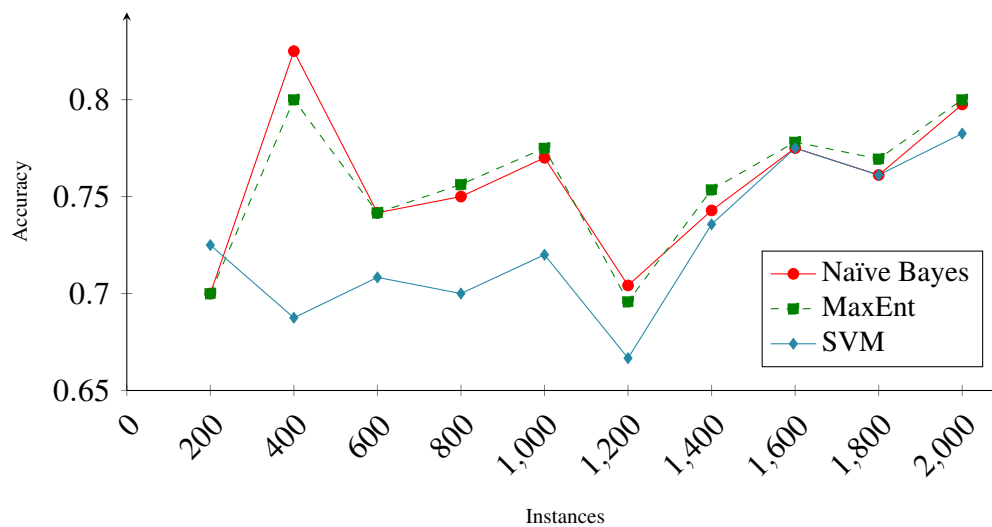


Figure 5.10: Naïve Bayes, MaxEnt and SVM accuracy on movie_reviews dataset using 1,000 most frequent unigrams.

Classifier	Accuracy	Precision		Recall		F-Measure	
		pos	neg	pos	neg	pos	neg
Naïve Bayes	0.7975	0.8287	0.7717	0.75	0.845	0.7874	0.8067
Maximum Entropy	0.8	0.8297	0.7752	0.755	0.845	0.7906	0.8086
SVM	0.7825	0.7678	0.7989	0.81	0.755	0.7883	0.7763

Table 5.6: Accuracy, precision and F-Measure scores for classifiers on 2,000 instances (movie_reviews dataset).

Results on the `movie_reviews` dataset (fig. 5.10) confirm that *Maximum Entropy* achieves the best performance. Nonetheless, *Naïve Bayes* is still highly competitive, and it can be often preferred especially when faster execution times are needed.

Tokenizers. We now compare the performances of different tokenization strategies on our tweet corpus. Results are obtained by simply modifying the tokenizer instance that we pass to the `parse_tweets_set()` function, as in the following example:

```
tokenizer = TweetTokenizer(preserve_case=False)
neg_docs = parse_tweets_set(negative_csv, label='neg', word_tokenizer=
    tokenizer)
```

Modifying the `TweetTokenizer` parameters on the first line we obtain different versions of the tweet dataset, from which we can subsequently extract our features (1,000 most frequent unigrams in this case). Using this approach we obtain the following results:

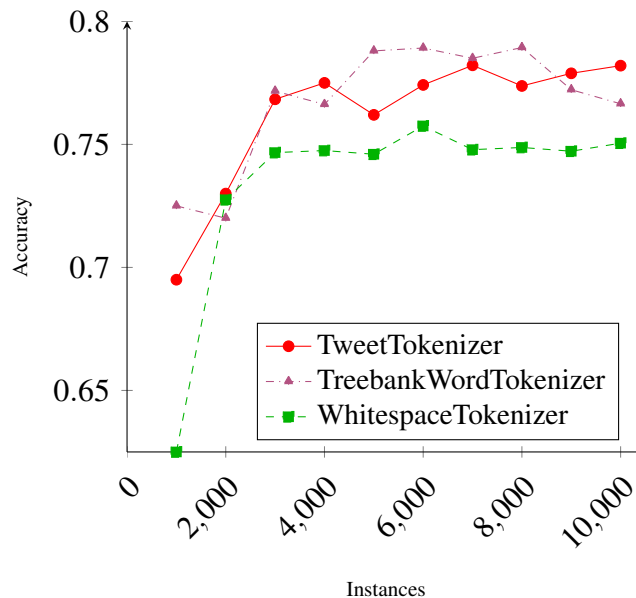


Figure 5.11: Naïve Bayes classification on our tweet dataset using 1,000 most frequent unigrams with different tokenizers. The `TweetTokenizer` assumes the same configuration of **Setting 1** in fig. 5.13.

As shown in 5.11, the Treebank-style tokenizer (see section 4.2) seems to perform better on a lower number of instances, but its performance drops while the number of instances increases. On the contrary, using a specialized `TweetTokenizer`, accuracy scores increase together with the number of instances.

Negation handling. As discussed in chapter 4, detecting negation scope and extracting relative features is a common technique in Sentiment Analysis. Our implementation allows us to handle negation using the following instructions:

```
all_words = sentim_analyzer.all_words([mark_negation(tweet) for tweet in
    training_tweets])

unigram_feats = sentim_analyzer.unigram_word_feats(all_words, top_n=1000)

analyzer.add_feat_extractor(extract_unigram_feats, unigrams=unigram_feats,
    handle_negation=True)
```

Using the `mark_negation()` method we handle negation scope in our training set, and we use this modified set to populate a list of unigrams which includes tokens with `_NEG` suffixes. We filter the most frequent 1,000 unigrams, and finally we add our customized feature extractor specifying that it should explicitly handle negation.

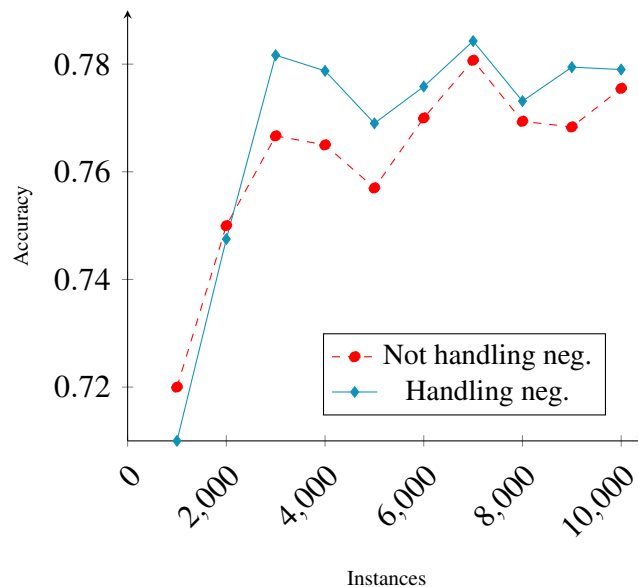


Figure 5.12: Naïve Bayes classification on tweet dataset, using 1,000 most frequent unigram features with and without handling negation (`TweetTokenizer(preserve_case=False)`).

Results reported in fig. 5.12 show significant improvements when negation scope is handled, confirming previous studies (Councill et al., 2010; Potts, 2011a,b; Das and Chen, 2001, 2007).

Micro-blogging tokenization. In our specific implementation, the `TweetTokenizer` class (which was already under development within *NLTK*) can identify and successfully tokenize most common emoticons, URLs and micro-blogging components (see section 4.9). We report results for different configurations of the *Natural Language ToolKit* `TwitterTokenizer`:

Setting name	TwitterTokenizer parameters
Setting 1	<code>preserve_case=False</code>
Setting 2	<code>preserve_case=True, reduce_len=True, strip_handles=True</code>
Setting 3	<code>preserve_case=True</code>
Setting 4	<code>preserve_case=False, reduce_len=True, strip_handles=True</code>

Table 5.7: `TweetTokenizer` configurations employed in fig. 5.13

Setting 1 results are obtained simply lower-casing all tokens. **Setting 2** results are obtained preserving token capitalization, normalizing word lengthening and removing twitter handles. **Setting 3** simply preserves capitalization. **Setting 4** results are obtained lower-casing all tokens, normalizing word lengthening and removing Twitter handles.

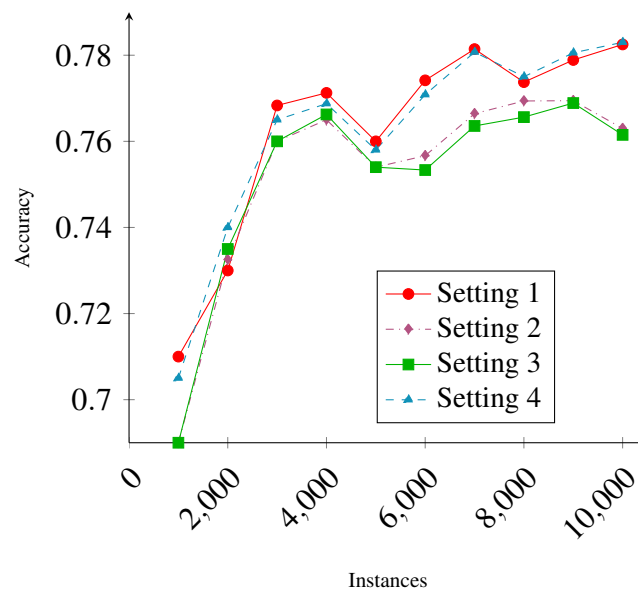


Figure 5.13: Naïve Bayes classification on our tweet dataset using top 1,000 unigram word features. Different settings for micro-blogging tokenization are employed, as described in table 5.7.

Vader and emoticons. *Vader* approach (see chapter 2 and section 3.1.6) tokenizes text using specific instructions, and does not therefore use any specific *NLTK* tokenizer

class. As it is possible to see in 5.14, results substantially improve when emoticons are present in the text.

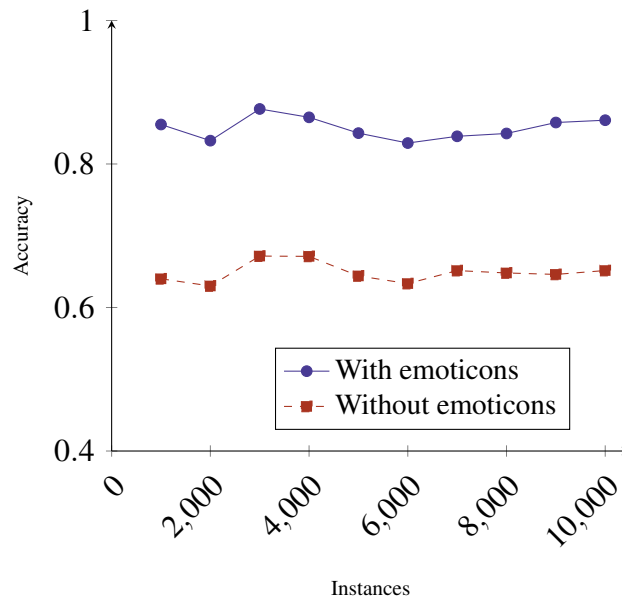


Figure 5.14: *Vader* accuracy on tweets dataset with and without emoticons

Note that this experiment cannot be directly compared with Machine Learning results on the same corpus: given our collection approach (described in chapters 3 and 5), every tweet contains at least one emoticon that directly determines its classification; for this reason, emoticons would weight too much on the final prediction, and accuracy scores would be extremely high. Moreover, while *Vader* results are based on the entire dataset, Machine Learning accuracy scores are only evaluated on a smaller test set.

Chapter 6

Evaluation and discussion

Adding Sentiment Analysis support into the *Natural Language Toolkit* platform is a task whose evaluation needs to take into consideration very specific aspects. Our task, in fact, does not directly implement a new approach to Sentiment Analysis, and cannot be therefore compared with previous methods using specific classification measures and scores.

As we described in section 5.4, our work is compatible with different forms of automatic evaluation. Using *Tox*, *Continuous Integration* and *Pylint* we have been in fact able to discover potential errors and correct them before the final Pull Request. However, as we discussed in chapters 1 and 5, our contributions can and should also be evaluated using parameters that are relative to open source practices and, more specifically, to the framework guidelines and requirements. We therefore decide to submit our final contribution to the online *NLTK GitHub* repository, asking for direct feedback by its authors and contributors.

6.1 Community feedback

The feedback received by our Pull Request mainly focuses on aspects that involve variable naming issues and package structure choices. We believe that all of these issues can be addressed with minor code modifications, and do not show any substantial flaw in our implementation.

Due to the community-based nature of *NLTK*, feedback can be considered as an extended stage of our project. We therefore aim to be actively involved in further discussion about our contributions, providing opinions and fixes in collaboration with other community members whenever it will be required.

6.2 Experience with *NLTK*

Collaborative environment. Working on *NLTK* entailed a continuous involvement into discussions regarding multiple aspects of the platform. During the implementation stages of our project we could exchange opinions even regarding aspects that were not directly related to our work, learning new procedures and improving our coding skills. We found a core group of passionate developers constantly providing support and feedback for new contributions, and we believe that this represents a substantial benefit for the whole community of users and developers.

During the implementation of our modules we sometimes found errors in existing code; in these cases we could immediately obtain collaboration from the community opening new Issues on GitHub. In our first Issue we asked for opinions about unusual results that we obtained when calling the `from_documents()` method of the `AbstractCollocationFinder` class. Feeding the method with a set of documents, in fact, we obtained n-grams that never appeared in any document. Further investigations led to discover that this behaviour was due to the `itertools.chain(*documents)` instruction, which collapses all documents returning a single instance from which we then erroneously extract bigrams. *NLTK* developers confirmed that this was to be considered as an issue, and started working on a possible solution.

Other Issues were opened regarding unusual *doctest* results. The discussion revealed that many modules, following an outdated recommendation from the *Developers Guide*¹, still provided *doctest* imports and calls in the last lines of the files. This practice was meant to be mostly replaced by *Tox* functionalities, that could also handle compatibility issues between different Python versions. The discussion contributed to updating the guide and to remove *doctest* calls from most of the modules.

Documentation. We think that *NLTK* provides useful documentation and explicative examples, which represent a useful source for those who wish to deepen their knowledge about the framework and Natural Language Processing in general. The amount of information that can be easily found on websites and guides can sometimes be confusing and disorienting, and it is therefore always advisable to become confident with the official documentation before looking for alternative solutions from other sources.

¹The specific line reported: “run *doctest* on the module if it is invoked on the command line”.

New standards. We think that it would be extremely useful to define a sort of standard structure for most of the corpora that are included into *NLTK*. At the present moment, corpus readers are created and re-created in order to adapt new parsing strategies to existing corpora; this is mostly done because datasets are often built by their authors using customised and fancy syntaxes which do not follow any existing standard. We think that *NLTK* could in some way raise this issue, engaging researchers and developers in a serious discussion about possible standards that would result in a more flexible structure for NLP tools. The role that *NLTK* plays in the academic environment, together with its visibility outside the academic boundaries, could be sufficient to publicize and spread new practices from which the entire field of Natural Language Processing could benefit.

6.3 Future work

Our implementation provides a significative base for Sentiment Analysis applications within *NLTK*. However, due to the extremely broad range of approaches and corpora that can be employed in Sentiment Analysis, several contributions can still be made to further extend the toolkit support for this field.

Aspect/feature-based approaches. Our corpus readers allow to use corpora that were not previously included into *NLTK*, some of which are specifically built for *aspect-based* analysis. However, time and scope constraints did not allow us to implement specific modules to apply the techniques described in related literature (see chapter 2). We believe that this is one of the most interesting areas in Opinion Mining from both a practical and theoretical point of view, and we therefore hope that our work will promote new contributions in this direction.

Visualization. At the present moment we only provide a single method to visualize sentiment information using lexicon categories. More advanced techniques can be developed, for example, to visualize detailed *semantic orientation* scores of single words and to summarize sentiment proportions in *document-level* analysis. Given the high correlation between the *Natural Language ToolKit* and academic research, we furthermore believe that providing visualization methods for specific syntaxes (e.g. LaTeX, markdown, etc.) would represent a valuable improvement as well.

Sentence-level analysis. Our `demo_liu_hu_lexicon()` method only implements a basic strategy for lexicon-based analysis, which can be considered as a naïve baseline rather than an effective classification method. We aim to improve this code in order to supply a better tool and a more sensible approach.

Lexicon generation and analysis support. We believe that all lexicon-based approaches would benefit from a common structure similar to the one we built for Machine Learning tasks. Users could be asked to choose their favorite approach (e.g. *Vader*, *Wilson*, *SO-Cal* etc.) to build new lexicons and employ them together with specific analysis techniques. This modular tool would allow easier and straightforward comparisons obtained simply modifying a limited number of parameters.

Micro-blogging POS-tagging support. Most POS-taggers are trained on data which does not relate to social-networking and micro-blogging domains². These models do not therefore produce significant results on texts which include emoticons, URLs, and other specific token categories that we described in chapter 4. We believe that adding specific support for this domain within *NLTK* would represent a valuable improvement, especially since manually-annotated corpora are already available for this purpose (see Gimpel et al., 2011).

²It is actually not clear on which corpus the *NLTK* default `pos_tag` module has been trained on (Alvations, 2015).

Chapter 7

Conclusions

Our project was aimed to add support for Sentiment Analysis within the *Natural Language ToolKit* platform. In order to achieve our objective, the following contributions were made:

- we reviewed a wide range of corpora and lexicons for Sentiment Analysis. Each dataset was evaluated taking into account factors such as contents, quality, structure, popularity and licensing terms of the data; In light of these aspects we subsequently provided final considerations about the suitability of each dataset for incorporation within *NLTK*;
- we had an active involvement in the incorporation of several of the described resources into the *Natural Language ToolKit* data distribution;
- new corpus reader classes were specifically built to parse and output the information contained in the new datasets. The design of these new readers also involved the creation of auxiliary classes that simplify the analysis of extracted information;
- we conducted a broad survey of commonly employed features and preprocessing techniques for Sentiment Analysis, providing an efficient implementation for some of the functionalities that were not previously available within *NLTK*;
- we constructed a framework that easily allows the implementation of different combinations of classifiers, tokenizers, preprocessing functions, feature extractors and corpora. Code is especially designed for teaching and research purposes;
- our design decisions about class inheritance and code structure allowed us to

consistently integrate our new features within the existing *NLTK* libraries and functionalities and libraries;

- our contributions conformed to *NLTK* guidelines and standards provided by the *NLTK-dev* community; this involved the development of regression test functionalities and detailed user-oriented documentation. Conformance to open source and Python best practices was also guaranteed.

We believe that our project can be considered a substantial base for further development of Sentiment Analysis functionalities within *NLTK*, and that it highly simplifies implementation and comparison of opinion mining tasks within the platform. We therefore believe that our work met all the objectives that were previously set and described.

Bibliography

Alec Go, R. B. and Huang, L. (2009a). Sentiment140.

<http://help.sentiment140.com/for-students>.

Alec Go, R. B. and Huang, L. (2009b). Sentiment140 forum.

<https://groups.google.com/forum/#!forum/sentiment140>.

Alvations (2015). Where did the nltk pos_tag model come from?

<https://github.com/nltk/nltk/issues/1063>.

Apache, S. F. (2004). Apache license 2.0.

<http://www.apache.org/licenses/LICENSE-2.0>.

Baccianella, S., Esuli, A., and Sebastiani, F. (2010). Sentiwordnet 3.0: An enhanced lexical resource for sentiment analysis and opinion mining. In *LREC*, volume 10, pages 2200–2204.

Bever, T. G. (1970). The cognitive basis for linguistic structures. *1970*, pages 279–362.

Bird, S., Klein, E., and Loper, E. (2009). *Natural language processing with Python*. "O'Reilly Media, Inc."

Councill, I. G., McDonald, R., and Velikovich, L. (2010). What's great and what's not: learning to classify the scope of negation for improved sentiment analysis. In *Proceedings of the workshop on negation and speculation in natural language processing*, pages 51–59. Association for Computational Linguistics.

CreativeCommons, C. (2013). Creative Commons - Attribution 4.0 International.

<https://creativecommons.org/licenses/by/4.0>.

Darroch, J. N. and Ratcliff, D. (1972). Generalized iterative scaling for log-linear models. *The annals of mathematical statistics*, pages 1470–1480.

- Das, S. and Chen, M. (2001). Yahoo! for Amazon: Extracting market sentiment from stock message boards. In *Proceedings of the Asia Pacific finance association annual conference (APFA)*, volume 35, page 43. Bangkok, Thailand.
- Das, S. R. and Chen, M. Y. (2007). Yahoo! for Amazon: Sentiment extraction from small talk on the web. *Management Science*, 53(9):1375–1388.
- Dave, K., Lawrence, S., and Pennock, D. M. (2003). Mining the peanut gallery: Opinion extraction and semantic classification of product reviews. In *Proceedings of the 12th international conference on World Wide Web*, pages 519–528. ACM.
- de Souza, S. C. B., Anquetil, N., and de Oliveira, K. M. (2005). A study of the documentation essential to software maintenance. In *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*, pages 68–75. ACM.
- Ding, X., Liu, B., and Yu, P. S. (2008). A holistic lexicon-based approach to opinion mining. In *Proceedings of the 2008 International Conference on Web Search and Data Mining*, pages 231–240. ACM.
- Dumais, S. and Chen, H. (2000). Hierarchical classification of web content. In *Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 256–263. ACM.
- EMC (2014). The digital universe of opportunities.
<http://www.emc.com/infographics/digital-universe-2014.htm>.
- Esuli, A. and Sebastiani, F. (2006). Sentiwordnet: A publicly available lexical resource for opinion mining. In *Proceedings of LREC*, volume 6, pages 417–422. Citeseer.
- Ganapathibhotla, M. and Liu, B. (2008). Mining opinions in comparative sentences. In *Proceedings of the 22nd International Conference on Computational Linguistics-Volume 1*, pages 241–248. Association for Computational Linguistics.
- Gantz, J. and Reinsel, D. (2012). The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the Future*, 2007:1–16.
- Gimpel, K., Schneider, N., O'Connor, B., Das, D., Mills, D., Eisenstein, J., Heilman, M., Yogatama, D., Flanigan, J., and Smith, N. A. (2011). Part-of-speech tagging

for Twitter: Annotation, features, and experiments. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: short papers-Volume 2*, pages 42–47. Association for Computational Linguistics.

GitHub (2014). GitHub glossary.

<https://help.github.com/articles/github-glossary>.

Go, A., Bhayani, R., and Huang, L. (2009). Twitter sentiment classification using distant supervision. *CS224N Project Report, Stanford*, 1:12.

González-Ibáñez, R., Muresan, S., and Wacholder, N. (2011). Identifying sarcasm in Twitter: a closer look. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: short papers-Volume 2*, pages 581–586. Association for Computational Linguistics.

Hale, J. (2001). A probabilistic earley parser as a psycholinguistic model. In *Proceedings of the second meeting of the North American Chapter of the Association for Computational Linguistics on Language technologies*, pages 1–8. Association for Computational Linguistics.

Harvard and MIT (2000). The harvard general inquirer.

<http://www.wjh.harvard.edu/~inquirer>.

Hatzivassiloglou, V. and McKeown, K. R. (1997). Predicting the semantic orientation of adjectives. In *Proceedings of the 35th annual meeting of the association for computational linguistics and eighth conference of the european chapter of the association for computational linguistics*, pages 174–181. Association for Computational Linguistics.

Hu, M. and Liu, B. (2004a). Mining and summarizing customer reviews. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 168–177. ACM.

Hu, M. and Liu, B. (2004b). Mining opinion features in customer reviews. In *AAAI*, volume 4, pages 755–760.

Hutto, C. (2014a). Vader on comp.social.

<http://comp.social.gatech.edu/papers>.

Hutto, C. (2014b). vaderSentiment (GitHub).

<https://github.com/cjhutto/vaderSentiment>.

Hutto, C. and Gilbert, E. (2014). Vader: A parsimonious rule-based model for sentiment analysis of social media text. In *Eighth International AAAI Conference on Weblogs and Social Media*.

Janardhana, R. (2008). How to build a Twitter sentiment analyzer.

<http://ravikiranj.net/posts/2012/code/how-build-twitter-sentiment-analyzer>.

Jiang, L., Yu, M., Zhou, M., Liu, X., and Zhao, T. (2011). Target-dependent Twitter sentiment classification. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*, pages 151–160. Association for Computational Linguistics.

Jindal, N. and Liu, B. (2006a). Identifying comparative sentences in text documents. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 244–251. ACM.

Jindal, N. and Liu, B. (2006b). Mining comparative sentences and relations. In *AAAI*, volume 22, pages 1331–1336.

Joachims, T. (1998). *Text categorization with support vector machines: Learning with many relevant features*. Springer.

Kennedy, A. and Inkpen, D. (2006). Sentiment classification of movie reviews using contextual valence shifters. *Computational intelligence*, 22(2):110–125.

Klein, D. and Manning, C. D. (2004). Corpus-based induction of syntactic structure: Models of dependency and constituency. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*, page 478. Association for Computational Linguistics.

Liberman, M. (2003). Zettascale linguistics.

<http://itre.cis.upenn.edu/myl/language-log/archives/000087.html>.

LinguisticsSE, L. S. E. (2011). Is “double positive meaning negative” a common phenomenon? - linguistics stack exchange.

<http://linguistics.stackexchange.com/questions/981/is-double-positive-meaning-negative-a-common-phenomenon>.

- Liu, B. (2004). Opinion mining, sentiment analysis, and opinion spam detection.
<http://www.cs.uic.edu/~liub/FBS/sentiment-analysis.html>.
- Liu, B. (2007). *Web data mining: exploring hyperlinks, contents, and usage data*. Springer Science & Business Media.
- Liu, B. (2010). Sentiment analysis and subjectivity. *Handbook of natural language processing*, 2:627–666.
- Liu, B. (2011). Sentiment analysis tutorial. *Talk given at AAAI-2011, Monday*.
- Liu, B., Hu, M., and Cheng, J. (2005). Opinion observer: analyzing and comparing opinions on the web. In *Proceedings of the 14th international conference on World Wide Web*, pages 342–351. ACM.
- Madnani, N. and Dorr, B. J. (2008). Combining open-source with research to re-engineer a hands-on introductory nlp course. In *Proceedings of the Third Workshop on Issues in Teaching Computational Linguistics*, pages 71–79. Association for Computational Linguistics.
- Manning, C. D., Raghavan, P., Schütze, H., et al. (2008). *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge.
- Marcus, M. P., Marcinkiewicz, M. A., and Santorini, B. (1993). Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19(2):313–330.
- Miller, G. A. (1995). Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41.
- Na, J.-C., Sui, H., Khoo, C. S., Chan, S., and Zhou, Y. (2004). Effectiveness of simple linguistic processing in automatic sentiment classification of product reviews.
- NLTK (2008). Nltk-dev forum.
<https://groups.google.com/forum/#!forum/nltk-dev>.
- NLTK (2015a). Nltk developers guide.
<https://github.com/nltk/nltk/wiki/Developers-Guide>.
- NLTK (2015b). Nltk package structure.
<https://github.com/nltk/nltk/wiki/Package-Structure>.

NLTK (2015c). Nltk wiki.

<https://github.com/nltk/nltk/wiki>.

Pang, B. and Lee, L. (2004). A sentimental education: Sentiment analysis using subjectivity summarization based on minimum cuts. In *Proceedings of the 42nd annual meeting on Association for Computational Linguistics*, page 271. Association for Computational Linguistics.

Pang, B. and Lee, L. (2005). Seeing stars: Exploiting class relationships for sentiment categorization with respect to rating scales. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 115–124. Association for Computational Linguistics.

Pang, B., Lee, L., and Vaithyanathan, S. (2002). Thumbs up?: sentiment classification using machine learning techniques. In *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10*, pages 79–86. Association for Computational Linguistics.

Pantone, P. (2015a). Abstractcollocationfinder from_documents() issues.

<https://github.com/nltk/nltk/issues/1033>.

Pantone, P. (2015b). Doctests and unicode failures.

<https://github.com/nltk/nltk/issues/1042>.

Pennebaker, J. W., Chung, C. K., Ireland, M., Gonzales, A., and Booth, R. J. (2007). The development and psychometric properties of liwc2007.

Pennebaker, J. W., Francis, M. E., and Booth, R. J. (2001). Linguistic inquiry and word count: Liwc 2001. *Mahway: Lawrence Erlbaum Associates*, 71:2001.

Pennebaker Conglomerates, I. (2007). Linguistic inquiry and word count (liwc).

<http://www.liwc.net>.

Perkins, J. (2010a). Streamhacker.

<http://streamhacker.com>.

Perkins, J. (2010b). Text classification for sentiment analysis - eliminate low information features.

<http://streamhacker.com/2010/06/16/text-classification-sentiment-analysis-eliminate-low-information-features>.

Perkins, J. (2011). Hierarchical classification.

<http://streamhacker.com/2011/01/05/hierarchical-classification>.

Perkins, J. (2014). *Python 3 Text Processing with NLTK 3 Cookbook*. Packt Publishing Ltd.

Potts, C. (2011a). On the negativity of negation. In *Semantics and Linguistic Theory*, number 20, pages 636–659.

Potts, C. (2011b). Sentiment symposium tutorial.

<http://sentiment.christopherpotts.net>.

Project, E. (2007). Experience project.

<http://www.experienceproject.com>.

Read, J. (2005). Using emoticons to reduce dependency in machine learning techniques for sentiment classification. In *Proceedings of the ACL student research workshop*, pages 43–48. Association for Computational Linguistics.

Riloff, E. and Wiebe, J. (2003). Learning extraction patterns for subjective expressions. In *Proceedings of the 2003 conference on Empirical methods in natural language processing*, pages 105–112. Association for Computational Linguistics.

Robb, A. (2014). How capital letters became internet code for yelling.

<http://www.newrepublic.com/article/117390/netiquette-capitalization-how-caps-became-code-yelling>.

Roe, C. (2012). The growth of unstructured data: What to do with all those zettabytes?

<http://www.dataversity.net/the-growth-of-unstructured-data-what-are-we-going-to-do-with-all-those-zettabytes>.

Saif, H., Fernandez, M., He, Y., and Alani, H. (2013). Evaluation datasets for Twitter sentiment analysis: a survey and a new dataset, the sts-gold.

Sanders, N. (2011). Sanders analytics Twitter sentiment corpus.

<http://www.sananalytics.com/lab/twitter-sentiment>.

SemEval (2014). Semeval-2014 task 9.

<http://alt.qcri.org/semeval2014/task9>.

- Somasundaran, S. and Wiebe, J. (2009). Recognizing stances in online debates. In *Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 226–234, Suntec, Singapore. Association for Computational Linguistics.
- Somasundaran, S. and Wiebe, J. (2010). Recognizing stances in ideological on-line debates. In *Proceedings of the NAACL HLT 2010 Workshop on Computational Approaches to Analysis and Generation of Emotion in Text*, pages 116–124, Los Angeles, CA. Association for Computational Linguistics.
- Stone, P. J., Dunphy, D. C., and Smith, M. S. (1966). The general inquirer: A computer approach to content analysis.
- Taboada, M., Brooke, J., Tofiloski, M., Voll, K., and Stede, M. (2011). Lexicon-based methods for sentiment analysis. *Computational linguistics*, 37(2):267–307.
- TASS (2013). Tass 2013 workshop on sentiment analysis at sepln.
<http://www.daedalus.es/TASS2013/corpus.php>.
- Turney, P. D. (2002). Thumbs up or thumbs down?: semantic orientation applied to unsupervised classification of reviews. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 417–424. Association for Computational Linguistics.
- Twitter (2015). Twitter terms of service.
<https://dev.twitter.com/overview/terms/agreement-and-policy>.
- University of Pennsylvania, D. o. L. (2003). Alphabetical list of part-of-speech tags used in the penn treebank project.
https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html.
- Villena Román, J., Lana Serrano, S., Martínez Cámara, E., and González Cristóbal, J. C. (2013). Tass-workshop on sentiment analysis at sepln.
- Wiebe, J. M., Bruce, R. F., and O’Hara, T. P. (1999). Development and use of a gold-standard data set for subjectivity classifications. In *Proceedings of the 37th annual meeting of the Association for Computational Linguistics on Computational Linguistics*, pages 246–253. Association for Computational Linguistics.

- Wilson, T., Wiebe, J., and Hoffmann, P. (2005). Recognizing contextual polarity in phrase-level sentiment analysis. In *Proceedings of the conference on human language technology and empirical methods in natural language processing*, pages 347–354. Association for Computational Linguistics.
- Wilson, T., Wiebe, J., and Hoffmann, P. (2009). Recognizing contextual polarity: An exploration of features for phrase-level sentiment analysis. *Computational linguistics*, 35(3):399–433.
- Witten, I. H. and Frank, E. (2005). *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann.
- Woodfield, S. N., Dunsmore, H. E., and Shen, V. Y. (1981). The effect of modularization and comments on program comprehension. In *Proceedings of the 5th international conference on Software engineering*, pages 215–223. IEEE Press.
- Yessenov, K. and Misailovic, S. (2009). Sentiment analysis of movie review comments. *Methodology*, pages 1–17.
- Yu, H. and Hatzivassiloglou, V. (2003). Towards answering opinion questions: Separating facts from opinions and identifying the polarity of opinion sentences. In *Proceedings of the 2003 conference on Empirical methods in natural language processing*, pages 129–136. Association for Computational Linguistics.