UNIVERSITÁ DI PISA
DOTTORATO DI RICERCA IN INGEGNERIA DELL'INFORMAZIONE

COMPUTING FRAMEWORKS AND APPLICATIONS FOR TEXT PROCESSING ON BIG DATA

DOCTORAL THESIS

Author
Tiziano Fagni

Tutor (s)
Prof. Avvenuti Marco, Dr. Esuli Andrea

Reviewer (s)
Prof. Di Pietro Roberto, Prof. Orlando Salvatore, Dr. Silvestri Fabrizio

The Coordinator of the PhD Program
Prof. Marco Luise

Pisa, November 2017

Cycle XXIX

Alla mia famiglia.

"Information is the oil of the 21st century, and analytics is the combustion engine”
Peter Sondergaard

Acknowledgements

This path toward PhD has been for me a very satisfactory experience which has greatly enriched me both personally and professionally.

My first big thank goes to Andrea Esuli, a competent advisor and a good friend which has shared with me this journey by providing suggestions and ideas on how to set up to the best all the contributions of the thesis. A second very great thank goes to Marco Avvenuti, always precise and present in his role of internal advisor, as well as an experienced researcher from whom you can always learn new things.

My gratitude also goes to all the persons I have collaborated with during these years. Alejandro Moreo Fernández, a great roommate which has worked with me on several of the ideas proposed in this thesis. All the people belonging to WAFI-CI research group, brilliant teammates sharing projects and research interests I am working on. All the guys and girls coming with me at lunch, often involving me in very interesting discussions on work topics or funny things.

Finally, I would like to say a very big thanks to my wife Valentina for being my first supporter in everything I do, to my children Niccolò and Leonardo for all the satisfactions they give us, and to my parents to allowed me to study and to become a better person.

Ringraziamenti

Questo percorso verso il conseguimento del dottorato è stato per me una esperienza estremamente appagante che mi ha molto arricchito sia personalmente che professionalmente.

Il mio primo grande ringraziamento va ad Andrea Esuli, un advisor competente e un buon amico che ha condiviso con me questo viaggio fornendo suggerimenti e idee su come impostare al meglio i lavori proposti nella tesi. Un secondo sentito ringraziamento va a Marco Avvenuti, sempre preciso e presente nel suo ruolo di advisor interno, oltre che un ricercatore esperto da cui è possibile sempre imparare cose nuove.

La mia gratitudine va anche a tutte le altre persone con cui ho interagito nel corso di questi anni. Alejandro Moreo Fernández, un caro collega di stanza che ha lavorato con me su alcune delle idee proposte in questa tesi. Tutte le persone del gruppo di ricerca WAFI-CI, brillanti compagni di team che condivono con me progetti di lavoro e tematiche di ricerche a cui sono interessato. Tutti i ragazzi e le ragazze con cui condivido l'ora del pranzo, che spesso mi coinvolgono in discussioni interessanti su tematiche di lavoro o cose particolarmente divertenti.

Infine, vorrei dire un grandissimo grazie a mia moglie Valentina per essere sempre il mio primo tifoso in tutto quello che faccio, ai miei figli Niccolò e Leonardo per tutte le soddisfazioni che ci danno e ai miei genitori per avermi permesso di studiare e di diventare una persona migliore.

Summary

The extreme popularity gained in the last years by Internet, and the Web in particular, as the main communication tool used by users and companies, has determined an exponential growth of generated information potentially useful for data analysis tasks. A large part of this generated data is represented by textual content, which is an information type completely unstructured (or marginally structured, e.g., HTML), which requires complex statistical and linguistic analyses to obtain helpful structured and semantic information from the original data. Analyzing large amount of texts is not a trivial task. Existing text mining software and libraries are generally optimized for sequential code execution on single machines. Furthermore, these software packages usually provide a limited set of specific features for text analyses, imposing to programmers to spend a relevant effort in the integration tasks required to build complete software solutions. In addition, no existing big data processing tools provide specialized libraries able to natively process textual contents. This issue's solution is demanded to developers, which have to provide ingenious integration strategies case per case, without the possibility to use a systematic and well defined approach.

In this thesis, we worked on overcoming these limitations by investigating three possible big data application scenarios in which we want to show how to approach text mining problems with effective integration strategies of different technologies and deployment of innovative software tools aiming at simplifying the work of developers.

In first scenario, covering standalone efficient tools operating with small/medium amount of data, we first introduce JATECS, a sequential text processing framework which offers many of the tools needed by developers to perform complete experimentations on text analytics problems. The software offers a powerful data structure called Index which allows programmers to easily access and manipulate analyzed information. Moreover, JATECS covers, in a "plug and play" programming style, all the steps and components that are usually part of a complete text mining pipeline: data acquisition, data preprocessing, data indexing, data models learning and data models evaluation.

We then move to analyze two specific use cases of such application scenario which make extensive use of JATECS as the core textual processing framework for the designed solutions. In the first use case we propose an automatic classifier for company

websites to be used as a support tool for classification task of companies by industry sector. As a second case study, we show how to build a mobile apps classification system working on custom taxonomies defined by users. In both cases, we use several of the methods and algorithms provided by JATECS to build and evaluate the systems.

We then explore a second application scenario in which the focus is on processing high amount of textual information using computation environments based on multi-thread single machines or of distributed type. In this context, we propose and evaluate Processfast, a new multi-thread data processing library offering the possibility to easily use task and data parallelism approaches inside the same application. We also present distributed solutions by proposing NLP4Spark, a data ingestion library built over Apache Spark and similar in spirit to JATECS, which offers a clean and concise API to conveniently access and transform managed data. As an interesting case study of "big data" technologies integration, we also show how to build a social media application ingesting data from Twitter and which is used as monitoring/support tool for emergency responders in crisis management of man-made/natural disasters.

We conclude our thesis by exploring a third application scenario in which deep learning methods are used to learn complex models over large amount of data. We thus present a novel deep learning method called Text2Vis which is a cross-media retrieval neural network able to translate a textual query into a visual space query and thus use this visual representation to search for similar images into a large storage of images.

Sommario

La grande popolarità ottenuta negli ultimi anni da Internet, e in particolare dal Web, come strumento principale di comunicazione usato dalle persone e dalle aziende, ha determinato una crescita esponenziale dei dati generati che potenzialmente potrebbero essere utilizzati a fini di analisi. Una larga parte di questi dati è di natura testuale, un tipo di contenuto non strutturato che richiede complesse analisi di tipo statistico e linguistico in modo da ottenere informazioni strutturate e semantiche rispetto ai dati analizzati. Processare elevate quantità di testo in modo efficiente non è un compito facile. I software e le librerie di text mining esistenti sono generalmente ottimizzate per eseguire il codice sequenzialmente su singole macchine. Inoltre, questi pacchetti software rendono disponibili un numero limitato di metodi o algoritmi specifici per l'analisi del testo, imponendo ai programmatori grandi sforzi in termini di integrazione di strumenti diversi al fine di poter realizzare soluzioni complesse e complete. In aggiunta a questo, nessuno strumento di big data processing fornisce librerie specializzate in grado di processare nativamente contenuti testuali. La soluzione a queste problematiche è di solito lasciata agli sviluppatori, i quali si devono ingegnare con strategie di integrazione da applicare caso per caso, senza la possibilità di utilizzare un approccio sistematico e ben definito.

In questa tesi abbiamo lavorato per superare queste limitazioni approfondendo queste tematiche su tre differenti scenari applicativi di tipo "big data". In questo contesto, vogliamo mostrare come approcciare problemi di mining su testo con strategie efficaci di integrazione di tecnologie diverse e mediante la realizzazione di strumenti software innovativi aventi lo scopo di semplificare in modo consistente il lavoro degli sviluppatori.

Nel primo scenario, focalizzato sulla realizzazione di strumenti standalone efficienti ed operanti con quantità di dati medio-piccole, per prima cosa introduciamo JATECS, un framework sequenziale di processamento testuale che fornisce molte delle funzionalità che tipicamente servono ai programmatori per realizzare sperimentazioni complete su problemi di analisi del testo. Il software realizzato permette, attraverso una struttura dati chiamata Indice, di accedere e manipolare facilmente i dati indicizzati. Inoltre, mediante l'ausilio di uno stile di programmazione "plug and play", riesce a coprire tutti

i passi necessari e a fornire componenti pronti per la realizzazione di una tipica pipeline di mining del testo: acquisizione dei dati, preprocessamento delle informazioni, indicizzazione dei dati, apprendimento di modelli e valutazione dei modelli appresi.

Successivamente andiamo ad analizzare due specifici casi di uso riconducibili a questo scenario applicativo e che utilizzano JATECS estensivamente come strumento di base di analisi del testo nelle souzioni proposte. Nel primo caso di studio proponiamo un sistema automatico di classificazione di siti Web come strumento di supporto per il problema delle classificazione di aziende per settore industriale. Nel secondo caso di studio invece proponiamo un software in grado fare classificazione automatica di applicazioni per dispositivi mobili considerando tassonomie di classi definite dagli utenti. In entrambi i casi, usiamo alcuni dei metodi e algoritmi forniti da JATECS per realizzare e valutare i sistemi proposti.

Esploriamo poi un secondo scenario applicativo dove lo scopo è processare elevate quantità di dati testuali utilizzando ambienti di esecuzione basati sia su approcci multi-thread a singola macchina sia in ambienti distribuiti. Iniziamo proponendo e valutando Processfast, una nuova libreria multi-thread di data processing che è caratterizzata dalla possibilità da parte del programatore di usare all'interno dello stesso applicativo di un approccio al parallelismo basato sia sui dati che sui task. Succesivamente presentiamo NLP4Spark, una nuova libreria distribuita costruita sopra Apache Spark che è caratterizzata da una API semplice e concisa per l'accesso e la manipolazione dei dati processati. Infine, come interessante caso di studio sulla integrazione di tecnologie "big data", facciamo vedere come progettare una applicazione su social-media in grado recuperare i dati da Twitter e di essere usata come strumento di monitoraggio/supporto per la protezione civile nella gestione di crisi dovute a catastrofi naturali o provocate dall'uomo.

Concludiamo questa tesi affrontando un terzo scenario applicativo nel quale si fa uso di metodi di deep learning su grandi quantità di dati per ottenere modelli di apprendimento particolarmente complessi. Presentiamo quindi una nuova architettura di deep learning chiamata Text2Vis che utilizza una rete neurale per fare cross-media retrieval tra testo e immagini. Il sistema è in grado di trasformare una query testuale in una query nello spazio visuale delle immagini, e successivamente di usare questa rappresentazione visuale per cercare immagini simili presenti su vaste collezioni di dati.

List of publications

International Journals

1. Fabio Carrara, Andrea Esuli, Tiziano Fagni, Fabrizio Falchi, Alejandro Moreo Fernández: "Picture It In Your Mind: Generating High Level Visual Representations From Textual Descriptions". Extended version of Neu-IR '16 SIGIR work (see below), published on "Information Retrieval Journal", Springer, October 2017, DOI: 10.1007/s10791-017-9318-6.

2. Marco Avvenuti, Stefano Cresci, Fabio Del Vigna, Tiziano Fagni, Maurizio Tesconi: "CrisMap: A Big Data Crisis Mapping System based on Damage Detection and Geoparsing". Currently in status "accepted with major revision" for publication on "Information Systems Frontiers" journal, Springer.

International Conferences/Workshops with Peer Review

1. Giacomo Berardi, Andrea Esuli, Tiziano Fagni, Fabrizio Sebastiani: "Multi-store metadata-based supervised mobile app classification". Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015.

2. Giacomo Berardi, Andrea Esuli, Tiziano Fagni, Fabrizio Sebastiani: "Classifying websites by industry sector: a study in feature design.". Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015.

3. Andrea Esuli, Tiziano Fagni: "Processfast, a java framework for development of concurrent and distributed applications". Proceedings of the 6th Italian Information Retrieval Workshop, Cagliari, Italy, May 25-26, 2015.

4. Fabio Carrara, Andrea Esuli, Tiziano Fagni, Fabrizio Falchi, Alejandro Moreo Fernández: "Picture It In Your Mind: Generating High Level Visual Representations From Textual Descriptions". Neu-IR '16 SIGIR Workshop on Neural Information Retrieval, July 21, 2016, Pisa, Italy.

Others

1. Andrea Esuli, Tiziano Fagni: "SparkBOOST, an Apache Spark-based boosting library". Technical report ISTI-CNR, cnr.isti/2016-TR-016, March 2016.

2. Andrea Esuli, Tiziano Fagni, Alejandro Moreo Fernandez: "JaTeCS, an open-source JAva TExt Categorization System". ArXiv e-prints:1706.06802, June 2017.

Contents

CHAPTER 1

Introduction

The last two decades have seen the massive adoption of Internet by people and organizations as the main tool for communications, work and fun in every imaginable context. Internet has now become the hub of all data communications, including not only human-produced data but also machine-produced data (e.g. sensors, logs, etc.). Twenty years ago the only people having the necessity to analyze massive amount of data were especially those belonging to scientific community and involved in specific research fields (e.g. math and physical simulations). The Internet has dramatically changed this situation. Today, thanks to the rapid evolution of digital technologies and communication systems, every company providing some type of service to people, in digital form or through conventional channels, has a direct link with its customers through the Web (e.g. Internet site, Facebook page, Twitter account, etc.). A company has therefore the opportunity to collect a great amount of feedback on their services in terms of usage, satisfaction and improvement in relation to what they offer. On the other hand, today every person is almost always connected through smart devices (e.g. smartphones, smartwatches, tablets, etc.), continuously exploiting the Web for getting or sharing information, for shopping, for online banking, etc. or for using social networks to share moments, videos, photos with their friends or their family. This people's explicit set of actions generates in turn also a massive data source of implicit and mostly unstructured actions related to users behaviors which, if combined with explicit actions and analyzed, could provide very useful insights about how people use and interact with available online services. We refer with the term "big data" to this huge amount of data produced and ready to be analyzed. In the following sections, we characterize this concept focusing specifically on textual data analysis, which is the main big data research domain covered in this thesis. Indeed text is a communication media that dominates on the Web, but it is by its nature an information that is completely unstructured, therefore

it presents several difficult challenges in order to extract useful information for analysis purposes.

1.1 What is "big data"?

Big data is a term that describes the large volume of data – both structured, semi-structured and unstructured – that is created on a day-to-day basis by users and/or machines due to usage of public/restricted sites/services available on Internet. Big data thematics have attracted a lot of economic interests in the last years and for good reasons. Exploiting correctly these new technologies and data analysis tools in order to acquire precious hidden insights about their business, can give to companies and organizations adopting it great advantages versus the competitors and let them to be ready to follow or adapt to new emerging trends in the market [42, 117, 144]. To be able to successful manage big data problems, we need to handle several challenges related to data acquisition, data storage, data cleansing, data analysis and data visualization. In the last years, data analysis is becoming increasingly important as it is the main tool to exploit hidden value from available data. Concepts like predictive analytics, user behavior analytics or some other advanced machine learning (ML) method could help find interesting data insights able to provide very useful strategic information about the analyzed problem, e.g. user behavior analytics over an existing service could suggest to business analysts new features to add in order to attract new users. As reported in several articles [83, 117, 118], the amount of information produced directly by the users or indirectly by other mechanisms (e.g. sensors, logs, etc.) has seen a massive increase in the last years and the predictions for the future are that this trend not only will continue but possibly it will be even more accelerated. In a recent published white paper [176], IDC[1] estimates that in 2025 the amount of data produced globally will reach the astronomical number of 163 Zettabytes, which is about 10 times higher than the amount of data generated in 2016. In addition, IDC confirms the trend that data growth in the next few years will follow an exponential curve, therefore suggesting that effective data analytics techniques will be a key factor for future business intelligence tasks.

It is very difficult to organically characterize this massive amount of data because it is generated by a multitude of very different data sources with completely different characteristics among them. That said, we can identify a series of precise characteristics which best describes the extremely complex nature of big data. Traditionally, this has been referred with the rule of Four V's[2] but in the last years others V's have been added and today we can count at least 7 V's. These important characteristics are described in the following:

Volume The quantity of data to be analyzed is massive and it can not be practical handled with neither the traditional data processing software tools (e.g. classic SQL DBMS) nor using a powerful single server machine (some type of distributed and horizontally scalable processing engine is needed for effective computations).

Velocity Velocity is the speed in which data is generated and available for processing. With the explosion in popularity of social networks and the ever increasing availability of smart devices and IoT (Internet of Things) devices, the data is created at

[1] https://www.idc.com/

[2] http://www.ibmbigdatahub.com/infographic/four-vs-big-data

very high speed rates and need to be processed in near-real time to be effectively used. 11

Variety Variety suggests that the type of data to be processed can be extremely variegated. The data could be in a well known structured format (e.g. XML, SQL table, etc.), in an unstructured form (e.g. a free text) or in a semi-structured form (e.g. an HTML page).

Variability Variability means that to reasoning correctly over available data you must always consider the input domain context where your problem arise. This simply means that the same data could have very different meaning and interpretations based on the considered domain.

Veracity Veracity indicates how much are reliable the information you have in relation to the problem you want to solve. You must ensure that your data is representative of your input domain and it does not contain wrong data (e.g. outliers).

Visualization Data visualization is a critical aspect in order to successful interpret this huge amount of information. For example, usage of charts to visualize large amounts of complex data is much more effective than to deal with spreadsheets and reports completely full of numbers and formulas.

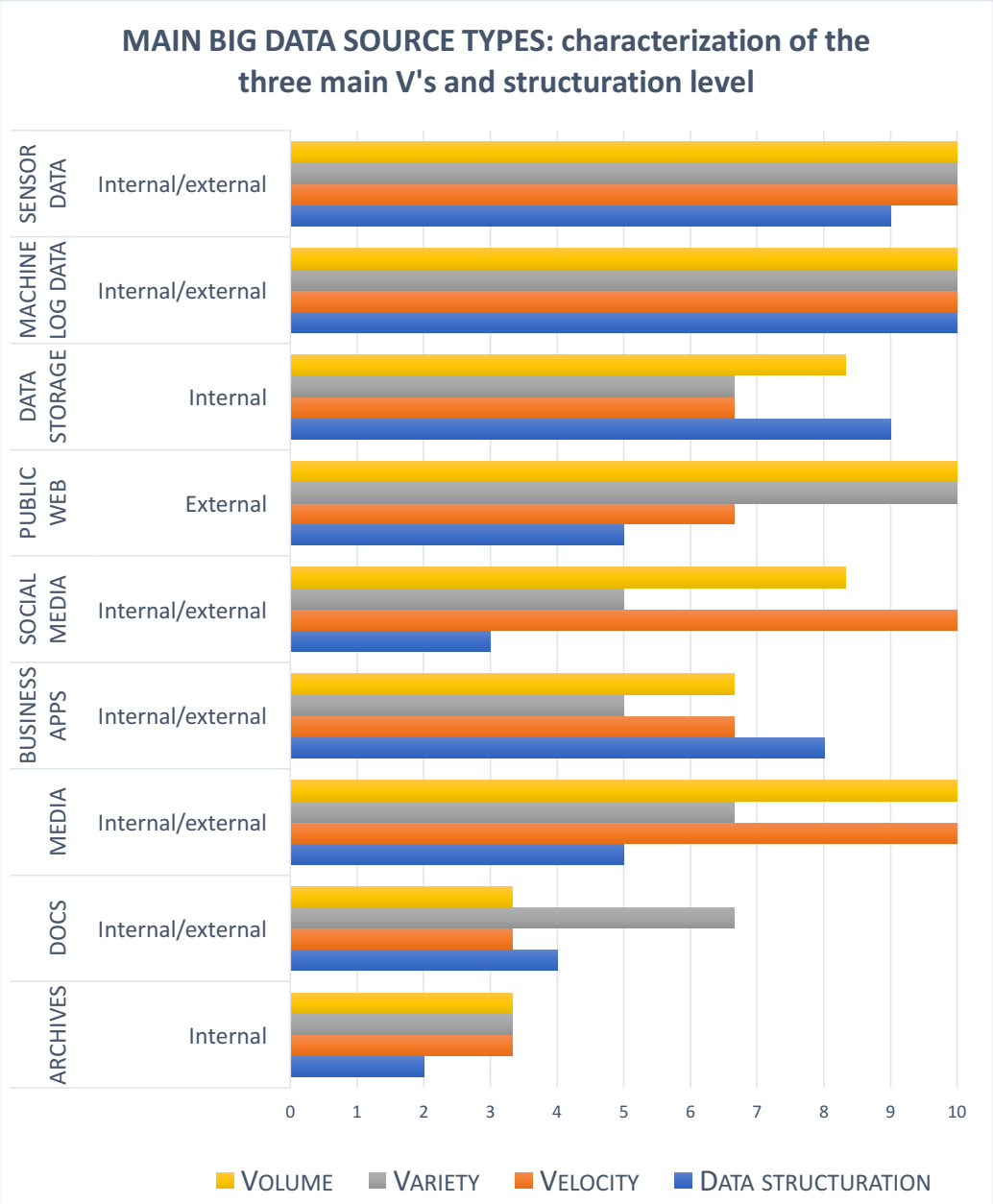
Value Added value is what you want to obtain from the data at the end of the game. Value is intended as strategic/insight information which you can spend to improve your existing business or to create new businesses opportunities.

Big data is everywhere and it can help organizations of any industry in many different ways. Nowadays there is so much data that existing hardware and conventional software are not able to deal with the vast amount of different types of data that is created at such a high speed. Therefore big data has become too complex and too dynamic to be processed, stored, analyzed and managed with traditional data tools. Historically, traditional data have been processed exploiting centralized databases (e.g. using SQL DBMS such as MySql[3], PostgreSQL[4], etc.) and performing all the needed complex computations using a powerful single computer system. This approach is very costly because it is based on vertical scalability, which rapidly determines a big growth of hardware cost and is anyway ineffective to process great amount of data in reasonable time. Another assumption made with traditional data which is no more valid in big data contexts is that the information to analyze is structured. In big data contexts we solve these problems by adopting a distributed system architecture exploiting horizontal scalability, both for data processing and for data distribution, by using commodity machines as singles nodes of the distributed computing platform. In order to handle data heterogeneity, it is essential also to switch to schema-less distributed data storage able to dynamically adapt at runtime to the structure of the data (e.g. NoSQL systems such as Apache Cassandra[5], MongoDB[6], etc.).

One interesting big data question is from where all this information is generated. The Figure 1.1, based on the data reported in [205], tries to answer this question by

[3] https://www.mysql.com/
[4] https://www.postgresql.org/
[5] http://cassandra.apache.org/
[6] https://www.mongodb.com

Figure 1.1: *Overview of the major data sources contributing to BIG DATA domain.*



highlighting the main data source types which contribute more in the generation of new information. For each source type is reported a) how much the three characteristics velocity, variety and volume affect the generation of data and b) the degree of data structuration of the specific data source. On the figure, for each data source type, it is also reported the typical data location (internal, external and both internal/external) in relation ot organization's firewall, i.e. where the data resides. The internal only resources are characterized by private archives of documents (e.g. scanned documents, statements, insurance forms, etc.) and prevalently structured records in organization

data storages (e.g. SQL, doc repositories, etc.). A lot of structured data is coming also from logs and sensor data, which are characterized by high volume, variety and volume, while other data sources like docs (file in specific formats like pdf, doc, csv, ppt, etc.) and business apps (data from ERP, CRM, HR systems, etc.) present information quite structured but with no high requirements in term of three big data characteristics considered. The most well-known big data sources are the public Web (intended as all the content available in conventional Web sites), the social media (all the information coming from social networks, e.g. Facebook, YouTube, etc.) and the media files (images, audios, videos, etc.). These data sources are characterized by contents from semi-structured to unstructured ones and the information produced requires a lot of efforts in order to be analyzed correctly (generally high values of volume, velocity and variety). In this thesis we have focused on these last 3 data source types, specifically concentrating on textual data in order to scale Natural Language Processing (NLP) techniques, and related Machine Learning (ML) and Information Retrieval (IR) methods, to various big data scenarios. In the following section we are going to describe why the textual part of data is so interesting and challenging for analysis purposes in this context.

1.2 Textual processing: challenges and opportunities

Textual data is a huge part of available unstructured data produced in big data contexts. This is mainly motivated by the explosions of Web social networks in the last decade where the interactions between users using this explicit type of communication are the majority of interactions occurring on these sites. But the text is also used as complementary information to enrich the knowledge of another object. Think for example at posted videos or pictures on Web sites, where a text description can be attached to a published object and the users can make comments about it. Text is everywhere and available for free in enormous quantities. Estimates [205] indicate that structured data covers only 20% of available information, therefore there is a massive hidden potential waiting to be leveraged in information of textual nature. The means to unlock this potential is through the usage of text mining techniques [3, 74, 153]. Text mining (also known as text analytics) is the computational process of deriving high-quality structured information from text. This can be achieved by examining large collections of documents through complex analysis methodologies based on statistical pattern learning or NLP tools. This step allow to transform unstructured text into a series of structured information, which can then be used as input for higher level analysis tasks. Among the main text mining tasks, we could mention text categorization [191], text clustering [4], concept/entity extraction [160], sentiment analysis [169] and text summarization [94].

Analyzing a text in order to catch useful structured information is a very hard task because natural languages have complex structures and rules, different from language to language. In general, correct interpretation or comprehension of a text is easy for a human but extremely difficult for an automated computer program. As reported in other works [33, 107, 142], this is due to several reasons. Usually, a text is full of ambiguous terms and phrases which often strongly relies on the context and background knowledge for defining and conveying meaning (e.g. apple as fruit vs Apple as company). Furthermore, in several contexts (e.g. a social network like Twitter) the used

natural language is enriched by artificial terms like abbreviations, emoticons, hashtags or slang terms which sometime are difficult to be interpreted correctly also by people. In addition to this, the complexity of a language allow people to express a same concept in completely different ways, using phrases presenting dissimilar structures and words (e.g. "AOL merges with Time-Warner" vs. "Time-Warner is bought by AOL"). Sometimes, a text can be expressed using a multilingual approach, e.g. a phrase written in italian can be enriched by terms coming from english dictionary because these terms are now of common use in the spoken and written language. As reported in [166, 196], some multilingual complex scenarios require the development of software systems able to treat both a) separated analyses of textual contents expressed in different languages and b) unified textual representations linking high level concepts and meanings between different languages. In such cases, it is necessary to provide software mechanisms to index and analyze the content in a specific language in specialized processing pipelines[7] and to adopt specific techniques (machine translation, content embeddings, etc.) to reduce the original multilingual problem to a monolingual one.

All these challenges have determined in the recent years an increasing interests by research community in NLP and information retrieval fields towards investigations in new or improved techniques able to overtake these difficulties. Typically, text analysis can be faced using several approaches. The simplest approach is completely statistic and does not take into consideration the syntactic/semantic characteristics of the target language a text is written in. The features vector of a text can be created using vector space models [184] obtained through popular text representation such as bag-of-words, n-grams words or n-grams characters [191]. These last vector representations are characterized by a high number of features and by extreme sparsity. Before being processed by a supervised or unsupervised ML method, vectors need to be reduced in the number of features (e.g. by feature selection or feature extraction methods [191]) in order to keep just the most significant ones from a statistical point of view. This reduction helps the subsequent applied ML methods to better manage the dimension of the problem, while at the same time guarantees a good effectiveness in the learned models. Sometimes, using only a statistical approach, it could be very difficult to effective handle a specific text mining problem. In these circumstances, it is necessary to make use of NLP techniques which relies on complex language analyses (i.e. lexical, grammatical, syntactic and semantic analyses) of the text in the target languages. NLP aim to resolve a lot of challenging tasks with the end-goals to provide more structured info and more insights about context and meaning of a textual content. These tasks include and are not limited to lemmatization, part-of-speech tagging, parsing, named entity recognition, natural language understanding, sentiment analysis and word sense disambiguation [49, 107][8]. In general, to obtain good results with textual analyses we must use both approaches and analyze a lot of data in order to get accurate statistical measures useful for the problem we are trying to solve. Often this process also involves a tedious and expensive task of feature engineering [190, 213] aimed to select the most effective textual features in order to successfully approach a specific problem. Feature engineering issues can be greatly mitigated by adopting deep learning meth-

[7] An example of software using such type of approach is JATECS, a text processing framework developed during this thesis and which will be presented on Chapter 2.

[8] A good overview is also available on Wikipedia at address `https://en.wikipedia.org/wiki/Natural_language_processing`.

ods [59, 189], a new research field based on renewed interest in neural networks with wide and deep architectures which has been applied successfully in the last few years on several application domains [60][9]. Deep learning algorithms work typically as black boxes by learning optimal dense representations of the input data based on specific target loss functions which are dependent of the problem to be solved (e.g. classification, regression, etc). In this way they almost mask the feature engineering problem and are automatically able to select internally the best available input information for problem's solution. Typical deep learning solutions in text processing problems include the usage of embeddings data representations [24, 151] and sophisticated neural network architectures such as convolutional neural networks (CNNs), multilayer recurrent neural networks and sequence-to-sequence models [48, 209, 221].

The ability of a text processing tool to analyze easily and quickly a lot of information is another key point to consider when choosing a text mining software. There are several software packages specialized on "small scale" data mining (e.g. Weka [97], Scikit-learn [170], etc.), generic NLP tasks (e.g. NLTK [28], OpenNLP [159], etc.), fast deep learning algorithms (e.g. Tensorflow [1], Theano [200], Keras [46], Caffe [110], etc.), or "big data" data mining (MlLib, Mahout, etc). A typical text mining problem usually imposes to programmers the choice of two or more of these packages in order to cover all the typical tasks and requirements required in the analytical process. This integration sometime is very difficult to be accomplished and require a great effort from the developers. Unfortunately, on the market there are no open source tools which offer a seamless integration between data mining and nlp, except some which provide limited functionalities in both worlds. And if we consider also big data domains, there are essentially no open source integrated tools available which cover nicely all these three requirements, i.e. providing NLP techniques and data mining algorithms to solve problems over big data domains. The same limitations on integration can also be applied to available deep learning tools. They offer greatly optimized algorithms but their start assumption is to work on already vectorized input data, basically ignoring all the issues related on how this vectorized representation of the data has been obtained.

1.3 Our contribution

Generally, facing a problem involving complex textual analyses over a great amount of data is not an easy task. In the case of big data scenarios, you need to deal with one or more textual big data sources with effective solutions, potentially able to retrieve and analyze a lot of information in a small amount of time. Such solutions are possible only if you approach your problem with a deep analysis of your requirements, providing, for each criticality found, a practical way (e.g. technology, algorithm, etc.) to handle that specific issue and globally an integrated product which "just works". In big data domains, the typical issues involved are related to architectural and technological aspects of designed solutions and integrations of used software tools, and data representation/engineering of textual information. All these things together determine that it is very difficult to define a single strategy which is always effective in processing textual data. Every text processing problem has its own peculiarities and difficulties,

[9] Deep learning algorithms have been used with great success on several data processing problems involving the analysis of images, texts, videos, audios, etc.

therefore every time a custom solution needs to be designed, exploiting different data processing techniques in order to maximize the obtained results.

In this thesis, we focused our attention on three major text processing application scenarios a user can be faced on, and which have very different requirements for effective solutions. These differences therefore impose to developers different choices in architectures, technologies and algorithms adopted in the designed software systems.

1.3.1 Scenario 1: standalone sequential text processing tool

In the first application scenario we want to design and develop specialized text processing tools which are effective for the specific problem handled and which are able to work in standalone mode on a sequential machine, without requiring complex integrations with other text processing related technologies. This context is typically characterized a small team of developers (even one single developer in many cases) which want to prototype/implement the software quickly and run it on a dedicated commodity machine with a possibly limited set of computational resources available. The quantity of data to be analyzed is generally in the small/medium scale and the software should be able to run the code efficiently and to exploit available resources in an optimal way. In this way, a product could be used with success even in particular cases which requires the processing of a quite high amount of data (e.g. in a standard classifier system, the learner works on thousands of training documents, the classifier works on hundred-thousands unlabeled documents). Our thesis's contribution on this application scenario is two-fold. First we focused our attention on the integration of text processing and NLP tools with popular machine learning algorithms used in typical data mining problems. To this purpose, we propose JATECS, a new Java framework optimized for sequential computations which can be used in generic text mining problems to perform quickly and easily a lot of typical tasks involved in these types of software solutions. The library has been designed following object-oriented programming paradigm, making extensively use of classes and interfaces for the provided functionalities. This flexibility can thus be exploited by developers to quickly develop prototypes or working products, and to evaluate their effectiveness on the field. The second contribution in this context is to show how to approach a couple of practical use cases that fit the characteristics of this first application scenario. We provide extensive analyses of such cases and develop the software solutions using the JATECS framework as the main text processing tool.

1.3.2 Scenario 2: big data text processing tool

The second application scenario we deal with is in the big data context, in the specific case when one have to efficiently manage huge amounts of textual data. In this big data context, the developed systems should be able to process information in batch from a consolidated data storage or from data source streams producing their information continuously and requiring near-real time (complex) analyses. This scenario moves the text processing solution into big data space, requiring architectural choices and usage of technologies suitable for this type of application domains. A fundamental aspect is thus the correct integration of standard text mining tools with these technologies, a purpose often very tricky to reach without spending a lot of time in engineering and implementation tasks. Sometime instead, for performance and scalability reasons, this integration need to go more deeply than the simple usage of sequential tools in a highly

parallel or distributed environment. This imposes the rewrite of specific algorithms (e.g. preprocessing or machine learning methods) in order to exploit the native APIs of the used big data tools. Indeed, only in this way we can access to all advantages provided by the used big data processing engine, such as transparent resources management and automatic fault tolerance. The programming models provided by each big data tool is often specific to those tools (e.g. map/reduce in Hadoop [57], RDD in Apache Spark [220], etc.), each presenting its own advantages and limitations, so this is another key point to consider when choosing the underlying engine used to build a text processing product. Our contribution in this scenario is two-fold. From one side we first investigate on how it is possible to scale text processing techniques and machine learning algorithms into big data contexts, approaching this problem both with custom multithread single-machine solutions and distributed ones based on consolidated tools like Apache Spark. We hence designed two possible general purpose data processing solutions covering both multithread and distributed cases, and we have provided a concrete example of how to optimize a popular family of sequential machine learning algorithms (boosting) for its usage in big data contexts. The other scenario's contribution is the design and analysis of a complete software solution covering a specific big data use case (using Twitter data to support emergency responders in crisis management of man-made/natural disasters) using many popular technologies like Apache Spark, Apache Kafka [127] and Elasticsearch [92]. This use case involves the creation of a complex big data software characterized by critical architectural choices and effective integration strategies among the various key components of the system in order to provide the desired functionalities.

1.3.3 Scenario 3: deep learning text processing tool

The third interesting application scenario we cover in this thesis is similar to first one (i.e. developing standalone text processing tools) but with very different requirements. In particular, the amount of data to be processed could potentially be high and the problem that need to be solved can be extremely complex, requiring rich models able to learn complicated correlations in the data. Such learning models are often obtained by using deep learning methodologies, a broader family of ML methods built over neural networks and based on learning data representations[10]. Deep learning networks are composed of multiple computational layers, each one including many nonlinear processing units performing feature extraction and transformation. The layers are organized in cascade, allowing the networks to learn hierarchical concepts with different levels of comprehension of problem's data. Deep learning methods are generally effective only if one can use a lot of data to "train" the network and the used architecture (wide, deep, or wide and deep) is complex enough, i.e. the model has a high number of parameters to tune during learning. On the market there exists several deep learning frameworks optimized to exploit powerful GPUs for computations and which offer a comprehensive set of ready architectures (e.g. feedforward and recurrent neural networks, convolutional neural networks, etc) and unit blocks (e.g. activation functions, dropout, etc.) to develop complex and effective models. Tensorflow (backed by Google) is a strong open source product offering all these characteristics and is currently the most popular choice among developers. For these reasons, in this thesis we adopted it to show how to use

[10] https://en.wikipedia.org/wiki/Feature_learning

the software to approach a specific deep learning use case which involves the extended analysis of a cross-media retrieval problem containing textual and image information.

1.4 Structure of the thesis

This thesis is composed of six main chapters and can be divided into three logical parts, each one corresponding to the application scenarios described in section 1.3. The Scenario 1 is covered by Chapters 2, 3 and 4, with the first chapter describing the software JATECS and the other two concentrating on two specific use cases (Web sites classification and mobile apps classification). In Chapters 5 and 6 we deal with Scenario 2, providing in the first chapter methodologies and tools for integration of text mining techniques into big data domains, and describing in the last chapter an interesting big data use case involving the creation of an articulated software used as monitoring and support tool for natural/man-made disasters management. We finally deal with Scenario 3 into Chapter 7 by providing the overview and analysis of an the interesting problem of cross-media retrieval facing how to correlate textual data with images information. Each of these chapter covers a detailed description of the analyzed problem, the current state of the art and an extended analysis of the proposed solution paired with a comprehensive experimentation and discussion of the results obtained. Following, we give a brief overview of each chapter included in the thesis.

In Chapter 2 we propose JATECS, an open source Java library optimized for sequential processing that supports research on automatic text categorization and other related problems, such as ordinal regression and quantification, which are of special interest in opinion mining applications. The library tries to merge the two worlds of data mining and NLP into a unique tool which offers to developers the possibility to quickly use a lot of popular techniques and algorithms to provide complete text processing software solutions. JATECS covers all typical phases required to be immediately productive, including data ingestion (from multiple input data sources), data preprocessing, data indexing, data learning and learning model evaluation. While presenting the software and its features, we will also show some code examples explaining how developers can use the provided Java API[11] to perform some common text processing tasks.

In Chapter 3 we address the problem of company classification by industry sector, a task usually performed manually but extremely important in finance sector for investment monitoring and planning purposes on local and/or global markets. We thus propose a system which tackles this task by automatically classifying company Web sites in order to assign to a company one or more relevant industrial sectors. In order to built an effective classifier, we extracted relevant features not only from textual content but also from other types of data (e.g. number of outgoing links, URL length, etc.). Moreover, some of features used were obtained by using external resources (e.g. Facebook) which helped to discovered more information about a specific Web site. The developed software can work in two different ways, one working as a standard classifier (autonomously assigning a label to a site) and the other working as a recommender systems (proposing a ranked list of relevant labels for a Web site). We provide an extensive evaluation of our system using a hierarchical dataset containing more than 20,000 company websites and designed expressly for market research purposes.

[11] API means Application Programming Interface. With this term we refer to the exposed library interfaces which can be used to write a program, i.e. the set of available classes and functions/methods invokable by programmers.

In Chapter 4 we propose a system able to classify mobile apps into user-defined classification schemes. The explosion in the number of mobile applications due to massive adoption of smart device such as smartphones or tablets has determined a growing difficulty for the users in locating the applications that meet their needs. This difficulty, besides the huge number of available applications, is due also to the possibility to search the apps on several stores (e.g. Google Play Store, iTunes, third-party Android app stores such as Aptoide or Uptodown, etc.) and to the different taxonomy structures of each store (used by developers to manually classify an application into an appropriate category, e.g. game, health, etc.). The possibility for the user to define its own taxonomy allow him to create[12] a classifier able to automatically assign labels from that taxonomy to new or unseen applications. The software has been tested using a dataset of more than 5,000 apps distributed in 50 classes. In order to extract the features to pass to used classifier, we gathered metadata information from Google Play Store or iTunes of each app contained in the dataset.

In Chapter 5 we focus our attention on big data application scenarios requiring the development of effective and scalable textual processing tools. We analyze the motivations and difficulties currently existing in order to integrate classic text mining and NLP tools with available general purpose big data processing solutions. We thus propose the analyses of two possible text processing application scenarios requiring such type of integrations and focused on big data solutions, but with different assumptions on runtime environments. In the first one, characterized by availability of a limited amount of computational resources, we propose an optimized multithread library called Processfast exposing advanced data processing capabilities and we compare it against some available distributed data processing solutions. We found that Apache Spark, one the big data processing software tested, provides very good performance also in these type of environments and basically it renders useless the effort spent developing customized solutions like Processfast. In the second application scenario, focused on conventional distributed environments, we show how we can exploit Apache Spark a) to design an elegant and effective framework called NLKP4Spark to efficiently index textual contents and b) to enhance the efficiency of a pair of textual classifiers belonging to boosting family by providing optimized distributed versions of the algorithms.

In Chapter 6 we tackle the problem on how to build an effective software system capable of supporting emergency responders (e.g. national Civic Protection agency) during crisis management of natural or man-made disasters. During these disasters a social network like Twitter is a precious informative channel which can provide in near-real time a lot of time-sensitive and actionable information about the event. The main idea of this work is to exploit information coming from this social network to build a software tool able to support emergency agencies in resource allocation and prioritization of available assets (rescuers, vehicles, etc.), in the first phases of the emergency, towards the areas most affected by the disaster, and for successive monitoring purposes. The system is designed to work in a big data environment. In particular, we propose a robust software architecture which uses some of the most popular big data software (e.g. Spark, Kafka, etc.) and it is able to cope effectively with massive input data without scalability/fault tolerance issues. The software supports emergency agencies by

[12] In supervised learning the user must create a labeled dataset in order to build a classification model through the use of some ML algorithm, e.g. support vector machines or boosting.

pointing out relevant tweets containing damage information about people or buildings, and by geo-tagging them. The interesting Twitter data can then be monitored and visualized in near-real time on specialized Web dashboards in order to support the work of rescuers. We studied the problem of damage detection in tweets by comparing the effectiveness of two different text processing approaches: one completely NLP-based and the other based on the usage of word embeddings. In geo-localization task of tweets, we assign instead a location to a tweet by exploiting online semantic annotation tools and collaborative knowledge-bases such as Wikipedia and DBpedia. The effectiveness of the system has been extensively tested on several training datasets related to different Italian natural disasters occurred in the last recent years.

In Chapter 7 we tackle the problem of image search when the query is a short textual description of the image the user is looking for. We developed a cross-media retrieval software able to translate a textual query into a visual space query and thus use this visual representation to search for similar images into a generic input pictures storage (e.g. a classic DBMS). Representing text into a visual space has an immediate double advantages in the creation of cross-retrieval media systems. The first is the independence of learned model (which translate the text into a visual space vector) from the vector representation of images collection used during the search process, therefore an improvement to the learned model has an immediate benefit also on search phase. The second advantage is that the visual feature representation of an image is language independent so it is possible to easily build specialized learning models, each one able to translate queries in a specific language (e.g. one for Italian, one for English, etc.). In this work we thus propose a family of neural network models on increasing complexities, ranging from simple regressor with dense representation to wide and deep architecture using both sparse and dense feature representations. The proposed models translate a textual query into visual vectors projected in the same space of those extracted from state-of-the-art deep CNN architectures (AlexNet, ResNet-152) trained on a pair of big image datasets. We provided an extensive experimentation of our proposed neural network models and a comparison with several other state-of-the-art algorithms.

In Chapter 8 we conclude, reporting a summary of our results and depicting the possible future path of the research in text analytics applied to big data contexts.

CHAPTER 2

JATECS: an open-source JAva TExt Categorization System

ABSTRACT

In this chapter we approach the issues related to the application scenario described in Section 1.3.1 by proposing a new software framework called JATECS *which can help programmers in writing new text processing applications quickly by including ready code templates and specific features used to solve many common text mining use cases.* JATECS *is an open source Java library that supports research on automatic text categorization and other related problems, such as ordinal regression and quantification, which are of special interest in opinion mining applications. It covers all the steps of an experimental activity, from reading the corpus to the evaluation of the experimental results. As* JATECS *is focused on text as the main input data, it provides the user with many text-dedicated tools, e.g.: data readers for many formats, including the most commonly used text corpora and lexical resources, natural language processing tools, multi-language support, methods for feature selection and weighting, the implementation of many machine learning algorithms as well as wrappers for well-known external software (e.g.,* SVM_{light}*) which enable their full control from code.* JATECS *support its expansion by abstracting through interfaces many of the typical tools and procedures used in text processing tasks. The library also provides a number of "template" implementations of typical experimental setups (e.g., train-test, k-fold validation, grid-search optimization, randomized runs) which enable fast realization of experiments just by connecting the templates with data readers, learning algorithms and evaluation measures.*

2.1 Introduction

The JAva TExt Categorization System (JATECS) has been developed in the past years by our research group as the main software tool to perform research on a broad range of automatic text categorization [191] and other text mining problems.

Research on text mining has gained new momentum in the last decade as the explosion of Social Networks (SNs[1]) largely increased the amount of textual data generated on the Web and also accelerated the speed at which it is generated and consumed. This scenario asked for novel methods to effectively and efficiently process such huge and novel streams of information, to sift down the relevant information with respect to a specific information need. Moreover, the textual content generated in SNs is rich of information related to personal opinions and subjective evaluations, which are of great practical and commercial interest. This aspect led to the growth of new disciplines such as Sentiment Analysis and Opinion Mining (SAOM, [141]). SAOM methods transform into a structured form the unstructured opinion-related information expressed by means of natural language in text, enabling the successive application of data mining methods on the structured information. This transformation can be performed by processing text at various levels: processing each document as a single and atomic entity (e.g., classification), performing aggregated analysis on entire collections (e.g., quantification), extracting multiple piece of information for a document (e.g., information extraction).

JATECS mainly consists of a Java library that implements many of the tools needed by a researcher to perform experiments on text analytics problems. It covers all the steps and components that are usually part of a complete text mining pipeline: acquiring input data, processing text by means of NLP/statistical tools, converting it into a vectorial representation, applying optimizations in the vector space (e.g., projections into smaller space through matrix decomposition methods), application of machine learning algorithms (and the optimization of their parameters), and the application of the learned models to new data, evaluating their accuracy. The development of JATECS started with a specific focus on topic-oriented text categorization, but soon expanded toward many other text-categorization related problem, including sentiment analysis, transfer learning, distributional language modeling and quantification.

JATECS is published at `https://github.com/jatecs/jatecs`, and it is released under the GPL v3.0 license[2]. In the following we describe the design concepts that drove the development of JATECS (Section 2.2), the core components (Section 2.3, and how they can be combined to solve different text mining problems (Section 2.4). We conclude comparing JATECS with similar software currently available (Section 2.5).

[1] Here we broadly mean any platform that acts as a large-scale gatherer of user-generated content, thus ranging from Twitter to TripAdvisor, from Facebook to Amazon's user feedback.

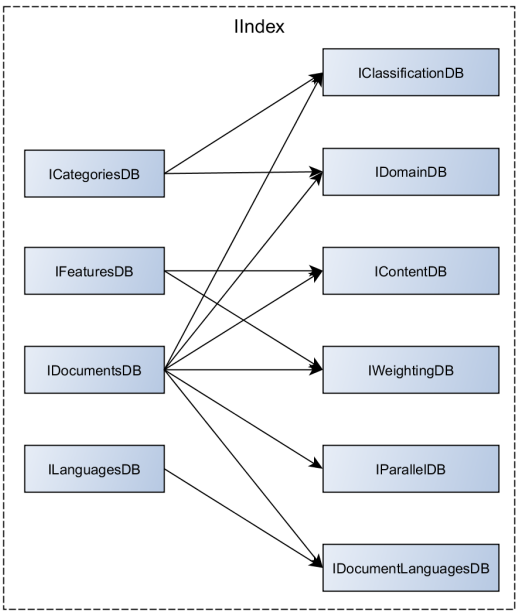
[2] https://www.gnu.org/copyleft/gpl.html

2.2 Design concepts

In an input dataset suitable for text analysis tasks, it is possible to describe its data by using 3 logical concepts and the relations among them. The *documents* are a logical representation of text instances that need to be processed. A single document is composed of a set of *features* where a single feature can be described by some attributes, e.g. a counter of the number occurrences of that feature in a specific document. Moreover, in supervised learning problems, a document can have attached to it a set of *labels* which best describe the content of that document in a custom taxonomy used in that specific problem. Given these three foundational concepts it is possible to combine them together to model the relations between them that allow to answer questions like "which are all the documents that contain the feature x?" or "Which are all the documents that contain feature x and belong to label y?". The relation that pairs documents with features is the one that define the actual content of each document (if seen focusing on a document) and how a feature distributes among documents (if seen focusing on a feature). The relation that pairs documents with labels models the actual classification of documents in the taxonomy of interest. The relation that pairs features with labels models how features distribute with respect to labels, and this information is the key of any supervised processing applied to data, from feature selection to the actual learning.

All these concepts and relation are modeled in JATECS by an API that defines each of them as an object with an interface to query information from it. The design of code in JATECS is corpus-centric: it is possible to trace back any feature, vector, label to the original corresponding entity in the corpus, in any moment of the computation. JATECS core interface is named *IIndex* and it keeps track of all data relations defined on a dataset (see Figure 2.1). The *IIndex* is a data container which gives a unified access

Figure 2.1: *Logical structure of a Jatecs index.*



to views of the above mentioned concepts and relations, plus additional information that is relevant to some text mining problems (e.g., language of each document) or

to some machine learning processes (e.g., feature weighting). An IIndex is linked to three concept DBs (*ICategoriesDB*, *IFeaturesDB* and *IDocumentsDB*) which contain information about the basic concepts of categories (labels), features and documents, and three relation DBs (*IClassificationDB*, *IContentDB* and *IDomainDB*) which keep track of the existing relations among the basic concepts. Additionally, JATECS brings support to multilingual collections, through the concept DB *ILanguagesDB*, and the relation DBs *IParallelDB* and *IDocumentLanguagesDB*. In the following we are giving a more detailed description of each DB.

ICategoriesDB This DB stores information about the set of all categories managed by the index. Each category in the DB is represented by a pair `<cID,categoryLabel>` where `cID` is a category unique ID assigned by the system and `categoryLabel` is the original string label associated with that category.

IFeaturesDB This DB stores information about the set of all features managed by the parent index. Each feature is represented by a pair `<fID,featureText>` where `fID` is a feature unique ID assigned by the system and `featureText` is the original text associated with this feature and extracted from original dataset.

IDocumentsDB This DB stores information about all the available documents managed by the parent index. Each document in the DB is represented by a pair `<dID,docName>` where `dID` is the document unique ID assigned by the system and `docName` is the document logical name linked to external input dataset.

ILanguagesDB This DB stores information about all the available languages managed by the parent index. Each language in the DB is represented by a unique predefined label, e.g., *en*, *it*, or *es*, for English, Italian, or Spanish languages, respectively.

IClassificationDB This sparse DB stores the relations existing in the input dataset between the categories and the documents, i.e. it links together the ICategoriesDB and the IDocumentsDB. The DB is able to handle multilabel contexes, so for each document it is possible to assign any numbers of labels in the form of list of pairs `<dID,cID>`. Here `dID` and `cID` have the usual meaning as described above.

IContentDB This sparse DB stores the relations existing in the input dataset between the documents and the features, i.e. it links together the IDocumentsDB and the IFeaturesDB. The presence of a specific feature (having `fID` identifier) in a given document (having `dID` identifier) is marked in this DB by keeping track the number of occurrences of the feature in the document. If a feature does not appear in a document then the DB will not store any entry for the considered feature and document.

IDomainDB This sparse DB stores the relations existing in the input dataset between the features and the categories, i.e. it links together the IFeaturesDB and the ICategoriesDB. Normally the index has a global representation of the features in all available categories, so each feature is valid in all categories. Instead sometime it is useful to give to each feature a specific valid category context, i.e. decide if a feature (having `fID` identifier) is or not valid in a specific category (having `cID`

identifier) [3]. In this last case, this DB facilitates this task by allowing to keep track of the validity of features inside each specific category.

IWeightingDB This DB is similar to IContentDB because stores the relations between the documents and the features. Differently from IContentDB, it allows to keep track of a weight associated with a feature in a specific document. Usually machine learning algorithms use the feature weights to build the learning model. As we will see in section 2.3.4, JATECS offers several methods to perform the weighting process over the features contained in an index.

IDocumentLanguagesDB This DB links together the IDocumentsDB with the ILanguagesDB, that is, it allows for specifying the language or languages in which a given document is written. This DB might only exist in multilingual indices, and could be obviated for monolingual ones.

IParallelDB This DB allows to store relations of parallelism among documents. Each relation is represented as a tuple `<dID_0,dID_1,...>`, indicating documents with ID `dID_0`, `dID_1`, ... consist of parallel versions about the same content[4].

The IIndex implementation currently available in JATECS is suitable for completely in-memory data processing. During initial data loading of an input dataset, the programmer can specify how to store data for *IClassificationDB* and *IContentDB*, choosing for these DBs a trade-off between RAM memory consumption and efficiency in the access of their data. Anyway, thanks to the extensive use of interfaces and abstract classes, a different IIndex implementation (e.g. disk-based with an effective caching mechanism) could be realized with little effort.
Following is a code example of usage of a JATECS index:

[3] This configuration is also called local representation of the features.
[4] Whether the sense of *parallelism* stands for sentence-level, document-level, or topic-level is an implementation decision.

Listing 2.1: *Example showing how to navigate indexed data with* JATECS

```
IIndex index = .... // Built from some input data source.

// Iterate over all documents.
IIntIterator documents = index.getDocumentDB().getDocuments();
while(documents.hasNext()) {
  int docID = documents.next();
  String documentName = index.getDocumentDB().getDocumentName(docID);

  // Select only the features which have number of occurrences >= 5 or weight > 0.15.
  ArrayList<String> wantedFeatures = new ArrayList<>();
  IIntIterator features = index.getContentDB().getDocumentFeatures(docID);
  while(features.hasNext()) {
    int featID = features.next();
    int frequency = index.getContentDB().getDocumentFeatureFrequency(docID, featID);
    double weight = index.getWeightingDB().getDocumentFeatureWeight(docID, featID);

    if (frequency >= 5 || weight > 0.15)
      wantedFeatures.add(index.getFeatureDB().getFeatureName(featID));
  }

  // Print information about the document.
  System.out.println("**************************");
  System.out.println("Document name: "+documentName);
  System.out.println("Most important features:
      "+Arrays.toString(wantedFeatures.toArray()));
}
```

2.3 Core library

In this section we report on the various components that are implemented in JATECS to support the development of a complete text processing pipeline. On figures 2.2 and 2.3 we show the logical architectures of two typical processes you can perform in JATECS, the workflows of respectively data indexing and data classification. On these figures, the light blue rectangles are logical components which the user of the library can adapt to their custom requirements through the usage of ready-made classes available in JATECS. In the following we provide, for each of these building blocks, a description of which components the library provides to programmers.

2.3.1 Data formats and corpora support

JATECS allows to import text from various data sources and different formats (component *CorpusReader* on Figure 2.2). All the classes related to data input (and output, i.e., saving back a IIndex to a specific format) are defined in the `jatecs.indexing.corpus` namespace. The abstract class `CorpusReader` defines the core methods to access a corpus, i.e., accessing its files, selecting the relevant set of documents (e.g., when there are well-known train/test splits), returning an iterator on its documents and their labels. JATECS provides specializations of the `CorpusReader` to quickly import data in different formats and have it available as a standard IIndex structure. This index can then be used in all subsequent stages of the experimentation and manipulated by the algorithms provided by the library, e.g. to perform a feature selection operation or to build a classifier.

The library implements readers for the following corpora:

Figure 2.2: *Logical architecture of* JATECS *indexing process.*

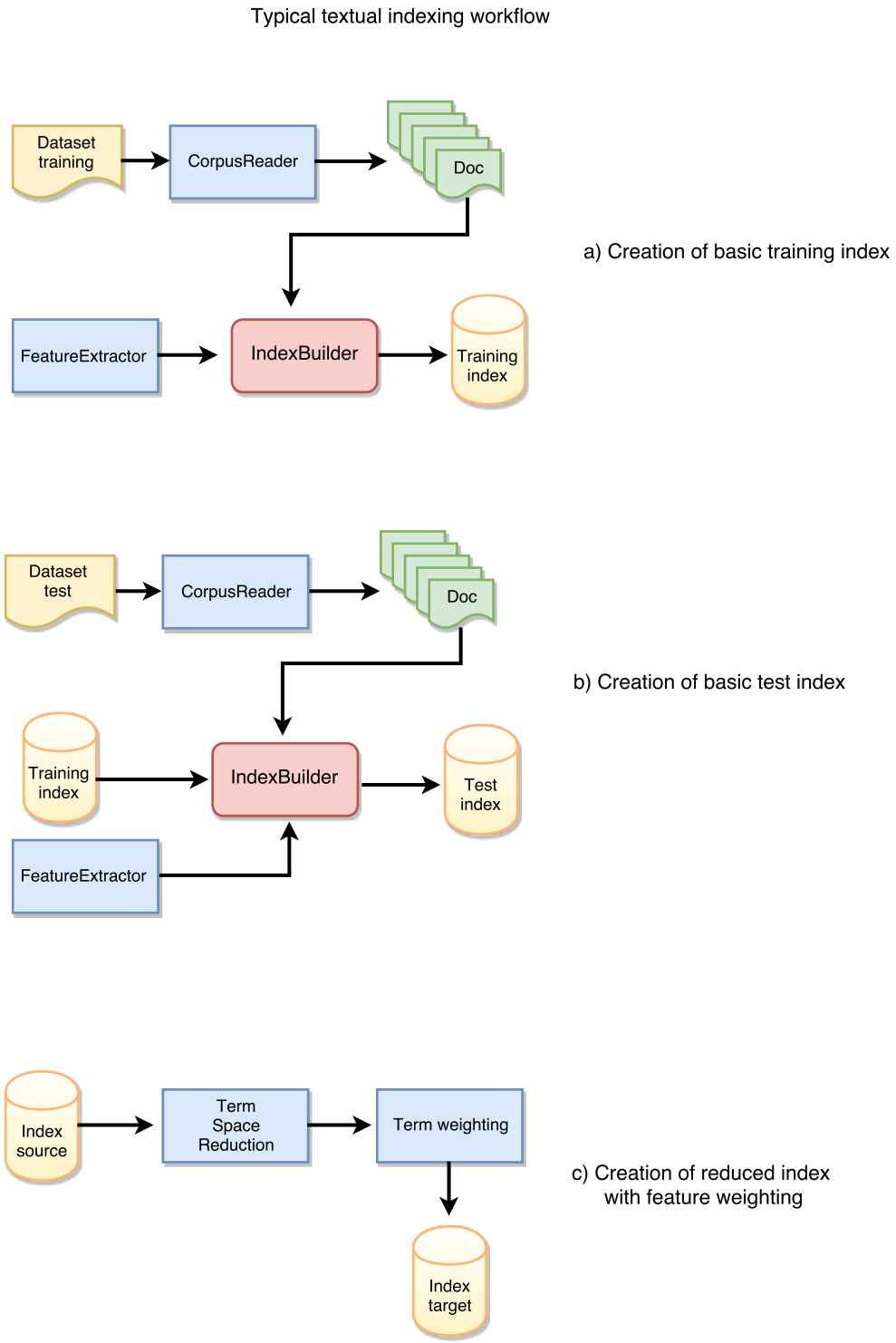
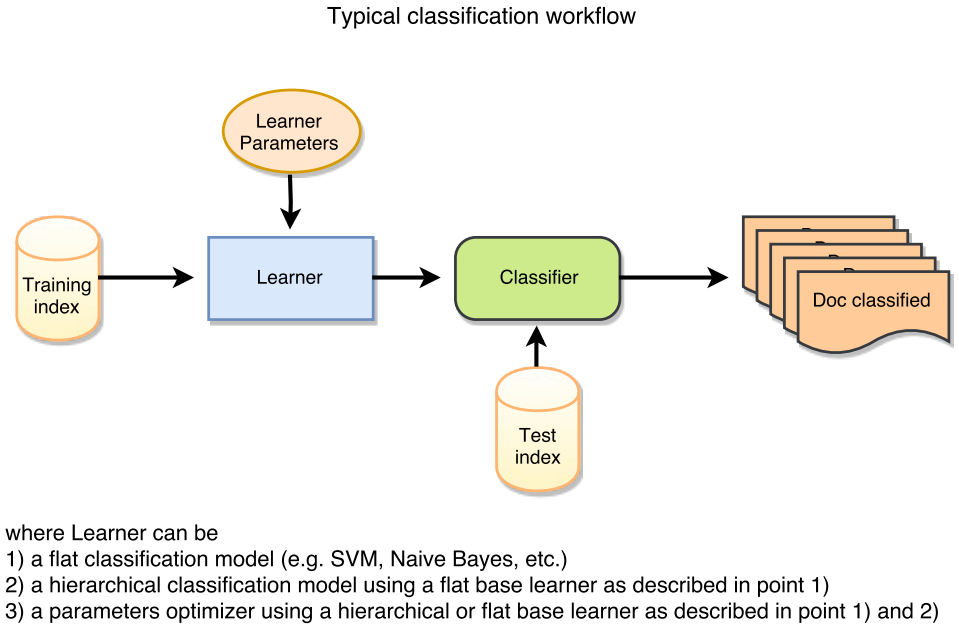


Figure 2.3: *Logical architecture of* JATECS *classification process.*



Reuters21578 [5] Currently it is probably the most widely used test collection for text categorization research. This dataset defines several data splits to be used for experimentation, the most used is the "ModApte" split which has 9,603 documents in training set, 3,299 documents in test set and a taxonomy consisting in 115 categories.

RCV1-v2/RCV2 RCV1-v2 [133] is another popular text categorization benchmark made available by Reuters and consisting of 804,414 news stories produced by Reuters from 20 Aug 1996 to 19 Aug 1997. The library provides access to "LYRL2004" data split, consisting in 23,149 training documents, 781,265 test documents and a taxonomy of 103 categories. RCV2[6] is a multilingual extension corpora containing over 487,000 news stories collected in the same time frame in thirteen different languages other than English (Dutch, French, German, Chinese, Japanese, Russian, Portuguese, Spanish, Latin American Spanish, Italian, Danish, Norwegian, and Swedish). RCV2 is a corpus comparable at topic level.

WipoAlpha [7] It is a large collection published by the World Intellectual Property Organization (WIPO) in 2003. The dataset consists of 75,250 patents classified according to version 8 of the International Patent Classification scheme and a hierarchical taxonomy of more than 69,000 codes.

Oshumed [8] The OHSUMED test collection is a set of 348,566 references from MED-

[5] http://www.daviddlewis.com/resources/testcollections/reuters21578/
[6] http://trec.nist.gov/data/reuters/reuters.html
[7] http://www.wipo.int/classifications/ipc/en/ITsupport/Categorization/dataset/wipo-alpha-readme.html
[8] http://davis.wpi.edu/xmdv/datasets/ohsumed.html

LINE, the on-line medical information database, consisting of titles and/or abstracts from 270 medical journals over a five-year period (1987-1991). The data is classified into a taxonomy of 23 cardiovascular diseases categories.

JRCAcquis JrcAcquis [197] is a parallel corpora available in more than 20 languages containing approximately 20,000 documents per language about legislative texts from EU legislation. The data is classified manually using the Eurovoc thesaurus, which consists of over 6,000 descriptor terms (classes) organized hierarchically into up to eight levels.

20 Newsgroup [9] another very popular collection of approximately 20,000 documents partitioned nearly evenly in 20 different Usenet discussion groups.

Multi-Domain Sentiment Dataset v.2 [10] [30], contains product reviews from Amazon.com for four domains (Books, DVDs, Electronics, and Kitchen appliances) and is commonly used as a benchmark for domain adaptation techniques. The dataset comprises 1000 positive reviews and 1000 negative reviews for each domain, and a set of unlabelled documents ranging from 3,586 to 5,945 per domain.

Webis-CLS-10 [11] [173] is a cross-lingual sentiment collection consisting of Amazon product reviews written in four languages (English, German, French, and Japanese), covering three product domains (Books, DVDs, Music). For each language-domain pair there are 2,000 training documents, 2,000 test documents, and from 9,000 to 50,000 unlabelled documents.

The library also implements readers for the following common formats:

SvmLight/LibSVM SvmLight [111] is a popular software providing Support Vector Machines (SVMs) algorithms for classification and regression problems. The software defines a standard format to manipulate input data and this is used also by LibSVM [39], another popular package used to perform SVM classification. A lot of datasets are already available in this specific data format[12]. JATECS support this format and make it easy to work with such type of data.

CSV This is the classic Comma Separated Values file format. The user can specify the character value to use as a separator and the data can be available in multiple files.

ARFF (Attribute-Relation File Format)[13] is a file format to describe instances of entities (documents, in our case) that share a set of attributes (features, in our case).

2.3.2 Text processing and feature extraction

On Figure 2.2, we now describe the component *FeatureExtractor* which allows to extract textual features from text. All the methods that typically compose the processing pipeline that converts text for strings of characters to vectors in a vectorial space are modeled through interfaces or abstract classes. Each interface/abstract class has then

[9] http://qwone.com/ jason/20Newsgroups/
[10] http://www.cs.jhu.edu/ mdredze/datasets/sentiment/
[11] http://www.uni-weimar.de/en/media/chairs/webis/research/corpora/corpus-webis-cls-10/
[12] https://www.csie.ntu.edu.tw/ cjlin/libsvmtools/datasets/
[13] http://weka.wikispaces.com/ARFF

a number of implementations for the most commonly used methods. For example, the basic activities of the process of feature extraction are implemented in the abstract class `FeatureExtractor`: stop word removal, stemming (supporting various languages through the use of Porter stemmers generated by Snowball [5] software), substitution of xml/html entities with proper characters. The `FeatureExtractor` is then extended to cover various cases:

BOWFeatureExtractor This is a basic extractor that splits of a given text content using not only spaces but also all the common punctuation signs and parenthesis like ",","." , ";", " (", etc.

CharsNGramsFeatureExtractor This extractor generates n-grams of characters (with n specifiable by the user) from a given text content. N-grams can be word-bounded or extracted continuously from the text string, regardless of word tokenization.

NLPFeatureExtractor This extractor use the Stanford POS tagger [145] and sentiment lexica [17, 31, 198] to extract sentiment-relevant features. The extractor marks the presence of terms with known sentiment polarity from sentiment lexica and also generates multi-word feature using POS patterns and negation propagation methods (as described in [16]).

SetFeatureExtractor This is an aggregator that allows to use a set of other extractors to combine the features generated by them, optionally marking such feature in a way that eventual identical features from different extractors are considered as different features (e.g., the word "and" from BOWFeatureExtractor is not confused with the 3-gram "and" from CharsNGramsFeatureExtractor).

Following is a JATECS code excerpt of loading an input data source using a CSV reader and performing feature extraction using the CharsNGramsFeatureExtractor:

```
// Load all labels (categories) of interest.
TroveCategoryDBBuilder categoryDBBuilder = new TroveCategoryDBBuilder();
FileCategoryReader categoriesReader = new FileCategoryReader(categoriesFile,
  categoryDBBuilder);
ICategoryDB categoryDB = categoriesReader.getCategoryDB();

// Prepare a character n-grams feature extractor.
CharsNGramFeatureExtractor extractor = new CharsNGramFeatureExtractor();
extractor.enableStemming(new EnglishPorterStemming());
extractor.enableStopwordRemoval(new EnglishStopword());
extractor.setNGramSize(4);

// Prepare a CSV corpus reader.
CSVCorpusReader corpusReader = new CSVCorpusReader(categoryDB);
corpusReader.setFieldSeparator("\t");
corpusReader.setInputFile(dataFile);
corpusReader.setDocumentSetType(SetType.TRAINING);

// Build the index.
TroveMainIndexBuilder mainIndexBuilder = new TroveMainIndexBuilder(
  categoryDB);
FullIndexConstructor mainIndexConstructor = new FullIndexConstructor(
  corpusReader, mainIndexBuilder);
mainIndexConstructor.setFeatureExtractor(extractor);
mainIndexConstructor.exec();
IIndex index = mainIndexConstructor.index();
```

2.3.3 Dimensionality Reduction

The number of different features resulting from the indexing process depends almost directly on the number of different terms in a corpus. It is typically a very high number with respect to the size of feature sets from data sources other than text, which can be a performance issue for many learning algorithms. Another typical difference from another media data sources, is that features follow a power-law distribution, i.e., few features appears a lot of time and a lot of features appear few times in text. For example, the relatively small set of 9,000 documents of the Reuters21578 dataset are composed of 20,123 distinct words (number excluded), but only 313 of them determine half of the half million word occurrences of that dataset. Features at the extremes of the distribution are typical less informative than those in the center, but it is not obvious which feature can be discarded after the feature extraction process.

Dimensionality Reduction [191] techniques aim at reducing the number of dimensions of the feature space by selecting the most informative features (component *Term Space Reduction* on Figure 2.2). There are mainly two approaches for reducing the dimensionality: Feature Selection and Distributional Semantic Models, and JATECS supports both of them.

Feature Selection (FS) methods attempt to select a reduced subset of informative features from the original set (thus discarding the rest), so that the size of the new subset is much smaller than the original one and so that the reduced set yields high classification effectiveness. In Text Classification (TC) the problem is usually tackled via "filtering" approaches, i.e., methods relying on a mathematical function measuring the contribution of each feature to the classification task. JATECS implements a number of *filtering* approaches for FS (see e.g., [213]), covering most popular Term Space Reduction (TSR) functions, including *Information Gain*, χ^2, *Pointwise Mutual Information*, or *Odds Ration*, among many others; as well as a number of well-known TSR policies, including the *Local* selection for each category, the *max*, *sum*, and *weighted* variants of the *Global* selection for all categories, and the *Round Robin* [78] policy. The following snippet shows a use information gain combined with round robin in JATECS:

```
//applies Round Robin policy to select the 5000 most important features according to
    Information Gain
RoundRobinTSR rrTSR = new RoundRobinTSR(new InformationGain());
rrTSR.setNumberOfBestFeatures(5000);
rrTSR.computeTSR(index);
```

Distributional Semantic Models (DSM) transform the original feature space into a new space of reduced dimensionality where semantic between terms is explicitly modeled thorough the concept of *distance* (between pairs of terms) and *position* in the space. Dimensions in a DSM are no longer associated to one particular term, as in a traditional bag of words model, but to *latent* features. JATECS implements a number of DSM, supporting Latent Semantic Analysis[14] [58] and most relevant Random Projections approaches [123] such as Random Indexing [120, 181], Lightweight Random Indexing [157], or the Achlioptas mapping [2]. The following snippet shows the use of random indexing in JATECS:

[14] As a wrapper of SVDLIBC, see `http://tedlab.mit.edu/~dr/SVDLIBC/`

```
// Use Random Indexing to map the original feature space into a 5000-dimensional one
    distributing 1% of non-zero values for each random index
int dim=5000;
RandomIndexing randomIndexing = new RandomIndexing(index, dim, dim*0.01);
randomIndexing.project();
index = randomIndexing.getLatentTrainindex();
```

2.3.4 Feature Weighting

Feature frequency (as reflected in *IContentDB*) may not suffice as an indicator of the importance of a term to the document content (e.g., Naïve Bayesian learner uses frequencies, SVM work better with a proper weighting).

For this reason JATECS maintains a dedicated data structure, the *IWeightingDB*, to explicitly quantify the relative importance of a term to a document thorough real values. The Feature Weighting functions (component *Term Weighting* on Figure 2.2) are responsible of setting those values, and JATECS implements most widely used weighting criteria, including the well-known *tf-idf* [183] (and many of its variants) and *BM25* [178]. The following snippet exemplifies how to apply the normalized *tf-idf* to bring to bear the terms importance to the documents.

```
// Weight the features using TF-IDF.
IWeighting weightingFunction = new TfNormalizedIdf(index);
index = weightingFunction.computeWeights(index);
// now index has a IWeightingDB with tfidf weights
```

2.3.5 Learning algorithms

JATECS implements many machine learning algorithms for classification, ordinal regression, and clustering. All those algorithms implement shared interfaces that enable the programmer to easily test different learning methods. Both interfaces and algorithms are defined in the `it.cnr.jatecs.classification` namespace. A learning algorithm (module *Learner* on Figure 2.3) must extend the `BaseLearner` abstract class by implementing the method

```
public abstract IClassifier build(IIndex trainingIndex);
```

that, given an IIndex return an IClassifier, which in turns requires the implementation of the methods

```
public ClassificationResult classify(IIndex testIndex, int document);
public ClassificationResult[] classify(IIndex testIndex, short category);
```

i.e., a method to classify a single document in a IIndex with respect to all the possible labels, and a method to classify all the documents in a IIndex with respect to a single label[15].

[15]These two methods can be use interchangeably to obtain a classification of all documents in an IIndex with respect to all possible labels. The choice of which one to use is an optimization parameter that can be set depending on how the used classifier works best. For example, SVMs works better classifying category by category, while AdaBoost works better classifying document by document.

Regarding classification, JATECS covers the most popular learning methods such as the Naïve Bayes [132], Rocchio, Logistic regression, Ridge regression [9], k-NN [212], AdaBoost and boosting methods in general [188], and Support Vector Machines (SVMs). SVMs support includes wrappers for the popular SVM^{light}[16] [111], SVM^{perf} [114], and LibSVM[17] packages. Classifiers committees (ensembles) and bagging methods are also implemented.

For ordinal regression, the library provides wrappers for ϵ-SVR and ν-SVR [47], as well as implementations of graph based methods, such as regression trees and D-DAGs [172]. A highly customizable implementation of k-means is implemented for clustering.

The following snippet offers an example on a basic text classification pipeline in JATECS:

```
IIndex train = .... // Training data.
IIndex test = .... // Test data.

ILearner learner = null;
if(useBayes)
learner = new NaiveBayesLearner();
else
learner = new SVMLearner();
// the rest of the code is independent of the selected learner
IClassifier classifier = learner.build(train);

// Classify test documents and get their predictions.
Classifier classifierModule = new Classifier(test, classifier);
classifierModule.exec();
IClassificationDB predictions = classifierModule.getClassificationDB();
```

2.3.6 Evaluation

JATECS provides a standard set of tools to experimentally measure the effectiveness of a system built using one of the available supervised algorithms. In case of evaluation of a classifier, we need to keep track of all correct/wrong predictions made by it over a test documents set. As explained in [191], when evaluating a multi-label classification system, one can use multiple *contingency tables* (one for each label is interested on) to compute various error measures like *precision*, *recall*, *accuracy*, *f1*, etc. JATECS provides all these measures both at the level of a single label and in a aggregate way in order to *micro/macro* evaluate [191] the effectiveness of the built system as a whole. In the same way, the software is also able to handle hierarchical classification systems by evaluating in a special way all internal classifier nodes acting as non-terminal labels.
Multi-class single-label or binary classifiers can be instead evaluated through the use of a *confusion matrix*, a structure relating all labels decisions and giving the exact repartition of wrong decisions for a specific label among all other labels. Our software offers a specific class to perform this task.
JATECS is also able to evaluate a regression model by providing an evaluator which computes, for each label, a simple difference, in absolute value, between the expected value and value estimated by the regressor.
Here is a small snippet of code showing how to perform classifier evaluation:

[16] http://svmlight.joachims.org/
[17] https://www.csie.ntu.edu.tw/~cjlin/libsvm/

```
IClassificationDB predictions = ... // Predictions made by a classifier.
IClassificationDB goldStandard = ... // Real documents labels.

// Evaluate classifier predictions.
ClassificationComparer comparer = new ClassificationComparer(predictions,
    goldStandard);
ContingencyTableSet tableSet = comparer.evaluate();

// Print evaluation measures.
ContingencyTable table = tableSet.getGlobalContingencyTable();
System.out.println("Global results (micro-averaged evaluation)");
System.out.println("tp = " + table.tp()
+ "\ttn = " + table.tn()
+ "\tfp = " + table.fp()
+ "\tfn = " + table.fn());

String res = String.format("p = %.3f\tr = %.3f\tf1 = %.3f\ta = %.3f",
table.precision(), table.recall(),
table.f1(), table.accuracy());
System.out.println(res);
```

2.4 Applications

2.4.1 Classification

JATECS handles the most common types of classification tasks [191], starting from simple binary classification, and including multi-label classification, multi-class single-label classification, and hierarchical classification. Multi-class, single-label classification is supported by the implementation of the *one-vs-all* [29], *one-vs-one* [29], and D-DAGs [172, 177] methods, on which any binary algorithm can be plugged in.
The JATECS index structure is able to manage taxonomies with a hierarchical structure, and implements a hierarchical classification learner [37, 216], which can use any learning algorithm as the basic learning device. The hierarchical learner supports the customization of the selection policy for negative examples for each node of the hierarchy, implementing the *Siblings*, *All*, *BestGlobal*, and *BestLocal(k)* policies of [71].
Classification on multilingual (comparable or parallel) corpora [85] is supported, including an implementation of the Multilingual Domain Models [89] technique. Optimization of parameters is supported by the implementation of well known exploration/-validation methods, including *K-Fold cross validation* (simple or stratified) [126], *leave-one-out* [126] and *grid-search* [26].
Following is an example of K-Fold cross evaluation using a TreeBoost classifier [68]:

```
// Build base internal learner using MP-Boost.
AdaBoostLearner internalLearner = new AdaBoostLearner();
AdaBoostLearnerCustomizer internalCustomizer = new AdaBoostLearnerCustomizer();
internalCustomizer.setNumIterations(iterations);
internalCustomizer.setWeakLearner(
new MPWeakLearnerMultiThread(threadCount));
internalCustomizer
.setInitialDistributionType(InitialDistributionMatrixType.UNIFORM);
internalLearner.setRuntimeCustomizer(internalCustomizer);

// Build treeboost using the specified internal learner.
TreeBoostLearner learner = new TreeBoostLearner(internalLearner);
TreeBoostLearnerCustomizer customizer = new TreeBoostLearnerCustomizer(
internalCustomizer);
learner.setRuntimeCustomizer(customizer);

// Prepare k-fold evaluator.
AdaBoostClassifierCustomizer internalClassifierCustomizer = new
    AdaBoostClassifierCustomizer();
internalClassifierCustomizer.groupHypothesis(true);
TreeBoostClassifierCustomizer classifierCustomizer = new
    TreeBoostClassifierCustomizer(
internalClassifierCustomizer);
SimpleKFoldEvaluator kFoldEvaluator = new SimpleKFoldEvaluator(learner,
customizer, classifierCustomizer, true);
kFoldEvaluator.setKFoldValue(10);

// Evaluate the TreeBoost learner over the specified training data.
IIndex training = ...
ContingencyTableSet tableSet = kFoldEvaluator.evaluate(training, null);
```

2.4.2 Active learning, training data cleaning, and semi automated text classification

JATECS provides implementations of a rich number of methods proposed for three classes of problems that are interrelated: Active Learning (AL), Training Data Cleaning (TDC), and Semi-Automated Text Classification (SATC).

In AL [102, 135, 201] the learning algorithm can select which documents to add to the training set at each step of an iterative process. The aim is to minimize the amount of human labeling needed to obtain high accuracy. JATECS implements the svm-based AL method proposed in [135]: Max-Margin Uncertainty sampling (MMU), Label Cardinality Inconsistency (LCI).

The TDC task [70, 161] consists of using learning algorithms to discover labeling errors in an already existing training set. In AL such information is not available, and thus TDC can leverage on more information, and different strategies from AL, to identify training labeling errors that once corrected will contribute to improve the performance of the classifier. JATECS implements a number of training data cleaning policies described in [70], e.g., cleaning by confidence, by committee disagreement, by k-nn labeling similarity.

SATC [25, 146] aims at reducing the amount of effort a human should invest while inspecting, and eventually repairing, the outcomes produced by a classifier in order to guarantee a required accuracy level. SATC task differs from both AL and TDC in the fact that the goal is not improving a classifier but the accuracy of a classification of a collection for its use in an external task (e.g., an e-discovery process), where such external task may determine a different importance in correcting difference type of

errors. JATECS implements SATC-oriented methods based on confidence ranking and utility ranking [25].

2.4.3 Transfer learning

Transfer learning [168] aims at enabling machine learning methods learn effective classifiers for a “target" domain when the only available training data belongs to a different “source" domain. This problem could be posed as how to bring closer both data distributions. Distances between domains are typically quantified as the $\mathcal{A}$-*distance* [22], a useful tool for transfer learning research that is incorporated in JATECS. The framework also includes an implementation of the Distributional Correspondence Indexing (DCI) [69] algorithm for cross-domain and cross-lingual learning. The following code example shows how a cross-domain classifier can be learned from a source domain and applied to a different target domain by using sets of unlabeled documents from both domain to define a common latent projection space, built using DCI methods. Note how the portion of code that implements the DCI projection can be removed without affecting the functionality of the rest of the code.

```
//reading indexes
IIndex train_s = ... // train documents from source domain
IIndex test_t = ... // test documents from target domain

// Start of DCI-related code
IIndex unlabel_s = ... // unlabeled documents from source domain
IIndex unlabel_t = ... // unlabeled documents from target domain

// cosine-based DCF
IDistributionalCorrespondenceFunction distModel_source= new CosineDCF(unlabel_s);
IDistributionalCorrespondenceFunction distModel_target= new CosineDCF(unlabel_t);

// other DCF models available:
// LinearDCF, PointwiseMutualInformationDCF, MutualInformationDCF
// PolynomialDCF, GaussianDCF

// projecting indexes into DCF space
DistributionalCorrespondeceIndexing dci = new
    DistributionalCorrespondeceIndexing(train_s,
distModel_source, distModel_target, customizer);
dci.compute();
train_s = dci.getLatentTrainIndex();
test_t = dci.projectTargetIndex(test_t);
// End of DCI-related code

//train learner
IClassifier classifier = Utils.trainSVMlight(train_s, svmconfig);

//classification
IClassificationDB predictions = classification(dci, classifier, test_t);

//evaluation
IClassificationDB trueValues = test_t.getClassificationDB();
Utils.evaluation(predictions, trueValues, resultsPath, targetTestName);
```

2.4.4 Quantification

Quantification [80] is the problem of estimating the distribution of labels in a collection of unlabeled documents, when the distribution in the training set may substantially

differ. Though quantification processes a dataset as a single entity, the classification of single documents is the building block on which many quantification methods are built. JATECS implements the three best performing classification-based quantification methods of [80] as well as their probabilistic versions, as proposed in [21]. The implementation is independent from the underlying classification method, which acts as a plug-in component, as shown in the following sample code:

```
int folds = 50;
IScalingFunction scaling = new LogisticFunction();
// any other learner can be plugged in
ILearner classificationLearner = new SvmLightLearner();

// learns six different quantifiers on training data
QuantificationLearner quantificationLearner = new QuantificationLearner(
folds, classificationLearner, scaling);
QuantifierPool pool = quantificationLearner.learn(train);

// quantifies on test returning the six predictions
Quantification[] quantifications = pool.quantify(test);
// evaluates predictions against true quantifications
QuantificationEvaluation.Report(quantifications,test);
```

2.4.5 Imbalanced Text Classification

The accuracy of many classification algorithms is known to suffer when the data are imbalanced (i.e., when the distribution of the examples across the classes is severely skewed). Many applications of binary text classification are of this type, with the positive examples of the class of interest far outnumbered by the negative examples. Oversampling (i.e., generating synthetic training examples of the minority class) is an often used strategy to counter this problem. JATECS provides a number of SMOTE-based implementations, including the original SMOTE approach [40], Borderline-SMOTE [98], and SMOTE-ENN [19], and the probabilistic topic model called DECOM [41]. JATECS also provides an implementation of the recently proposed Distributional Random Oversampling (DRO – [156]), an oversampling method specifically designed for classifying data (such as text) for which the distributional hypothesis holds.

2.5 Related work

On the market there are several software similar in spirit to JATECS and optimized for data mining and NLP tasks. Some of these are open source, others are commercial products and/or closed source. In this section we will concentrate only on open source software because, being freely available to obtain and use, it is easier to test and evaluate the quality and the features of the software.

Waikato Environment for Knowledge Analysis (Weka) [97] is a very popular Java software suite of ML algorithms useful for generic data mining tasks. It is probably one of the oldest open source data mining tool available in the market (first public Java version released in 1999) and, due to its free availability and the vast amount of available features, has gained a lot of traction in the past years, having being used extensively both in academic and industrial contexts. Weka, differently from JATECS that is focused on textual data, supports several standard data mining tasks on several data types (e.g. numeric, nominal, etc.), by including algorithms for data preprocessing, feature selection, classification, regression, clustering and data visualizations. The

software uses the Attribute-Relation File Format (ARFF) as the default format used to ingest data from input datasets, although as JATECS it supports other popular input data formats like CSV or LibSVM/SVMLight sparse representation. All the algorithms provided by Weka assume that each input data source (dataset) is represented in terms of instances (documents), where each instance is composed of the same number of attributes and each attribute has its own type (e.g. an attribute named "age" has type "numeric"). A dataset can be manipulated through the use of filters, which allow to preprocess the data in some way (e.g. remove frequent/infrequent attributes, discretize numeric attributes, etc.) before passing it forward into the ML pipeline. As JATECS, Weka has been mainly designed with in-memory[18] batch-learning in mind, where all the input data is available before processing algorithms start using it. A minimal support to online learning with streaming data is also available, but the number of provided algorithms is very small and not comparable to variety and quantity of available batch learners.[19] Weka allows an easy access to its underlying functionalities by providing to its users several GUIs usable to quickly perform data mining tasks. These tools allow the user to visually define experiments by selecting data sources and filters to be applied, build supervised or unsupervised models, and graphically evaluate models and results obtained on the specific problem. On the other hand, JATECS does not have an equivalent set of tools, but it relies only on the Java API in order to provide access to all its own features. Another important difference between our software and Weka is that JATECS, while loading data from an input source during indexing phase, build an in-memory compressed index (see the IIndex structure in section 2.2) over the input data allowing a) fast on-the-fly queries on the data; b) low memory consumption by exploiting the high sparseness of the textual data. In particular, the point a) allow to design ML algorithms conveniently and with few compromises in terms of performance, the point b) allow to work with datasets rather big even on a simple desktop machine. Lastly, another important aspect differentiating JATECS from Weka is that our software provides ML algorithms which are text-domain specific and cover more advanced use cases than traditional data mining tasks (e.g. text quantification, crosslingual text classification, etc.).

Scikit-learn [170] is a popular open source Python module integrating a wide range of state-of-the-art ML algorithms for medium-scale supervised and unsupervised problems.[20] The software is heavily used in the Python community because the API is simple and very well documented and the code is fast and optimized.[21] The set of provided functionalities is very similar to Weka, covering all the typical phases of a data mining tool like dataset loading, data transformations, learning, hyper-parameters optimization and models evaluation. Like Weka and differently from JATECS, it is a generic data mining library so direct support for text analytics is very limited. For these purposes, Scikit-learn is often integrated with other Python libraries specialized in NLP tasks, e.g. NLTK library. Another difference from JATECS is that Scikit-learn does not have an indexed representation of its input data for fast preprocessing data or build ML models with an higher level API. Instead it offers to developers a bunch of specialized

[18] All the data available from input dataset is loaded in RAM memory before the processing task starts its own work.

[19] See `http://weka.sourceforge.net/packageMetaData/` for a comprehensive list of ML algorithms available in Weka.

[20] See `http://scikit-learn.org/stable/` for a complete list of what the software offers to developers.

[21] Scikit-learn is built over the Numpy and Scipy libraries. These libraries provide optimized data representation and fast operations on dense/sparse arrays and matrices, together with efficient algorithms for linear algebra.

transformers for data cleaning, dimensionality reduction, etc, that cover a lot of common cases. If required a developer can write custom transformers but the entire process is more complex if compared to the usage of an higher level API like the one provided by JATECS. Finally, as Weka, Scikit-learn covers typical data mining scenarios but it is inadequate to deal with advanced NLP applications like textual transfer learning or text quantification. upport Several libraries available in the market are specialized just on few specific aspects of a typical generic ML process. NLTK [28] and OpenNLP[22] are ML-based toolkits focused on NLP tasks. They provide solutions to most common text analytics problems like tokenization, sentence segmentation, part-of-speech tagging, named entity recognition, and so on, yet they lack a rich set of ML algorithms and tools like those provided by Weka and Scikit-learn. spaCy [195] is another notable Python NLP library focusing only on labelled dependency parsing, named entity recognition and part-of-speech tagging but providing extremely fast implementations (according to several benchmarks, the authors claim that their system is the fastest in the world) and top-class accuracy. JATECS offers some of these NLP capabilities in its core library, anyway the framework design allows, in the case a specific feature is missing, to easily plug in a specific implementation of the wanted algorithm (or a wrapper around the specific tool) in its workflow.

LibSVM [39] and SvmLight [112] are very popular software packages providing in-memory fast implementations of various types of SVMs (Support Vector Machines), suitable for multiclass/binary classification and regressions and supporting several kernel types (linear, polinomial, RBF, etc.). Both these software have been integrated into JATECS by wrapping their respective public interfaces. SHOGUN [194] is another machine learning toolbox focuses on large scale kernel methods, especially on Support Vector Machines. It provides similar algorithms as LibSVM and SvmLight but it offers an extended set of kernels including various recent string kernels like Fisher, TOP, Spectrum, etc. Moreover the software also implements a number of linear methods like Linear Discriminant Analysis (LDA), Linear Programming Machine (LPM), (Kernel) Perceptrons and features algorithms to train Hidden Markov models.

2.6 Conclusions

JATECS is a Java machine learning framework designed to support text mining processes. It is highly modular, with the definition of interfaces for any information processing step of a typical ML pipeline, thus supporting a plug and play style exploration of different algorithms in a point of the pipeline with almost no adaptation required for the other components in the rest of the pipeline.

JATECS also implements most of its interfaces through abstract classes, which provide a typical implementation of the basic functionalities of the interface, reducing the amount of code needed to be written when creating a new implementation of an interface. For example, the loops that apply a classification model to a set of documents, either document-by-document first or category-by-category first, are implemented in the Classifier class, and their rewriting for a new classification algorithm is needed only when none of these two methods of classification fits the way the classification algorithm works, a case we never faced.

[22] `http://opennlp.apache.org`

A rich number of text processing methods are implemented, from tokenization and parsing, to weighting and feature selection. Similarly, a rich number of ML algorithm are implemented or wrapped from state-of-the-art libraries. JATECS provides complete implementations to run the typical tasks of classification and clustering, as well as other tasks that are not support by most of the ML frameworks currently available, such as active learning, training data cleaning, transfer learning, and quantification.

CHAPTER 3

Classifying Websites by Industry Sector: A Study in Feature Design

ABSTRACT

In this chapter we explore a first application type which can be brought back to application scenarios defined in Section 1.3.1 and which make uses of the framework JATECS *described in Chapter 2 for text processing purposes.*
Classifying companies by industry sector is an important task in finance, since it allows investors and research analysts to analyse specific subsectors of local and global markets for investment monitoring and planning purposes. Traditionally this classification activity has been performed manually, by dedicated specialists carrying out in-depth analysis of a company's public profile. However, this is more and more unsuitable in nowadays's globalised markets, in which new companies spring up, old companies cease to exist, and existing companies refocus their efforts to different sectors at an astounding pace. As a result, tools for performing this classification automatically are increasingly needed. We address the problem of classifying companies by industry sector via the automatic classification of their websites, since the latter provide rich information about the nature of the company and market segment it targets. We have built a website classification system using JATECS *as the underlying text processing solution and we tested its accuracy on a dataset of more than 20,000 company websites classified according to a 2-level taxonomy of 216 leaf classes explicitly designed for market research purposes. Our experimental study provides interesting insights as to which types of features are the most useful for this classification task.*

3.1 Introduction

Having companies and activities classified by industry sector is important to research analysts, fund managers, and investment managers, since this allows them to partition the market into homogeneous segments and to monitor and compare industry trends across different sectors and subsectors.

Market trends indicate that stocks within the same industry sector often perform similarly during the same temporal interval; for this reason, it is important for investment managers to know the industry sector in which a given company operates. Additionally, this allows companies to have up-to-date views of who their competitors are, and to carry out competitive intelligence actions.

Classification by industry sector has a long tradition, and several taxonomies for classifying companies by industry sector have been developed and maintained over the years. One important such taxonomy is the ICB (Industry Classification Benchmark)[1], created in 2005 by Dow Jones and the FTSE Group, and used by the NASDAQ, NYSE and other stock markets worldwide. ICB consists of 184 nodes organised into a 4-level taxonomy; an example path (from root to leaf) of such a taxonomy is Financials $\rightarrow$ Insurance $\rightarrow$ Nonlife Insurance $\rightarrow$ Property & Casualty Insurance.

Other such taxonomies are the Global Industry Classification Standard (GICS)[2], a 4-level taxonomy of 256 nodes jointly developed by MSCI Inc. and Standard & Poor's and widely used in the financial sector, and the Thomson Reuters Business Classification (TRBC)[3], a 5-level taxonomy of 1065 nodes operated by Thomson Reuters.

Traditionally, the activity of classifying companies under these taxonomies has been performed manually, by dedicated specialists who perform an in-depth analysis of a company's public profile. One problem with this way of operating is that manual classification work is expensive, especially when carried out by senior employees. Even more importantly, manual classification work is increasingly unsuitable in nowadays's globalised market, in which new companies spring up, old companies cease to exist, and existing companies refocus their efforts to different sectors at an astounding pace. For some applications, manual classification work may also simply be too slow for responding to suddenly arisen business needs. As a result, tools for performing this classification automatically are sorely needed.

We address the problem of classifying companies by industry sector via the automatic classification of their websites. Website classification is different from webpage classification since the decision under which class the website should be classified should be taken based on the nature of the entire website, and not of a webpage alone.

We have built a website classification system based on supervised learning techniques and exploiting algorithms provided by JATECS framework; the system can work both in "hard classification mode", placing each company website into exactly one leaf node in the taxonomy, and in "soft classification mode", returning a ranked list of the k leaf nodes that appear the most plausible for the website. The system thus lends itself to working as a fully autonomous classifier (hard classification), or as assistive tech-

[1] http://www.icbenchmark.com/
[2] http://www.msci.com/products/indexes/sector/gics/
[3] http://thomsonreuters.com/business-classification/

nology to users who then need to take the final classification decision themselves (soft classification).

We have tested the accuracy and efficiency of our system on a dataset of more than 20,000 URLs identifying company websites. The URLs are classified according to a 2-level taxonomy of about 250 leaf classes explicitly devised for market research purposes. The results of our experiments indicate that website classification can be performed at a high accuracy level and with good efficiency. In reporting this experimental study we place particular emphasis on the description of the impact that specific types of features have on classification accuracy. In particular, we study the impact of *endogenous* features (i.e., features extracted from the website itself), and the impact of *exogenous* features (i.e., features extracted from sources external to the website itself, but somehow referring to the website). Our experimental study provides interesting insights as to which types of features are most useful for this classification task.

The rest of the chapter is structured as follows. In Section 3.2 we describe our website classification system, especially focusing on the feature extraction phase. Section 3.3 describes our experimental setup, and Section 3.4 presents and discusses the results of our experiments. Section 3.5 discusses related work, while Section 3.6 concludes, pointing at avenues for future research.

3.2 A System for Website Classification

In this section we describe our website classification system in detail, focusing in particular on the feature extraction phase. The system has mainly developed by using the tools/features provided by the JATECS library presented in Chapter 2. In Figure 3.1 we present the logical architecture of the software. In the following we explore the characteristics of the main submodules included in our system.

3.2.1 Crawling websites

The first step of the process consists in crawling the URLs in our dataset (module *Crawler* in figure 3.1). The crawler we have used "pretends" to be the "googlebot" Google crawler, since some websites provide more information about themselves when the crawler is a search engine crawler, and this additional information may be useful to the classification process. Preliminary experiments that we had run by crawling the URLs in the standard way had shown inferior accuracy.

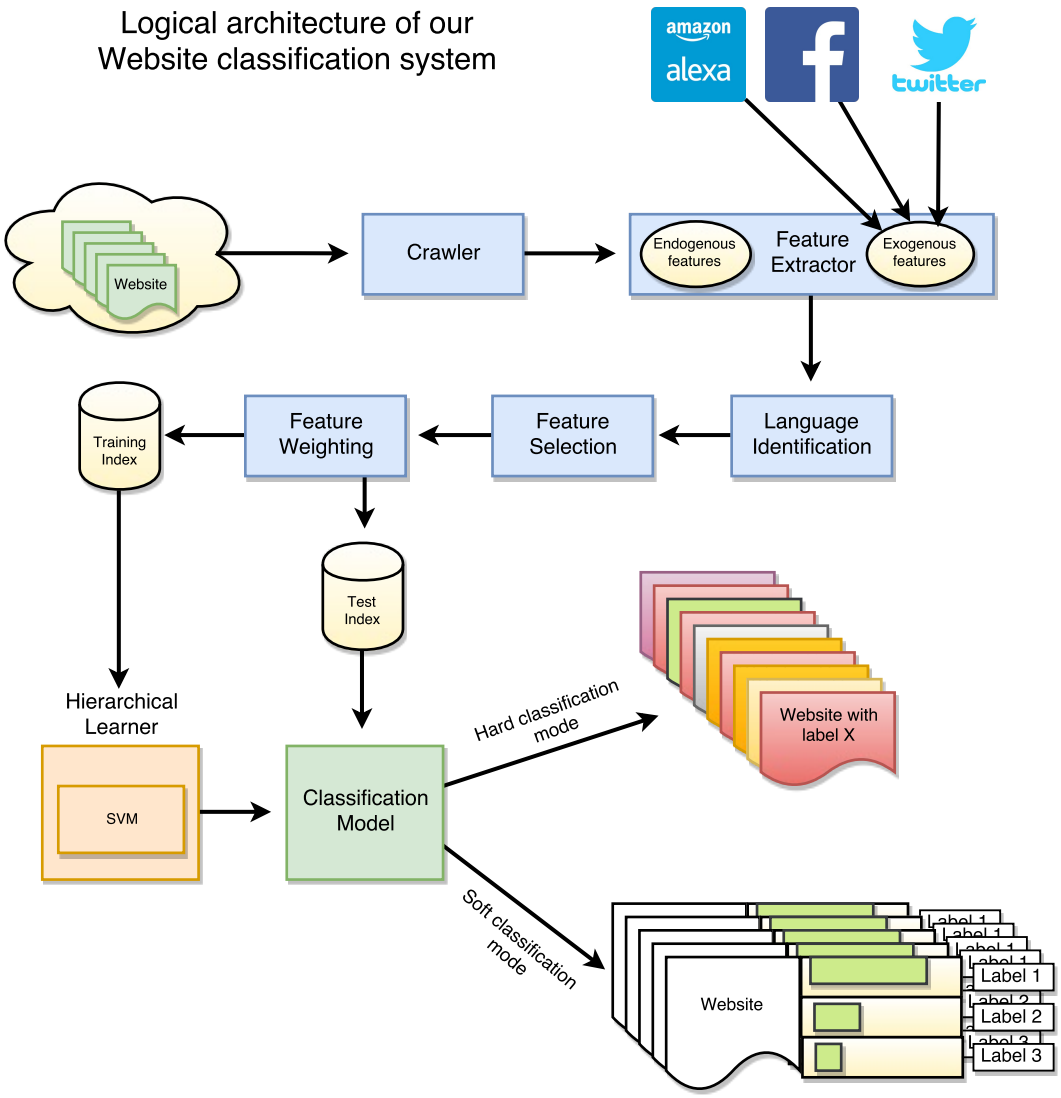
The crawler works in two phases:

1. It retrieves the homepage and the subpages of the websites[4]. In order to keep the computational cost of the modules downstream within reasonable limits, we limit the crawl to the first 10 subpages found for each website, in breadth-first order.

2. It retrieves exogenous information, such as Twitter profiles or Facebook pages, if linked from the homepage, and Alexa queries, if present; see Section 3.2.2 for details.

[4]We only retrieve HTML data, and do not instead download any image / video content, nor perform any Javascript code interpretation. Javascript interpretation could in principle be used by websites to perform AJAX-based dynamic content loading. However this is usually adopted to perform background loading of secondary information (e.g., widgets, social buttons, multimedia content) which is unlikely to contribute to our task, and is rarely used to load the main content of webpages, since this latter content should be loaded as fast as possible.

Figure 3.1: *Design of our Website classification system.*



3.2.2 Feature Extraction

The aim of the module *FeatureExtractor* of Figure 3.1 is to extract the features used during learning and classification phases. These features are extracted according to two different criterion, as described in the following subsections.

Endogenous features

The first type of features we extract are *endogenous* features, i.e., information to be found in the website itself. We extract features from the following fields marked up in the HTML structure:

- **URL**: Given a URL u, after downcasing all its characters we extract all *character n-grams* (for any $n = 3, 4, 5$) and all "words" contained in u, where a character n-gram is any sequence of n characters different from separators (defined as the characters ".", "/", "_") and a "word" is any maximal such sequence, i.e., a sequence of non-separator characters that occurs in u between separators. For instance, from the URL `http://www.google.com/`, the words extracted are `www`, `google`, and `com`, while the 4-grams extracted are `goog`, `oogl`, and `ogle`.

  The rationale is that some of these sequences may be highly indicative of class membership; for instance, the 3-gram `xxx` and the 4-gram `porn` clearly point to the Adult leaf class.

  The URL field is the only field from which we extract character n-grams; the rationale is that other fields, such as Title and Body, do not constrain the user to be as concise as the URL field does, which means that in fields other than URL it makes sense to consider the word as the minimal unit of content.

- **Title**: Given the title of a webpage, we extract all the words contained in it and perform stop word removal and stemming.

- **MetaDescription**: Same as for "Title".

- **MetaKeywords**: Same as for "Title".

- **Body**: Same as for "Title".

- **Headings**: Same as for "Title", and carried out only for tags `<h1>`, `<h2>`, `<h3>`, `<h4>`. The extracted features are pooled together and treated as belonging to generic "Headings" field.

In order to keep the computational cost of the modules downstream within reasonable limits, for each of "MetaDescription", "MetaKeywords", "Body", we limit the content extraction for each webpage to the first 2000 characters.

We also extract a number of "structural" features, such as

- **UrlLength**: Represents the length (number of characters) of the URL of the website home page. This might be useful, e.g., to identify websites (such as `https://bitly.com/`) belonging to class UrlShorteners, since such websites typically have very short URLs.

- **LinksCount**: Represents the maximum (across the pages in the website) number of outgoing links. For instance, a high number of outgoing links might be indicative of class WebDirectories.
- **HeadingsCount**: We compute the maximum (across the pages in the website) numbers of headings of type `<h1>`, `<h2>`, `<h3>`, `<h4>` (these are kept separate and thus give rise to four different subtypes of features).
- **ParagraphsCount**: Represents the maximum (across the pages in the website) number of paragraphs, as identified by the `<p>` tag.

All of these structural features are framed as binary features of the form $c \leq a_i$, where c is the count and a_i is a natural number. For instance, **UrlLength** ≤ 8 is a feature whose value is 1 if the length of the URL of the website home page is smaller or equal than 8, and 0 otherwise. For feature type **UrlLength**, we add one binary feature for each different home page URL length a_i instantiated in the dataset; same for the other three feature types.

Exogenous features

The second type of features we extract are *exogenous* features, i.e., information that is to be found outside the website but that refers to the website anyway. The exogenous features we extract are the following:

- **Alexa**: For any given website we query (the freely accessible version of) Alexa[5], a popular service for Web analytics, and obtain back from it the 5 most frequent queries among the ones addressed to major web search engines that have resulted in clicks on a link to our website. In other words, these queries are the ones which have sent the most traffic to this specific website during the months before we crawled it. The rationale of using this information is that, if we know that a user has issued, e.g., the query "replacement car battery" to Google and has clicked on a link, among the ones returned by Google, leading to our website, we may hypothesize that our website may belong to class CarParts. From these queries we obtain features by extracting all the words contained in them and performing stop word removal and stemming.
- **Facebook and Twitter**: Many companies now have a Facebook or a Twitter page which contains a short textual description of the company and of its mission; these pages are often referred to from within the company's website, typically from the home page. Whenever we find a link to a Facebook or a Twitter page within a website, we extract the univocal name the company adopts in the specific social network and we then exploit the specific web API for retrieving the desired data. We then extract all the words contained in the retrieved page and perform stop word removal and stemming.

  Facebook company pages have several fields, of which we consider the following: `about` (information about the page); `category` (the page's main category, e.g. Product/Service, Computers/Technology); `company_overview` (the company

[5] `http://www.alexa.com/`

overview); `description` (the description of the page); `general_info` (general information provided by the page); `mission` (the company mission); `name` (the name of the page); `products` (the products of this company). Twitter company pages only have `name` and `description` fields.

We qualify all the features we extract, endogenous and exogenous, by a prefix that indicates which field the feature comes from, with the goal of placing different emphasis on features coming from different fields (e.g., a word may have more importance if it was extracted from the page title). For example, the word `google` extracted from the URL `http://www.google.com/` is represented as `URL:google`, while when the same word is encountered in the title of the webpage it is represented as `TITLE:google`.

3.2.3 Language Identification

We analyse each webpage in the website, and each page downloaded from Facebook or Twitter, in order to classify it according to the language the page is worded in (module *LanguageIdentification* in the Figure 3.1). We detect language using the Language Detection library[6]. This library implements a naïve Bayesian learner trained on a corpus of Wikipedia pages in different languages; the learner exploits the character n-grams extracted from the texts as features, and builds a single-label multi-class classifier for the languages in the training set. For the purposes of this experimentation we discard from consideration all websites none of whose pages are deemed to be in English by this language detector. For all other websites, only pages deemed to be in English are retained.

3.2.4 Feature Selection

The feature extraction process described in Section 3.2.2 returns a number of features too high to be used in practice. In order to keep the computational effort to a manageable level, and in order to avoid overfitting, we perform *feature selection* (FS – see e.g., [96, 143]) as depicted by module *FeatureSelection* of Figure 3.1. This latter consists in identifying a subset $S \subset T$ of the original feature set T (which coincides with the set of features that occur in at least one training instance) such that $|S| \ll |T|$ ($\xi = |S|/|T|$ being called the *reduction factor*) and such that S reaches the best compromise between (a) the efficiency of the learning process and of the classifiers (which is, of course, inversely proportional to $|S|$), and (b) the effectiveness of the resulting classifiers. We did feature selection via the usual "filtering" approach [116], which consists in (a) scoring each feature t_k according to its estimated contribution to discriminating class c_j from the other classes, and (b) retaining only the highest-scoring ones. As the scoring function we use *information gain* (also known as *mutual information*), defined as

$$\begin{aligned} IG(t_k, c_j) &= H(c_j) - H(c_j|t_k) = \\ &= \sum_{c \in \{c_j, \bar{c}_j\}} \sum_{t \in \{t_k, \bar{t}_k\}} P(t,c) \log_2 \frac{P(t,c)}{P(t)P(c)} \end{aligned}$$

[6] `http://code.google.com/p/language-detection/`

Here $H(\cdot)$ indicates entropy, $H(\cdot|\cdot)$ indicates conditional entropy, and probabilities are evaluated on an event space of training instances, where $P(t_k)$ and $P(\bar{t}_k)$ represent the fractions of instances that contain feature t_k and do not contain feature t_k, respectively, and $P(c_j)$ and $P(\bar{c}_j)$ represent the fractions of instances that belong to class c_j and do not belong to class c_j, respectively. $IG(t_k, c_j)$ measures the reduction in the entropy of c_j obtained as a result of observing t_k, i.e., measures the information that t_k provides on c_j.

We then adopt a "round robin" policy [79] in which the n (internal and leaf) classes take turns in picking a feature from the top-most elements of their class-specific orderings, until the desired number x of features are picked. In this way, for each class c_j the final set of selected features contains at least[7] the $\frac{x}{n}$ features that are best at discriminating c_j, which means that all the classes are adequately championed in the final feature set.

In the experiments reported in this chapter we retain 30% of the original features, bringing the total number of 1,325,645 features to a slightly more manageable number of 397,693 features.

3.2.5 Feature Weighting

Through the module *FeatureWeighting* of Figure 3.1, all the selected features are weighted by means of the well known BM25 weighting function (see e.g., [179]); initial experiments carried out on a development set and using standard $tf * idf$ had returned inferior results. In BM25, the weight $w(t, d)$ of feature t for website d is defined by

$$w(t,d) = \frac{tf(t,d) \cdot (k_1 + 1)}{tf(t,d) + k_1 \cdot (1 - b + b \cdot \frac{|d|}{avgdl})} \cdot idf(t) \tag{3.1}$$

where $tf(t, d)$ is the number of occurrences of feature t in website d, $avgdl$ is the average number of non-null features (endogenous and exogenous) for a website, k_1 and b are parameters, and $idf(t)$ is

$$idf(t) = log \frac{(N - |\{d : t \in d\}|) + 0.5}{|\{d : t \in d\}| + 0.5} \tag{3.2}$$

where N is the total number of websites in the collection.

In our experiments we have set k_1 and b to the values suggested in [179], namely, $k_1 = 1.2$ and $b = 0.5$.

3.2.6 Classifier Training

As the learning algorithm we have used the TREESVM configuration described in [72] (see module *HierarchicalLearner* in Figure 3.1). Essentially, this consists of running a binary SVM learner[8] at each nonroot node x in the hierarchy, using as negative examples for x all examples belonging to the parent node of x that belong to some sibling of x and not to x itself. As the binary SVM-based learner we have used the freely

[7] The "at least" here means that, since the same feature may be a high-scoring feature for more than one class, strictly more than $\frac{x}{n}$ features per class may eventually get selected.

[8] In preliminary experiments we had tried to use MPBOOST [66] as the base learner, with inferior results.

available `libsvm` software package[9]. We have used a linear kernel with the parameter `C` (the penalty parameter of the error term) set to 1; this parameter setting, while simple, achieves a good trade-off between model complexity and effectiveness, and is the default setting used in the `libsvm` software.

3.2.7 Classification

At the end of the learning process, TREESVM has generated a hierarchical classifier in the form of a tree of binary classifiers, one for each nonroot node (module *ClassificationModel* in Figure 3.1). The classifiers work in "Pachinko machine" style, i.e., only the instances that have been attributed to class x by the associated binary classifier are fed to the binary classifiers associated with the children of x. For a given instance a binary classifier returns both a binary decision ("assign" or "not assign") and a classification score, where high score means high probability of belonging to the class. In "hard classification" mode the highest-scoring leaf class is assigned to the instance, while in "soft classification" mode the k highest-scoring leaf classes are ranked in decreasing order of their score and associated to the instance.

We have modified the original behaviour of TREESVM to better handle the simultaneous presence of the hard and soft classification modes. The original TREESVM is designed to work as a multi-label classifier, so that zero, one, or several leaf classes at the same time can be attributed to an instance. This causes problems in our application context, since the original TREESVM (a) does not guarantee that the instance is attributed to at least one leaf class, thus making hard classification problematic, and (b) does not even guarantee that classification scores for at least k leaf classes are generated for an instance[10], thus making soft classification problematic.

To overcome these two problems, when classifying an instance we heuristically force, at each level in the hierarchy, the assignment of at least three internal nodes. The instance can thus be fed to all the children of these internal nodes, and this tends to guarantee that a large enough number of leaf classifiers are invoked and thus return a classification score for the instance. When in hard classification mode, the top-scoring class is selected; when in soft classification mode, a ranked list of the top-scoring k classes is returned (in our experiments we take $k = 5$).

3.3 Experiments

For our experiments we have used a classification scheme and a dataset of URLs that were provided to us by a customer. Note that we had no control on the design of the classification scheme and on the choice of the dataset; we thus take both as given[11].

3.3.1 The Classification Scheme

The classification scheme consists of a taxonomy of industry sectors of depth 2, with 27 internal (depth-1) classes and 248 (depth-2) classes. All leaf classes are at depth 2, and all depth-1 classes have at least two children classes. The taxonomy is severely

[9] `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`

[10] To see this, assume that the taxonomy has two levels, that the instance is rejected by all level-1 nodes but one, and that this one node has only $(k-1)$ leaf classes as its children; in this case, only for these $(k-1)$ leaf classes a classification score is generated.

[11] Unfortunately the data and the classification scheme are property of this customer, and we are not in a position to make them publicly available.

imbalanced, with internal nodes having from a minimum of 2 to a maximum of 21 children nodes[12].

One aspect that makes classification under this taxonomy difficult is that the taxonomy mixes issues of topic and genre[13]. For instance, the leaf classes ShoppingSearch and ShoppingBlogs have two different parent classes (WebPortals&Search and Blogging, respectively); some distinctions at level 1 are based on topic (e.g., Automotive vs. ComputersAndInternet), while some others are based on genre (e.g., Blogging vs. NewsAndMedia). Classification within this taxonomy is thus very challenging, since it is sometimes the case that different nodes are very similar in meaning, making the task of telling them apart extremely difficult.

3.3.2 The Dataset

Our dataset of URLs originally contained 69,100 URLs, each labelled with exactly one leaf class from our taxonomy. After the crawling phase we discarded from consideration the 46,893 websites that proved unreachable. Most of the errors returned by unreachable hosts were due to bad HTTP status (e.g., "404" errors) or unknown IP address[14]. We further discarded the 2,169 websites that our language recognition module deemed void of English-language content. This left us with (69,100-(46,893+2,169)) = 20,038 websites to experiment with. Of the original 248 leaf classes in the taxonomy we discarded the 32 that proved empty (i.e., none of the 20,038 websites was classified under them), thus leaving 216 leaf classes to work with.

3.3.3 Evaluation measures

As mentioned in previous sections, our website classification system can work both in "hard classification mode", placing each company website into exactly one leaf class in the taxonomy, and in "soft classification mode", returning a ranked list of the k leaf classes that appear the most plausible for the website. We thus evaluate our system according to two different evaluation measures, each suitable for a different mode.

For evaluating the effectiveness of the system at hard classification we have used the well-known F_1 function, defined as

$$F_1 = \frac{2\pi\rho}{\pi+\rho} = \frac{2TP}{2TP+FP+FN} \tag{3.3}$$

F_1 corresponds to the harmonic mean of *precision* (π) and *recall* (ρ); here, TP, FP, and FN, stand for the numbers of true positives, false positives, and false negatives, respectively. Note that F_1 is undefined when $TP = FP = FN = 0$; in this case, consistently with most other works in the literature, we take F_1 to equal 1.0, since the classifier has correctly classified all instances (as negative examples). It is customary to average the class-specific F_1 scores across the classes by computing both *microaveraged* F_1 (denoted by F_1^{μ}) and *macroaveraged* F_1 (F_1^{M}). F_1^{μ} is obtained by (i) computing the class-specific values TP_i, (ii) obtaining TP as the sum of the TP_i's (same for FP

[12] For reasons internal to their organization, our customer decided not to avail themselves of the standard taxonomies discusses in the introduction, and thus developed their own custom taxonomy.

[13] Classification by genre, especially in the web domain, is sometimes called "functional classification"; see e.g., [174].

[14] The dataset contains a considerable number (20,911) of URLs directed to Google static content, i.e., `gstatic.com`, which turned out to be unreachable during the crawling.

and FN), and then (iii) applying Equation 3.3. F_1^M is obtained by first computing the F_1 values specific to the individual classes, and then averaging them across the c_j's. Both averages are computed across *all* classes, internal and leaf alike.

For evaluating the effectiveness of the system at soft classification we instead use a variant of *reciprocal rank* (RR). The RR of an instance d_i is defined as

$$RR(d_i) = \frac{1}{r(d_i)} \tag{3.4}$$

where $r(d_i)$ is the rank attributed by the system to d_i's true class. We use a variant of RR that we call RR_k, defined as

$$RR_k(d_i) = \begin{cases} \frac{1}{r(d_i)} & \text{if } r(d_i) \leq k \\ 0 & \text{if } r(d_i) > k \end{cases} \tag{3.5}$$

That is, no credit at all is given to the system for placing d_i's true class at rank $(k+1)$ or higher, since we only return the top k classes for each website.

Again, we compute both microaveraged and macroaveraged versions of RR_k, noted RR_k^μ and RR_k^M, respectively. RR_k^μ is obtained as the average of $RR_k(d_i)$ across all the test instances, while RR_k^M is obtained by first computing the class-specific averages of RR_k and then averaging the results across the classes. Again, both averages are computed across *all* classes, internal and leaf alike.

3.4 Results

The results of our experiments are reported in Table 3.1; all results were obtained by 10-fold cross-validation (10FCV).

The first observation that can be made by looking at the microaveraged figures (F_1^μ and RR_k^μ) reported in the table is that the accuracy is surprisingly high. For instance, the system equipped with all feature types at the 30% reduction level (last row, columns 7-11) obtained $F_1^\mu = 0.808$ and $RR_k^\mu = 0.852$, which are very high values. Just to put these values into context, the $RR_k^\mu = 0.852$ value resulted from the true class of the website being ranked in 1st place 80.8% of the cases, 5.9% of the cases in 2nd place, 2.9% in 3rd place, 1.4% in 4th place, 0.9% in 5th place, and in only 8.1% of the cases it was ranked outside the top 5 positions. This is a very good result especially in the light of the fact that there are no less than 216 classes to pick from, and picking the single one that applies to the website certainly qualifies as finding a needle in a haystack.

Figures for macroaveraged measures (F_1^M and RR_k^M) are lower, but this was to be expected in the light of the severely imbalanced nature of the dataset. In fact, the micro- and macro-averaged versions of a measure yield the same value only when the dataset is perfectly balanced, while the former yields higher values than the latter when the dataset is imbalanced. In our case, the most frequent class (BloggingServices) totals 3347 instances while the least frequent one (AccountancyAndTaxServices) has only 1 instance, which is an indication of the severe level of imbalance of this dataset.

3.4.1 Effects of Different Feature Types

One of the key aspects in designing a website classification system is feature design, i.e., (a) deciding what type of features to extract, and (b) which among the extracted

Table 3.1: *Web Site classification - Accuracy of classifiers obtained by using all extracted features (Columns 2 to 6) or by using selected features only (Columns 7 to 11). In Column 7, percentages indicate the survival rate of the specific feature type. Boldface indicates best results for a specific combination of feature selection level (no selection or 30% selection) and evaluation measure.*

	No Feature Selection					With Feature Selection					
	#Features	F_1^M	F_1^μ	RR_5^M	RR_5^μ	#Features		F_1^M	F_1^μ	RR_5^M	RR_5^μ
URL	670,340	0.497	0.797	0.532	0.843	211,250	(31.5%)	0.491	0.797	0.534	0.842
Title	38,801	0.367	0.639	0.420	0.705	12,076	(31.1%)	0.331	0.621	0.390	0.694
Body	319,355	0.329	0.604	0.370	0.670	78,913	(24.7%)	0.328	0.618	0.373	0.686
Headings	153,552	0.319	0.597	0.364	0.653	46,858	(30.5%)	0.320	0.606	0.361	0.660
MetaDescription	42,897	0.299	0.492	0.321	0.542	15,297	(35.7%)	0.291	0.505	0.316	0.553
MetaKeywords	32,139	0.201	0.297	0.232	0.341	11,464	(35.7%)	0.200	0.322	0.229	0.371
LinksCount	96	0.019	0.161	0.034	0.215	96	(100.0%)	0.019	0.161	0.034	0.215
ParagraphsCount	39	0.014	0.171	0.028	0.198	39	(100.0%)	0.014	0.171	0.028	0.198
HeadingsCount	39	0.012	0.141	0.029	0.185	39	(100.0%)	0.012	0.141	0.029	0.185
UrlLength	12	0.002	0.076	0.009	0.088	12	(100.0%)	0.002	0.076	0.009	0.088
All endogenous feature types	1,257,270	0.469	**0.791**	0.517	**0.841**	376,044	(29.9%)	0.472	0.797	0.523	0.844
Alexa	36,061	**0.552**	0.754	**0.567**	0.804	9,309	(25.8%)	**0.522**	0.696	**0.548**	0.773
Facebook	24,164	0.201	0.204	0.196	0.241	9,689	(40.1%)	0.203	0.254	0.197	0.300
Twitter	8,150	0.181	0.112	0.174	0.131	2,641	(32.4%)	0.175	0.156	0.172	0.197
All exogenous feature types	68,375	0.527	0.726	0.554	0.792	21,639	(31.6%)	0.499	0.679	0.540	0.764
All feature types	1,325,645	—	—	—	—	397,693	(30.0%)	0.490	**0.808**	0.539	**0.852**

features to retain. The reason is that the size of the full set of features extracted via a process like the one of Section 3.2.2 may be daunting, even for datasets of modest size. For the dataset that we use in the present work, no less than 1,325,645 features are generated from the 20,038 websites the dataset consists of.

In an attempt to better understand the quality of the features contributed from each of the fields identified in Section 3.2.2, for each such field we have run experiments by using the features extracted from that field only; the results are reported in Table 3.1, each row representing a specific feature type. While these experiments do not account for the subtle interactions that may take place as a result of the co-presence of features of different types, they nonetheless give an indicative idea of the relative contribution that the different types of features can give to the overall classification task. We have run such experiments (a) with the full feature set (Columns 2-6 of Table 3.1), and (b) with the set resulting from the feature selection process (Columns 7-11 of Table 3.1). This latter process consisted in putting into a common pool all the extracted features of all types, and then selecting the best ones irrespective of their type, so that different feature types compete with each other. In Column 7 we report, for each feature type, the *survival rate* of a given feature type, i.e., the percentage of features of that type that made it into the final selected set.

The first observation we can make is that, as it could be expected, the "All feature types" setting is overall the best, i.e., there is no single feature type, or group of feature types, that outperforms it; while the "Alexa" setting slightly outperforms it for macro measures, it is substantially outperformed by "All feature types" for micro measures.

A second observation is that some individual endogenous feature types (e.g., "URL", "Title", "Body") deliver very good accuracy by themselves. Using URL features alone is actually competitive with using *all* feature types, which is somehow surprising, given that URLs are hardly a convenient means for authors to convey semantics. The success of this feature type has likely to do with the need, on the part of authors, to cram as much information as possible in a short string that must be both descriptive and evocative to prospective customers. "Title" features also perform well, and this is intuitive, since authors tend to use highly significant words in page titles; the same argument applies, although to a slightly smaller degree, to "Headings" features. Also the fact that "Body" features are helpful is unsurprising, since the body of the page is where content is meant to be conveyed.

"MetaDescription" and "MetaKeywords" features deliver smaller accuracy levels than the previously discussed types. We believe the main reason for this to be the fact that, for many webpages, these fields are left completely empty by their authors. This means that, if all pages of the website have these fields empty, the website is classified completely at random by the "MetaDescription" and "MetaKeywords" classifiers. As for the remaining endogenous feature types ("LinksCount", "UrlLength", "HeadingsCount", "ParagraphsCount"), each of them in isolation delivers very little accuracy, but this is obviously due to the fact that these feature types only contain a few dozen features each.

As for the exogenous features, the "Alexa" features deliver extremely high performance, even surpassing (as already noted above) "All feature types" when it comes to macro measures. The fact that these features excel for macro measures means that they particularly excel for infrequent classes; this is intuitive, since queries tend to be highly descriptive of the content of websites that they send clicks to, and the information they convey can thus make up for the scarcity of endogenous information, a problem that especially affects infrequent classes.

The "Facebook" and "Twitter" features are much less helpful than the "Alexa" features. However, this need not mean that pages from social media are less informative than queries. Indeed, one of the problems here is that, while "Alexa" queries are available for most websites in the dataset (18,478 websites), only 4,370 websites in the dataset have a corresponding Facebook page, and only 4,237 have a Twitter page. This means that, similarly to what happens for the "MetaDescription" and "MetaKeywords" classifiers, the "Facebook" and "Twitter" classifiers classify websites that do not have a Facebook and Twitter page completely at random. So, the percentage of websites that have a Facebook and Twitter page is the *de facto* upper bound to the percentage of websites that get correctly classified by the "Facebook" and "Twitter" classifiers.

3.4.2 Effects of Feature Selection

As for feature selection, the results indicate that it is beneficial when applied to endogenous features (where small improvements are observed for all four measures), but the contrary is true for exogenous features, where deteriorations are observed for all four measures as a result of feature selection. This is likely due to the fact that endogenous features are almost 20 times as many as exogenous features, which means that the former are potentially responsible for overfitting much more than the latter are, and are thus much more in need of trimming.

Some feature-specific classifiers tend to lose accuracy as a result of feature selection. One of them is "Title"; the likely reason for this is that, since titles consist of few words only, if all the words contained in the title of a website have been filtered out by the feature selection process, this website will be classified at random by the "Title" classifier. Also the "Alexa" classifier loses accuracy as a result of feature selection, which is yet another confirmation of how informative queries are, especially for infrequent classes.

Other feature-specific classifiers instead benefit from feature selection. One such example is "Body"; this confirms well-known results from text classification [214], which had shown that bag-of-words classifiers (i.e., classifiers using as features the words in the body of the text) can benefit from moderately aggressive feature selection. Even more interestingly, two other such examples are "Facebook" and "Twitter"; their case is similar to "Body", in the sense that company Facebook and Twitter pages consist of free text, which obviously contains many non-discriminative words too.

As for "LinksCount", "UrlLength", "HeadingsCount", and "ParagraphsCount" features, none of them is filtered out by the selection process (they obtain 100% survival rate), which means that, even if they are very few, all of them are important for the learning process.

3.4.3 Learning curve

We have run additional experiments in order to plot a learning curve for our classifiers, with the goal of finding out if we are likely to obtain improvements should we provide additional training data. To run these tests, in each of the 10 "folds" of the 10FCV experiment the test set is retained in its entirety while only a randomly selected percentage p of the training set is retained. We have run tests with $p \in \{20\%, 40\%, 60\%, 80\%\}$, in "All feature types" mode; if the results obtained with $p = 100\%$ (which we have reported in the preceding sections) are higher than the ones for $p = 80\%$, this means that we probably have not yet reached an effectiveness plateau, and that there are still likely margins of improvement.

The learning curve displayed in Figure 3.2 (see also Table 3.2) indeed shows that effectiveness is still on the rise, both for the hard classification (F_1) and the soft classification (RR) modes. This indicates that the results discussed in the previous sections, while already good in themselves, could likely still be improved upon if more training data were available.

	F_1^M	F_1^μ	RR_5^M	RR_5^μ
20%	0.168	0.635	0.211	0.701
40%	0.270	0.706	0.303	0.763
60%	0.365	0.751	0.389	0.802
80%	0.423	0.781	0.454	0.828
100%	0.490	0.808	0.539	0.852

Table 3.2: *Web Site classification - Accuracy of classifiers (using all feature types) obtained by using increasing percentages of training data.*

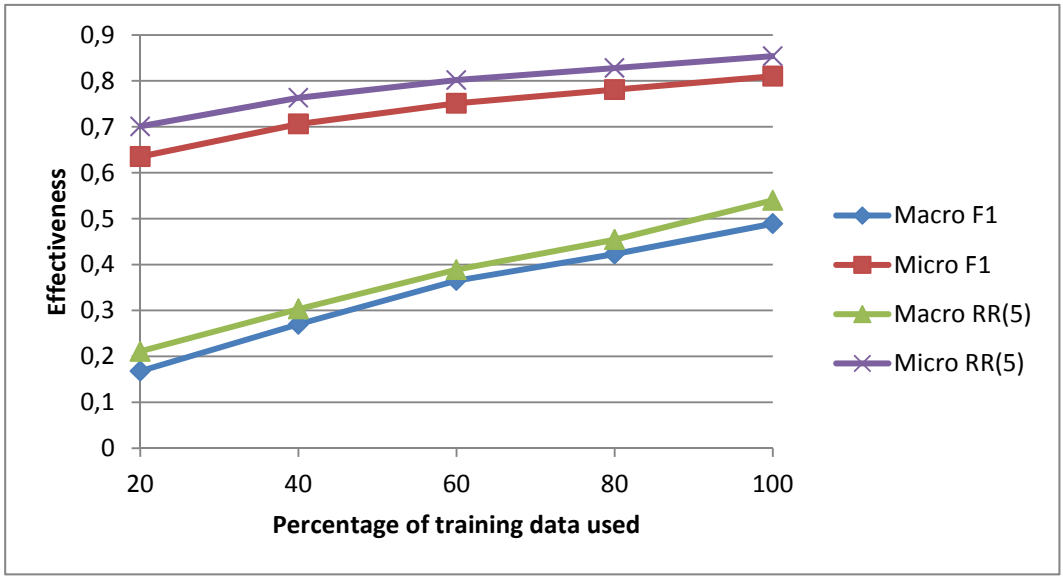


Figure 3.2: *Web Site classification - Learning curves for F_1^{μ}, F_1^{M}, RR_k^{μ}, RR_k^{M}, expressed as a function of p.*

3.4.4 Efficiency issues

Our experiments have been run on a commodity machine Intel Core i7-980X Processor Extreme Edition equipped with six 3.33Ghz cores and 24GB RAM. All experiments were run with sequential code except for the crawling part, where a multithreaded process was employed.

From the point of view of processing time:

1. Crawling the entire dataset (69,343 websites + info about them crawled from Alexa, Twitter, Facebook) required $\approx$ 3 days and 4 hours (at a rate of $\approx$ 5 websites per minute). This time also includes the effort wasted for attempting to crawl the 47,136 websites that, in the end, proved unreachable.

2. Content extraction for the entire dataset required less than 2 hours (at a rate of $\approx$ 185 websites per minute); this includes noise removal, feature extraction, indexing, and language identification.

3. For each of the 10 folds in the 10FCV experiment, training (on 90% of the dataset, i.e., 18,034 websites) required less than 4 hours, while classification (of 10% of the dataset, i.e., 2,003 websites) required $\approx$ 18 mins (at a rate of $\approx$ 108 websites per minute).

The above figures allow us to estimate that coding 1,000 new websites (after the classifier has already been trained) requires $\approx$ 3 hours and 35 mins. Of these, $\approx$ 3 hours and 20 mins (93.0% of the total effort) are needed for crawling, $\approx$ 6 mins (2.8%) are needed for extracting content, and 9 minutes (4.2%) are needed for classification. So, the largest amount of time is spent for crawling, a phase which is not entirely under our

control since the speed at which it can be performed depends e.g., on the throughput of the server on which the website is hosted, and on the speed of the network connection. However, the crawling process is parallelized (each task is entrusted with a host, i.e., a website), so it can be sped up almost at will by increasing the level of parallelism. It should be remarked that the system is still in prototype form, and that margins of improvement in terms of efficiency are thus still large.

3.5 Related Work

Website classification. Website classification is tackled in the work of Yang et al. [215] by framing the problem as one of *webpage* classification: for each website, the first 50 pages (in breadth-first fashion) from the site are crawled and merged into a single page, which is classified via webpage classification methods. In their work the emphasis is comparing different methods for making sense of hyperlinks pointing out of the website, and no sophisticated technique is attempted for either content extraction or feature selection; the problem of language identification is not mentioned in their paper, which seems to suggest that all the websites in their datasets are from the English-speaking world.

Webpage classification. A survey of techniques for webpage classification can be found in [174]. The issue of crawling and preprocessing webpages for their classification is akin to that of doing the same for their retrieval or for their clustering; on crawling and preprocessing for webpage clustering, see [36, Section 4]. The already mentioned work by Yang et al. [215] indeed tackles webpage classification according to industry sector, using a set of 4,285 URLs classified under a coarse-grained taxonomy of 28 classes and a more fine-grained one of 255 classes called Hoovers-28 and Hoovers-255, respectively; the datasets used in this research had originally been assembled by Ghani et al. [88]. Such a dataset, being almost 15 years old, is unfortunately unusable nowadays, since it may safely be assumed that many of the URLs in that dataset point to websites that do not exist anymore, or whose content has radically changed since then. As a consequence, results obtained nowadays are not comparable with the results obtained then, even if obtained on the same dataset; this is, of course, one of the problems of performing experiments on a moving target such as the Web.

Another attempt at webpage classification according to industry sector is the one in [147], which uses a depth-2 hierarchy of 71 classes and a dataset of 6,440 webpages. It should be noted, however, that [147, 215] are not real attempts at maximizing the accuracy of an operational website classifier, since they only use the datasets mentioned for investigating one specific aspect of this task (text classification under hierarchically-organized classification schemes in the case of [147], hypertext classification in the case of [215]), leaving other aspects unexplored. The use of information extracted from the URL for purposes of webpage classification is studied in [119].

3.6 Conclusions

We have presented a study on automatically classifying company websites by industry sector; this is a challenging classification task since it combines issues of topic with issues of genre, the two aspects being closely intertwined in classification schemes

used for this task. The experiments we have carried out on a dataset of more than 20,000 websites classified according to a 2-level taxonomy of 216 classes have shown that the system can obtain very high accuracy values, guessing the true class of the website more than 80% of the times; this is an important feat, since picking the correct class from no less than 216 available classes amounts to finding the classic needle in the haystack.

An interesting improvement of this work could be related to extending the system to address the classification of websites expressed in languages other than English. This is an important step for today's globalised market.

CHAPTER 4

Multi-Store Metadata-Based Supervised Mobile App Classification

ABSTRACT

In this chapter we provide an experimental analysis of another interesting use case treatable as an example application for scenario defined in Section 1.3.1. As in Chapter 3, we used the framework JATECS *as the core text processing technology to develop the software system.*

The mass adoption of smartphone and tablet devices, together with the consolidation of the related platforms – iOS, Android, and more recently Windows Phone – have boosted the growth of the mobile applications market. Confronted with a huge number of choices, users may encounter difficulties in locating the applications that meet their needs. This problem is made even more severe by the fact that, in addition to the official "stores" for the main platforms, a number of independent app distribution channels are springing up.

Sorting applications into a predefined classification scheme would be of help to the app discovery process. Some stores do classify apps into classification schemes, but the class for a given app is often specified by the app developer; especially in stores with permissive publication policies, this may mislead users. To make things more difficult for the user, the classification schemes differ from store to store. Systems for automatically classifying apps into a user-defined classification scheme would thus be of significant help.

Methods for automated app classification have been proposed that rely on tracking how the app is actually used on users' mobile devices; however, this approach can lead to privacy issues. We present a system for classifying mobile apps into user-defined

classification schemes which instead leverages information publicly available from the Web and from the online stores where the apps are marketed. We present experimental results obtained on a dataset of 5,993 apps manually classified under a classification scheme consisting of 50 classes. Our results indicate that automated app classification can be performed with good accuracy, at the same time preserving users' privacy.

4.1 Introduction

Mobile instruments such as smartphones or tablets are widely used nowadays, and for many people they represent the main entry point into the world of information technologies. Many software companies are focusing their development activity on mobile platforms, since these latter represent one of the most promising markets in information technology[1].

In this sector the software distribution model is radically different from the standard distribution model of computer software. Mobile applications are mainly available, if not exclusively, on specific channels called "markets" or "stores". The user experience is then simplified: users can browse these stores, locate specific apps, and download them immediately, which means that the software is directly transferred from the producer to the consumer. Each mobile platform, such as Android or iOS, has its own store (e.g., the Play Store for Android apps and the Apple Store for iOS apps), which is usually accessible from the Web or from a dedicated app. However, many new stores are springing up for each platform, especially for Android[2]. This fragmentation of the software distribution channels may negatively affect the user experience when searching for an app to satisfy one's needs.

In order to help the user in locating the apps she needs, stores are internally organized according to a pre-determined classification scheme. That is, each app is labelled according to one or more classes it belongs to, which helps the user in exploring the store more effectively.

However, it is the developers themselves which label their own apps; especially in stores with permissive publication policies, this may mislead users. Additionally, each store relies on its own classification scheme, depending, for example, on the functionality of the store and the target for which it is intended; for instance, the Amazon Appstore only hosts apps for the Kindle eBook reader, which means it is targeted more to readers than to generic users. In a scenario in which the user wants to explore and find an app suitable for her own needs, it would instead be more useful to have a unique, user-defined classification scheme according to which all apps, independently of the store they are marketed on, are classified.

In this chapter we present a system for automatic app classification. Methods for automated app classification have been proposed that rely on tracking how the app is actually used on users' mobile devices [223]; however, this approach can lead to privacy issues. Our solution enables users and developers to automatically classify apps, independently of their distribution platform and under a user-defined classification scheme,

[1] http://goo.gl/zqPTa7

[2] http://en.wikipedia.org/wiki/List_of_mobile_software_distribution_platforms

at the same time preserving users' privacy. The software has been implemented mainly using JATECS library presented in Chapter 2, and, where necessary, it has been integrated with the usage of other external NLP tools.

We tackle automated app classification (a.k.a. "categorization") by leveraging the metadata (textual and numeric data) contained in the page that describes the app (which is thus identified by its URL), as contained in a specific app store. We approach the app classification task through supervised learning: given a custom pre-defined classification scheme and a set of apps manually classified according to it to be used as training data, our system trains a classifier capable of automatically assigning classes in the classification scheme to new, unseen apps. We assume that each app can be assigned to one or more classes, which make this an instance of *multi-class multi-label classification* ("multi-class" referring to the fact that there are more than 2 classes in the classification scheme, "multi-label" referring to the fact that more than one class can be attributed to the same app).

Classifying apps can be seen as a subtask of *app discovery*[3]. Stores contain several hundred thousands apps, and the process of searching for the desired app (i.e., the one with the right functionality) is more complex than retrieving simple textual documents. Supporting discovery with classification is already employed by popular stores such as the Apple Store and Google Play, which have recently expanded their classification scheme to a richer set of classes and subclasses[4]. Mobile apps and services for app discovery are becoming popular. One example is Xyo[5], which creates a stream of recommended apps; a user can navigate through the stream, and each app description is enriched with subclasses automatically assigned by Xyo. Appcrawlr[6] is another service which makes use of machine learning and NLP technologies in order to capture app genres and functionality. All these developments in app discovery date back to last year; this task is going to be central in the growth of the stores, and the tools for classifying apps are consequently becoming of key importance.

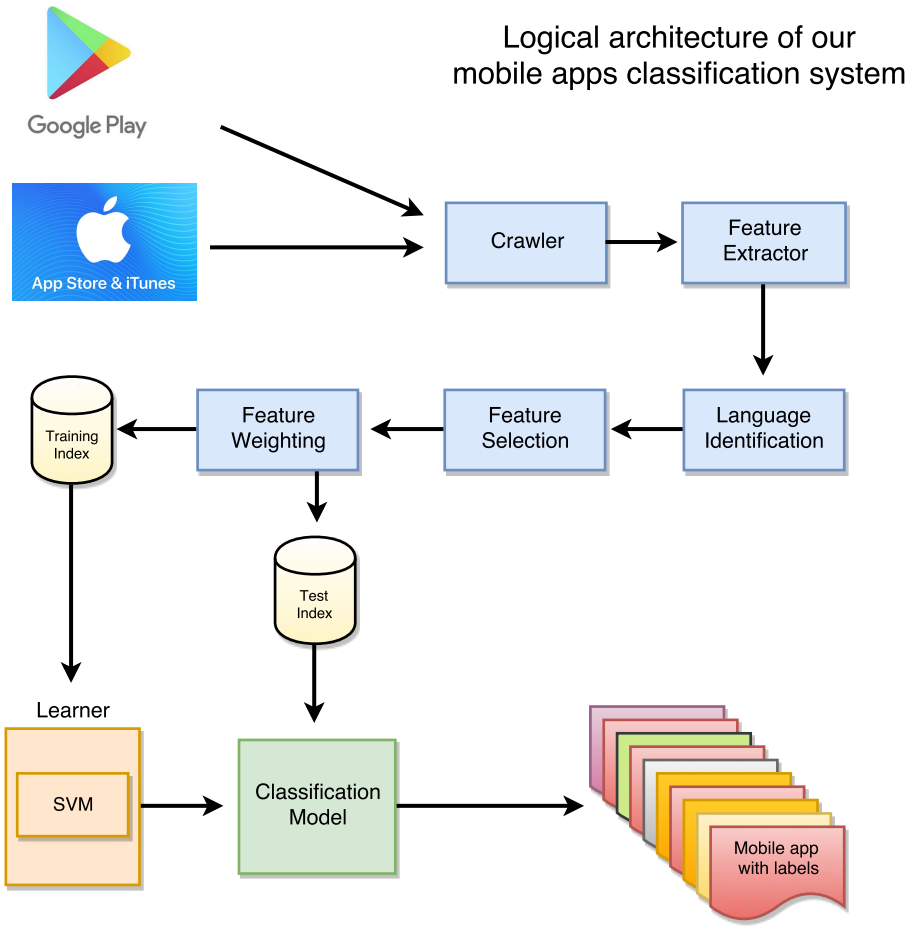
The rest of the chapter is structured as follows. In Section 4.2 we describe our app classification system, especially focusing on the feature extraction phase. Section 4.3 describes the setup for the experiments we carry out, whose results we then discuss in Section 4.4. Section 4.5 discusses related work, while Section 4.6 concludes.

4.2 A system for App Classification

In this section we describe in detail the main components of the system for App Classification that we have developed. The most important modules of the software have been created by using the JATECS framework described in Chapter 2. The remaining secondary modules have been developed using external state-of-the-art open source software libraries (e.g. the *Language Detection Library for Java* used as the library for language detection) freely available on the Internet. In Figure 4.1 we show the logical architecture of the developed system. In the following we explore the characteristics of the main submodules included in our software.

[3] https://goo.gl/4kyDLG
[4] http://bit.ly/1xtFV7F
[5] http://goo.gl/07Q8Jy
[6] http://appcrawlr.com/app/technology

Figure 4.1: *Design of our mobile apps classification system.*



4.2.1 Crawling App Stores

The first step of the process consists in gathering the metadata of the mobile applications we are interested in (module *Crawler* in Figure 4.1). In this work we focus on Android and iOS apps, disregarding for the moment apps for the Windows Phone.

The metadata of a specific application can be retrieved from the Google Play store[7] or the Apple iTunes store[8], depending on which stores the app is published on.

To access metadata of an Android application, we use the open source Java library 'marketapi'[9]. Given an Android application unique address (e.g., the Gmail app address is `com.google.android.gm`), the library, by querying Google servers, allows to quickly retrieve the various metadata fields about the application that are relevant for our purposes (see Section 4.2.2).

In order to retrieve metadata about iOS applications, we use the public Apple service 'Search API'[10]. This Web REST API can be queried by submitting the unique application ID (e.g., `https://itunes.apple.com/lookup?id=5654554`), which results in a JSON response containing all metadata related to that application. The JSON response can then be easily analysed to extract all information of interest.

4.2.2 Content Extraction

App metadata are processed in order to extract features to be fed to the learning algorithm (module *FeatureExtractor* in Figure 4.1). The metadata fields from which we extract features are the following:

- **Description**: The app description contains important textual information. This field briefly presents the app, and it usually lists its characteristics and functionality. This information, expressed in natural language, is relevant for our task not only because of the explicit clues it provides (e.g., "IMAP e-mail client"), but also because of the stylistic and genre-specific use of language that can characterise some classes of apps (e.g., adventure games are often described using an emotion-laden language, while music-streaming apps often cite famous artists' names). We extract all the words contained in the "description" field and perform "stop word removal" (i.e., we remove content-neutral words such as articles and prepositions) and "stemming" (i.e., we map a word into its morphological root, so as to collapse e.g., "computers" and "computational" into the same feature).
- **AppleClassScheme**: We include the Apple Store class assigned by the app developer as a feature. Each app is assigned exclusively to one class (e.g., Business, Education, Games, ...) in the store, so this yields a single feature whose value is the app's class name.
- **GoogleClassScheme**: Similarly as for "AppleClassScheme", we include the Google Play store class assigned by the app developer as a feature.
- **Name**: Given the name of an app, we extract all the words contained in it and perform stop word removal and stemming.

[7] `https://play.google.com/store`
[8] `http://www.apple.com/itunes/overview/`
[9] `https://code.google.com/p/android-market-api/`
[10] `http://goo.gl/iYaIcA`

- **AverageUserRating**: User ratings define the quality of an application, and they can thus be used as a clue, e.g., to detect scam applications; for instance, for an app with a low AverageUserRating we might not want to trust the developer-provided information in the AppleClassScheme / GoogleClassScheme field. We extract the average user rating of each application, and we define a numerical feature which takes values in the $[0, 5]$ interval, the greater the value the higher the rating.

  It should be mentioned that we do not use the total number of downloads as an additional feature, since this number is known to be strongly correlated to average user ratings [77], and would thus likely represent duplicate information.

- **UserRatingCount**: The number of user ratings define the popularity of an application. Together with AverageUserRating this feature type represents user-related information, as it quantifies the feedback an app receives. Again, this is a single numerical feature with positive integer values.

- **FileSize**: An app file size is an indicator of the complexity of the application, so it is an ideal feature for discriminating apps with different targets of use (e.g., the file size of games is usually large). Again, we split the range of file sizes into three subranges, measured in bytes: $[0, 10^6]$, $(10^6, 10^7]$ and $(10^7, +\infty)$. Again, this is a single numerical feature with positive integer values (number of bytes).

We qualify all the features we extract by a prefix that indicates which field the feature comes from, with the goal of placing different emphasis on features coming from different fields. For instance, word "maps" may have less importance if it is extracted from the Description field (in which case it is represented as `DESCRIPTION:maps`) and more importance if it is extracted from the Name field (in which case it is represented as `NAME:maps`).

4.2.3 Language Identification

We analyse each app description in order to determine the language it is written in (module *LanguageIdentification* in Figure 4.1). We detect language using the Language Detection library[11]. This library implements a naïve Bayesian learner trained on a corpus of Wikipedia pages in different languages; the learner exploits as features the character n-grams (i.e., sequences of n consecutive characters, possibly across different words) extracted from the texts, and builds a single-label multi-class classifier for the languages in the training set. For the purposes of this experimentation we discard from consideration all apps whose description is deemed not to be in English by the Language Detection library.

4.2.4 Feature Selection and Weighting

The feature extraction process described in Section 4.2.2 returns a high number of features. For the dataset that we use in the present work (see Section 4.3 for details), no less than 50,379 features are generated from the 5,792 apps the dataset consists of. In order to keep the computational effort to a more manageable level, and in order to avoid

[11] `http://code.google.com/p/language-detection/`

overfitting, we perform feature selection using information gain in the same way as described in section 3.2.4. Related to feature weighting, we use BM25 in the same way as described in Section 3.2.5.

Feature selection and feature weighting are performed respectively by modules *FeatureSelection* and *FeatureWeighting* of the architecture described in Figure 4.1.

4.2.5 Classifier Training

As the learning algorithm we have used the support vector machines (SVMs) [113], and more specifically the implementation provided by the `libsvm` software package[12]. This functionality is implemented on module *Learner* presented in Figure 4.1. Since `libsvm` requires features to be numeric, we convert set-based features (such as "AppleClassScheme" and "GoogleClassScheme") into binary ones. For instance, feature "AppleClassScheme" has 46 possible values, since the classification scheme used in the Apple Store consists of 46 classes and each app is classified under exactly one of them. We thus feed `libsvm` with 46 binary features, each indicating whether the app has the corresponding class or not.

We train a binary classifier on each class of the classification scheme; more than one classifier can return a positive decision for the same app, which implements the multi-label character of this classification task. In `libsvm` we have used a linear kernel with the parameter `C` (the penalty parameter of the error term) set to 1; this configuration, while simple, achieves a good trade-off between model complexity and effectiveness, and is the default setting used in `libsvm` software.

4.3 Experimental Setting

For our experiments we have used a classification scheme and a dataset that were provided to us by a customer. Note that we had no control on the design of the classification scheme and on the choice of the dataset; we thus take both as given[13]. The dataset consists of 5,792 app IDs classified according to a flat classification scheme consisting of 50 classes; each app belongs from a minimum of 1 to a maximum of 4 classes. The dataset is very imbalanced; for instance, a single class ("Games") accounts for 4,903 apps (84.6% of the total), while the other 49 classes have just an average of 34.2 apps each. This makes learning accurate classifiers for these 49 classes a difficult problem.

We perform feature selection with $\xi = 30$, i.e., we retain 30% of the original features, bringing the original number of 50,379 features to a more manageable number of 15,114 features.

As a measure of effectiveness that combines the contributions of *precision* (π) and *recall* (ρ) we have used the well-known F_1 function, defined as

$$F_1 = \frac{2\pi\rho}{\pi + \rho} = \frac{2TP}{2TP + FP + FN} \tag{4.1}$$

which corresponds to the harmonic mean of precision and recall, where TP stands for true positives, FP for false positives, and FN for false negatives. Note that F_1 is

[12] Freely downloadable from `http://www.csie.ntu.edu.tw/~cjlin/libsvm/` . In preliminary experiments we had tried to use MPBOOST [66] as the learning algorithm, with inferior results.

[13] Unfortunately we have no rights on the data and on the classification scheme, so we are not in a position to make them publicly available.

undefined when $TP = FP = FN = 0$; in this case, consistently with most other works in the literature, we take F_1 to equal 1.0, since the classifier has correctly classified all apps (as negative examples).

In multi-label classification it is customary to average the class-specific F_1 scores across the classes by computing both microaveraged F_1 (denoted by F_1^{μ}) and macroaveraged F_1 (F_1^M). F_1^{μ} is obtained by (i) computing the class-specific values TP_i, (ii) obtaining TP as the sum of the TP_i's (same for FP and FN), and then (iii) applying Equation 4.1. F_1^M is obtained by first computing the F_1 values specific to the individual classes, and then averaging them across the c_j's.

4.4 Results

Table 4.1 presents the effectiveness results we have obtained in our experiments; all results were obtained by 10-fold cross-validation (10FCV).

Table 4.1: *Mobile App classification: accuracy of classifiers obtained by using all extracted features (Columns 2 to 4) or by using selected features only (Columns 5 to 7). In Column 5, percentages indicate the survival rate of the specific feature type. Boldface indicates best results.*

	No Feature Selection			With Feature Selection			
	#Features	F_1^M	F_1^{μ}	#Features		F_1^M	F_1^{μ}
Description	45652	0.219	0.878	14076	(30.8%)	0.229	0.880
AppleClassScheme	46	0.118	0.862	29	(63.0%)	0.048	0.816
GoogleClassScheme	38	0.128	0.853	30	(78.9%)	0.095	0.834
Name	4632	0.246	0.849	968	(20.9%)	0.187	0.826
All feature types	50379	0.311	**0.895**	15114	(30.0%)	**0.320**	**0.895**

The first observation that can be made by looking at the microaveraged results (F_1^{μ}) is that we have obtained very high accuracy for both the NFS and WFS configurations. This is the case not only for the configuration containing all feature types ($F_1^{\mu} = 0.895$), but also for the configurations based on specific feature types (e.g., AverageUserRating, with just 4 distinct features, provides a surprisingly good result, $F_1^{\mu} = 0.792$). The macroaveraged results (F_1^M) are lower, but this was to be expected in the light of the severely imbalanced nature of the dataset. In fact, the micro- and macro-averaged versions of a measure yield the same value only when the dataset is perfectly balanced, while the former yields higher values than the latter when the dataset is imbalanced. In our case, the most frequent class (Games) totals 4,902 instances while the least frequent one (Tethering) has only 1 instance, which is an indication of the severe level of imbalance of this dataset.

A second fact that emerges is that feature selection not only does not harm accuracy, but even gives some marginal accuracy improvements, which indicates that many features that had originally been extracted are useless for the classification process. Using feature selection is thus a win-win approach, also due to the fact that the computational

cost of both the learning and the classification phases scale linearly with the number of features used [115].

4.4.1 Effects of Different Feature Types

In an attempt to better understand the quality of the features contributed from each of the fields identified in Section 4.2.2, for each such field we have run experiments by using the features extracted from that field only; the results are reported in Table 4.1, each row representing a specific feature type. While these experiments do not account for the subtle interactions that may take place as a result of the co-presence of features of different types, they nonetheless give an indicative idea of the relative contribution that the different types of features can give to the overall classification task. We have run such experiments (a) with the full feature set (Columns 2-4 of Table 4.1), and (b) with the set resulting from the feature selection process (Columns 5-t of Table 4.1). The feature selection process consisted in putting into a common pool all the extracted features of all types, and then selecting the best ones irrespective of their type, so that different feature types compete with each other. In Column 6 we report, for each feature type, the *survival rate* of a given feature type, i.e., the percentage of features of that type that made it into the final selected set.

The first observation we can make is that, as it could be expected, the "All feature types" setting is the best overall, i.e., there is no single feature type that outperforms it. While this is true for both measures, this is particularly evident for F_1^M, both in the NFS and WFS configurations, which indicates that infrequent classes (which are the ones that get most attention from macroaveraged measures) need all bits of available information to be put to work in order to be correctly handled.

A second observation is that some individual feature types deliver reasonably good accuracy by themselves. While some of them produce good results for F_1^μ only ("AppleClassScheme" and "GoogleClassScheme"), other types ("Name" and "Description") provide fairly good values for both F_1^M and F_1^μ. "AppleClassScheme" and "GoogleClassScheme" give us information about the class assigned to the application in their corresponding stores; this is clearly very useful semantic information to learn from, especially for very popular classes (this should explain the very good micro accuracy obtained in the results). "Name" features also perform well, and this is intuitive, since authors tend to use highly significant words in app names. Also the fact that "Description" features are helpful is unsurprising, since the content of this field gives a very detailed information about the aim and functionality of the application, i.e., it is the field where content is meant to be conveyed.

The remaining feature types ("AverageUserRating", "UserRatingCount" and "FileSize"), when used in isolation of the others, deliver smaller accuracy levels (especially for F_1^M); this is partly due to the fact that these types consist of only a handful of features each.

4.4.2 Effects of Feature Selection

The results show that the feature selection process can globally improve the accuracy of the generated classifiers on the infrequent classes.

The "Description" feature-specific classifier benefits from feature selection. This feature type contains the majority of features extracted from the dataset (45,652 out of

50,379), so a reduction in the number of features is necessary to prevent the problem of data overfitting.

For the "Name" feature type we observe a substantial cut, due to the fact that many app names have an evocative rather than a descriptive nature, and only few of them contain terms related to the functionality of the application. Some examples of useful app names, in terms of features, are "Google Calendar", "Jorte Calendar", "Aviary Photo Editor", "Photo Collage Editor". After the cut, app with fancy names cannot be classified with this only feature type, for this reason we see a large drop in F_1^M. The features related to store classification schemes receive a smaller cut than for "Names" features, but due to the uniqueness of these features (only one classification scheme feature per app exists), some apps are not more represented by these features after feature selection. We thus see the same drop in effectiveness. This happens because apps with store classes which are not discriminating for our classification scheme (i.e., classes which do not have a corresponding concept in our classification scheme), cannot be classified because these features are cut with high probability.

We do not include figures for the remaining feature types ("AverageUserRating", "UserRatingCount" and "FileSize") since, when used in isolation of the others, they turn out to have no discriminatory power at all, since each of the 50×3 binary classifiers generated out of them is the majority class classifier (i.e., Games is always assigned and none of the other classes is ever assigned).

4.4.3 Learning curve

We have run additional experiments in order to plot a learning curve for our classifiers, with the goal of finding out if we are likely to obtain improvements should we provide additional training data. To run these tests, in each of the 10 "folds" of the 10FCV experiment the test set is retained in its entirety while only a randomly selected percentage p of the training set is retained. We have run tests with $p \in \{20\%, 40\%, 60\%, 80\%\}$, in "All feature types" mode for both configurations tested (NFS and WFS); if the results obtained with $p = 100\%$ (which we have reported in the preceding sections) are higher than the ones for $p = 80\%$, this means that we probably have not yet reached an effectiveness plateau, and that there are still likely margins of improvement.

The learning curve displayed in Figure 4.2 (see also Table 4.2) indeed shows that effectiveness is still on the rise, both for the full features set and the reduced features set configurations. This indicates that the results discussed in the previous sections, while already good in themselves, could likely still be improved upon if more training data were available.

4.5 Related Work

Several tasks rely on mining app data, which can be defined not only by the descriptive information in the stores or in the apps itself, but also by the usage data extracted from users' devices. In the field of app data mining the task of app classification is not very popular in literature, while solutions to the tasks of app recommendation and app search are widespread, this type of applications are the primary way for app discovery and diffusion. Automated classification in the app ecosystem is mainly used as an enabling technology, e.g., in understanding user needs (e.g., analysing app reviews)

	No Feature Selection		With Feature Selection	
	F_1^M	F_1^μ	F_1^M	F_1^μ
20%	0.044	0.842	0.051	0.845
40%	0.115	0.861	0.132	0.863
60%	0.180	0.875	0.196	0.877
80%	0.247	0.885	0.259	0.886
100%	0.311	0.895	0.320	0.895

Table 4.2: *Mobile App classification: accuracy of classifiers (using all feature types) obtained by using increasing percentages of training data.*

or pointing out privacy and security aspects of the apps. However, classifying by app genre is sometimes used within applications devoted to app discovery.

Stores already provide, in each app page, suggestions about similar apps, but there is a high demand for more effective and personalized recommendation. App recommendation is primarily based on user' preferences, therefore many solutions exploit usage information or user' hints [121, 138]; some solutions also make use of app data. One relevant aspect of the apps is the set of features they implement, especially how new versions are improved with new characteristics. This is taken into account in [139], where recommendation is based on what the user desires from an app, thus analysing both textual information from versioning descriptions and the class each version is associated with. An system of this type is described in [218], in which app features are extracted through topic modelling of app descriptions, so the probability of a user to be attracted by a new app is proportional to features overlap with her installed apps. In [27] app metadata is used in order to build a graph weighted by app similarities, so as to recommend new apps by exploiting the graph structure.

Recommendation can be also meant as an instrument for searching apps; in fact, app search turns out to be more effective when users' behaviour is taken into account. In [217] apps are ranked by scoring them according to their by their interoperabiliy. Application dependencies (namely the consecutive execution of correlated apps) are extracted from usage patterns. App metadata is particularly exploited in [154], where apps are indexed by their store pages; an inverted index of app classes is also created, in order to consider the correlation between words and classes, so as to finally boost retrieval accuracy through the relatedness between user contexts (which have a textual representation) and classes.

The classification of app data may be carried out for different reasons, and is often used as an enabling technology for carrying out other tasks. One popular such task is mining app feature requests from user feedbacks, which are abundant in app reviews [167], so as to help developers improve mobile apps. For example, in [43] the authors process app reviews, filtering out the ones with poor user feedback and then ranking them. A developer can then read, for each app feature, the most representative reviews in which the feature is commented upon or requested. A task in which user feedback is exploited is the prediction of app popularity; in [224] a system for predicting app popularity is described which models the sequence of popularity observations, which

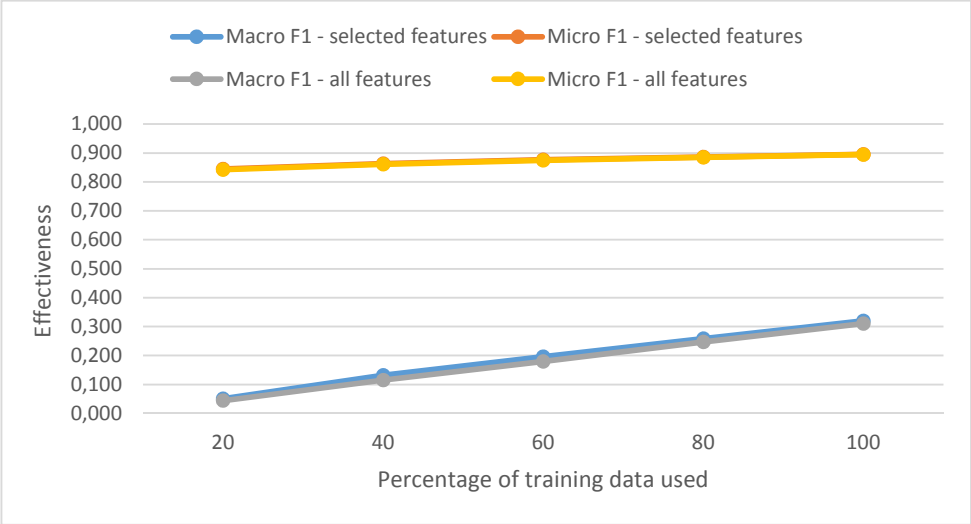


Figure 4.2: *Learning curves for F_1^μ and F_1^M, expressed as a function of p and differentiated by the feature set used. Note that the two F_1^μ curves (the one with and the one without feature selection) are barely distinguishable; same for the two F_1^M curves.)*

are represented by app rankings, ratings and review topics.

Another task in which automated classification is widely used is in app security verification. Analysing user behaviour, such as calls or browser navigation, is possible in order to detect anomalies which can be associated to malicious software [55, 62]. In this scenario, apart from usage information, the most important features to extract from app metadata are the permissions, the actions an app can take in order to access to private data of the user [81].

Few works on app classification by genre, or target of use, exist. The task is faced in [223], in which classification is obtained with a model trained on usage patterns of the users, together with external knowledge retrieved from the Web. Their experimental setting is very similar to ours, in terms of data and objectives, but no app metadata is used. It is an invasive approach, which is dependent on users' private data. In another approach in [185] the authors extract features both from the app file and store metadata. They perform a series of experiments on a small dataset, with seven classes, and with only Android applications. In our experimental setting we instead deal with a large classification scheme, which is more effective in diversifying apps. An interesting approach to malicious apps detection in [91] performs clustering by app genres. The classification system initially creates clusters of apps by classes generated by means of a topic modelling method, so as to group apps with similar functionality, with the final objective of identifying anomalous apps in each cluster.

4.6 Conclusions

We have developed a system for the automatic classification of mobile apps by genre, and we have evaluated experimentally its effectiveness. We have found that the metadata retrieved from app stores is of fundamental importance in representing the content of apps, but also that classification via supervised learning under non-trivial classification schemes is a difficult task. Accuracy is still less than optimal; there are still large margins of improvement. In the future we plan to explore many directions: (i) finding additional sources of app-relevant information, e.g., adding more stores, identifying relevant information on social networks, such as product pages, fan pages, comments, (ii) improving the feature extraction phase, e.g., being able to process multilingual information, (iii) adopting learning algorithms specifically devised for extremely unbalanced datasets, (iv) exploring the use of hierarchical classification schemes, (v) testing the use of methods for the (semi-)automatic generation of additional training examples.

CHAPTER 5

Facing on text mining problems on big data contexts

ABSTRACT

In this chapter we concentrate our attention to the issues related to analyzing textual data in big data setups, i.e., the Scenario 2 described in Section 1.3.2. With the explosion of data availability occurred in the last years due to the exponential growth of Web and Internet usage by people and companies, an important today's requirement is the availability of efficient big data processing tools able to build effective and scalable software solutions. One of the most important type of content available in abundance is text which can appears in multiple different contexts (e.g. social networks, news articles, images descriptions, etc.). Textual contents are unstructured data by nature and they required very complex analyses in order to extract useful information and semantic from them. Current available text processing tools are usually designed to work in sequential environments, requiring a lot of efforts from programmers in order to integrate them with general purpose big data processing tools which instead are optimized for parallel or distributed environments. In this chapter, we analyze two possible text processing application scenarios requiring such type of integrations and focused on big data solutions, but with different assumptions on runtime environments. In the first one, characterized by availability of a limited amount of computational resources, we propose an optimized multithread library called Processfast exposing advanced data processing capabilities and we compare it against some available distributed data processing solutions. In the second application scenario, focused on conventional distributed environments, we show how we can exploit Apache Spark a) to design an elegant and effective framework called NLP4Spark to efficiently index textual contents and b) to en-

hance the efficiency of two textual classifiers belonging to boosting family by providing optimized distributed versions of the algorithms.

5.1 Introduction

In the last years we have seen an explosion of data available to be used as valuable information potentially providing insights and new opportunities for the ideation/improvements of users services. This dramatic growth of data availability has been determined by several factors. Internet and the Web have seen a global scale adoption by users and companies with the creation of a huge numbers of different sites and services covering basically all aspects of our life. Today, a vast number of world's population is always "online", ready to communicate with each others at any time by using the most disparate tools (e.g. social networks, email, sms, etc.) provided by smart devices (e.g. smartphone, smartwatch, tablet, etc). In addition, the IoT revolution [93] is determining that our society is directed towards an ever increasing interconnection between physical sensors and people's life. A vast amount of this produced data has textual nature: documents, social network posts, messages, objects descriptions, and so on are just small examples of what you can find today on the Web and Internet in general. Consequently, it is extremely important to provide effective textual processing tools able to work efficiently in big data contexts. Building such type of tools is not an easy task for programmers. Advanced text mining tools make use of sophisticated techniques and algorithms to provide effective textual representations and models to be used in problems solutions. Often completely different approaches can be taken (e.g. statistical vs. NLP) for textual processing solutions. Many existent software (e.g. Weka [97], Nltk [28], Scikit-learn [170], OpenNLP, etc.) provide a limited set of such functionalities, forcing programmers to integrate themselves the tools in a coherent way. In addition, the great majority of this available text processing tools is written using sequential code, with data structures which sometime, in parallel or distributed environments, render the integration task very complicated or almost impossible. On the other hand many existing big data processing framework (e.g. Hadoop [211], Storm [202], Spark [220], Flink [35], Beam[1], etc.) provide usually APIs for general purpose programming, and only in few cases some generic machine learning (ML) library with no specific support for textual analyses. Given these limitations, it is thus extremely important to describe which are the most convenient integration approaches to take when handling different application scenarios in big data contexts.

In this chapter we focus on two typical big data processing scenarios that one can faced on. The first one assumes that the amount of data to be analyzed relatively large but the available computational resources are very limited, typically we have access only to a single powerful commodity machine. In this scenario it was not clear if an existing big data processing tool could work relatively well compared to a native multithread solution. We indeed explore this possibility by proposing Processfast, a new generic data processing library based on a multithread engine, and by comparing

[1] `https://beam.apache.org/`

it with a major big data software player such as Apache Spark. In the second scenario instead we face on big data textual processing problems exploiting distributed solutions based on Spark. We then show how it is possible to use this popular software to build an effective textual indexing framework called NLP4Spark which uses the concept of index (similar to JATECS in Chapter 2) to represent textual information and allow to preprocess index data using a powerful functional approach based on Spark APIs. In addition, as another interesting example of Spark integration, we also describe how to write a distributed software version of two popular textual classification algorithms based on boosting (AdaBoost.MH and MP-Boost), showing that such implementations can greatly exploit the computing power of distributed environments.

5.2 Processfast, a parallel approach for big data processing

Most of the frameworks for parallel and distributed computing focus on a single parallel computing paradigm, e.g., stream processing [155,163,202], map-reduce [57,211,220], task parallelism [61]. Complex applications could benefit from using a combination of different approaches. For example, a text mining system that classifies a stream of tweets can be decomposed into a set of parallel tasks (crawling, NLP processing, indexing, classification, aggregation, report) connected via streams, with each task possibly exploiting data parallelism to efficiently apply the same computation to batches of data. Implementing such a system usually requires to combine several frameworks, with the added burden of implementing a communication layer among them. Moreover, the target architecture of the system, e.g., a single multi-core machine or a distributed environment, will typically result into different choices of frameworks, since each framework is usually targeted to produce its maximum efficiency on a specific architecture. Changing the runtime architecture will often require to change the underlying parallel computing framework[2], with non trivial implementation cost. In addition, it is not clear at all if it is reasonable from a performance perspective to adopt one of the existing big data tools available on the market to work in environments characterized by a limited amount of computational resources (e.g. a single "server" machine). This type of runtime environment, which is targeted by the work in this specific section, could indeed be better exploited by using light tools optimized for local computations only.

In order to answer these open questions, we here propose Procesfast (PF), a Java/-Groovy framework that aims at providing a seamless integration of different parallel computing models, by combining the functionalities provided by different parallel processing frameworks into an homogeneous API. PF does not aim at implementing yet another parallel computing framework, its purpose is to act as a higher-level API that abstracts the functionalities of current parallel computing frameworks, allowing to write scalable applications that once developed can be deployed, and executed efficiently, on different architectures by only switching the PF runtime layer that implements the API on the better suited frameworks. This section introduces the PF main concepts and structures together with some code examples showing how the PF API can be used to write concurrent applications. In addition, we run some experiment which show how the software performs in relation to other technologies, including big data oriented ones.

[2]Many tools have a standalone mode that simulates a distributed environment on a single machine, but it is mainly thought as a development tool, not for production use.

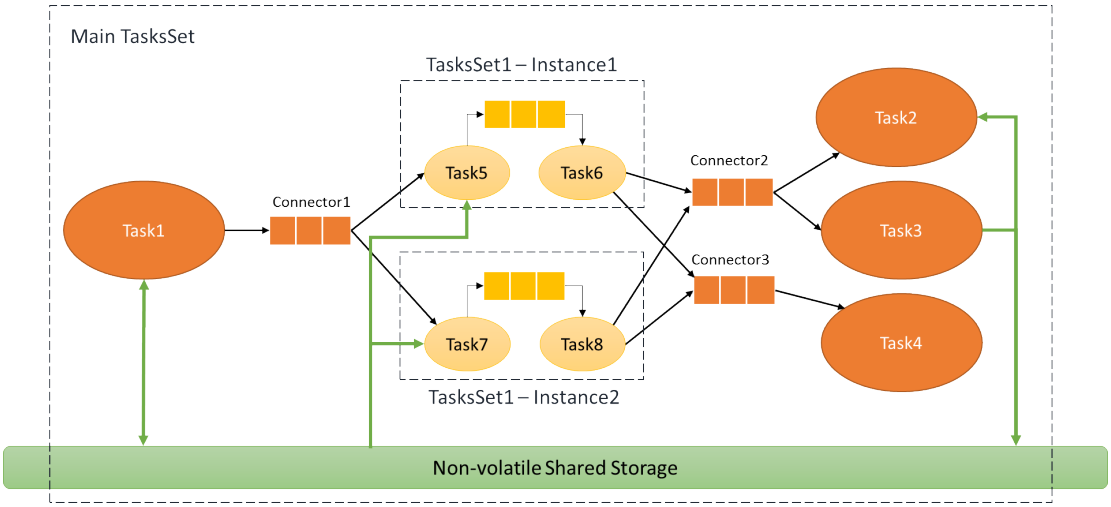


Figure 5.1: *Schema of a Processfast application*

5.2.1 An overview of the programming model

The PF API[3] defines how the developer can write applications that use and combine task/stream parallelism and data parallelism. It provides a lock-free programming model in which an application can be defined in terms of a set of asynchronous processes that are able to intercommunicate.

TaskSet, a logical container and a reusable "black box"

As shown in Figure 5.1, the topology of an application is defined inside a *TaskSet*. A TaskSet is a logical container of *Tasks* (detailed in Section 5.2.1) and *Connectors* (detailed in Section 5.2.1). A TaskSet is also a Task, and thus can be used as a "black box" inside another TaskSet. A TaskSet allows to implement *task/stream parallelism* through the use of Connectors to let the contained Tasks communicate together. *Barriers* can be used to synchronize the execution of Tasks.

Task, a stateful and asynchronous logical process

A *Task* is the minimal unit of execution in PF. Every Task is a stateful logical process which runs asynchronously with respect to the other tasks in the application and can be created in multiple instances inside a single program. A Task is able to communicate through Connectors only with the other Tasks defined in the same TaskSet, or externally using the input and output Connectors of the parent TaskSet (which define the entry and exit points of the TaskSet taken as a black box). As shown in Figure 5.2, a Task internally can exploit the *data parallelism* by operating concurrently on *PartitionableDatasets* (PDs). The concept of PD is very similar to that of RDD in Spark [220]. A PD is a read-only data structure which can be split in n partitions, where every partition can then be processed concurrently as a data stream. PDs can be processed by applying two types of operations, following a map-reduce model:

[3]Processfast public repository: `https://github.com/tizfa/processfast-api`

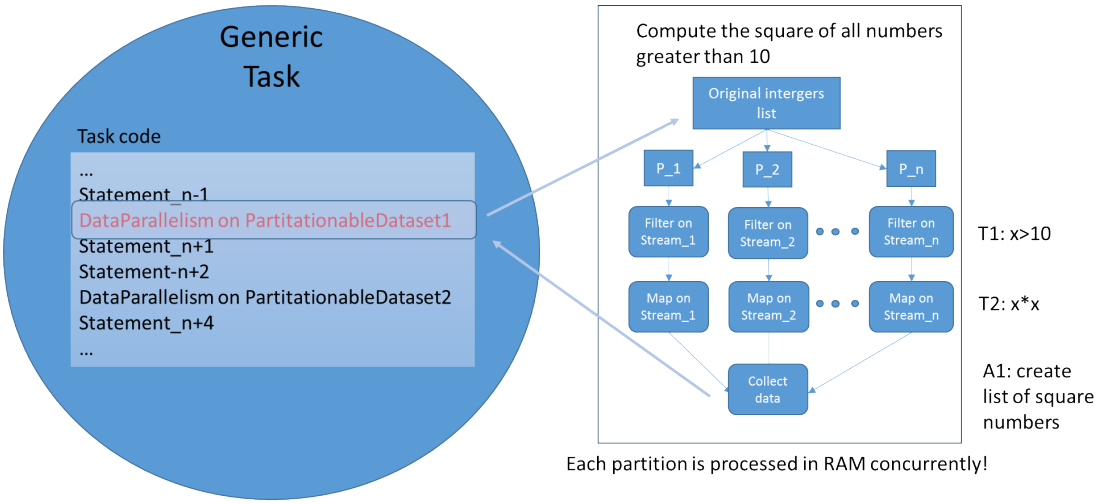


Figure 5.2: *The internal structure of a Task*

- *transformations*: each item in the input stream is transformed in some way resulting in a new item in the output stream (e.g. T1 and T2 in Figure 5.2).
- *actions*: items from input data stream are collected and processed to produce an aggregated result that is returned to the caller task (e.g. A1 in Figure 5.2).

Connectors, interprocess communication based on queues

A Connector is a shared queue, belonging to a TaskSet, uniquely identified by a name. Connectors can have single or multiple Tasks attached as readers and writers. A Task can consume exclusively an item read from a Connector (*first come, first served*, data parallelism) or the Connector can provide to each reading Task the same complete sequence of items as written to the connector (*broadcasting*, task parallelism). Write operations are generally asynchronous (a Task after posting a message on a specific connector can continue its computation) while the read are synchronous and blocking, though is it also possible to define synchronous read/write connections.

Shared permanent storage

The PF API defines a shared permanent storage which allows direct access to some high levels data structures. The main purpose of the permanent storage is to reduce the amount of information transmitted through connectors and to rely instead on solutions that are best fit for the target architectures. The supported data structures are:

- *Array*: a direct access unidimensional array. The structure supports PD views, thus it is ready for parallel processing.
- *Matrix*: a direct access bidimensional array. The structure supports PD views, allowing parallel processing by rows or by columns.
- *Dictionary*: a key/value collection.
- *DataStream*: a byte stream used to load/store data.

5.2.2 Code examples using the provided API

In this section, we provide some code examples in Java/Groovy which show how to use Processfast API to develop concurrent applications. For each example, we annotate the code with some useful comment describing what we do inside the program.

In the following Java code, we exploit task parallelism to show how to implement a simple master-worker scheme with one task acting as the dispatcher of messages to be processed and a set of worker tasks which process the incoming messages by reading them from a shared named queue:

```
// Create runtime.
ProcessfastRuntime runtime = ....

// Define the main tasks set of the program.
TaskSet ts = runtime.createTaskSet();

// Create a named queue of type load balancing.
ts.createConnector("DISTRIBUTOR", ConnectorType.LOAD_BALANCING_QUEUE);

// Declare dispatcher task.
ts.task((TaskContext tc) -> {
  // Get output named queue and write on it 10,000 random numbers.
   ConnectorWriter dist =
       tc.getConnectorManager().getConnectorWriter("DISTRIBUTOR");
   for (int i = 0; i < 10000; i++)
    dist.putValue(new Random().nextInt(1000));
   // Signal that I finished using the output named queue.
   dist.signalEndOfStream();
}).withConnectors(wci -> {
  // Declare that I will use the named queue "DISTRIBUTOR" in write mode.
  wci.getConnectorManager().attachTaskToConnector(wci.getTaskName(),
      "DISTRIBUTOR", ConnectorCapability.WRITE);
});

// Declare worker task.
ts.task((TaskContext tc) -> {
  // Get input named queue and read indefinitely work packages from it.
  ConnectorReader dist =
      tc.getConnectorManager().getConnectorReader("DISTRIBUTOR");
   while(true) {
    // Take an input message and process it.
    ConnectorMessage cm = dist.getValue();
     if (cm == null)
      break;
     int val = (int) cm.getPayload();
     if (val % 2 == 0)
       tc.getLogManager().getLogger("TEST").info("The number "+val+" is
           odd");
     else
       tc.getLogManager().getLogger("TEST").info("The number "+val+" is
           even");
   }
}).withConnectors(wci -> {
  // Declare that I will use the named queue "DISTRIBUTOR" in read mode.
  wci.getConnectorManager().attachTaskToConnector(wci.getTaskName(),
      "DISTRIBUTOR", ConnectorCapability.READ);
}).withNumInstances(4, 4); // The runtime will create 4 instances of this
```

```
    worker.

// Run the application.
runtime.run(ts);
```

In this other example, we instead exploit data parallelism by showing an extract of Groovy code inside a task which process a list of random numbers by using some transformations and actions provided through PartitionableDataset interface:

```
ts.task { tc ->
  ...

   // Build a random list of numbers.
   Random r = new Random()
   def l = []
   (1..10000000).each {
      l << r.nextInt(1000)
   }

   // Create a partitionable dataset of the list.
   def pd = tc.createPartitionableDataset(new
       GenericIteratorDataSourceIterator<Integer>(l.iterator()))

   // Apply some transformations and cache it...
   def squareResults = pd.filter { tdc, item -> item.v2 % 2 == 0 }
   .map { tdc, item -> item.v2**2 }
   .cache(CacheType.RAM)

   // Apply some actions to obtain something back.
   int numItems = squareResults.count()
   List<Integer> first100 = squareResults.take(0, 100)

   ...
}
```

5.2.3 Experimental results and comparison with existing big data tools

We implemented a multi-thread runtime based on GPars library[4] which is able to exploit the parallelism of a multicore processor of a single machine. We tested two use cases on a machine with a CPU AMD FX-8350 4.1 Ghz, 8 cores and SATA disk. The first one is a simple product of two matrices, $C = A * B$ where A is a matrix $10000 * 100$ and B a matrix $100 * 10000$. The sequential computation takes about 50 s to run, the parallel version using a map-reduce approach through PDs and using 8 cores takes about 9.5 s to run.
The second case addressed is indexing and counting the word occurrences from a collection containing a total of about 10 Gb of textual data. A more articulated set of experiments on this use case is reported on Figure 5.3 where it is shown the execution time in seconds of the programs implemented with different technologies, multithread (Processfast and Java 8 Stream API[5]) and distributed (Apache Spark [220] and Apache Hadoop [211]) solutions. For Hadoop version we tested two different deployment options: a) single node setup (Hadoop-EachLine-SingleNode configuration in the figure),

[4] http://www.gpars.org/
[5] https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html

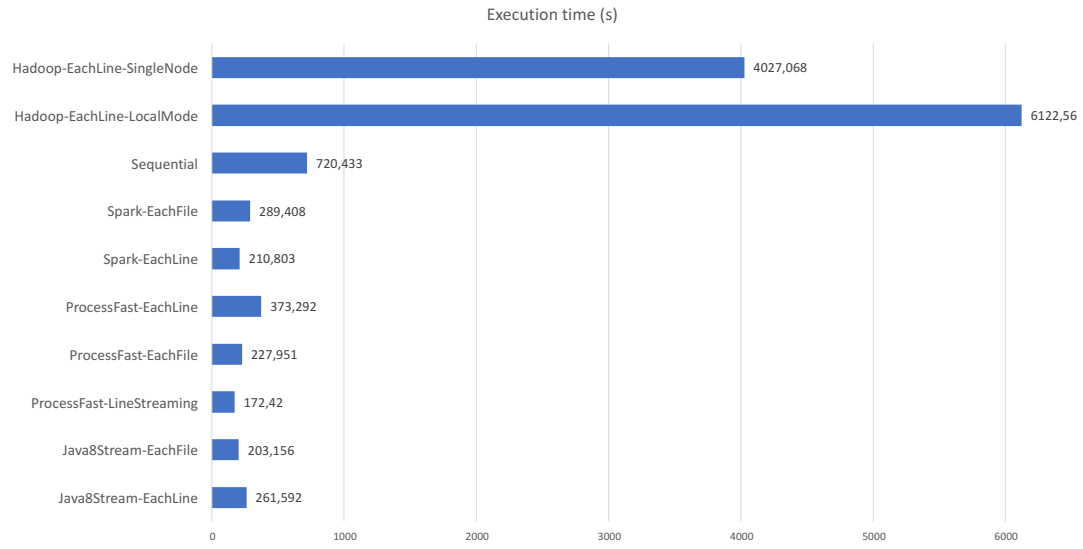


Figure 5.3: *Word occurrences count: Processfast performance vs Java 8 Stream, Spark and Hadoop versions.*

which runs Hadoop processes into a single Java virtual machine (JVM) instance and b) local mode (Hadoop-EachLine-LocalMode in the figure), which runs each Hadoop process in its own JVM instance. Spark instead has been always tested using the local mode running option[6]. For Processfast, Spark and Java 8 Stream we tested two variants of the algorithm, one suffixed with "EachLine" which process each file of the input collection in parallel (data parallelism on files) and the other, suffixed with "EachLine", which read files line by line and process the lines in parallel (data parallelism on lines). In addition, with Processfast we tested another variant suffixed with "LineStreaming" which process each file line by line but with a streaming architecture exploiting a dispatcher-workers-collector schema. Some interesting results emerge from this experimentation. First, Hadoop is a big data tool which should be avoided in contexts where limited computational resources are available: in both configuration tested it provides terrible performance even when compared to a pure sequential version of the code. The other systems works quite well in terms of scalability, with a slight disadvantage of Spark probably due to the type of optimization of the engine (engineered for distributed scenarios). Another interesting difference, probably due on how the Spark engine is organized internally, is that the local optimized code versions (Processfast, Java 8 Stream) works better in modality "EachFile" while the Spark version is faster in modality "Each Line". Anyway, the best results are obtained with "Processfast-LineStreaming" which, due to the particular ingestion architecture used, it is able to efficiently mask the latency necessary to read data from disk with effective processing of the working messages.

The most important key point emerging from this experimentation, also confirmed by other tests not reported here, is the goodness of Apache Spark as a general purpose data processing product. Indeed this software, even though being a tool optimized for distributed environments and specifically focused on processing of enormous amount of

[6] `https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-local.html`

data, is able to work remarkably well also in mono-machine environments with relative small amount of information to process, providing performance very similar to tools specifically optimized for this type of environments. In the optic of having a unique tool which could be used always for all data processing requirements, this suggests it is easier and convenient to adopt it instead of trying to build an alternative product such as Processfast, which at the end could use Spark internally (as a runtime engine) but by masking Spark features with its own API. This last point is very important because masking Spark API means also the impossibility to use transparently the ecosystem software consisting in all the third-party tools developed for this platform[7], so limiting in some way the simplicity of integration and code reuse for the developers. We can also note that Spark is a tool which mainly exploit data parallelism to perform scalable computations but this limitation can be easily overcome. Indeed, in cases where adopting task parallelism and asynchronous communications is convenient, its is easily integrable with other distributed technologies such as Kafka [127] or ZeroMQ [101] message brokers, allowing to build complex real-time data processing systems with reasonable effort. For all these reasons emerging from the above discussion, we decided to move to distributed textual processing by using Apache Spark as the main developing framework and exploiting its specific API to build something similar in spirit to what we have done with JATECS in Chapter 2.

5.3 Apache Spark: a small overview

In this section we provide a quick overview of Apache Spark [220][8], the big data framework we identified as the core technology to develop our scalable text processing tools.

Apache Spark is one of the leading open source big data processing frameworks currently available on the market. It is built around speed, ease of use, and sophisticated analytics, giving to developers a complete tool sets to perform complex processing tasks using a multitude of different approaches. Compared to other popular big data processing frameworks like Apache Hadoop or Apache Storm, Spark offers several advantages. First, it provides a unified programming environment which allow to easily deal with problems having datasets of different nature (e.g. textual content, connected graphs, etc.) and analyzed in different ways (batch vs. near real-time processing). The software is also very optimized in terms of performance and memory usage, allowing to obtain very efficient code both for in-memory and on-disk data processing.[9]. Spark lets you quickly write applications using various programming languages (currently Scala, Java, Python and R) and using a set of over 80 high level operators (like map, filter, groupBy, etc.) or interactively query data through a specific command shell. In addition, the software offers a set of high level libraries built over its core, to query/process data through SQL and in tabular way (SparkSQL), to apply machine learning methods (MLlib), to process streaming data (Spark Streaming) and to perform graphs analytics (GraphX). Lastly, Spark seamless integrates with the Hadoop stack of related technologies, allowing to share the same cluster installation to run its programs, and providing read/write connectors to typical external storage systems like HDFS, HBase, etc.

[7] `https://spark.apache.org/third-party-projects.html`

[8] `http://spark.apache.org/`

[9] According to Spark main site, the software, when compared to Hadoop, is faster up to 100x for in-memory processing and up to 10x for on-disk processing.

RDD (Resilient Distributed Dataset) [219] is the data structure used by Spark to operate concurrently and in a fault-tolerant way over an input dataset. It is a read-only partitioned collection of records, distributed over the nodes of the cluster the program is running on (typically on every node of a distributed filesystem, e.g. HDFS). Every partition can be processed in parallel through a set of available *transformations* and *actions* [10] . These operations can be chained together in a pipeline, allowing Spark to make smart optimizations on data distribution and data loading during processing tasks. The partial results obtained during the various transformations applied on an RDD can also be cached (in memory or on disk), thus avoiding to repeat multiple times costly computations on data.

The main entry point of a Spark application running on a cluster is a process called *Driver* (see Spark architecture in figure 5.4). This process is responsible, through the

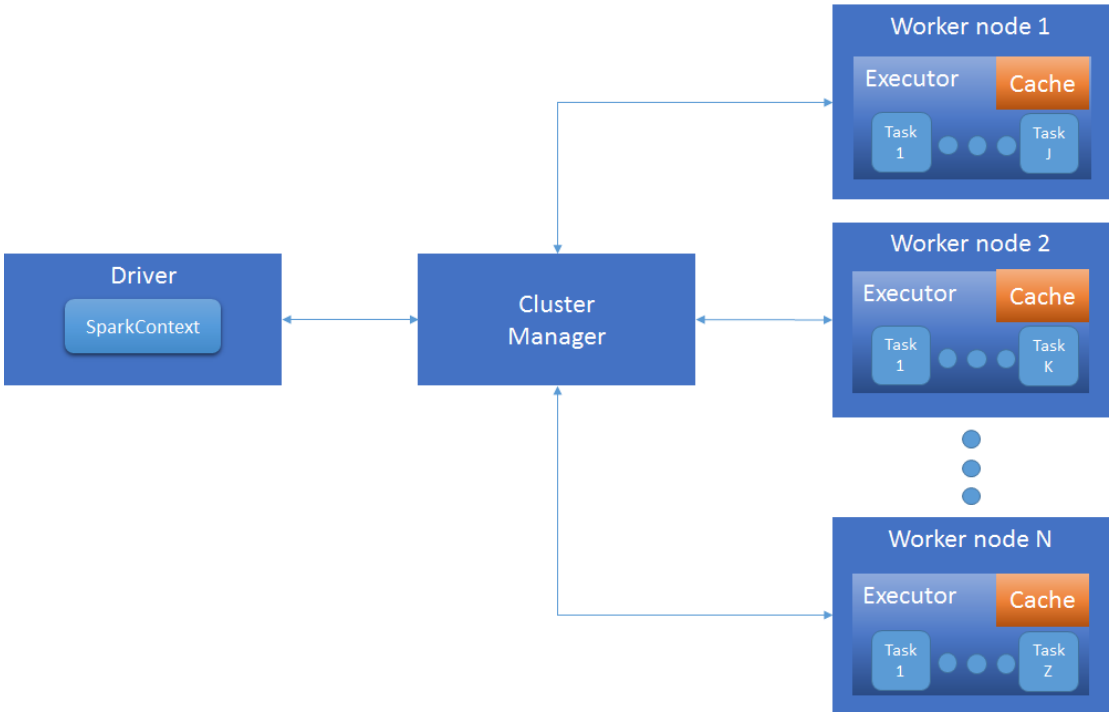


Figure 5.4: *Logical architecture of a Spark application running on a cluster with N nodes.*

mediation of *Cluster Manager*, to run the program code, including coordinating the execution of all parallel computations (defined in terms of RDDs transformations and actions) over the computational resources available on the cluster. Each partition of an RDD is assigned to an executor of a specific *Worker node*, which can in turn process the relative data using the set of operations expressed by the programmer. Each transformation or action takes the form of a *Task* in worker nodes, and it can be possibly run concurrently with other tasks in the same executor.

An example of using an RDD counting the word frequencies of a subset of a textual collection is the following:

[10] See `https://spark.apache.org/docs/latest/programming-guide.html#rdd-operations` for a comprehensive list of transformations and actions available in Spark.

```
JavaSparkContext sc = ...; // The context used to access Spark runtime.

// Create an RDD pointing to all text files contained
// in the input directory.
JavaRDD<String> textLines = sc.textFile("path/to/dir/*");

// Extract only the lines which contain the substring "apple"
// and cache the results.
JavaRDD<String> appleOnlyLines = textLines.filter(line ->
    line.toLowerCase().contains("apple")).cache();

// Count how many lines need to be processed.
long numLines = appleOnlyLines.count();

// Get the list of all distinct words, each filled with
// their number of occurrences.
List<Tuple2<String, Integer>> words = appleOnlyLines.flatMap(line -> {
  return Arrays.asList(line.toLowerCase().split(" "));
}).mapToPair(word -> {
  return new Tuple2<>(word, 1);
}).reduceByKey((v1, v2) -> v1 + v2).collect();

// Print results.
System.out.println("Number of lines processed: "+numLines);
for (Tuple2<String, Integer> item : words) {
  System.out.println("Word: "+item._1()+", Occurrences: "+item._2());
}
```

5.4 NLP4Spark, a textual indexing framework using Apache Spark

With this section, we move to "real" big data contexts where the problems to be solved need to process great amount of information and the solutions proposed should be able to exploit distributed computing to scale in computational power. We thus want to show how to exploit Spark engine as core data processor to build scalable text processing tools. In particular, we would like to build a text mining library similar in spirit to JATECS framework, which allow to index and preprocess textual data in a quick and convenient way. We designed NLP4Spark, a Scala library written on top of Spark which inherits the idea of index (same concept as in JATECS) as convenient data representation for its access and allows the developers to express in few lines of code programs able to quickly ingest and manipulate even great amount of textual contents. While designing the software, the guiding principle we followed was to provide a native API very simple and convenient, while at the same time avoiding to mask the Spark API. This design modality has the dual advantage of not imposing to programmers the necessity to learn a new complex API and to allow them to use their Spark skills directly toward the solution of the specific text mining problem. In the following we want to explore the design solutions we used for NLP4Spark, discussing the major issues encountered and how we solve them from a technical point of view. We will also show how we can use the exposed API to process textual data, providing several detailed code examples.

NLP4Spark uses the concept of index as a unique point of access to information indexed from a given data source. Our idea is to design a wrapper library API that is as

simple as possible, delegating to Spark's functional API the task to implement all the complex logic in terms of data processing expressed by developers in their programs. With software version 2.x, the Dataset API[11] has emerged as the more convenient interface to manipulate data in a Spark program. This API maintains all the advantages that RDDs have, such as strong type with compilation time error checks and possibility to use powerful lambda functions to manipulate data. Moreover, the API benefits from Catalyst's optimizer, the Spark query planner which transforms a pipeline of Spark calls (transformations and actions) into a running plan which will be executed efficiently on available computational resources. For these reasons, we thus adopted Dataset API as the way to manipulate the data inside our index structure.

An index structure (type TextualSparseDataIndex in our API) can be obtained from a custom data source by processing data through the static method *TextualSparseDataIndex.create* as shown in the following example:

```
// Create a Reuters21578 specific data source reader for training data.
val providerTraining = new
    Reuters21578DataSourceProvider("/Users/tiziano/Documents/datasets/reuters21578",
    DocumentSetType.TRAINING)

// Define feature extraction rules: a) puntuaction + english stopwords
// remover and b) characters 3-grams.
val extractionRules = Array(
DocFeatureExtractorPolicy(new PuntuactionRawFeatureExtractor(), Array(new
    EnglishStopwordRemover())),
DocFeatureExtractorPolicy(new CharacterNGramsFeatureExractor(3)))
val featureExtractorTraining = new StandardDocumentFeatureExtractor(extractionRules)

// Define the label extractor rule for multilabel documents.
val labelsExtractorTraining = new StandardLabelExtractor(";")

// Create a new training index based on previous defined
// data source and rules.
val trainingIndex = TextualSparseDataIndex.create(sparkSession,
providerTraining, featureExtractorTraining, labelsExtractorTraining)

// Save the index as JSON to file system or HDFS path.
trainingIndex.saveToJSON("/Users/tiziano/test/nlp4spark/reuters21578/training", true)

// Create a Reuters21578 specific data source reader for test data.
val providerTest = new
    Reuters21578DataSourceProvider("/Users/tiziano/Documents/datasets/reuters21578",
    DocumentSetType.TEST)

// Create for test data a feature extractor based on previous defined rules and
    already indexed training data.
val featureExtractorTest = new IndexDependantDocumentFeatureExtractor(trainingIndex,
    extractionRules)

// Define the label extractor rule for multilabel documents.
val labelsExtractorTest = new IndexDependantLabelExtractor(trainingIndex, ";")

// Create test index.
val testIndex = TextualSparseDataIndex.create(sparkSession,
providerTest, featureExtractorTest, labelsExtractorTest)
```

In the code we show how to use the framework to index a specific training data collection (Reuters21578 dataset[12] in this example) and how to use the training index

[11] https://spark.apache.org/docs/latest/sql-programming-guide.html#datasets-and-dataframes

[12] http://www.daviddlewis.com/resources/testcollections/reuters21578/

data (e.g. reuse features already extracted) to index test data. To create an index, we can specify in various ways how to extract relevant text features. In the example, we extract two types of features using a) a classic Bag-of-word approach with stop words removed and b) 3-grams characters. The test index has been instead created reusing the data and the feature extraction rules defined for training set. An index can be stored on HDFS in form of JSON or Parquet formats. Note the similarity with the relative JATECS code from Chapter 2.

Having access to an index instance, we can manipulate its data by mainly using the Scala methods reported on table 5.1.

Table 5.1: *Main data manipulation methods offered by our index API.*

METHOD NAME	AIM OF THE METHOD
transformByDocuments	Filter the data of the index by using a custom function which a) manipulate data through Dataset API and b) return manipulated data as a Dataset view in terms of documents.
transformByFeatures	Filter the data of the index by using a custom function which a) manipulate data through Dataset API and b) return manipulated data as a Dataset view in terms of features.
transformByLabels	Filter the data of the index by using a custom function which a) manipulate data through Dataset API and b) return manipulated data as a Dataset view in terms of labels.

The key idea is to provide data manipulation methods which work as code frameworks already working except for the part where the custom data processing logic must be expressed by programmers. This custom logic can be given by developers by using typed high order functions written using Spark operators. In this way, the only task mandatory for the programmers is to write the data processing logic, leaving all the rest of dirty work to the library code. To better understand how this works in practice, let's analyze the Scala code of *transformByDocuments* method:

```
override def transformByFeatures(tr: (TSparseDataIndex[TextualFeature]) =>
    Dataset[FeatureView],
  recomputeDocumentIDs: Boolean = false,
  recomputeFeatureIDs: Boolean = false,
  recomputeLabelIDs: Boolean = false): TSparseDataIndex[TextualFeature] = {

 // Check input parameters validity.
 Cond.verifyNotNull(recomputeFeatureIDs, "recomputeFeatureIDs")
 Cond.verifyNotNull(recomputeLabelIDs, "recomputeLabelIDs")
 Cond.verifyNotNull(tr, "tr")

 // Apply the custom data manipulation logic expressed with Spark operators.
 val dsFeatView = tr(this)

 // Transform the features view as reported by custom processing logic into
     documents view.
 var ds :Dataset[DocumentView[TextualFeature]] =
     featureViewToDocumentView(dsFeatView)

 // If required, recompute all unique IDs for documents, features and labels.
 if (recomputeDocumentIDs)
   ds = compressDocumentIDs(ds)
 if (recomputeFeatureIDs)
   ds = compressFeatureIDs(ds)
 if (recomputeLabelIDs)
   ds = compressLabelIDs(ds)
```

```
24
25    // Create a new index reflecting the new available data.
26    new TextualSparseDataIndex(ds)
27  }
```

The method works as a wrapper around the custom processing logic defined by "tr" function input parameter. After having verified the validity of input parameters (lines 7-9), we apply the custom logic defined by the programmer to current index data (line 12). The data returned by this function, expressed as a view over features, need to be transformed into an equivalent view over documents (line 15) because this is the default view used while instantiating a new index object (line 26). In the case the user has request to recompute unique IDs, we perform this task in lines 18-23.

After having seen the details on how the library works internally, we now want to show some examples of how to use the API and Spark functions to easily manipulate an index (i.e. create new data views on data subsets of the original index):

```
1  // Create a new index by filtering out all documents not having at least 10
2  // features which have a) number of occurrences > 3 or b) associated
3  // weight >= 0.15.
4  var interestingDocsIndex = trainingIndex.transformByDocuments(idx => {
5    val matchingDocs = idx.documents.filter(doc => {
6      val validFeats = doc.features.filter(feat => feat.occurrences > 3
7        || feat.weight >= 0.15).size
8      return validFeats > 10
9    })
10   return matchingDocs
11 })
12
13
14 // The same example as above, just more concise!
15 interestingDocsIndex = trainingIndex.transformByDocuments(idx =>
16   idx.documents.filter(
17   doc=>doc.features.filter(
18     feat=>feat.occurrences > 3 || feat.weight >= 0.15).size > 10
19   )
20 )
21
22
23 // Create a new index by filtering out all features not occurring at least
24 // in 50 distinct documents. We also request to recompute the IDs of
25 // documents and features, i.e. removing sparseness in IDs allocation.
26 interestingDocsIndex = trainingIndex.transformByFeatures(idx => {
27   idx.features.filter(feat=>feat.documents.size >= 50)
28 }, recomputeDocumentIDs = true, recomputeFeatureIDs = true)
29
30
31 // Create a new index by keeping only the top 10 leaf labels with the
32 // greater number of distinct features.
33 interestingDocsIndex = trainingIndex.transformByLabels(idx=>{
34   idx.labels.getLeafLabels().map(l => {
35     val numDistinctFeatures =
         l.documents.flatMap(doc=>doc.features).map(f=>f.featureID).distinct.size
36     (l, numDistinctFeatures)
37   }).sort(desc("_2")).limit(10).map(l=>l._1)
38 }, recomputeDocumentIDs = true, recomputeFeatureIDs = true, recomputeLabelIDs =
     true)
```

Every call to one of the NLP4Spark data manipulation methods takes as input a custom lambda function providing an index object ("idx" parameter). The *TextualSparseDataIndex* index type exposes the properties *documents*, *features* and *labels* to provide respectively a Spark's Dataset data view centered on documents, features and labels.

These properties are used as entry points to start manipulating data using Spark operators. In line 5, after having access to index documents, through Dataset *filter* transformation we filtered out all the docs not matching the condition of having at least 10 "good" features (lines 6-8). Here valid features are counted as the ones having number of occurrences greater than 3 or specific weight greater equals to 0.15. We can also note that the same example can be easily rewritten in a more concise and convenient way (lines 15-20). In lines 26-28 and 33-38 we provide two other examples using features and labels as base views for data manipulation. In the first example we simply compute a new index where we exclude all the features not comparing in at least 50 distinct documents. While creating the new index, we request also to recompute unique IDs both for features and documents in order to have a continuous distribution of IDs, often a practical requirement for optimized computations. The last example is more articulated and shows how use several Spark calls to perform easily complex computations. In line 34, we access to all leaf labels (i.e. labels not having children in a hierarchical taxonomy) and for each label we compute the number of distinct features occurring inside the documents belonged to that specific label (lines 35-36). The resulting couples (<label,feature_occurrences>) are ordered descending by feature occurrences and for the ordered list we take the first top 10 labels, which are the only one labels remaining in the new computed index (line 37).

As final code excerpt, we show the equivalent Scala example of data navigation as shown for JATECS in Listing 2.1:

```
val index: TextualSparseDataIndex = .... // Built from some input data source.
index.documents.foreach(doc=>{
  val documentName = doc.name

  // Extract all interesting features.
  val wantedFeatures = doc.features.filter(f=>f.occurrences >= 5 && f.weight > 0.15)
    .map(f=>f.name)

  // Print information about this document.
  println("*****************************")
  println("Document name: "+documentName);
  println("Most important features:"+wantedFeatures.mkString(","));
})
```

As we can see from previous examples, our library, thanks also to expressivity of Scala language and Spark APIs, results in a very simple and concise code, which can be formulated naturally by programmers and its comparable in clarity, if not better, with the corresponding sequential code expressed by using JATECS.

5.5 Use case: porting boosting algorithms to Apache Spark

As another interesting example of usage of Apache Spark as the core framework to develop text processing tools, we here describe how we used it to develop a distributed implementation of a popular family of ML algorithms called boosting. This type of iterative algorithms[13], based on supervised learning [191] and used to build automatic classifiers, works very well in textual domains and can benefit very much in terms of computational efficiency if parallelized with a effective tool such as Spark. Another reason why we targeted boosting as an interesting use case is, to the best of our knowl-

[13] https://en.wikipedia.org/wiki/Iterative_method

Input: A training set $Tr = \{\langle d_1, C_1\rangle, \ldots, \langle d_g, C_g\rangle\}$ where $C_i \subseteq C = \{c_1, \ldots, c_m\}$ for all $i = 1, \ldots, g$.

Body: Let $D_1(d_i, c_j) = \dfrac{1}{gm}$ for all $i = 1, \ldots, g$ and for all $j = 1, \ldots, m$

For $s = 1, \ldots, S$ do:

- pass distribution $D_s(d_i, c_j)$ to the weak learner;
- get the weak hypothesis $\hat{\Phi}_s$ from the weak learner;
- set $D_{s+1}(d_i, c_j) = \dfrac{D_s(d_i, c_j)\exp(-\Phi(d_i, c_j)\cdot\hat{\Phi}_s(d_i, c_j))}{Z_s}$

  where $Z_s = \sum_{i=1}^{g}\sum_{j=1}^{m} D_s(d_i, c_j)\exp(-\Phi(d_i, c_j)\cdot\hat{\Phi}_s(d_i, c_j))$

  is a normalization factor chosen so that $\sum_{i=1}^{g}\sum_{j=1}^{m} D_{s+1}(d_i, c_j) = 1$

Output: A final hypothesis $\hat{\Phi}(d, c) = \sum_{s=1}^{S}\hat{\Phi}_s(d, c)$

Figure 5.5: *The* ADABOOST.MH *algorithm.*

edge, the lack of specific implementations of these methods over Spark, in particular of the algorithms AdaBoost.MH and MP-Boost, two of the more effective variants when working with textual data.

In this section we provide a detailed description of AdaBoost.MH and MP-Boost algorithms together with technical details on how the code has been parallelized with Spark. Furthermore, we show a detailed experimentation of the implemented algorithms using a small computer cluster. All the developed code is available in open source form at address `https://github.com/tizfa/sparkboost`.

5.5.1 Boosting algorithms descriptions

An introduction to AdaBoost.MH

ADABOOST.MH [187] (see Figure 5.5) is a multi-label *boosting* algorithm, i.e. an algorithm that generates a highly accurate classifier (also called *final hypothesis*) by combining a set of moderately accurate classifiers (also called *weak hypotheses*). The input to the algorithm is a training set $Tr = \{\langle d_1, C_1\rangle, \ldots, \langle d_g, C_g\rangle\}$, where $C_i \subseteq C$ is the set of categories to each of which document d_i belongs.

ADABOOST.MH works by iteratively calling a *weak learner* to generate a sequence $\hat{\Phi}_1, \ldots, \hat{\Phi}_S$ of weak hypotheses; at the end of the iteration the final hypothesis $\hat{\Phi}$ is obtained as a sum $\hat{\Phi} = \sum_{s=1}^{S}\hat{\Phi}_s$ of these weak hypotheses. A weak hypothesis is a function $\hat{\Phi}_s : D \times C \rightarrow \mathbf{R}$. We interpret the sign of $\hat{\Phi}_s(d_i, c_j)$ as the prediction of $\hat{\Phi}_s$ on whether d_i belongs to c_j, i.e. $\hat{\Phi}_s(d_i, c_j) > 0$ means that d_i is believed to belong to c_j while $\hat{\Phi}_s(d_i, c_j) < 0$ means it is believed not to belong to c_j. We instead interpret the absolute value of $\hat{\Phi}_s(d_i, c_j)$ (indicated by $|\hat{\Phi}_s(d_i, c_j)|$) as the strength of this belief.

At each iteration s ADABOOST.MH tests the effectiveness of the newly generated weak hypothesis $\hat{\Phi}_s$ on the training set and uses the results to update a distribution D_s of weights on the training pairs $\langle d_i, c_j\rangle$. The weight $D_{s+1}(d_i, c_j)$ is meant to capture how effective $\hat{\Phi}_1, \ldots, \hat{\Phi}_s$ have been in correctly predicting whether the training document

d_i belongs to category c_j or not. By passing (together with the training set Tr) this distribution to the weak learner, ADABOOST.MH forces this latter to generate a new weak hypothesis $\hat{\Phi}_{s+1}$ that concentrates on the pairs with the highest weight, i.e. those that had proven harder to classify for the previous weak hypotheses.

In ADABOOST.MH the weak hypotheses generated by the weak learner at iteration s are decision stumps of the form

$$\hat{\Phi}_s^{(k)}(d_i, c_j) = \begin{cases} a_{0j} & \text{if } w_{ki} = 0 \\ a_{1j} & \text{if } w_{ki} = 1 \end{cases} \tag{5.1}$$

where t_k (called the *pivot term* of $\hat{\Phi}_s^{(k)}$) belongs to $\{t_1, \ldots, t_r\}$, a_{0j} and a_{1j} are real-valued constants., and $w_{ki} = 1$ (resp. $w_{ki} = 0$) means that term t_k occurs (resp. does not occur) in d_i. The choices for t_k, a_{0j} and a_{1j} are in general different for each iteration s, and are made according to an error-minimization policy as described by Algorithm 1 in Figure 5.6.

ALGORITHM 1. THE ADABOOST.MH WEAK LEARNER

1. For each term $t_k \in \{t_1, \ldots, t_r\}$, select, among all the weak hypotheses $\hat{\Phi}^{(k)}$ that have t_k as the pivot term, the one (indicated by $\hat{\Phi}_{best}^k$) for which Z_s is minimum.
2. Among all the hypotheses $\hat{\Phi}_{best}^{(1)}, \ldots, \hat{\Phi}_{best}^{(r)}$ selected for the r different terms in Step 1, select the one (indicated by $\hat{\Phi}_s$) for which Z_s is minimum.

Figure 5.6: *Algorithm 1: the* ADABOOST.MH *weak learner.*

Step 1 is clearly the key step, since there are a non-enumerable set of weak hypotheses with t_k as the pivot term. Schapire and Singer [186] have proven that, given term t_k and category c_j,

$$\hat{\Phi}_{best}^{(k)}(d_i, c_j) = \begin{cases} \frac{1}{2} \ln \frac{W_{+1}^{0jk}}{W_{-1}^{0jk}} & \text{if } w_{ki} = 0 \\ \frac{1}{2} \ln \frac{W_{+1}^{1jk}}{W_{-1}^{1jk}} & \text{if } w_{ki} = 1 \end{cases} \tag{5.2}$$

where

$$W_b^{xjk} = \sum_{i=1}^{g} D_s(d_i, c_j) \cdot [\![w_{ki} = x]\!] \cdot [\![\Phi(d_i, c_j) = b]\!] \tag{5.3}$$

for $b \in \{-1, +1\}$, $x \in \{0, 1\}$, $j \in \{1, \ldots, m\}$ and $k \in \{1, \ldots, r\}$, and where $[\![\pi]\!]$ indicates the characteristic function of predicate π (i.e. the function that returns 1 if π is true and 0 otherwise). For term t_k and for these values of a_{xj} we obtain

$$Z_s = 2 \sum_{j=1}^{m} \sum_{x=0}^{1} (W_{+1}^{xjk} W_{-1}^{xjk})^{\frac{1}{2}} \tag{5.4}$$

In practice, the value $a_{xj} = \frac{1}{2} \ln \frac{W_{+1}^{xjk} + \epsilon}{W_{-1}^{xjk} + \epsilon}$ is chosen in place of $a_{xj} = \frac{1}{2} \ln \frac{W_{+1}^{xjk}}{W_{-1}^{xjk}}$, since this latter may produce outputs with a very large or infinite absolute value when the denominator is very small or zero[14].

The output of the final hypothesis is the value

$$\hat{\Phi}(d_i, c_j) = \sum_{s=1}^{S} \hat{\Phi}_s(d_i, c_j) \tag{5.5}$$

obtained by summing the outputs of the weak hypotheses.

MP-Boost, an improved AdaBoost.MH variant with multiple pivot terms

MP-BOOST [67] is an improved version of ADABOOST.MH, which differentiate from the latter for the possibility to select multiple pivot terms (the best term for each category) at each iteration of the algorithm.

Looking at Equation 5.1 we may note that, at each iteration s, choosing a weak hypothesis means choosing (i) a pivot term t_k, *the same for all categories*, and (ii) for each category c_j, a pair of constants $\langle a_{0j}, a_{1j} \rangle$. Choosing the same term for all categories is clearly suboptimal, a better strategy is, at every iteration s, to choose a different pivot term $t_{\langle s,j \rangle}$ for each category c_j allowing the weak hypothesis to provide customized, improved treatment to each individual category.

In MP-BOOST the weak hypotheses generated by the weak learner at iteration s are thus of the form

$$\hat{\Phi}_s(d_i, c_j) = \begin{cases} a_{0j} & \text{if } w_{\langle s,j \rangle i} = 0 \\ a_{1j} & \text{if } w_{\langle s,j \rangle i} = 1 \end{cases} \tag{5.6}$$

where term $t_{\langle s,j \rangle}$ is the pivot term chosen for category c_j at iteration s.

To see how MP-BOOST chooses weak hypotheses of the form described in Equation 5.6, let us first define a *weak c_j-hypothesis* as a function

$$\hat{\Phi}^j(d_i) = \begin{cases} a_{0j} & \text{if } w_{ki} = 0 \\ a_{1j} & \text{if } w_{ki} = 1 \end{cases} \tag{5.7}$$

that is only concerned with classifying documents under c_j; a weak hypothesis is the union of weak c_j-hypotheses, one for each category $c_j \in C$. At each iteration s, MP-BOOST chooses a weak hypothesis $\hat{\Phi}_s$ as illustrated by Algorithm 2 in Figure 5.7.

Note the difference between Algorithm 1, as described in Section 5.5.1, and Algorithm 2; in particular, Step 2 of Algorithm 2 is such that weak c_j-hypotheses based on different pivot terms may be chosen for different categories c_j.

For reasons analogous to the ones discussed in Section 5.5.1, Step 1 is the key step; it is important to observe that $\hat{\Phi}^j_{best(k)}$ is still guaranteed to have the form described in Equation 5.2, since the weak hypothesis generated by Equation 5.6 is the same that Equation 5.1 generates when $m = 1$, i.e. when C contains one category only.

Note also that the "outer" algorithm of Figure 5.5 is untouched by our modifications, except for the fact that a normalization factor Z_s^j local to each category c_j is used (in

[14] In [188] the value for ϵ is chosen by 3-fold cross validation on the training set, but this procedure is reported to give only marginal improvements with respect to the default choice of $\epsilon = \frac{1}{gm}$, which we adopt in this work.

ALGORITHM 2. THE MP-BOOST WEAK LEARNER

1. For each category c_j and for each term $t_k \in \{t_1, \ldots, t_r\}$, select, among all weak c_j-hypothesis $\hat{\Phi}^j$ that have t_k as the pivot term, the one (indicated by $\hat{\Phi}^j_{best(k)}$) which minimizes

$$Z^j_s = \sum_{i=1}^{g} D_s(d_i, c_j) \exp(-\Phi(d_i, c_j) \cdot \hat{\Phi}^j(d_i)) \tag{5.8}$$

2. For each category c_j, among all the hypotheses $\hat{\Phi}^j_{best(1)}, \ldots, \hat{\Phi}^j_{best(r)}$ selected in Step 1 for the r different terms, select the one (indicated by $\hat{\Phi}^j_s$) for which Z^j_s is minimum;
3. Choose, as the weak hypothesis $\hat{\Phi}_s$, the "union", across all $c_j \in C$, of the weak c_j-hypotheses selected in Step 2, i.e. the function such that $\hat{\Phi}_s(d_i, c_j) = \hat{\Phi}^j_s(d_i)$.

Figure 5.7: *Algorithm 2: the* MP-BOOST *weak learner.*

place of the "global" normalization factor Z_s) in order to obtain the revised distribution D_{s+1}; i.e.

$$D_{s+1}(d_i, c_j) = \frac{D_s(d_i, c_j) \exp(-\Phi(d_i, c_j) \cdot \hat{\Phi}^j(d_i))}{Z^j_s}$$

The main difference in the algorithm is thus in the "inner" part, i.e. in the weak hypotheses that are received from the weak learner, which now have the form of Equation 5.6, and in the way they are generated.

5.5.2 Implementation details of boosting algorithms over Spark

In this section we will give some details about the Spark implementations of boosting learners and classifiers. The logical architecture of the solution is presented in Figure 5.8. In the figure the names "host", "task" and "driver" have the same meaning as those used to describe the generic Spark architecture presented in Figure 5.4.

As shown in Figure 5.5, the boosting learning algorithms are clearly iterative methods, a type of algorithms that Spark can address very well. The trainer implementation we made for Spark uses basically the same sequential code for the main external loop which generates the sequence of weak learners and update the learning model (module *Learner* in Figure 5.8). What changes is that now we have parallelized the code that generates the weak hypothesis. Being in the context of supervised NLP with sparse feature representations, we assume that the number of features extracted from a textual dataset is generally greater, also of several order of magnitude, than the number of documents. For this reason we decided to exploit parallelism over the features by creating a dataset RDD operating at feature level. This representations allow us to divide the input data into n partitions, where each one is analyzed independently of each others. To implement the weak learner in the way described in algorithms 1 and 2, we use a *map-reduce* approach[15], as exemplified in the following snippet of pseudo-code:

[15]In Spark API it is available the *map* transformation and the *reduce* action which have similar meaning as the corresponding map-reduce operations you perform on Hadoop.

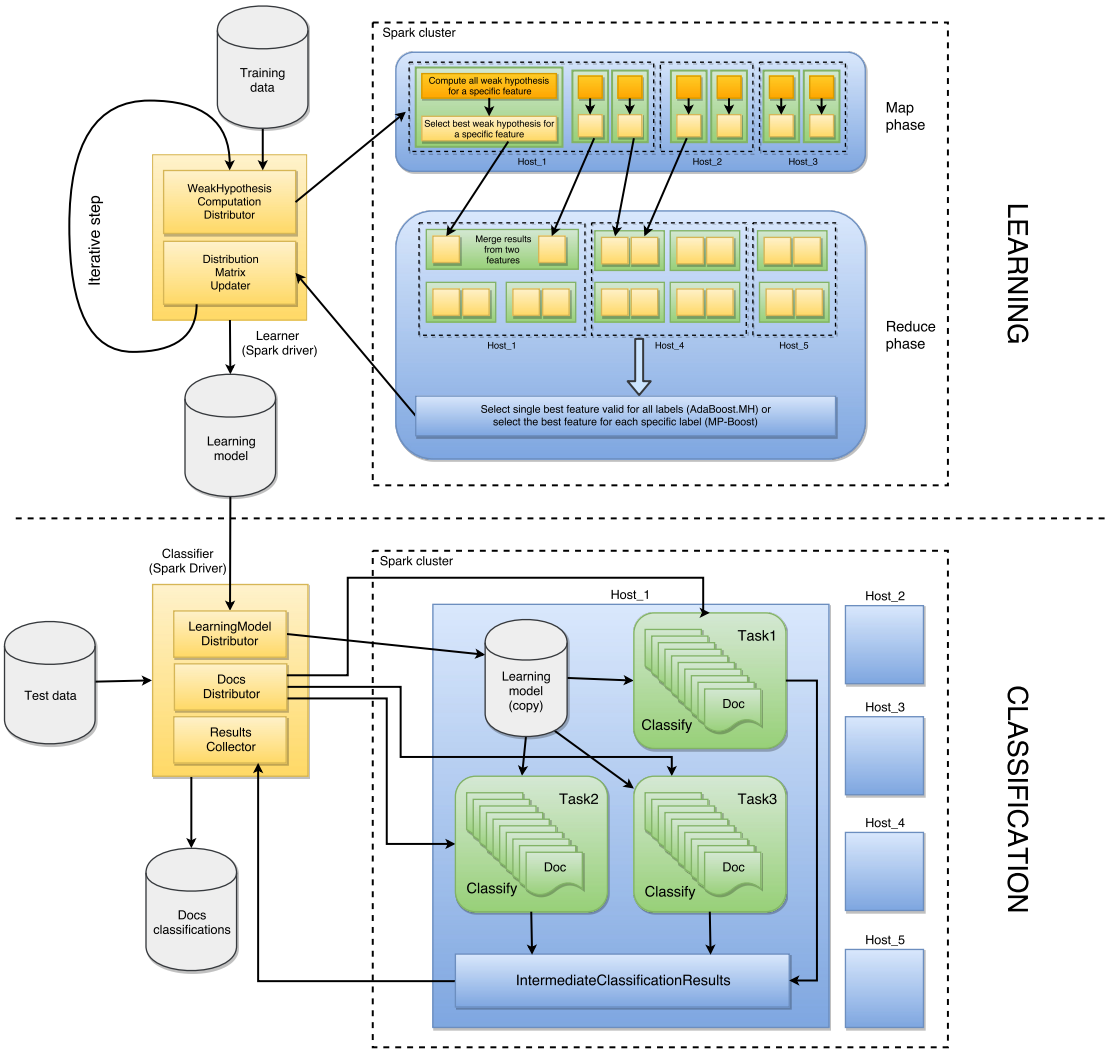


Figure 5.8: *Logical architecture of the Spark implementation of boosting learner and classifier. This schema can be applied without modifications both to AdaBoost.MH and MP-Boost algorithms.*

```
// Input RDD used to parallel processing features. Each item in the
// RDD is a feature representation that gives access to all
// information needed to process that feature independently from
// the others.
RDD[FeatureDocuments] rdd = ...

// MP-Boost: get best weak hypothesis for each category.
weakHypothesis = rdd.map(feature -> {
  // Given a feature, we compute its best weak hypothesis,
  // and we pass that result to the reduce() code.
  fwh = computeBestWeakHypothesis(feature)
  return fwh
}).reduce(fwh1, fwh2 -> {
  // For each label, we compare two feature weak hypothesis
  // and we choose that it gives us the minimum Z_s value.
  wh = []
  for (int i = 0; i < numLabels; i++) {
```

```
    if (fwh1.getZ_s(i) < fwh2.getZ_s(i) {
      wh[i] = getWeakHypothesisData(fwh1, i)
    } else {
      wh[i] = getWeakHypothesisData(fwh2, i)
    }
  }
  return wh
})
```

In the *map* function, for both algorithms, we compute for each feature the corresponding best weak hypothesis $\hat{\Phi}_{best}^{(k)}$ according to equation 5.2. In the *reduce* function, we differentiate the code for the 2 algorithms by a) computing the best feature among all categories for AdaBoost.MH and b) computing the best feature for each category in the case of MP-Boost (as shown in the above pseudo-code). In Figure 5.8, each green block of map-reduce phases is a single Spark task which which can be executed in parallel on one of the nodes (typically a multithread machine) of the available cluster.

The classifier code is the same for both algorithms because we use the same data format for saving the model parameters. Here it is the corresponding pseudo-code adhering to equation 5.6:

```
// The set of input documents to be classified.
RDD[MultilabelPoint] rdd = ...

// The learned boosting model.
BoostingWeakHypothesis model = ...

// Classify each document in parallel through the RDD map() function.
rdd.map(doc ->{
  DocScore docScore = ....
  // For each computed weak hypothesis..
  for (wh in model) {
    // For each label learned...
    for (lab in model.labels()) {
      if (doc.contains(wh.feature()))
        // Sum positive contribution.
        docScore.sum(wh.c1(lab))
      else
        // Sum negative contribution.
        docScore.sum(wh.c0(lab))
      }
    }
  }

  // docScore has now stored all scores for all labels. If the
  // score for a specific label j is > 0 (<= 0) then the
  // document is deemed to belong (to not belong) to code j.
  return new DocResults(doc, docScores)
})
```

As shown in Figure 5.8, the code exploits the implicit data parallelism by dividing the set of documents to be classified into several partitions, each one processed independently by a specific Spark task. As optimization, it is also important to note that the learned model is copied to each single node of the cluster just one time and reused in read-only mode by each task executed in parallel over that specific node. The docu-

ments scores are computed by summing all contributions (negatives and positives) coming from every learned weak hypothesis. Depending on the built classifier, every weak hypothesis could contribute to the document differently for each label (MP-BOOST) or give the same contributions independently of the selected label (ADABOOST.MH).

5.5.3 Experiments

In this section we will show the scalability results obtained by the implemented algorithms measured over a small computer cluster. Related to learning methods, we focused just on algorithm MP-BOOST because the two learning algorithms are very similar in terms of code, sharing the same ideas for parallelization and having negligible differences in terms of execution cost.

We have setup a Spark installation on a dedicated cluster built using 3 VirtualBox virtual machines and using the popular software Cloudera Manager[16] as cluster manager. Each machine has been allocated on a different physical Ubuntu server. Each server has different characteristics from each other in terms of CPU, RAM, and hard disk capacity. In particular the CPUs, although all having 6 physical cores (12 virtual cores with hyperthreading[17] enabled), are different Intel models, with notable differences in terms of speed clocks and processor architectures. Moreover, to the best of our knowledge, VirtualBox does not allow to force assignment of guest machine virtual cores to host physical cores, that is it does not guarantee the a machine virtual core will be always assigned to an host real core. Therefore, we have setup each VirtualBox virtual machine to have the same quantity of RAM (16 GB) and storage capacity (100 GB), and we assigned to each machine 12 virtual cores, enabling us the possibility to explore how the scalability is influenced by using real physical cores together with hyperthreading technology. Finally, the machines are interconnected by a 100 Mbit wired network shared with the 3 physical servers.

As performance measure for scalability, we used the throughput speedup formula[18], specifically considering execution time of sequential code vs. parallel code, as given by the following equation:

$$Speedup = \frac{T_s}{T_p} \tag{5.9}$$

where T_s is execution time of sequential code and T_p is the execution time of parallel code. In our experiments, we both considered vertical and horizontal speedup configurations. In vertical scalability tests, we fixed a single machine (the Spark master machine, which runs on the fastest physical server) and progressively test the algorithms using an increasing number of cores. In horizontal scalability tests, we test the algorithms by using an increasing number of machines (always using all available cores in each machine).

We tested the scalability performance of the algorithms using 2 different datasets, both available in LibSvm format at address `https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html` . The first, *kdd2010-algebra*, defines a binary classification problem using data coming from KDD Cup 2010, an educational data mining competition. This dataset is composed of a quite huge training set (8,407,752

[16] `http://www.cloudera.com/`
[17] `https://it.wikipedia.org/wiki/Hyper-Threading`
[18] `https://en.wikipedia.org/wiki/Speedup`

documents and 20,216,830 distinct features), perfect to be used to measure horizontal learning scalability in a small cluster like our. The same training set has also been used to measure vertical and horizontal classification scalability. This task, being highly parallel in the number of documents but with a small computational cost associated with a single document classification, is an interesting case to see how much on approach based on vertical scalability could compare in terms of speedup to an approach based on horizontal scalability. The second dataset we used is *news20.binary*, another binary classification problem which is quite small in terms of number of documents (19,996) but with an high number of distinct features (1,355,191). In this case, being a dataset learnable in reasonable time on a single machine, we try to compare the differences in terms of learning speedup for horizontal and vertical scalability. At the same time, we will show results on vertical classification scalability measured on a single machine. We also tried to scale the algorithm horizontally but we basically obtained no improvements compared to using a single multi-core machine (more on this later).

We made some assumptions for this experimentation. To enforce fair comparisons among the tested configurations , we fixed the number of partitions inside Spark RDDs data to be always equals to the maximum number of cores available in the configurations tested. For vertical scalability experiments the number of partitions was fixed to 12, while for horizontal ones we fixed 36 as the number of partitions. This guarantees that the RAM usage will be the same in every experiment of the same type, while at the same time it enforces the maximum usage of the available computational resources defined in the considered experiment. Moreover, we have used Spark with its own default parameters, without trying to optimize some specific aspects of the runtime[19] while running the tests. We avoided this time-consuming step because our main goal was to show that Spark can be quickly used as a tool to successfully parallelize with good performance a machine learning algorithm like boosting. Lastly, to try to obtain fair results from experimentation, we ensure that at the time we run experiments each physical server was actively running just the VirtualBox virtual machine without any other heavy user applications in execution.

The experimentation results are outlined on Table 5.2 and the corresponding speedup graphs are shown from Figure 5.9 to Figure 5.14.

Let's start analyzing the results obtained using the smaller dataset (news20.binary). Related to vertical learning (Figure 5.12), the algorithm scales reasonably well, even showing superlinearity scalability in the case of using all 12 virtual cores and in relation to the number of real cores available on the server the virtual machine is running on. In the case of horizontal learning (Figure 5.13), adding more machines implies that the speedup just improves marginally, without giving significant performance advantages if compared to the quantity of added computational resources. These last results could be caused by several reasons. First, passing to configurations with more than 1 machine involved in the experiment, implies that at each iteration of the algorithm the network latency (limited by 100 MBit speed) to distribute the working loads (distribution matrix and possibly other used data structures) and to retrieve the results (the set of weak hypothesis) has probably a not negligible negative impact on the performance. Moreover, by having cluster's machines with different computational power in a context of an iter-

[19] Often, to maximize application performance, one need to tune several runtime settings which have a great impact on application performance, e.g. tune how aggressively the garbage collector should run, type of serializer to use for managing temporary data, etc.

	KDD2010-ALGEBRA	NEWS20.BINARY	KDD2010-ALGEBRA	NEWS20.BINARY
# CORES	TIME (S)	TIME (S)	TIME(S)	TIME (S)
	VERTICAL LEARNING		VERTICAL CLASSIFICATION	
1	—	1,015	517	26
2	—	510	275	20
4	—	321	164	19
6	—	247	117	18
8	—	201	106	16
10	—	181	100	15
12	—	163	98	14
	HORIZONTAL LEARNING		HORIZONTAL CLASSIFICATION	
12	5,765	163	98	—
24	4,874	150	51	—
36	4,513	135	47	—

Table 5.2: *Performance results for learning and classification tasks over the 2 used datasets.*

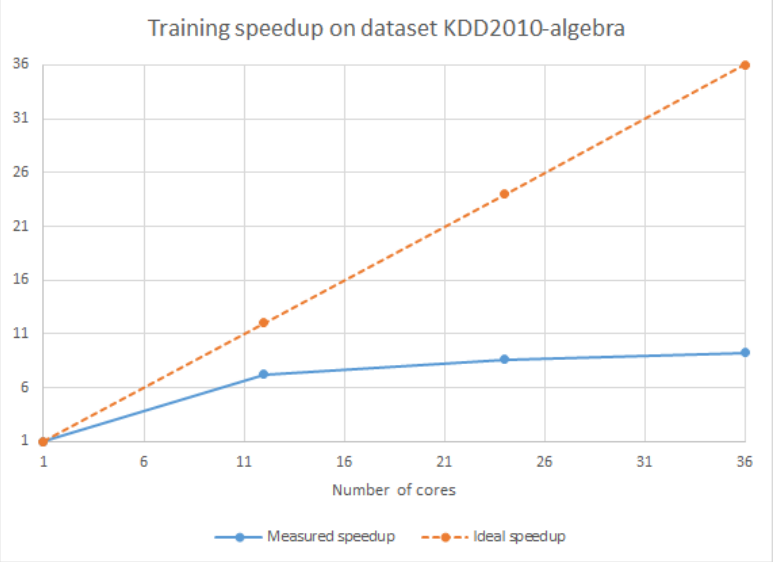


Figure 5.9: *Learning horizontal speedup on dataset KDD2010-algebra.*

ative algorithm[20], the slowest machine become a bottleneck in the computation slowing down all (or part of) the cores of other machines to wait for its completion before starting to analyze new data. Another possible performance issue is due to the smaller size of each single RDD partition because the entire dataset is divided into 36 partitions instead of 12 (the case of vertical learning). This determines that the computational

[20] At the end of each iteration, there is a synchronization barrier with the aim of waiting for all workers to have processed all data.

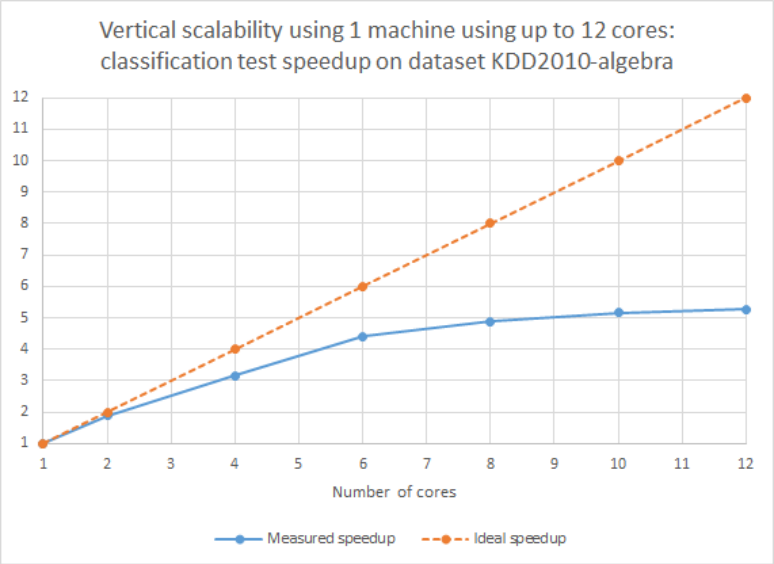


Figure 5.10: *Classification vertical speedup on dataset KDD2010-algebra.*

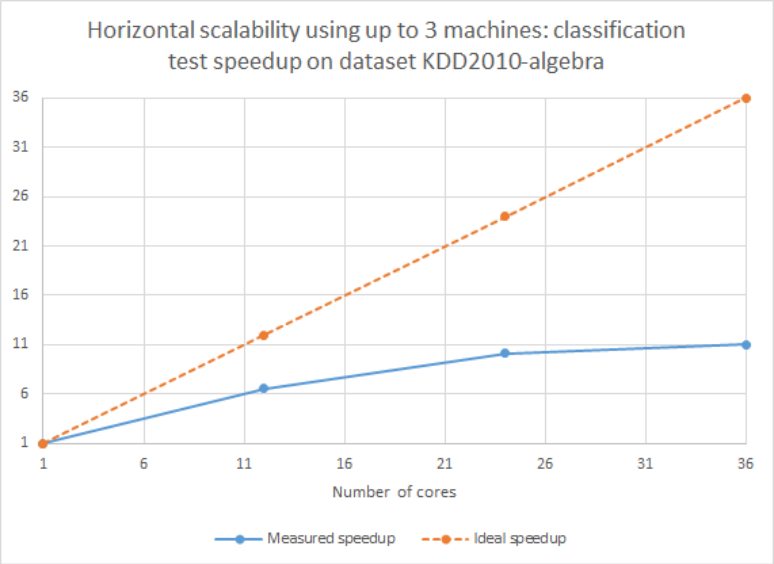


Figure 5.11: *Classification horizontal speedup on dataset KDD2010-algebra.*

load for each partition (core) is greatly reduced with more time spent by the system to handle workers data allocation/monitoring, determining suboptimal resources utilization (e.g. excessive context switches among threads). The scalability is not shining also if considering vertical classification speedup (Figure 5.14) but in this case this is justified by the small number of documents to be classified: in fact, even sequential code is able to classify all 19,996 documents in a very small time (about 26 seconds). In these conditions probably the time spent by Spark handling data distribution among workers is much higher than the time spent by a single worker processing its data.

The results on the biggest dataset (kdd2010-algebra) are similar in trends to those obtained on news20.binary data only just slightly better, justified by the fact that the documents to process are now many more than before. On horizontal learning experi-

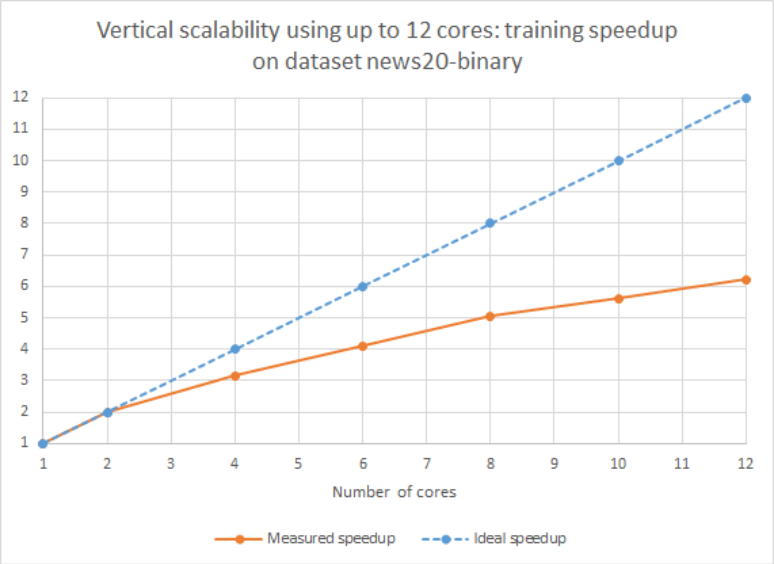


Figure 5.12: *Learning vertical speedup on dataset news20.binary.*

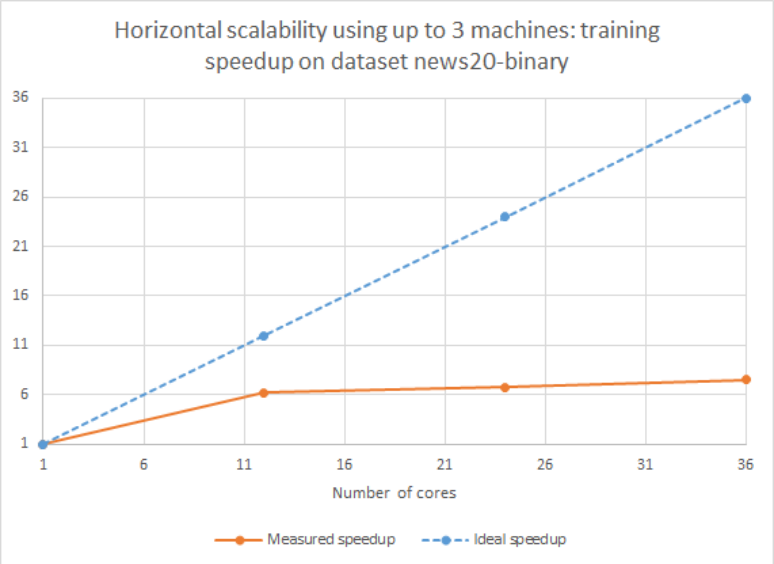


Figure 5.13: *Learning horizontal speedup on dataset news20.binary.*

ments (Figure 5.9), the algorithm scales better and all the weak points found previously in the other dataset in the same configuration are in this case attenuated by notable growth of the time spent by each worker for real data processing. The same type of improvements can be seen also for vertical and horizontal classification configurations (Figures 5.10 and 5.11). Here the algorithm achieves interesting performance, especially considering the type of Spark installation used (virtual machines communicating with a slow network connection) and no fine tuning of Spark runtime.

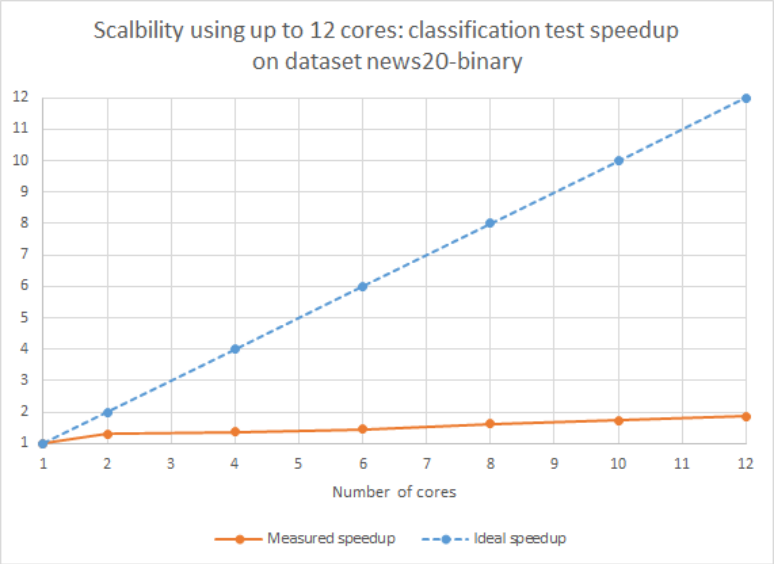


Figure 5.14: *Classification speedup on dataset news20.binary.*

5.6 Related work

In the last years, with the data explosion due to the increasing Internet popularity and the availability of connected smart devices, several general purpose software and specialistic ML tools for big data processing have emerged in the market. Among general purpose tools, we can identify three possible macro types of processing: batch, streaming and hybrid (performing both batch and streaming processing).

In batch processing contexts, the oldest and more consolidated big data tool available is the software open source Apache Hadoop [211] which includes a software ecosystem (HDFS, Yarn, etc.) often used at the core of a typical big data software stack. The programming model offered by Hadoop is based on map-reduce paradigm [57], which allows to solve a problem by processing data in two successive steps called *map* and *reduce*. In map phase the original problem data is transformed into a series of couples key-value while in the reduce phase all values sharing the same key are in same way combined together. This approach is conceptually very simple but it allows anyway to express and solve several practical problem types. However, sometime instead, it is not possible to use Hadoop conveniently because a) the problem to be solved can not be expressed (or can be expressed in suboptimal ways) through map-reduce or b) it is very inefficient to use Hadoop for certain problem types (e.g. iterative algorithms) because it uses heavily disk I/O (also for temporary data) during data processing. Microsoft Dryad [106] is another general purpose batch-oriented engine for coarse-grain data-parallel applications. It uses the concept of DAG dataflow to define computational graphs where the vertexes implement the various data processing logics and the arcs are the communication channels linking the processing nodes. Dryad runtime has been designed to work well on several different runtime configurations, ranging from multithread single machines up to computer clusters. Dryad had the limit to be a closed-source software, not available for public download. Apache Tez [180] on the other hand is an open source software which implements a lot of ideas proposed in Dryad paper.

FlumeJava [38] is another parallel data processing tools which abstracts the usage of an existing big data engine (e.g. Hadoop) and provides an API to exploit data parallelism through a set of predefined operators on read-only data collections. To perform efficient parallel computations, the tool works in a lazy way, abstracting the programmers the way on how a specific operation is done and generating execution plans specific and optimized for used runtime engine.

Another interesting research direction that was recently taken is the study of technologies related to streaming data processing. Several DSPEs (Distributed Streaming Processing Engine) have been proposed in the last years, with some having gained a good popularity among developers and have been used also to develop very complex and near-real time data processing software products. Apache Storm [202] is probably the most popular pure DSPE available on the market. The applications over Storm are expressed by designing topologies, real-time computations represented in the form of computational graphs which will be executed on Storm clusters. A topology is characterize by the presence of nodes of type *spouts* (emitting data tuples to be processed) and nodes called *bolts* which process streams of these tuples according to the logic defined by programmers. The data flows from a node to others nodes through communication channels, each one characterized by a specific policy defining data distribution among all involved workers (e.g. broadcast, round-robin, etc). Storm guarantees at-least-once delivery semantic[21] and it is suitable for contexts where low-latency, scalable and fault-tolerant processing are very important requirements. Another streaming software also featuring these last features is Apache Samza [164]. The processing techniques adopted by the software are similar to Storm but in addition Samza supports specifically stateful data processing in order to quickly recover from failure situations. Yahoo Samoa [155] in another interesting meta-DSPE which is able to work with different existent streaming engines (currently it supports Apache Storm, Apache Flink, and Apache Samza). The software exposes a generic API based on the same concepts as used in Storm (e.g. topologies for the computational graphs, processors for workers, streams for communication channels, etc.) but it also provides several ready machine learning algorithms (e.g. bagging, vertical Hoeffding tree classifier, clustering, etc.) to be used for the resolution of typical data mining problems.

Another very interesting class of DSPEs is that characterized by systems using an hybrid data processing approaches, providing an API suitable for both batch and streaming processing. Some of these softwares provide an unified API for transparently processing data in streaming or batch mode. This characteristic is very important because allow to use such tools effectively to design software solutions based on very complex architectures (e.g. lambda architecture [99, 125]). Apache Spark [219], as already amply described in section 5.3, is the leading framework of this type on the market. It provides not only an almost unified API for batch and streaming processing but also a big software ecosystem built around the tool which allow to work easily with SQL data sources (SQL and Dataframe lib), machine learning problems (MLlib lib), graph analysis (GraphX lib) and a myriad of other interesting uses cases available through numerous third-party integrations (e.g. Elasticsearch, HBase, etc.). Another very interesting software is Apache Flink [35] which is similar in terms of available features to Spark. In particular, it offers the possibility to work with streaming and

[21] `https://goo.gl/UQemSY`

batch contexts with slightly different APIs and, like Spark, it provides native high level libraries to work with ML algorithms (FlinkML lib), SQL data sources (Table lib) and graph processing (Gelly lib). The big difference between Spark and Flink is that this last engine is a "real" streaming engine, allowing to process data as it arrives in near real-time. The batch processing capabilities are built over streaming ones, aggregating data stream into data windows seen as bounded datasets. Spark on the other hand takes the opposite approach being mainly a batch data processor and building streaming capabilities by grouping data in timed windows (micro-batches). Consequently Spark is only suitable in application contexts where the latency required for data processing is not under the second while Flink is more effective when very low latency (order of few milliseconds) is a very stringent requirement for the designed applications. Moreover, Flink provides a more advanced management of time windows over working packages, allowing to easily analyze out-of-order messages thanks to advanced techniques like watermarks [6] and differentiating between event and time processing timestamps for processed data. Spark instead can count on a more big software ecosystem and a greater support from community developers, being a more mature product than Flink. Another interesting approach for hybrid data processing has been proposed by Google with its Dataflow model [7] which it is currently implemented in an open source software called Apache Beam[22]. This software provides an unified model for defining both batch and streaming data-parallel processing pipelines. The most interesting feature of Beam is that abstract the runtime engine where effectively the defined program's dataflow will be executed. Indeed it currently transparently supports the execution of programs on four different distributed backends: Spark, Flink, Apache Apex[23] and Google Cloud Dataflow[24]. The programming model provided by Beam try to provide the best concepts and features available from used engines. This gives to developers a powerful API that lets them to concentrate on how to represent and process the data they are interested on, leaving to specific runtime used all the cumbersome optimizations necessary for run efficiently the code on the target runtime architecture.

On the market that are several ML big data tools available which can be used to facing on typical data mining problems. Unfortunately no one of these software packages are ready to work directly into textual domains. This hard task is leaved to programmers, which need to integrate the various technologies together in order to provide effective solutions. MLlib[25] is a ML library built over Apache Spark which offers common ML and statistical algorithms. It includes components to perform basic statistics (correlations, stratified sampling, hypothesis testing, etc.), classification and regression (SVM, logistic regression, Naive Bayes, etc.), collaborative filtering, clustering (k-means, LDA, etc.), and other basic methods for feature extraction and transformation. As most of the other tools we have cited, it is a generic ML tool, so it does not provide algorithms targeting specifically NLP and text analytics. H_2O[26] is another generic fast and scalable ML software suite which puts a lot of emphasis in the integration with other popular third party software (mainly Apache Spark and Apache Hadoop) and the possibility to use its features in several different ways (from Java, R,

[22] https://beam.apache.org/
[23] https://apex.apache.org/
[24] https://cloud.google.com/dataflow/
[25] https://spark.apache.org/mllib/
[26] https://h20.ai/

Python, Excel, etc.). It provides several common supervised and unsupervised learning methods (generalized linear modeling, distributed random forest, Naive Bayes, k-means, etc.) and a deep learning classifier/regressor based on an optimized distributed implementation of a classic feed forward neural network. Another generic ML software quite popular in the community is Apache Mahout[27]. It provides implementations of distributed or otherwise scalable machine learning algorithms focused primarily in the areas of collaborative filtering, clustering and classification. The software initially was implemented on top of Hadoop map-reduce, but in the last two years, thanks to the emerging of new fast data processing software like Apache Spark or Apache Flink, several algorithms have been ported to these new systems.

5.7 Conclusions

In this chapter we faced on the problem of how to design effective textual processing tools able to work in big data application scenarios. We firstly started from a scenario where we have to process many information but we are limited in terms of computational resources available, typically a powerful but single multicore machine. In this scenario, it was not clear if an existing big data processing software could provide near-optimal performance or a specific designed multithread solution could perform much better. We thus proposed Processfast, a Java/Groovy framework that aims at providing a seamless integration of different parallel computing models, by combining the functionalities provided by different parallel processing frameworks into an homogeneous API. We described the logical architecture of the software, showing some examples of its API usage to implement programs exploiting data or task parallelism. Subsequently we showed a quick experimentation comparing the performance of Processfast with Hadoop and Spark in a relatively small data processing task. The obtained results suggest that Spark is very solid product providing very good performance also in specific contexts like this, thus making useless the effort required to develop new products like Processfast. Basing on these results, we moved to distributed data processing adopting Spark as the core technology to provide scalable and fault-tolerant text processing solutions. We designed a new library called NLP4Spark which aims to provide a simple API to quicky ingest and index textual data by adopting the concept of index as used by JATECS in Chapter 2 and by using powerful Spark APIs to manage data. We motivated the technical solutions adopted in the software and we showed several code examples of API usage for data ingestion, data preprocessing and data navigation cases. These examples demonstrated that the library, thanks also to Scala and Spark APIs, allows the programmers to write in few lines of code very expressive and effective textual processing programs. Finally, as another interesting usage of Spark, we analyzed in depth how to port two popular boosting algorithms implementing text classifiers (AdaBoost.MH and MP-Boost) from a sequential software version to a distributed one. After having described the characteristics of the algorithms and how to parallelize the code exploiting Spark API, we provided an extensive experimentation of the algorithms using a computer cluster. We thus verified that such machine learning methods can obtain good benefits in terms of performance and data scalability if they are parallelized with effective tools like Spark.

[27] `https://mahout.apache.org/`

CHAPTER 6

CrisMap: A Big Data Crisis Mapping System based on Damage Detection and Geoparsing

ABSTRACT

In this chapter we continue exploring Scenario 2 described in section 1.3.2 by showing a complex use case which requires to develop a big data solution deeply integrating several sophisticated software modules together, including a specific one covering an advanced automatic text classifier. The shown solution also proposes an interesting software architecture which allows the system to be easily maintainable, scalable and fault-tolerant.

Natural disasters, as well as man made disasters, can have a deep impact on wide geographic areas, and emergency responders can benefit from the early estimation of emergency consequences. This chapter presents `CrisMap`*, a Big Data crisis mapping system capable of quickly collecting and analyzing social media data.* `CrisMap` *extracts potential crisis-related actionable information from tweets by adopting a classification technique based on word embeddings and by exploiting a combination of readily-available semantic annotators to geoparse tweets. The enriched tweets are then visualized in customizable, Web-based dashboards, also leveraging ad-hoc quantitative visualizations like choropleth maps. The maps produced by our system help to estimate the impact of the emergency in its early phases, to identify areas that have been severely struck, and to acquire a greater situational awareness. We extensively benchmark the performance of our system on two Italian natural disasters by validating our maps against authoritative data. Finally, we perform a qualitative case-study on a recent devastating earthquake occurred in Central Italy.*

6.1 Introduction

Computational solutions capable of overcoming societal sustainability challenges have always been looked at with great interest from both Academia and practitioners [208]. In recent years, one of the fields that has attracted more attention is the one related to the exploitation of user-generated information for disaster management [11]. Within this context, Social Media (SM) data revealed to be particularly valuable in the aftermath of those events, typically natural and man-made disasters, which trigger massive participation of affected communities in sharing time-sensitive and actionable information [15, 84]. Both types of disasters require a timely intervention by emergency responders, who are in charge of providing support and relief to the affected population. The scarcity of resources, such as human resources, food, and water influences significantly the time required for the intervention, affecting the effectiveness of rescue operations. In many practical situations, the scarcity of key resources – temporal, economic, and human resources above all – imposes dire limitations to the extent and the effectiveness of the emergency management process. For this reason, tools capable of supporting resource allocation and prioritization can have a significant impact towards the effectiveness of emergency management operations. Among these tools there are the SM-based crisis mapping systems, which increase situational awareness by enabling the real-time gathering and visualization of data contributed by many SM users. Such type of system is a platform able to collect text and multimedia content from a variety of sources, such as Twitter and Facebook, to analyze and aggregate collected data, and to visualize relevant facts on a map. Notably, during many recent disasters, civil protection agencies developed and maintained live Web-based crisis maps to help visualize and track stricken locations, assess damage, and coordinate rescue efforts [150].

Indeed, recent work demonstrated the possibility to create crisis maps solely using geolocated data from SM, to better understand and monitor the unfolding consequences of disasters [12, 90, 150]. All these SM-based crisis mapping systems face the fundamental challenge of *geoparsing* the textual content of emergency reports in order to extract mentions of places/locations, thus increasing the number of messages to exploit. Geoparsing involves binding a textual document to a likely geographic location which is mentioned in the document itself. State-of-the-art systems, such as [150], perform the geoparsing task by resorting to a number of preloaded geographic resources containing all the possible matches between a set of place names (toponyms) and their geographic coordinates. This approach requires an offline phase where the system is specifically set to work in a geographically-limited region. Indeed, it would be practically infeasible to load associations between toponyms and coordinates for a wide region or for a whole country. Moreover, not all geolocated data is useful towards understanding the severity of the emergency and, indeed, only a small fraction of messages convey information about the consequences of the emergency on communities and infrastructures. Current crisis mapping systems typically detect the most stricken areas by considering the number of messages shared and by following the assumption that more emergency reports equals to more damage [150, 210]. Although this relation exists when consid-

ering densely and uniformly populated areas [136], it becomes gradually weaker when considering wider regions or rural areas. These challenges, related to the detection of damage and to geoparsing, are reflected by the current limitations of state-of-the-art SM-based crisis mapping systems [12].

Here, we propose solutions to overcome the main drawbacks of current state-of-the-art crisis mapping systems. Our proposed system exploits both situational assessments and position information contained in tweets produced during an emergency. One interesting novelty of our approach is the analysis of emergency-related tweets from a twofold perspective: (i) a damage detection component exploits word embeddings and a SVM classifier to detect messages reporting damage to infrastructures or injuries to the population; (ii) a message geolocation component performs the geoparsing task by exploiting online semantic annotation tools and collaborative knowledge-bases such as Wikipedia and DBpedia. Information extracted by the damage detection and the message geolocation components are combined together to produce interactive, Web-based crisis maps.

Contributions. We describe `CrisMap`: a system capable of producing crisis maps in the aftermath of mass emergencies by simultaneously adopting word embeddings for tweet filtering and classification, and by exploiting semantic annotators for tweet geoparsing. In particular:

- we propose an architectural solution for crisis mapping, based on scalable and resilient Big Data technologies;
- we address the problem of damage detection in SM messages;
- we compare a damage detection approach based on natural language processing (NLP) versus one based on word embeddings (WE), highlighting the advantages of WE towards language independence and fast processing;
- we propose and benchmark a geoparsing technique based on readily-available semantic annotators;
- we validate the reliability of the crisis maps generated by our system against authoritative data for 2 past emergencies;
- we perform a qualitative case-study on a recent severe earthquake in Italy.

The deployment of our proposed `CrisMap` system can help to quickly map damage scenarios in order to concentrate rescue efforts and organize a prompt emergency response.

6.2 System architecture

The software architecture of our crisis mapping system is presented in Figure 6.1. As with any system that needs to cope with the massive amount of data collected from social networks, special requirements are imposed in the design by both the real-time constraints and the heterogeneity of data. For these reasons, our `CrisMap` system deploys several Big Data technologies to process incoming SM data efficiently, without sacrificing scalability and fault-tolerance.

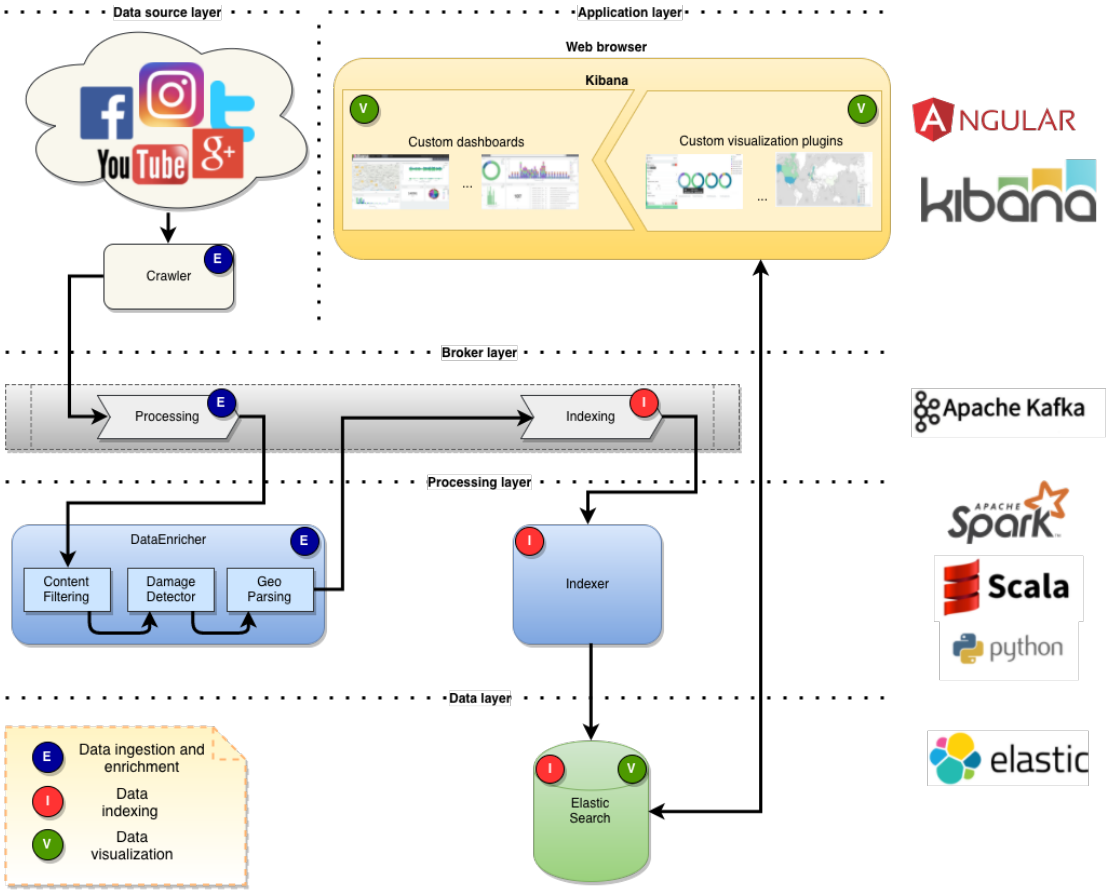


Figure 6.1: *The logical architecture of our crisis mapping system. On the left-hand side, the architecture is organized in a stack of layers: from the topmost data source and application layers of our system, to the broker, processing, and data layers. On the right-hand side, for each layer are reported the technologies adopted to design and implement the components of the system.*

The functional workflow is divided into three logical steps: (i) Data Ingestion and Enrichment (labelled with a blue circle), where SM data is collected and processed in order to select and geoparse messages containing information about damages; (ii) Data Indexing (labelled with a red circle), in which useful data is conditioned and saved into the internal storage; (iii) Data Visualization (labelled with a green circle), in which stored data is retrieved and used to create maps or, more generally, to provide results to the end users through a Web dashboard.

In the following, we give an overall description of the functionalities of each of these steps. A detailed technical description and evaluation of the solutions we adopted and implemented to address the main issues related to the design of a crisis mapping system, namely *mining messages to search for damage*, *geoparsing* and *visualize data*, are given in Sections 6.4, 6.5 and 6.6, respectively.

6.2.1 Data Ingestion and Enrichment

The first logical step of the system consists into ingesting data coming from available SM data sources, possibly enriching it with additional information not directly available from the data source and which can provide useful information exploitable subsequently during the visualization phase. Data ingestion occurs using platform specific crawling/scraping software. For the sake of simplicity, in this work we focused our attention on Twitter solely. However, nothing prevents the adopter to deploy the system using a different data source.

Our system is able to process both real-time data fetched from the Twitter stream and historical data acquired from data resellers. In a practical application scenario, the system is fed with real-time data, while historical data can be used to run simulations on past emergency events. The *Crawler* component exploits Twitter's Streaming API[1] to perform data acquisition from the social network. The Streaming API gives low latency access to Twitter's global stream of messages. Collected messages can be optionally filtered by search keywords.

Collected messages are then forwarded toward the Processing layer through a specific queue (named "Processing") placed at the Broker layer. Our implementation choice fell on Kafka[2], a distributed publish-subscribe messagging platform. As described on the website, Kafka "is used for building real-time data pipelines and streaming apps. It is horizontally scalable, fault-tolerant, wicked fast, and runs in production in thousands of companies". Its characteristics help the system to sustain very high traffic rates by mitigating the back-pressure data problem [199] and to process data resiliently, with no loss in case of system failures (e.g. hardware faults).

SM messages are fetched from the *Processing* queue and processed by an enrichment component called *DataEnricher*. As depicted in the figure, the *DataEnricher* is organized internally into a pipeline of three sub-modules that process, filter, and enrich incoming data. The *ContentFiltering* sub-module analyzes the tweets in order to select those that are relevant to the problem (e.g., whose contents relate to natural disasters). Relevant tweets are then analyzed by the *DamageDetector* sub-module, which in turn discriminates between data carrying or not carrying damage information. The tweets containing damage information are finally forwarded to the *GeoParsing* sub-module,

[1] https://dev.twitter.com/streaming/overview
[2] https://kafka.apache.org

which possibly enriches each tweet with information about the discovered geolocation. Upon completion of the pipeline, the enriched SM messages are pushed into another queue, the *Indexing* queue, waiting to be indexed and stored into the system. The *DataEnricher* is implemented using Spark[3] and uses a streaming approach to quickly process incoming data from the Processing queue.

6.2.2 Data Indexing

In the second step, enriched data is moved from the *DataEnricher* to a permanent storage through the *Indexing* queue. Data fetched from the queue is instantaneously parsed and pushed into a search and analyitcs engine. The choice of Elasticsearch[4] (ES) was driven by its capability to scale horizontally, as well as to perform fast search and aggregation on textual data leveraging its internal Apache Lucene[5] engine. In fact, the integration of ES with other software like Kibana and Logstash makes it a valid solution as storage system.

The role of the *Indexer* module is to map the data structure of a Twitter message into an optimized ES index, where each field type is treated differently to exploit the features of the engine to provide fast search and real-time analytics on stored data (e.g., ES tokenizes textual fields and provides an inverted terms list to efficiently search for in the related fields). Data stored in ES is available for queries in near-real time, introducing a latency of just 1 second before the indexed data could be retrieved by users. Indeed, the query latency is negligible in our context, having no practical consequences for the purposes of crisis mapping.

6.2.3 Data Visualization

In the third step, data stored in the index can be browsed and queried through the Kibana[6] software. With Kibana, it is possible to build real-time visualizations with very useful insights, like real-time volumes, maps, bar charts and word clouds. Those visualizations can be also joined to build complex dashboards that allow the user to track changes over time for the most important metrics of the dataset. Moreover, Kibana supports the possibility to easily extend the internal visualization types in order to customize graphical views using the most appropriate visual analytics for data. To our purposes, we developed a plugin to replace the native visualization maps. The new plugin[7] supports multi-resolution choropleth maps and the possibility to normalize data according to population, offering different options for what concerns the scales and their customizations.

6.3 Datasets

The datasets used for this work are composed of Italian tweets, collected in the aftermath of 5 major natural disasters. For our experiments we considered different kinds of disasters, both recent and historical: 3 earthquakes, a flood, and a power outage. Specifically, the `L'Aquila` and the `Emilia` datasets are related to severe earthquakes that

[3] `http://spark.apache.org`
[4] `https://www.elastic.co/products/elasticsearch`
[5] `https://lucene.apache.org/`
[6] `https://www.elastic.co/products/kibana`
[7] The plugin is publicly available at `https://github.com/marghe943/kibanaChoroplethMap.git` .

struck rural areas of Italy in 2009[8] and 2012 respectively[9]. The Amatrice dataset is related to a recent earthquake that struck central Italy in 2016[10]. The Sardinia dataset has been collected in the aftermath of a flash flood occurred in the Sardinia island in 2013[11]. Finally, the Milan dataset describes a power outage occurred in the metropolitan city of Milan (northern Italy) in 2013. To investigate a wide range of situations we picked disasters having variable degrees of severity: some caused only moderate damage, while other produced widespread damage and casualties.

The datasets were created by using the Twitter's Streaming API[12] for recent disasters, and the Twitter resellers' Historical APIs[13] (GNIP) for past disasters. The APIs give access to a global set of tweets, optionally filtered by search keywords. We exploited a different set of search keywords for every different disaster in order to collect the most relevant tweets about it. Whenever possible, we resorted to hashtags specifically created to share reports of a particular disaster, such as the *#allertameteoSAR* hashtag for the Sardinia dataset. In this way, we were able to select only tweets actually related to that disaster. However, for historical disasters we couldn't rely on specific hashtags and had to exploit generic search keywords already proposed in literature, see [13,14,182]. This is the case of the L'Aquila dataset, for which we exploited the "terremoto" (*earthquake*) and "scossa" (*tremor*) Italian keywords. In addition, we only used "fresh" data shared in the aftermath of the disasters under investigation. For instance, all the 3,170 tweets in the Emilia dataset were posted in less than 24 hours since the earthquake occurred.

Tweets in the L'Aquila, Emilia, and Sardinia datasets have been manually annotated for mentions of damage according to the 3 following classes: (i) tweets related to the disaster and carrying information about damage to infrastructures/communities (*damage*); (ii) tweets related to the disaster but not carrying relevant information for the assessment of damage (*no damage*); (iii) tweets not related to the disaster (*not relevant*). The inclusion of a class for tweets that are not related to a disaster (*not relevant*) is necessary because the automatic data collection strategy we adopted does not guarantee that all the tweets collected are actually related to the disaster under investigation. This is especially true for the datasets collected with generic search keywords and represents a further challenge for our classification task. The manual annotation of damage mentions among tweets is exploited to train and validate our damage detection classifier, as thoroughly explained in Section 6.4. Furthermore, following the same approach adopted in [86, 150], we carried out an additional manual annotation of 1,900 random tweets of the aforementioned datasets with regards to mentions of places/locations. This further annotation is exploited to validate our geoparsing results, as described in Section 6.5. The Milan dataset is also used for a comparison of geoparsing techniques in Section 6.5, since it was already exploited in previous work [150]. The Emilia and Sardinia datasets are also used in Section 6.6 to quantitatively validate our crisis maps against authoritative data. Finally, the Amatrice dataset is exploited in Section 6.6 as a case study of our system in the aftermath of the recent central Italy earthquake.

[8] https://en.wikipedia.org/wiki/2009_L'Aquila_earthquake
[9] https://en.wikipedia.org/wiki/2012_Northern_Italy_earthquakes
[10] https://en.wikipedia.org/wiki/August_2016_Central_Italy_earthquake
[11] https://en.wikipedia.org/wiki/2013_Sardinia_floods
[12] https://dev.twitter.com/docs/api/streaming
[13] http://gnip.com/sources/twitter/historical

Notably, the total number of 15,825 tweets in our datasets, shown in Table 6.1 along with other details, is greater than those used in other related works, such as [150] (6,392 tweets across 4 datasets), and [87] (2,000 tweets for a single dataset).

Table 6.1: *Characteristics of the Datasets.*

				tweets					used in
dataset	**type**	**year**	**users**	*damage*	*no damage*	*not relevant*	*GPS*	*total*	**sections**
L'Aquila	Earthquake	2009	563	312 (29.4%)	480 (45.2%)	270 (25.4%)	0 (0%)	1,062	6.4, 6.5, 6.6
Emilia	Earthquake	2012	2,761	507 (16.0%)	2,141 (67.5%)	522 (16.5%)	205 (6.5%)	3,170	6.4, 6.5
Milan	Power outage	2013	163	-	-	-	15 (3.8%)	391	6.5
Sardinia	Flood	2013	597	717 (73.5%)	194 (19.9%)	65 (6.6%)	51 (5.2%)	976	6.4, 6.5, 6.6
Amatrice	Earthquake	2016	7,079	-	-	-	21 (0.2%)	10,226	6.6

To better understand the importance of geoparsing in a crisis mapping task, in Table 6.1 we also reported the number of tweets natively geolocated (GPS column). Geolocation of these tweets is performed directly by Twitter whenever a user enables GPS or WiFi geolocation. Statistics on our datasets confirm previous findings reporting that only a small percentage ($1\% \div 4\%$) of all tweets are natively geolocated [45, 53]. As introduced in Section 7.1, the low number of natively geolocated tweets drastically impairs crisis mapping, hence the need of a geoparsing operation. Noticeably, none of the 1,062 tweets of the L'Aquila dataset, dating back to 2009, are natively geolocated.

6.4 Mining text to search for damage

The detection of damage in SM messages is a challenging task due to the almost completely unstructured nature of the data to be analyzed [54]. The *DataEnricher* component of Figure 6.1 analyzes the content of tweets with the twofold goal of discarding irrelevant tweets and labeling the relevant ones according to the presence (or lack thereof) of damage mentions. In our system, "damage" refers both to damage to buildings and other structures and to injuries, casualties, and missing people. In other words, damage encompasses all harmful consequences of an emergency on infrastructures and communities.

In this work, we approach the damage detection problem as a two-levels binary classification task. To our purposes, we are interested in identify 4 different classes of tweets:

- *Not relevant*: tweets not related to a natural disaster.
- *Relevant*: tweets related to a natural disaster.
- *Without damage*: tweets related to a natural disaster but which do not convey information relevant to damage assessment.
- *With damage*: tweets related to a natural disaster which convey information relevant to damage assessment.

At the first level, the binary classifier (sub-module *ContentFiltering* in Figure 6.1) acts as a filter to discriminate between not relevant and relevant tweets, allowing only the latter ones to pass over to the second level. The classifier at the second level (sub-module

DamageDetector in Figure 6.1) discriminates between tweets containing relevant information about damage and those not containing information relevant for damage assessment.

We built the two binary classifiers using the Support Vector Machines (SVM) algorithm [50] with a simple linear kernel as machine learning method[14]. The set of features used by the single classifier was obtained from tweets by analyzing the textual content of a tweet using an approach based on word embeddings [24].

The NLP (Natural Language Processing) research field has gained a lot of attention in the last years, due to the renewed interest in neural networks technologies (e.g., deep learning), the continuous growth of the computational power of the CPUs and GPUs, and the explosion of available data that can be used to train these neural networks in an unsupervised way. In this work we specifically used the approach proposed by Mikolov *et al.* [151] describing the `word2vec` software, which is currently the most popular model for embeddings used in NLP-related tasks. Word embeddings techniques are an elegant solution to the problem of the features sparseness in document vectors created by using classic approaches like bag-of-words, char-N-grams, or word-N-grams [192]. From one side, they aim to create a vector representation with a much lower dense dimensional space. On the other side, they are useful to extract semantic meaning from text, to enable natural language understanding by learning the latent context associated with every specific word extracted from training data. More formally, distribute word representations (word embeddings) learn a function $W : (word) \rightarrow R^n$ that maps a word into a vector where each dimension models the relation of that specific word with a specific latent aspect of the natural language, both in syntactic or in semantic way. The vectors are learned through the training of neural language models together with the parameters of the network from a set of unannotated texts and according to an objective function (e.g., distributional hypothesis[15]) [23]. The resulting word vectors are computed so as to maintain the semantic/syntactic relationship existent between words which in turn allow to (i) visualize the vectors of similar words very close into a given metric space (e.g., visualize the word embeddings space on 2-D space through techniques like t-SNE), (ii) compute algebraic operations on vectors to point out some specific characteristic of the data (e.g., $W(\text{“}queen\text{”}) \cong W(\text{“}king\text{”}) - W(\text{“}man\text{”}) + W(\text{“}woman\text{”})$).

The approach we used in our system to build the damage-detection component is completely different from the one presented in our recent work on the same research topic [12]. In our previous work, we built this component using a multiclass classifier based on SVM with linear kernel but operating with features extracted from training data using classic NLP techniques [54]. The classifier based on classic NLP guarantees a good accuracy at classification time, but it has some significant drawbacks that we aim to mitigate in this work. In particular, in the past work we used 5 different classes of features (e.g., lexical text features, morphosyntactic features, sentiment-analysis features, etc.) that are almost language-dependent and extracted with a quality level strongly dependent from the set of NLP tools and resources available to analyze the textual data. The choice of which features to extract (often referred to as the “feature engineering"

[14] As software implementation we used the SVC class available in the `scikit-learn` Python package.

[15] The meaning of this hypothesis is that words appearing in similar contexts often have similar meaning.

Table 6.2: *NLP classifier vs. Embeddings classifier in terms of $F1$ effectiveness compared over the three available labeled datasets.*

	dataset	damage	no damage	not relevant
NLP	L'Aquila	0.89	0.87	0.73
	Emilia	0.90	0.87	0.49
	Sardinia	0.89	0.46	0.29
EMB	L'Aquila	0.85	0.82	0.72
	Emilia	0.85	0.87	0.52
	Sardinia	0.82	0.43	0.33

problem) is a non-trivial task that must be solved in order to provide the classifier a set of features sufficiently informative for the resolution of the specific problem, given the input domain. Moreover, the total number of features extracted in this way is very high (in the order of several hundred-thousands features) and it has a severe impact on the system in terms of performance both at training and classification time, being this dependent both from the complexity of the SVM algorithm (which is linearly increasing with the number of features) and the time spent to use external NLP tools to enrich data. Conversely, using our new approach based on word embeddings helps handling this type of issues because this technique completely avoids the feature engineering process and makes the system almost independent from any specific language. The only requirement affecting this model is the language coherence on a set of unannotated textual data used for the training of the embeddings, a condition often satisfiable very easily and with no human effort on many application domains (such as the Twitter domain). Another useful implication of word embeddings is the reduced number of features used by classifier, being often in the order of few hundreds, and therefore contributing to speed-up learning new models and classifying new documents.

To validate the goodness of our new proposal based on embeddings, we compared the $F1$ effectiveness [192] obtained with a 10-fold cross evaluation of both approaches using the multiclass single-label configuration proposed in our previous work, as shown in Table 6.2. The NLP classifier (reported as NLP) has been tuned as described in [12] while the embeddings classifier (reported as EMB) has been tuned as in the following. The word embeddings vectors have been obtained by training the system on a dataset composed of the union of the tweets coming from `L'Aquila`, `Emilia`, `Sardinia`, and `Amatrice` using the CBOW method [151], and the size of the embeddings was set to 100. Given a tweet, to obtain a single vector representing the tweet content, we computed the tweet vector as the averaged sum of the vectors of the words contained in the text of the tweet[16]. The SVM classifier working over embeddings has been set to use the linear kernel with the parameter $C = 1$ and we have handled unbalanced data distribution among labels by assigning to each label a weight proportional to its popularity[17]. As reported in the Table 6.2, the embeddings classifier obtains very similar results to the NLP classifier in all tested datasets, confirming that our new approach provides all previously discussed advantages without sacrificing too much the accuracy

[16]We did not use more sophisticated methods like "Paragraph Vector" [131] because these statistical methods do not work well for small texts like tweets.

[17]We used the 'balanced' value for class weight, see `scikit-learn` documentation at `http://bit.ly/2g5QSqk`. In this way we indicate to SVM to treat the various labels in different ways during training phase, giving more importance to class errors (measured with used loss function) made for skewed classes.

of the damage detection system.

This comparison also suggests that in order to improve the accuracy of the embeddings classifier we can operate at two different levels. Firstly, we can use more training data to train the classifier: a simple and effective solution is to merge the four separated datasets (`L'Aquila`, `Emilia`, `Sardinia`, and `Amatrice`) into one bigger training dataset. Secondly, to simplify the task of classification model's learning, we can change the target problem from a multiclass single-label classification problem into a pair of separated binary classification problems, as described at the beginning of this section. This last modification has also the advantage to clearly separate the filter part that identifies relevant tweets from the damage detection phase. This separation enables parallel execution of the damage classification task and the geoparsing task, decreasing the total time required to entirely process a single tweet.

We tested the system with the proposed improvements using a 10-fold cross evaluation and by optimizing the model parameters in order to build a reasonable good set of classifiers. The measure used to drive the choice in the best configuration values is the micro averaged $F1$ [192]. The set of parameters subject to optimization were the following:

- *Embeddings dataset*: the dataset used to learn word embeddings. We have used 5 possible different datasets: `Amatrice`, `L'Aquila`, `Emilia`, `Sardinia`, and the union of all previous ones (`All`).

- *Embeddings size*: the size used to represent word embeddings vectors and consequently the vector size of a single tweet. The possible set of values are 50, 100, 200, and 400. The default value is 100.

- *Class weight*: the weight of the errors associated with the positive class used in the two binary classifiers. The positive classes were “Not relevant" for filter classifier and “With damage" for damage classifier. The possible set of values were 1.0, 1.5, 2.0, 2.5 and “Balanced" (same meaning as explained in footnote 17). The default value is 'Balanced'.

- *C*: indicates the cost penalty associated with a misclassification. The possible set of values are $\in$ [0, 12] with a step increment of 0.1. The default values is 1.0.

To avoid testing all possible combinations of the above parameters, and in order to choose a reasonable good set of parameter values, we followed a simplified procedure as illustrated in the following. Using the order of the parameters as described above, we optimize one parameter at a time testing the full set of values for that specific parameter and using the default values for the other parameters. In case one of the other parameters has been optimized already, we use instead its best found value. The dataset used to evaluate the system[18] has been generated in two different versions, one for filtering and one for damage detection. In the case of filtering, every tweet in the dataset originally labeled with “With damage" or “Without damage" labels has been relabeled with “Relevant" label, resulting in a final dataset containing 4,351 relevant tweets and 857 not relevant tweets. In the case instead of damage detection, every tweet in the

[18]The dataset is the union of the `L'Aquila`, `Emilia` and `Sardinia` datasets.

Table 6.3: *Choice of Optimal Parameters for Embeddings Classifier Filtering Relevant/not Relevant Tweets.*

		Not relevant			Relevant			Micro avg results		
	Configuration	**Pr**	**Re**	**F1**	**Pr**	**Re**	**F1**	**Pr**	**Re**	**F1**
Emb. dataset	`Amatrice`	0.28	0.73	0.41	0.92	0.63	0.75	0.82	0.65	0.69
	`L'Aquila`	0.26	0.89	0.40	0.96	0.50	0.65	0.84	0.56	0.61
	`Emilia`	0.28	0.82	0.41	0.94	0.58	0.72	0.83	0.62	0.67
	`Sardinia`	0.21	0.85	0.34	0.93	0.37	0.53	0.81	0.45	0.50
	`All` **	0.36	0.79	0.49	0.95	0.72	0.82	0.85	0.73	0.76
Emb. size	50	0.35	0.80	0.48	0.95	0.71	0.81	0.85	0.72	0.76
	100 **	0.36	0.79	0.49	0.95	0.72	0.82	0.85	0.73	0.76
	200	0.35	0.79	0.49	0.94	0.72	0.81	0.85	0.73	0.76
	400	0.35	0.79	0.49	0.95	0.71	0.81	0.85	0.73	0.76
Class weight	1.0	1.00	0.00	0.00	0.84	1.00	0.91	0.86	0.84	0.76
	1.5	0.42	0.23	0.30	0.86	0.94	0.90	0.79	0.82	0.80
	2.0 **	0.44	0.54	0.48	0.91	0.86	0.88	0.83	0.81	0.82
	2.5	0.42	0.59	0.49	0.91	0.84	0.88	0.83	0.80	0.81
	Balanced	0.36	0.79	0.49	0.95	0.72	0.82	0.85	0.73	0.76
Best C	9.1	0.43	0.56	**0.49**	0.91	0.86	**0.88**	0.83	0.81	**0.82**

dataset originally labeled with “Not relevant" has been relabeled with “Without damage" label, resulting in a final dataset containing 3,672 tweets marked with “Without damage" label and 1,536 tweets marked with “With damage" label.

In Table 6.3 and Table 6.4 we report the experimental results obtained from the optimization process, respectively while building the filter classifier and the damage classifier. Except from the C parameter, which we report only as the best value found, we have marked with ** the best configuration found among all tested ones. In the case of parameters “Embeddings dataset" and “Embeddings size" in both type of classifiers the best results are obtained with the full dataset (`All`) and 100 as the dimension of embeddings[19]. The optimal weight class values are different for each classifier reflecting the fact that the data distribution is very different between the two used datasets. In particular, the dataset used for filter classifier is more unbalanced towards relevant tweets, resulting in more variance in the results obtained for each tested configuration and in general providing quite low F1 values for class “Not relevant" (~ 0.5). Anyway, the results also show that the errors made for this last class by the filter classifier are partially recovered by the damage classifier, considering that the F1 accuracy on both damage classes are remarkably high (> 0.8 in both cases), resulting into an effective and useful implementation of the damage detection component.

6.5 Geoparsing

Geoparsing – namely, the resolution of toponyms in a textual document to a set of geographic coordinates – is typically considered the focal point of crisis mapping and has been faced since the diffusion of the Web. This task is typically solved extracting toponyms from text and looking up for matches in gazetteers containing all the possible

[19]In case of configurations with equal results in terms of F1 we prefer to choose those having more balanced values between precision and recall measures.

Table 6.4: *Choice of Optimal Parameters for Embeddings Classifier Identifying With Damage/Without Damage Tweets.*

		Without damage			With damage			Micro avg results		
	Configuration	**Pr**	**Re**	**F1**	**Pr**	**Re**	**F1**	**Pr**	**Re**	**F1**
Emb. dataset	Amatrice	0.92	0.75	0.82	0.58	0.85	0.69	0.82	0.77	0.78
	L'Aquila	0.87	0.89	0.88	0.72	0.69	0.70	0.83	0.83	0.83
	Emilia	0.95	0.88	0.91	0.76	0.89	0.82	0.89	0.88	0.89
	Sardinia	0.91	0.89	0.90	0.74	0.79	0.76	0.86	0.86	0.86
	All **	0.96	0.87	0.91	0.75	0.92	0.83	0.90	0.89	0.89
Emb. size	50	0.96	0.87	0.91	0.75	0.92	0.82	0.90	0.88	0.89
	100 **	0.96	0.87	0.91	0.75	0.92	0.83	0.90	0.89	0.89
	200	0.96	0.87	0.91	0.75	0.92	0.82	0.90	0.88	0.89
	400	0.96	0.87	0.91	0.75	0.92	0.82	0.90	0.88	0.89
Class weight	1.0	0.93	0.92	0.92	0.81	0.84	0.82	0.89	0.89	0.89
	1.5 **	0.95	0.89	0.92	0.78	0.88	0.83	0.90	0.89	0.89
	2.0	0.96	0.88	0.92	0.76	0.91	0.83	0.90	0.89	0.89
	2.5	0.96	0.87	0.91	0.75	0.92	0.82	0.90	0.88	0.89
	Balanced	0.96	0.87	0.91	0.75	0.92	0.83	0.90	0.89	0.89
Best C	0.4	0.95	0.90	**0.92**	0.78	0.88	**0.83**	0.90	0.89	**0.89**

matches between a set of place names and their geographic coordinates [150]. This approach requires an offline phase where the geoparsing system is specifically set to work in a geographically-limited region. Although this solution is effective for limited areas and offers a fast response, it is practically infeasible to load associations between toponyms and coordinates for a wide region or for a whole country [12]. Another challenge related to geoparsing is that of toponymic polysemy – that is, the situation in which a toponym might have different meanings, thus possibly referring to different places, according to the context in which it is used (e.g., the word “Washington" may refer to the first US president, to the US capital, to the US state, etc.)[20]. This last problem is particularly relevant for geoparsing systems based on gazetteers lookups, since with this approach, there is no way to perform a disambiguating operation of the toponyms in order to understand their actual meanings [12, 53].

To overcome these limitations, the *GeoParsing* sub-module of the *DataEnricher*, shown in Figure 6.1, adopts semantic annotators in the geoparsing process. Semantic annotation is a process aimed at augmenting a plain-text with pertinent references to resources contained in knowledge-bases such as Wikipedia and DBpedia. The result of this process is an enriched (annotated) text where mentions of knowledge-bases entities have been linked to the corresponding Wikipedia/DBpedia resources. This annotation process is highly informative since it enables the exploitation of the rich information associated with the Wikipedia/DBpedia resources that have been linked to the annotated text. Here, we exploit semantic annotations for our geoparsing task by checking whether knowledge-bases entities, which have been linked to our tweets, are actually places or locations. Semantic annotation also has the side effect of alleviating geoparsing mistakes caused by toponymic polysemy. In fact, some terms of a plain-text can potentially be linked to multiple knowledge-bases entities. Semantic annotators automatically perform a disambiguating operation and only return the most likely reference

[20] http://en.wikipedia.org/wiki/Washington

	GPS		DBpedia Spotlight		Dexter		TagMe	
L'Aquila	0	(0%)	271	(25.5%)	578	(54.4%)	721	(67.9%)
Emilia	205	(6.5%)	975	(30.8%)	1,037	(32.7%)	1,671	(52.7%)
Milan	15	(3.8%)	99	(25.3%)	320	(81.8%)	364	(93.1%)
Sardinia	51	(5.2%)	530	(54.3%)	784	(80.3%)	582	(59.6%)

Table 6.5: *Contribution of our GeoParsing sub-module, used in conjunction with the different semantic annotators, on the number of geolocated tweets.*

to a knowledge-base entity for every annotated term. Overall, our proposed geoparsing technique overcomes 2 major problems affecting current state-of-the-art crisis mapping systems: (i) it avoids the need to preload geographic data about a specific region by drawing upon the millions of resources of collaborative knowledge-bases such as Wikipedia and DBpedia, (ii) it reduces the geoparsing mistakes caused by toponymic polysemy that are typical of those systems that perform the geoparsing task via lookups in preloaded toponyms tables. Another additional useful characteristic of our geoparsing technique is that it is unsupervised, unlike the one presented in [87].

Because of these reasons, our geoparsing technique is particularly suitable for being employed in a system aimed at producing crisis maps *impromptu*, such as the one that we are proposing. The possibility to link entities mentioned in emergency-related messages to their pages, also allows to exploit the content of their Wikipedia/DBpedia pages in order to extract other useful information about the unfolding emergency. Most commonly used semantic annotators also provide a confidence score for every annotation. Thus, it is possible to leverage this information and only retain the most reliable annotations, discarding the remaining ones.

Among all currently available semantic annotators, `CrisMap` is currently based on `TagMe` [76], `DBpedia Spotlight` [149], and `Dexter` [203], three well-known, state-of-the-art systems [204]. `TagMe` is a service of text annotation and disambiguation developed at the University of Pisa. This tool provides a Web application[21] as well as a RESTful API for programmatic access and can be specifically set to work with tweets. Since `TagMe` is based on the Wikipedia knowledge-base, the annotated portions of the original plain-text are complemented with the ID and the name of the linked Wikipedia page. `TagMe` also returns a confidence score *rho* for every annotation. Higher *rho* values mean annotations that are more likely to be correct. After annotating a tweet with `TagMe`, we resort to a Wikipedia crawler in order to fetch information about all the Wikipedia entities associated with the annotated tweet. In our implementation we sort all the annotations returned by `TagMe` on a tweet in descending order according to their *rho* value, so that annotations that are more likely to be correct are processed first. We then fetch information from Wikipedia for every annotation and check whether it is a place or location. The check for places/locations can be simply achieved by checking for the *coordinates* field among Wikipedia entity metadata. We stop processing annotations when we find the first Wikipedia entity that is related to a place or location and we geolocate the tweet with the coordinates of that entity. The very same algorithmic approach is employed for the exploitation of the other semantic annotators: `DBpedia Spotlight` and `Dexter`. Indeed, it is worth noting that our

[21] `https://tagme.d4science.org/tagme/`

proposed geoparsing technique does not depend on a specific semantic annotator, and can be implemented with any annotator currently available, or with a combination of them.

We used our *GeoParsing* sub-module to geocode all the tweets of our datasets. Then, following the same approach used in [150] and [86], we manually annotated a random subsample of 1,900 tweets to validate the geoparsing operation. Noticeably, our system achieves results comparable to those of the best-of-breed geoparsers with an $F1 = 0.84$, whether the systems described in [150] and [86] scored in the region of $F1 \sim 0.80$. Furthermore, to better quantify the contribution of our *GeoParsing* sub-module, we report in Table 6.5 the number of natively geolocated tweets (GPS column) and the number of tweets geolocated by our *GeoParsing* sub-module via `TagMe`, `DBpedia Spotlight`, and `Dexter`. As shown, our system geoparses the highest number of tweets based on the annotations of `TagMe`, for all datasets, except for the `Sardinia` one, for which the best results are achieved with `Dexter`'s annotations. In any case, our *GeoParsing* sub-module managed to geoparse from a minimum of 25.3% tweets, to a maximum of 93.1% tweets of the `Milan` dataset, meaning that almost all tweets of that dataset were associated with geographic coordinates, allowing to use such tweets in our crisis maps.

6.6 Mapping data

Among the diverse data visualization techniques, one that is commonly employed to represent the geographic distribution of a statistical variable, is the *choropleth map*. A choropleth map is a thematic representation in which subareas of the map are filled with different shades of color, in proportion to the measurement of the given variable being displayed[22]. This visualization technique is usually exploited to depict the spatial distribution of demographic features such as population, land use, crime diffusion, etc. In `CrisMap` we exploit the same visualization technique to show the spatial distribution of damage in the aftermath of an emergency. A clear advantage of exploiting choropleth maps instead of the typical on/off maps used in previous works [150], lies in the possibility to apply different shades of color to the different areas of the map, according to the estimated extent of damage suffered by that area. This complements well with the prioritization needs that arise in the first phases of an emergency response. Most notably, our system can be easily extended to produce different end-results. In other words, we can choose to produce a choropleth crisis map, or any other visualization of the analyzed tweets exploiting the high flexibility of the Kibana interface.

Notably, `CrisMap` is capable of producing choropleth crisis maps with a spatial resolution at the level of *municipalities*. Anyway, when tweets are accurate enough, it is also possible to precisely identify objects (e.g., a specific building) that suffered damage. It is also worth noting that the choice to produce crisis maps showing the estimated degree of damage among the different municipalities is not due to a region-level only geoparsing. Indeed, the exploitation of semantic annotators potentially allows to geocode every entity that has an associated Wikipedia/DBpedia page. So, in those cases when a tweet contains detailed geographic information, it is possible to geoparse it to building- or even street-level. Our choice to produce crisis maps at the level of munici-

[22] `https://en.wikipedia.org/wiki/Choropleth_map`

Table 6.6: *Binary detection of damaged municipalities for the* `Emilia` *earthquake.*

	evaluation metrics					
task	*Precision*	*Recall*	*Specificity*	*Accuracy*	*F-Measure*	*MCC*
Detection of all damaged areas	1.000	0.178	1.000	0.797	0.303	0.375
Detection of areas that suffered significant damage	1.000	0.813	1.000	0.992	0.897	0.898

palities is instead motivated by an effort to rigorously compare our crisis maps to data officially released by the Italian Civil Protection agency, which reports damages at the municipality-level.

Here, we first show the accuracy of our visualizations referring to two case studies, the `Emilia` earthquake and the `Sardinia` flood. We compare the maps realized with SM data against those produced using authoritative data provided by Italian civil protection agency. Finally, we present results obtained by applying `CrisMap` to study the `Amatrice` earthquake. Unfortunately, there is no fine economic loss estimation for the `Amatrice` earthquake, since the same area suffered a second severe shake just few months after the first one, when official damage surveys still had to be completed. Nonetheless, in the case of the `Amatrice` earthquake, we are still able to provide a qualitative case-study of our crisis maps.

6.6.1 Quantitative validation

The authoritative data that we used for the comparison is the economic loss/damage (quantified in millions of euros) suffered by the different municipalities, as assessed by the Italian Civil Protection agencies of Emilia Romagna[23] and Sardinia[24].

It is possible to perform a first quantitative evaluation of our crisis maps following the approach used in [150], that is, performing the evaluation as a classification task. Under this hypothesis, the goal of the system is to detect damaged municipalities disregarding of those requiring prioritized intervention – namely, those that suffered the most damage. Thus, we can exploit well-known machine learning evaluation metrics to compare crisis maps generated by our system with those obtained from official data. The comparison is performed by checking whether a municipality with associated damage in authoritative data also appears as damaged in our crisis maps.

Table 6.6 reports the results of this comparison for the `Emilia` earthquake. We first consider all the municipalities of the affected region, namely the Emilia Romagna region, and then we repeat the comparison by only considering those municipalities that suffered a significant degree of damage (more than 10% of the damage suffered by the Ferrara municipality, which is the maximum value for the `Emilia` earthquake). As clearly highlighted by Table 6.6, the proposed crisis mapping system is able to accurately identify the areas where damage actually occurred. However, not all the damaged municipalities are identified by the system, as represented by the low Recall value in the first row of the table. Anyway, when we remove from the comparison those municipalities that suffered the lowest damage, the Recall metric reaches a value of 0.813, showing the system's ability in detecting areas that suffered a significant amount of

[23] `http://www.openricostruzione.it` - Web site maintained by the Emilia Romagna regional district.

[24] `http://www.regione.sardegna.it/documenti/1_231_20140403083152.pdf` - Italian Civil Protection report about damage to infrastructures and agricolture.

Table 6.7: *Binary detection of damaged municipalities for the* `Sardinia` *flood.*

	evaluation metrics					
task	*Precision*	*Recall*	*Specificity*	*Accuracy*	*F-Measure*	*MCC*
Detection of all damaged areas	0.833	0.128	0.993	0.814	0.222	0.280
Detection of areas that suffered significant damage	0.500	1.000	0.995	0.995	0.667	0.705

damage. In other words, the great majority of the mistakes of our system occurred in municipalities that suffered relatively low damage, and not on those requiring immediate attention. The same also applies for the `Sardinia` flood, as reported in Table 6.7, with an improvement of the Recall metric from 0.128 to 1 when considering municipalities that suffered more than 2% of the maximum damage (i.e. the damage suffered by the municipality of Olbia).

Overall, the results obtained by our system with respect to the detection of damaged areas are comparable to those reported in [150]. However, our system operated with a fine geographic resolution on 2 case studies of natural disasters that affected wide, rural, and sparsely populated areas. Conversely, the system presented in [150] has a fine resolution only for an emergency affecting a densely and uniformly populated area (Manhattan, New York) while it shows coarse resolution results for a disaster striking a wide area (the state of Oklahoma).

In addition to detecting damaged areas, `CrisMap` also visually sorts municipalities using a color hue that is proportional to the intensity of the damage suffered. In other words, it order ranks municipalities based on the (normalized) number of tweets conveying damage information. This unprecedented feature opens up the possibility to perform a finer evaluation of our crisis maps than that carried out in previous works. Indeed, it is possible to compare the ranking of damaged municipalities as obtained from tweets, with a ranking derived from authoritative sources, such as those provided by civil protection agencies. A crisis mapping system that is able to rapidly identify the most damaged areas would become a valuable tool in the first phases of emergency response, when resource prioritization plays a dominant role. A possible way of performing such evaluation is by employing metrics that are typically used to assess the performance of ranking systems, such as search engines. Search engines are designed to return the most relevant set of results to a given user-submitted query. In our scenario, we can consider `CrisMap` as a basic “search engine" that returns a list of areas and that is specifically designed to answer a single, complex query: “which areas suffered the most damage?". Search engines are evaluated with several metrics and indices, aimed at capturing a system’s ability to return desired resources (e.g.; Web pages, text documents, etc.) among the first results. We can then evaluate the ability of `CrisMap` to correctly identify areas that suffered a high degree of damage by employing evaluation metrics of search engines. Specifically, among such well-known metrics are the *normalized Discounted Cumulative Gain* ($nDCG$) [108] and the *Spearman’s Rho coefficient*. The $nDCG$ measures the performance of a “recommendation" (or ranking) system based on the graded relevance of the recommended entities. It is the normalized version of the *Discounted Cumulative Gain* and ranges from 0 to 1, with 1 representing the ideal ranking of the entities. This metric is commonly used in information retrieval

Table 6.8: *Ranking evaluation of most damaged municipalities.*

dataset	$nDCG$	*Spearman's Rho*
Emilia	0.770	0.698
Sardinia	0.647	0.408

to evaluate the performance of web search engines:

$$DCG_k = \sum_{i=1}^{k} \frac{2^{rel_i} - 1}{log_2(i+1)}$$

$$nDCG_k = \frac{DCG_k}{IDCG_k}$$

where rel_i is the graded relevance of the result at position i and $IDCG_k$ is the maximum possible (ideal) DCG for a given set of entities.

Spearman's Rho instead measures the correlation between two variables described using a monotonic function, and it is evaluated as:

$$\rho = 1 - \frac{6 \sum_i D_i^2}{N(N^2 - 1)}$$

where $D_i = r_i - s_i$ is the difference between actual position (given by the system) and expected position (given by the reports). For instance, it measures the correlation between the ideal output of the system (Civil Protection ordering) with the result of the system and describes how likely one variable is going to change (tweets with damage) given the other (damage amount). Being a correlation coefficient, it ranges from -1 to 1, with values in the region of 0 indicating no correlation. Using these metrics we assessed the ability of our system in detecting the most stricken areas against authoritative data based on the economic damage suffered by the affected municipalities.

Our experiments confirm that there is a considerable agreement between tweet-derived rankings and those based on authoritative data, as reported in Table 6.8.

A simple test for statistical significance of our ranking results with Spearman's Rho further supports our claims, achieving a confidence score $> 99\%$ for both the Emilia earthquake and the Sardinia flood. Overall, results of our system in detecting all damaged areas, as well as the most damaged ones, demonstrate the applicability and the usefulness of CrisMap also in wide, rural, and sparsely populated regions.

6.6.2 The qualitative Amatrice case-study

Figure 6.2 shows an excerpt of the CrisMap dashboard in the aftermath of the Amatrice earthquake. All the visualizations shown in figure are related to the Amatrice dataset, collected during the first hour from the earthquake occurrence. On top of Figure 6.2 are two choropleth maps – visualizations 6.2(a) and 6.2(b) – obtained by embedding the choropleth plugin, that we specifically developed for CrisMap, inside the Kibana interface. Figure 6.2(a) shows the map generated from all the tweets that we collected, while Figure 6.2(b) is obtained only from *damage* tweets. Our choropleth maps clearly highlight the most damaged municipalities, namely Norcia, Amatrice and Accumoli,

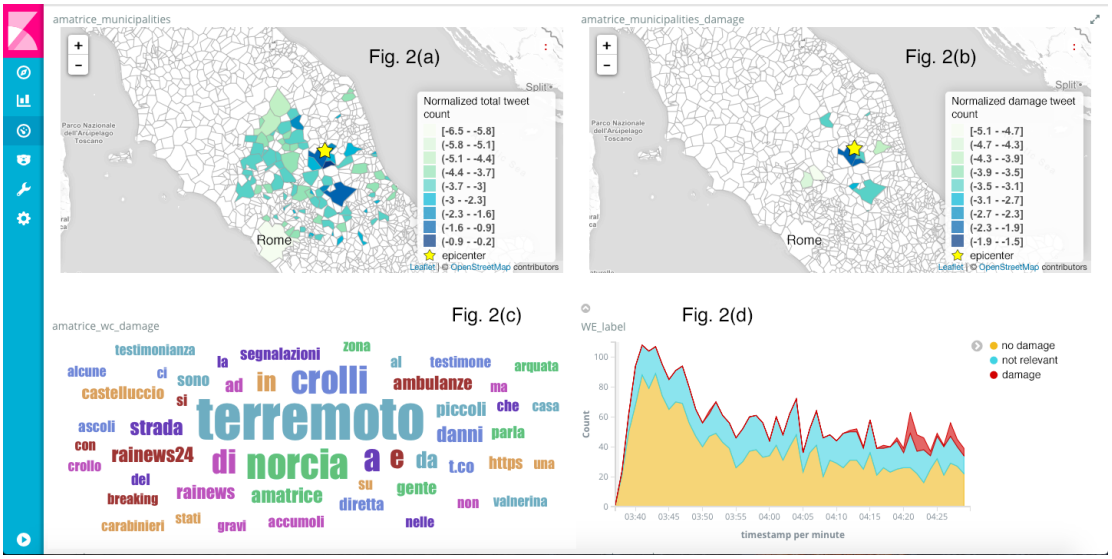


Figure 6.2: *Excerpt of the* `CrisMap` *dashboard in the aftermath of the* `Amatrice` *earthquake. The dashboard comprehends our ad-hoc choropleth maps showing the spatial distribution of tweets, a temporal view of collected and classified tweets per minute, and a word-cloud. The dashboard is easily extensible and customizable, allowing end-users to include more visualizations by choosing from the many ones natively provided by Kibana.*

located in the vicinity of the epicenter. The map in Figure 6.2(b), related only to damage tweets, is rather sparse, since in the first hour after the earthquake only a very small fraction of tweets conveyed damage reports. This is also clearly visible from Figure 6.2(d), showing the volume of tweets collected and classified by `CrisMap` every minute. As shown, tweets reporting damage (red-colored) have been shared almost only in the last minutes of the first hour. Yet, despite the very few tweets reporting damage, the word-cloud of Figure 6.2(c) highlights the key consequences of the earthquake: (i) the 3 most damaged villages Norcia, Amatrice, Accumoli; (ii) and several mentions of damage, such as “crolli" (collapsed buildings) and “danni" (widespread damage).

Notably, all the visualizations of Figure 6.2 are interactive and are updated in real-time, as new data is collected and analyzed by `CrisMap`. Regarding user interactions, for instance it is possible to click on a specific municipality in the choropleth maps in order to update all other visualizations by showing only data that is related to the selected municipality. Alternatively, one could click on a word in the word-cloud to visualize the temporal and spatial distribution of tweets containing that word. These functionalities open up the possibility to promptly perform drill-down analyses of the most relevant facts.

6.7 Related Work

The possibility to exploit social media data for crisis mapping has been first envisioned in a few trailblazing works [84, 90, 148] and further backed up by recent research [15]. Since the early works, there has been a growing interest by both practitioners and scholars in all areas related to crisis mapping: from data acquisition and management, to analysis and visualization [12].

6.7.1 Practical experiences

The great interest of practitioners and emergency responders towards the exploitation of SM data is testified by the efforts of the Federal Emergency Management Agency (FEMA) and the United States Geological Survey (USGS), which already led to interesting results with practical applications to earthquake emergency management[25] [32, 65]. Regarding already deployed applications, well-known crisis mapping platforms are Ushahidi[26], Mapbox[27], Google's Crisis Map[28], ESRI ArcGIS[29], and CrisisCommons[30] [20]. The main features of these platforms are related to data acquisition, data fusion, and data visualization. Such platforms represent hybrid crowdsensing systems where users can voluntarily load data onto the system in a participatory way, or the system can be configured so as to automatically perform data acquisition in an opportunistic way. The same hybrid data collection strategy has also been employed in a fully automatic system recently benchmarked in the earthquake emergency management field [10]. Another already-deployed application that exploits crowdsourced data is USGS's "Did You Feel It?" (DYFI) system[31]. This system, although not relying on SM data, exploits citizen reports and responses to earthquakes in order to automatically assess potential damage. It is foreseeable that in the near future such system could instead be fed with SM data. Indeed, there is already interesting research – from both USGS itself and from other laboratories – moving towards this direction [52, 95, 129, 130].

6.7.2 Academic works

Recent scientific literature has instead switched the focus from data acquisition and data fusion to in-depth data analysis. This is typically done by leveraging powerful machine learning techniques, and resulted in novel solutions being proposed to overcome critical crisis mapping challenges such as geoparsing and extracting situational awareness from microtexts [53].

Specifically, [150] presents a state-of-the-art system that matches preloaded location data for areas at risk to geoparse real-time tweet data streams. The system has been tested with data collected in the aftermath of New York's flooding (US – 2012) and Oklahoma's tornado (US – 2013) and achieved promising results. Among the key features of [150] is the possibility to match toponyms at region-, street-, or place-level. This is achieved by preloading already existing geographic databases (e.g.: the Geonames and GEOnet Names global gazetteers) for areas at risk, into the system. Crisis maps are then generated by comparing the volume of tweets that mention specific locations with a statistical baseline. Although presenting state-of-the-art solutions, [150] still has several drawbacks. The system can only work on a specific geographical area at a time since it has to load and manage external data for that area. Tweets mentioning locations outside the predefined area cannot be geolocated and consequently disasters cannot be monitored outside the area's boundaries. Moreover, the width of the area cov-

[25] https://blog.twitter.com/2014/using-twitter-to-measure-earthquake-impact-in-almost-real-time
[26] https://www.ushahidi.com/
[27] https://www.mapbox.com/
[28] https://www.google.org/crisismap/
[29] http://www.esri.com/arcgis/
[30] https://crisiscommons.org/
[31] http://earthquake.usgs.gov/research/dyfi/

ered by the system has direct implications on the amount of data to load and manage. This impacts on system's performances thus resulting in limitations on the maximum geographical area that can be monitored with [150]. Furthermore, the system in [150] does not take into account the problem of toponymic polysemy [12, 53]. In addition, crisis maps generated by [150] only consider tweet volumes and may result less accurate than those obtained by analyzing the content of tweets. For example in the case of severe earthquakes, where the shaking is perceived also hundreds of kilometers far from the epicenter, the majority of tweets comes from densely populated areas, such as big cities. Anyway, locations that have suffered most of the damage might be small villages in rural areas around the epicenter, which risk remaining unnoticed if the analysis only considers tweet volumes [12, 53].

In addition to the fully functional crisis mapping system described above, other solutions for the geoparsing task have been recently proposed in [56,86,87] where authors experimented with heuristics, open-source named entity recognition software, and machine learning techniques. Furthermore, other works emphasized the extraction of actionable and time-sensitive information from messages. For instance, authors of [206] apply natural language processing techniques to detect messages carrying relevant information for situational awareness during emergencies. In [105] is described a technique to extract "information nuggets" from tweets – that is, self-contained information items relevant to disaster response. While these works present fully automatic means to extract knowledge from texts, in [207] is proposed a hybrid approach exploiting both human and machine computation to classify messages. All these linguistic analysis techniques for the extraction of relevant information from disaster-related messages have however never been employed in a crisis mapping system.

Finally, a survey presented an extensive review of current literature in the broad field of social media emergency management, and can be considered for additional references [104].

6.8 Conclusions

In this chapter we presented `CrisMap`: an effective software system capable of supporting emergency responders (e.g. national Civic Protection agency) during crisis management of natural or man-made disasters. The system is able to process incoming SM data from Twitter in order to quickly produce crisis maps useful to prioritize the allocation of available resources, especially in the first phases of the crisis, towards the populations and territories most affected by the specific disaster. The proposed solution is designed and built using Big Data technologies, allowing the system to be scalable, fault-tolerant, and able to process incoming data in near-real-time. To overcome the limits resulting from the unstructured nature of Twitter data and to identify useful information for our purposes, we analyze the data using a two-fold perspective. On the one hand, we introduced a damage detection component exploiting word embeddings and a SVM classifier to detect messages reporting damage to infrastructures or injuries to the population. On the other hand, we proposed a message geolocation component that performs the geoparsing task by exploiting online semantic annotators and collaborative knowledge-bases. The approach using word embeddings has also been compared with a traditional one based on classic NLP techniques, pointing out

the potential advantages of the former in relation to complexity and performance of the proposed method. The accuracy and the reliability of the system was validated analytically comparing the experimental results of `CrisMap` against the authoritative data for 2 past disasters. Furthermore, we also performed a qualitative evaluation of the system on a case-study of a recent severe earthquake in Italy for which authoritative data are not available.

The proposed system offers some room for further improvements, to increase the readability of the maps and, in general, for acquiring a better situational awareness of unfolding events. With regards to geoparsing results, recent developments of semantic annotation tools open up the possibility to provide more implementations of our proposed geoparsing technique. Therefore we envision the possibility to simultaneously exploit multiple semantic annotators in an ensemble or voting system. In the future this approach could allow to obtain even better results and to overcome the possible limitations of a single annotator. Moreover, to date, `CrisMap` only exploits textual data, but we believe that multimedia data, like images and live-videos, could contribute critical information for emergency responders. Thus, in the future we aim at providing analytic and visual support for multimedia content.

Other avenues of future experimentation might be related to multi-source mining. Indeed, although Twitter is nowadays one of the preferred SM sources, given its "open" policies on providing data to third parties, data collection from multiple sources could mitigate the bias introduced by the analysis of a single SM.

CHAPTER 7

TEXT2VIS: Projecting Textual Descriptions Into Abstract Visual Representations Using Deep Learning

ABSTRACT

In this chapter we tackle the problem of image search when the query is a short textual description of the image the user is looking for. We choose to implement the actual search process as a similarity search in a visual feature space, by learning to translate a textual query into a visual representation. Searching in the visual feature space has the advantage that any update to the translation model does not require to reprocess the, typically huge, image collection on which the search is performed. We propose TEXT2VIS*, a neural network that generates a visual representation, in the visual feature space of the fc6-fc7 layers of ImageNet, from a short descriptive text.* TEXT2VIS *optimizes two loss functions, using a stochastic loss-selection method. A visual-focused loss is aimed at learning the actual text-to-visual feature mapping, while a text-focused loss is aimed at modeling the higher-level semantic concepts expressed in language and countering the overfit on non-relevant visual components of the visual loss.*

7.1 Introduction

Using a textual query to retrieve images is a very common cross-media search task, as text is the most efficient media to describe the kind of image the user is searching for. The actual retrieval process can be implemented in a number of ways, depending on how the shared search space between text and images is defined. The search space can be based on textual features, visual features, or a joint space in which textual and visual features are projected into.

Using textual features is the most common solution, specially at the Web scale. Each image is associated with a set of textual features extracted from its context of use (e.g., the text surrounding the image in the Web page, description fields in metadata), and eventually enriched by means of classifiers that assign textual labels related to the presence or certain relevant entities or abstract properties in the image. The textual search space model can exploit the actual visual content of the image only when classifiers for the concepts of interest are available, thus requiring a relevant number of classifiers; this also requires to reprocess the entire image collection whenever a new classifier is made available.

On the other side, the visual and joint search spaces represent each image through visual features extracted from its actual content. The method we propose in this paper adopts a visual space search model. A textual query is converted into a visual representation in a visual space, where the search is performed by similarity. An advantage of this model is that any improvement in the text representation model, and its conversion to visual features, has immediate benefits on the image retrieval process, without requiring to reprocess the whole image collection.

A joint space model requires instead a reprocessing of all images whenever the textual model is updated, since the projection of images into the joint space is influenced also by the textual model part. It also requires managing and storing the additional joint space representations that are used only for the cross-media search.

In this chapter we present the results on learning TEXT2VIS, a neural network model that converts textual descriptions into visual representations in the same space of those extracted from deep Convolutional Neural Networks (CNN) such as ImageNet [128]. TEXT2VIS achieves its goal by using a stochastic loss choice on two separate loss functions (as detailed in Section 7.2), one for textual representations autoencoding, and one for visual representations generation. Results show that the produced visual representations are able to capture the high level concepts expressed in the textual description.

7.2 Generating visual representations of text

In this section we describe the architecture of our TEXT2VIS network. Our idea is to map textual descriptions to high-level visual representations. As the visual space we used the *fc6* and *fc7* layers of the Hybrid network [222] (i.e., an AlexNet [128] trained on both ImageNet[1] and Places[2] datasets). We tested two vectorial representations for the textual descriptions: TEXT2VIS$_1$ uses simple bag-of-words vectors that mark with a value of one the positions that are relative to words that appear in the textual description and leave to zero all the others; TEXT2VIS$_N$ adds a bit text structure info by

[1] http://image-net.org
[2] http://places.csail.mit.edu/index.html

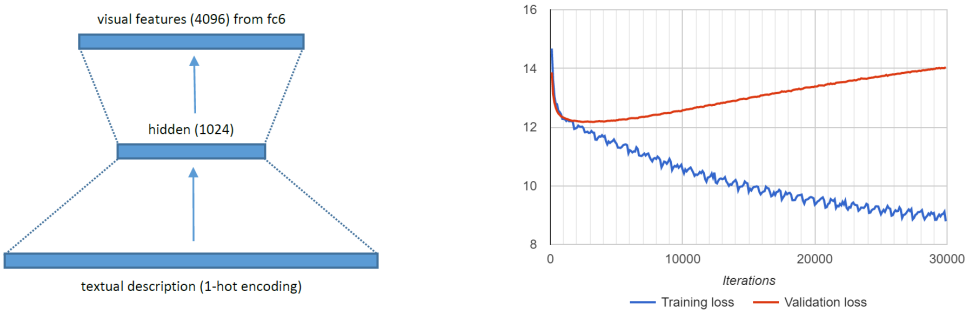


Figure 7.1: *Overfitting of a simple regressor model with one hidden layer of size 1024.*

considering also N-grams for a selection of part-of-speech patterns[3]. TEXT2VIS$_N$ is a first approach at modeling text structure into the input vectorial representation, which differentiates the task of search from detailed/complex textual description we aim at from the traditional keyword search.

We have also investigated the use of pre-trained word embeddings, representing the textual description as the average of the embeddings of the words composing the description (see Equation 1 in [64]), but we have not observed any improvement. Generating the word embeddings is an additional cost, and the fitness of the embeddings for the task depends on the type of documents they are learned from. For example, an 11% improvement in MAP is reported in [34] from learning embedding from Flickr tags compared to learning them from Wikipedia pages. The direct use of bag-of-words vectors in TEXT2VIS removes the variable of selecting an appropriate document collection to learn the embedding and its learning cost.

As described in the following, TEXT2VIS actually learns a description embedding space that is able to reconstruct both the original description and the visual description. To reach this, we started with a simple regressor model (Figure 7.1, left) trained to directly predict the visual representation of the image associated with the textual input. We observed a strong tendency to overfit (Figure 7.1, right), thus degrading the applicability of the method to unseen images.

We explained this overfitting with the fact that a visual representation keeps track of every element that appears in the image, regardless of their semantic relevance within the image, while a (short) textual description is more likely focused on the visually relevant information, disregarding the secondary content of the image, as shown in Figure 7.3.6. As the learning iterations proceed, the simple regressor model starts capturing secondary elements of the images that are not relevant for the main represented concept, but are somewhat characteristic in the training data.

Our TEXT2VIS proposal to contrast such overfitting is to add a text-to-text autoencoding branch to the hidden layer (Figure 7.2, left), forcing the model to satisfy two losses: one visual (text-to-visual regression) and one linguistic (text-to-text autoencoder). The linguistic loss works at higher level of abstraction than the visual one, acting as a regularization constraint on the model, and preventing, as confirmed by our experiments, overfitting on the visual loss (Figure 7.2, right). As detailed in the next section, we implemented the use of the two losses with a stochastic process, in which

[3]We considered the part-of-speech patterns: 'NOUN-VERB', 'NOUN-VERB-VERB', 'ADJ-NOUN', 'VERB-PRT', 'VERB-VERB', 'NUM-NOUN', and 'NOUN-NOUN'.

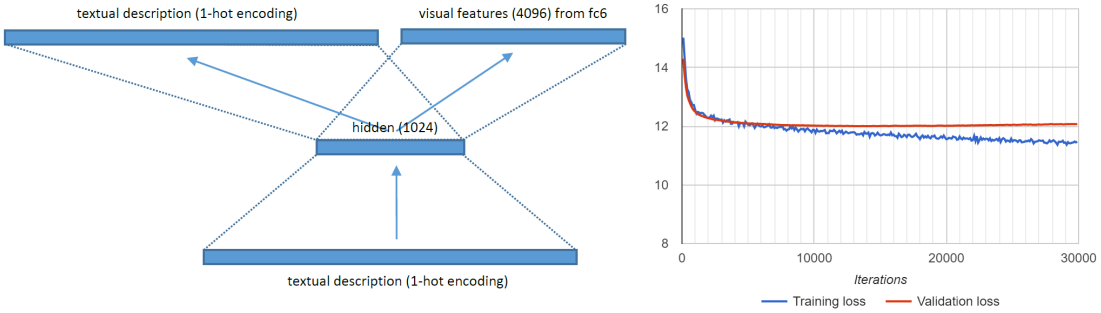


Figure 7.2: *Our proposed* TEXT2VIS *which controls overfitting by adding an autoencoding constraint on the hidden state.*

at each iteration one of the two is selected for optimization.

7.2.1 TEXT2VIS

TEXT2VIS consists[4] of two overlapped feedforward neural nets with a shared hidden layer. The shared hidden layer causes a regularization effect during the combined optimization; i.e., the hidden state is constrained to be a good representation to accomplish with two different goals. The feedforward computation is described by the following equations:

$$z = ReLU(W_1 t_{in} + b_1) \tag{7.1}$$
$$t' = ReLU(W_2 z + b_2) \tag{7.2}$$
$$v' = ReLU(W_3 z + b_3) \tag{7.3}$$

where t_{in} represents the sparse one-hot encoding (bag of words) for the textual descriptor given as input to the net, z is the hidden representation, v' and t' are the visual and textual predictions, respectively, obtained from the hidden representation z, $\Theta = \{W_i, b_i\}_{i \in \{1,2,3\}}$ are the model parameters to be learned, and $ReLU$ is the activation function, defined by $ReLU(x) = \max\{0, x\}$.

Both predictions v' and t' are then confronted with the expected outputs (i) the visual representation v corresponding to the $fc6$ or $fc7$ layers of [128], and (ii) a textual descriptor t_{out} that is semantically equivalent to t_{in}. We used the *mean squared error* (MSE) as the loss function in both cases:

$$\mathcal{L}(x, y; \Theta') = MSE(x, y) = \frac{1}{n} \sum_{i=1}^{n} (x_i - y_i)^2 \tag{7.4}$$

The model is thus multi-objective, and many alternative strategies could be followed at this point in order to set the Θ parameters so that both criteria are jointly minimized. We rather propose a much simpler, yet effective, way for carrying out the optimization search, that consists of considering both branches of the net as independent, and randomly deciding in each iteration which of them is to be used for the gradient descend optimization.

[4] A Tensorflow implementation of TEXT2VIS is available at `https://github.com/AlexMoreo/tensorflow-Tex2Vis`.

Let thus define $\Theta_t = \{W_i, b_i\}_{i\in\{1,2\}}$ and $\Theta_v = \{W_i, b_i\}_{i\in\{1,3\}}$ as the model parameters of each independent branch. The optimization problem has two objectives (Equations 7.5 and 7.6), and at each iteration, a random choice decides which of them is to be optimized. We call this heuristic the *Stochastic Loss* (SL) optimization.

$$\Theta_{\widehat{t}} = argmin_{\Theta_t}\mathcal{L}_t(t_{out}, t'; \Theta_t) \tag{7.5}$$

$$\Theta_{\widehat{v}} = argmin_{\Theta_v}\mathcal{L}_v(v, v'; \Theta_v) \tag{7.6}$$

Note that the net is fed with a triple $\langle v, t_{in}, t_{out}\rangle$ at each iteration. When $t_{out} = t_{in}$ the text-to-text branch is an *autoencoder*. It is also possible to have $t_{in} \neq t_{out}$, with the two pieces of text been semantically equivalent (e.g., t_{in} ="`a woman cutting a pizza with a knife`", t_{out} ="`a woman holds a knife to cut pizza`") then the text-to-text branch might be reminiscent of the *Skip-gram*- and *CBOW*-like architectures. The text-to-image branch is, in any case, a regressor. The SL causes the model to be co-regularized. Notwithstanding, since our final goal is to project the textual descriptor into the visual space, the text-to-text branch might be though as a regularization to the visual reconstruction (and, more specifically, to its internal encoding) which responds to constrains of linguistic nature.

7.3 Experiments

7.3.1 Datasets

We used the *Microsoft COCO* dataset (MsCOCO[5] [140]). MsCOCO was originally proposed for image recognition, segmentation, and caption generation. Although other datasets for image retrieval exist (e.g., the one proposed in [103]), they are more oriented to keyword-based queries. We believe MsCOCO to be more fit to the scenario we want to explore, since the captions associated to the images are expressed in natural language, thus semantically richer than a short list of keywords composing a query.

MsCOCO contains 82.783 training images (*Train2014*), 40.504 validation images (*Val2014*), and about 40K and 80K test images corresponding to two different competitions [44] (*Test2014* and *Test2015*). Because MsCOCO was proposed for caption generation, the captions are only accessible in the *Train2014* and *Val2014* sets, while they are not yet released for *Test2014* and *Test2015*. We have thus taken the *Train2014* set for training, and split the *Val2014* into two disjoint sets of 20K images each for validation and test.

Each image in MsCOCO has 5 different captions associated. Let $\langle I, C\rangle$ be any labeled instance in MsCOCO, where I is an image and $C = \{c_1..c_5\}$ is a set of captions describing the content of I. Given a $\langle I, C\rangle$ pair, we define a labeled instance in our model as $\langle v, t_{in}, t_{out}\rangle$, where $v \in \mathbb{R}^{4096}$ is the visual representation of the image I taken from the *fc6* layer (or *fc7*, in separate experiments) of the Hybrid network [222]; t_{in} and t_{out} are two textual descriptors from C representing the input and output descriptors for the model, respectively. During training, t_{in} and t_{out} are uniformly chosen at random from C (thus t_{in} and t_{out} are not imposed to be different). Note that the number of training instances one could extract from a given $\langle I, C\rangle$ amounts to 25, which increases the variability of the training set along the different epochs.

[5] Publicly available at `http://mscoco.org/`

7.3.2 Training

We solve the optimization problems of Equations 7.5 and 7.6, using the *Adam* method [124] for stochastic optimization, with default parameters (learning rate $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 1e^{-0.8}$). Note that there are two independent instances of the Adam optimizer, one associated to $\mathcal{L}_t$ (Equation 7.5) and other for $\mathcal{L}_v$ (Equation 7.6). In this preliminary study we decided to set for the SL an equal selection probability to both $\mathcal{L}_t$ and $\mathcal{L}_v$; different distributions will be investigated in future research.

We set the size of the training batch to 100 examples. We set the maximum number of iterations to 300.000, but apply an early stop when the model starts overfitting (as reflected in the validation error). The training set is shuffled each time a complete pass over all images is completed.

All the Θ parameters have been initialized at random according to a truncated normal distribution centered in zero with standard deviation of $\frac{1}{\sqrt{n}}$, where n is the number of columns. The biases have all been initialized to 0.

The vocabulary size is 10,358 for TEXT2VIS$_1$ after removing terms appearing in less than 5 captions. For TEXT2VIS$_N$ we considered the 23,968 uni-grams and N-grams appearing at least in 10 captions. Since the number of units in the hidden and output layers are 1024 and 4096, respectively, the total number of parameters of the models amount to 25.4M in TEXT2VIS$_1$ and 53.3M in TEXT2VIS$_N$.

7.3.3 Evaluation Measures

Image retrieval is performed by similarity search in the visual space, using Euclidean distance on the l2-normalized visual vectors to generate a ranking of images, sorted by closeness. We measure the retrieval effectiveness of the visual representations produced from textual descriptions by our TEXT2VIS network by means of the *Discounted Cumulative Gain* (DCG [109]), defined as:

$$DCG_p = \sum_{i=1}^{p} \frac{2^{rel_i} - 1}{\log_2(i+1)} \tag{7.7}$$

where rel_i quantifies the *relevance* of the retrieved element at rank position i with respect to the query, and p is the rank at which the metric is computed; we set $p = 25$ in our experiments, as was done in related research [64, 103].

Because the rel values are not provided in the MsCOCO, we estimate them by using the $ROUGE_L$ [137] metric. $ROUGE_L$ is one of the evaluation measures for the MsCOCO caption generation competition[6] [44]. We compute $rel_i = ROUGE_L(t_{in}, C_i)$, where t_{in} is the query caption, and C_i are the 5 captions associated to the retrieved image at rank i. This caption-to-caption relevance model is thus aimed at measuring how much the concepts expressed in the query appear as relevant parts of the retrieved images.

7.3.4 Results

We compared the performance of TEXT2VIS$_1$ and TEXT2VIS$_N$ models against: *RRank*, a lower bound baseline that produces a random ranking of images, for any query; *VisSim*, a direct similarity method that computes the Euclidean distances using the original

[6] https://github.com/tylin/coco-caption

fc6, or *fc7*, features for the image that is associated to query caption in MsCOCO; and *VisReg*, the text-to-image regressor described in Figure 7.1.

Table 7.1 reports the averaged DCG scores obtained by the compared methods. These results show a significant improvement of our proposal with respect to the compared methods. When using $fc6$ as the visual space, TEXT2VIS$_1$ obtains a 8.51% relative improvement with respect to *VisSim* and 1.40% over *VisReg*. The improvements of TEXT2VIS$_N$ are respectively of 8.08% and 0.94%. When using $fc7$ as the visual space it is TEXT2VIS$_N$ that obtains, yet by a small margin, the best result. The relative improvements of TEXT2VIS$_1$ over emphVisSim and *VisReg* are respectively of 8.48% and 0.97%, and for TEXT2VIS$_N$ respectively of 8.60% and 1.09%.

Method	fc6	fc7
RRank	1.524	1.524
VisSim	2.150	2.180
VisReg	2.317	2.359
TEXT2VIS$_1$	**2.350**	2.382
TEXT2VIS$_N$	2.339	**2.385**

Table 7.1: *Performance comparison of the different methods in terms of averaged DCG*

In addition to the averaged performance, we also investigated how often the ranking produced by TEXT2VIS is more relevant (according to DCG) than those produced by *VisSim* and *VisReg*. Figure 7.3 indicates that in 69.2% of the cases, the ranking of TEXT2VIS$_1$ was found more relevant than *VisSim* (see Figure 7.3). The same happens in 58.1% of the cases when comparing TEXT2VIS to *VisReg*.

7.3.5 Why Stochastic Loss?

TEXT2VIS uses two independent optimizers to optimize the visual ($\mathcal{L}_v$) and the textual ($\mathcal{L}_t$) losses, based on a stochastic choice at each iteration (SL, section 7.2.1). Previous approaches to multimodal learning relied instead on a unique aggregated loss (typically of the form $\mathcal{L} = \mathcal{L}_v + \lambda\mathcal{L}_t$) that is minimized by a single optimizer [75, 162]. We compared the two approaches on the case of equal relevance of the two losses ($\lambda = 1$, uniform distribution for SL). SL better optimizes the two losses (Figure 7.4), and is less prone to overfit.

We deem that SL allows to model in a more natural way the relative relevance of the various losses that are combined, i.e., by selecting the losses in proportion to the assigned relevance, whereas the numeric aggregation is affected by the relative values of losses and the differences in their variation during the optimization (e.g., a loss that has a large improvement may compensate for another loss getting worse). SL is also computationally lighter than the aggregated loss, as SL updates only a part of the model on each iteration.

7.3.6 Visual comparison

Figure 7.5 show a few samples[7] that highlight the differences in results from the three compared methods. In all the cases results from the *VisSim* method are dominated by the main visual features of the images: a face for the first query, the content of the screen

[7]More results at `https://github.com/AlexMoreo/tensorflow-Tex2Vis`

for the second query, an outdoor image with a light lower part, plants, people and a bit of sky in the third one. The two text based methods obtains results that more often contain the key elements of the description. For the first query, TEXT2VIS retrieves four relevant images out of five, one more that *VisReg*. For the other two queries the results are pretty similar, with TEXT2VIS placing in second position an image that is a perfect match for the query, while *VisReg* places it in fifth position.

7.4 Related work

Deep Learning and Deep Convolutional Neural Networks (DCNNs) in particular, have recently shown impressive performance on a number of multimedia information retrieval tasks [100, 128, 193]. Deep Learning methods learn representations of data with multiple levels of abstraction. As a result, the activation of the hidden layers has been used in the context of transfer learning and content-based image retrieval [63, 175] as high-level representations of the visual content. Somewhat similarly, distributional semantic models, such as those produced by Word2Vec [152], or GloVe [171], have been found useful in modeling semantic similarities among words by establishing a correlation between *word meaning* and *position* in a vector space.

In order to perform cross-media retrieval, the two feature spaces (text and images in our case) should become comparable, typically by learning how to properly map the different sources. This problem has been attempted in different manners so far, which could be roughly grouped into three main variants, depending on whether the mapping is performed into a common space, the textual space, or the visual space.

Mapping into a common space: The idea of comparing texts and images in a shared space has been investigated by means of Cross-modal Factor Analysis and (Kernel) Canonical Correlation Analysis in [51]. In a similar vein, Corr-AE was proposed for cross-modal retrieval, allowing the search to be performed in both directions, i.e., from text-to-image and viceversa [75]. The idea is to train two autoencoders, one for the image domain and another for the textual domain, imposing restrictions between the two. As will be seen, the architecture we are presenting here bears resemblance to one of the architectures investigated in [75], the so-called *Correspondence full-modal autoencoder* (which is inspired by the multimodal deep learning method [162]). Contrarily to the multimodal architectures though, we apply a stochastic criterion to jointly optimize for the two modals, thus refraining from combining them into a parametric single loss.

Mapping into the textual space: The BoWDNN method trains a deep neural network (DNN) to map images directly into a bag-of-words (BoW) space, where the cosine similarity between BoWs representations is used to generate the ranking [18]. Somehow similarly, a dedicated area of related research is focused on generating captions describing the salient information of an image (see, e.g., [73, 122]).

Two other important examples along these lines are DeViSE [82] and ConSE [165]. Both methods build upon the higher layers of the convolutional neural network of [128]; the main difference lies on the way both methods treat the last layer of the net. Whereas DeViSE replaces this last layer with a linear mapping (thus fine-tuning the whole network) ConSE, on the other side, directly takes the outputs of the last layer and learns a projection to the textual embedding space.

Mapping into the visual space: Our proposal TEXT2VIS belongs to this group where, to the best of our knowledge, the only example up to now was a method dubbed *Word2VisualVec* [64], which was reported just very recently. There are some fundamental points where their method and ours differ, though. On the one hand, their *Word2VisualVec* takes combinations of Word2Vec-like vectors as a starting point, thus reducing the dimensionality of the input space; we directly start from the bag-of-words vector encoding of the textual space, as we did not observed any improvement in pre-training the textual part. On the other, they build a deep network on top of the textual representation. As shall be seen, our TEXT2VIS is much shallower, as we found the net to be capable of mapping textual vectors into the visual space quite efficiently, provided that the model is properly regularized; an issue on which we focused our attention.

7.5 Conclusions

Results of our experiments indicate that our method produces better rankings than those produced by similarity search apply directly on the visual features of a query image. This is an indication that our text-to-image mapping produces better prototypical representations of the desired scene than the representation of a sample image itself. A simple explanation of this result is that textual descriptions strictly emphasizes the relevant aspects of the scene the user has in mind, whereas the visual features, directly extracted from the query image, are keeping track of all the information that is contained in that specific image, causing the similarity search to be potentially confused by secondary elements of the scene.

The TEXT2VIS model also improved, yet by a smaller margin, over the *VisReg* model , showing that an auto-enconding branch in the network is useful to avoid overfitting on visual features.

We also found that combing losses in a stochastic fashion, rather than numerically, improves both the effectiveness and efficiency of the system.

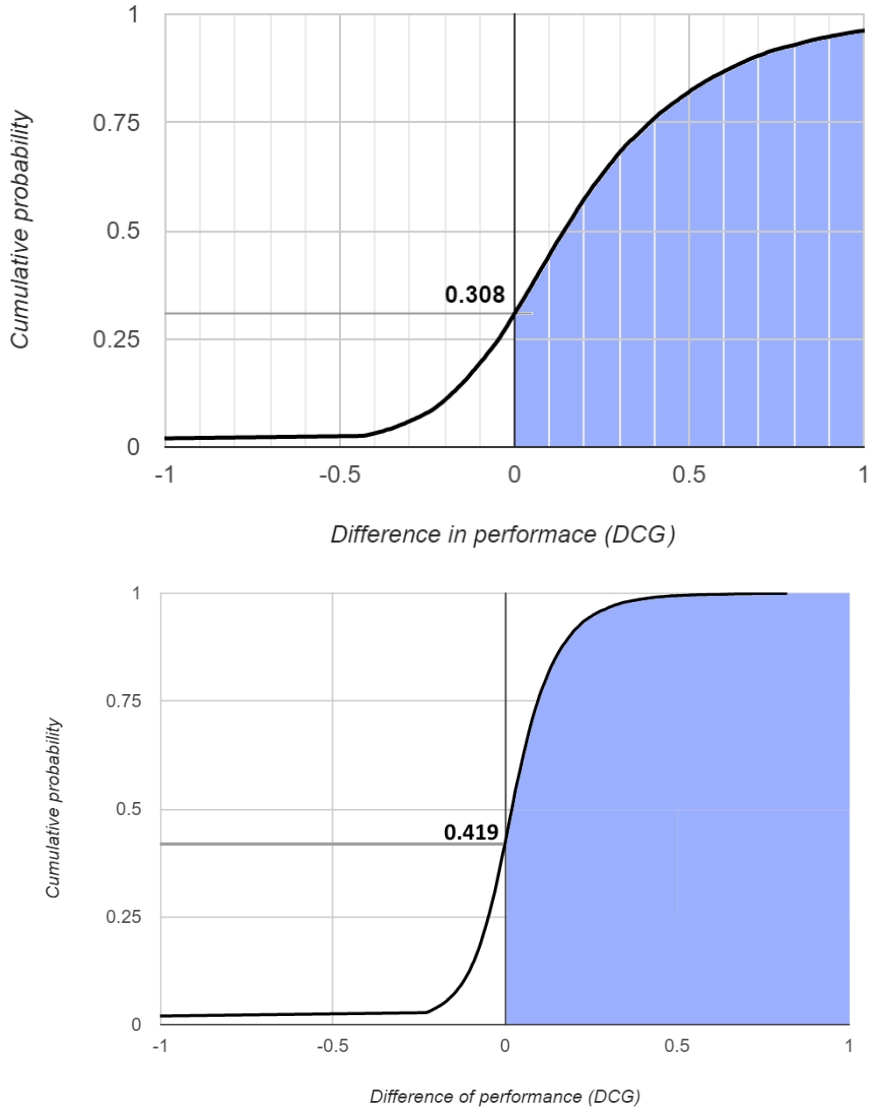


Figure 7.3: *Cumulative probability distribution of the difference in performance of our* TEXT2VIS$_1$ *with respect to* VisSim *(upper plot) and* VisReg *(lower plot), on* $fc6$*. Positive differences mean* TEXT2VIS *obtained a better ranking score than* VisSim *or* VisReg *(resp. 69.2% and 58.1% of cases, shadowed region).*

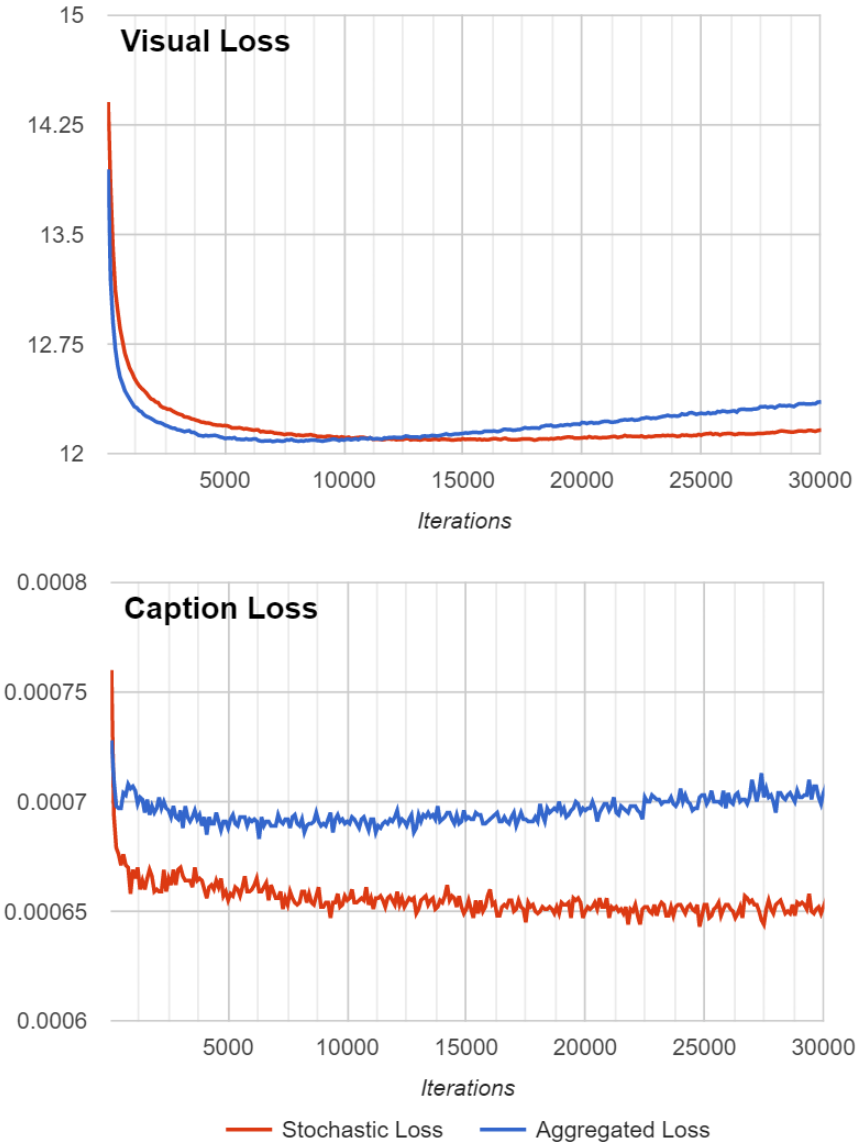


Figure 7.4: *Validation loss for $\mathcal{L}_v$ and $\mathcal{L}_t$, optimizing on a linear combination of losses (blue) or using two optimizers with stochastic loss selection (SL, red).*

Figure 7.5: *Examples of search results from the three compared methods.*

CHAPTER 8

Conclusions

The extreme popularity gained in the last years by Internet, and the Web in particular, as the main communication tool used by users and companies has determined an exponential growth of generated information potentially useful for data analysis tasks. A large part of this generated data is represented by textual content, which is an information type completely unstructured, which requires complex statistical and linguistic analyses to obtain helpful structured and semantic information from the original data. Efficiently analyze large amount of texts is not a trivial task. Existing text mining software and libraries are generally optimized for sequential code execution on single machines. Furthermore, these software packages usually provide a limited set of specific features for text analyses, imposing to programmers great efforts in integration tasks required to build complex and complete software solutions. In addition, no existing big data processing tools provide specialized libraries able to natively process textual contents. This issue's solution is demanded to developers, which have the option of a) try to adapt existing text mining or NLP tools to the adopted big data software (usually a far from trivial task) or b) rewrite from scratch specific algorithms or techniques using the features provided by the used big data processors.

8.1 Summary of results

The main aims of this thesis were to propose original and effective textual processing tools and solutions for complex problems involving the analyses of "big data" data sources. Given the issues described previously, we focused on several critical aspects which usually render the design and deployment of such solutions a very tricky task for programmers. We thus identified three interesting big data application scenarios where implementing effective text mining analyses are common critical factors in order to

provide effective solutions. Each scenario differs from the others in terms of amount of data to process, amount of computational resources available and complexity of the problem to be solved.

In the first application scenario, covered by Chapters 2, 3 and 4, we focused on design and implementation of standalone text processing tools, optimized for sequential execution on single commodity machines and able to process efficiently small to medium amount of data. Our contribution in this context has been two-fold.

At first, in Chapter 2 we proposed JATECS, a new Java framework specialized on the creation of generic text analytical software covering all the typical phases of such solutions: data acquisition, data preprocessing, data indexing, data models learning and data models evaluation. The software has been designed to be highly modular, allowing to quickly test ingestion and ML processing pipelines where each analytical module can be easily changed on the fly in a plug and play style. The library is able to ingest data from multiple different data sources (e.g. standard datasets like Reuters21578, RCV1v2, etc. or generic data sources like CSV and ARFF formats) and provides many standard statistical and NLP methods for features extraction, dimensionality reduction and features weighting.

In JATECS, differently from all the other existing text processing software, there is clear and unique point of access to managed data through the concept of index, an in-memory data structure managing all the relations existing from the main objects (documents, categories and features) resulting from the data ingestion process. We have shown through several code examples that index structure allows the programmers to quickly and efficiently manipulate and access all the important information related to the analyzed data. Moreover, it allows to write machine learning algorithms using an higher level view over data, avoiding working directly with feature vectors and thus simplifying the implementation task in many cases. The set of ML algorithms and models optimization techniques provided in JATECS allow to cover many typical textual processing use cases. Indeed, it easily allows developers to handle many types of classification tasks (e.g. flat/hierarchical classifiers of type multilabel, multiclass or binary), clustering, textual quantification and some other advanced tasks like transfer learning, active learning, training data cleaning and semi-automated text classification.

In Chapters 3 and 4, as additional contribution for first application scenario, we have shown that JATECS library has been used with success as core text analytics technology to facing on two specific interesting use cases. Considering that classification of companies by industry sector is an important task in finance, in Chapter 3 we proposed an automated classification system which handle this specific problem by performing classifications of company websites. The proposed system can work in autonomous modality or as an assistive technology for the users. In the first case, the software works in "hard classification mode", placing each company website into exactly one leaf node in the taxonomy, while in the second case works in "soft classification mode", returning a ranked list of the k leaf nodes that appear the most plausible for the website. The features extraction phase has been approached generating data features from two possible domain sets. Endogenous features have been extracted analyzing the internal website data (e.g. page url, page title, page body, number of outgoing links in the page, etc.) while exogenous features have been obtained integrating analyses of data coming from resources external to companies sites (i.e. Facebook, Twitter and Alexa).

Our experimental results, obtained from a hierarchical dataset containing more than 20,000 websites, show that our approach, based on TreeSVM algorithm, can guarantee levels of classification accuracy extremely interesting, both in hard and in soft classification mode. The best results were obtained using all types of extracted features but the results also suggest that some of these taken alone (e.g. url or title) contribute remarkably in terms of effectiveness on the built classifier. In Chapter 4, we took a similar approach as described for Chapter 3 to build an automatic mobile apps classifier based on custom taxonomies defined by users. In particular, we extracted mobile applications features from meta-information gathered from Google PlayStore and iTunes stores and we tested our system on a flat dataset consisting of more than 5,000 different mobile apps. In this case, we found that classification via supervised learning under non-trivial taxonomies is a difficult task. The obtained system accuracy was indeed suboptimal; in particular, it seems that in addition to used metadata information, more informative features need to be obtained in order to better model the input domain. We left this insight for future works on this research topic.

In the thesis we dealt with a second application scenario where the main aim is to efficiently process huge amount of textual data both in batch and streaming contexts. Facing on this scenario requires proper architectural choices and valid integration strategies to put together the various technologies involved in the creation of the software. The running environment is also another key point to consider when deciding which tools to adopt in order to deploy effective solutions. In Chapter 5 we explored various ways to design scalable textual processing products. A first approach, targeting running environments having few computational resources available (typically a single multicore machine), tried to understand if an existing distributed big data processing tool could be used efficiently on such environments or if an optimized multithread solution can works much better. We thus proposed Processfast, a new multithread Java/Groovy framework that aims at providing a seamless integration of task and data parallelism computing models using an homogeneous and functional API. We have benchmarked it against two popular existing big data software like Hadoop and Spark and our experimentation showed that, although our proposed solution was the top performer in this context, the performance difference from Spark was rather small. These results suggested us to adopt Spark when moving to distributed solutions, using it as the core technology to provide higher level textual processing functionalities. A contribution in this sense was the design and implementation of the text mining library called NLP4Spark, focused on data ingestion phase and built around the concept of index (as used in JATECS) to quickly access and manipulate indexed information. Thanks to usage of Scala programming language and Spark APIs as the way to custom process available data, we showed that programs can be written using a very simple and concise code, which can be formulated naturally by programmers and its comparable in clarity, if not better, with the corresponding sequential code expressed by using JATECS. A second contribution in providing distributed solutions was to study how to use Spark to develop a distributed implementation of a popular family of ML algorithms called boosting, in particular focusing on AdaBoost.MH and MP-Boost classifiers. We described extensively how these algorithms work and how to exploit Spark APIs to parallelize the code, and we provided an extensive experimentation showing the vertical and horizontal scalability obtainable from distributed implementation on a small computer cluster.

The results have shown that such type of iterative algorithms can greatly benefit from using big data processing tools optimized for this type of problems, confirming that in many cases scalable solutions can be achieved quite easily assuming you choose the right tools.

In Chapter 6 we described how to design and build a complex social media analytics platform aimed to support emergency responders in crisis management of man-made/natural disasters. The implementation of such system has been an interesting experience, because beside providing single advanced processing modules (e.g. data filtering and damage detection, geolocation of textual contents, etc.), a great effort has been dedicated in the architectural design of the software and on the adopted integration strategies to put together the various used, and unrelated, big data technologies. Indeed, the chosen architecture and software stack allowed us to deploy a scalable solution which is fault-tolerant and also easily adaptable to different use cases requiring fast streaming computations. The software supports emergency agencies by pointing out relevant social media messages (tweets coming from Twitter in our case) containing damage information about people or buildings, and by geo-tagging them. The interesting data can then be monitored and visualized in near-real time on specialized Web dashboards in order to support the work of rescuers. We detected damage information on unstructured tweets using an SVM classifier exploiting word embeddings as textual feature representation. We have proven that this approach allow to obtain similar effectiveness to a standard one based on NLP techniques but it has the advantages of being essentially language independent, avoiding the "feature engineering" task and being very cheap computationally. We tested the system goodness analytically against authoritative data coming from two past Italian disasters and we also provided a qualitative study on Amatrice's italian earthquake for which authoritative data is not currently available. The results obtained are very promising, showing that our system is able to circumscribe rather correctly the most damaged areas in stricken territories. Moreover, another interesting thing is that the software is able to generate credible damage maps of a disaster after few tens of minutes the event has occurred. For instance, authoritative data of a disaster is usually available several months or years after the event, so the software rapidity in generating approximate but realistic maps is an important added value.

The third application scenario we were interested to explore in this thesis was related to implementation of standalone "big data" textual processing solutions focused on the resolution of very complex problems. Such type of solutions require advanced information analytics on high amount of data in order to learn rich ML models exposing complicated and hidden correlations in the analyzed data. Deep learning methods are usually effective end efficient techniques to facing on such type of problems. In Chapter 7 we explored the usage of such techniques by proposing a cross-media retrieval software able to translate a textual query into a visual space query and thus use this visual representation to search for similar images into a generic input pictures storage (e.g. a classic DBMS). The strategy adopted of mapping textual features into visual space has a pair of advantages respect to using other existing approaches (e.g. using textual features for image retrieval or using a common latent space for text and images in retrieval process). A first advantage is that the images features representation is independent from the model projecting text into visual features, i.e. an improve-

ment in model projections alone has immediate benefits in retrieval process. A second important advantage is that we can build specialized projection models specific for each natural language we are interested in, e.g. one specific for italian, one for english, and so on. We thus proposed Text2Vis, a neural network regressor consisting in two overlapped feedforward nets (one for text autoencoded features and one for visual features) sharing a common hidden layer providing regularization effects during learning process. Being a multi-objective model , as the optimization function we adopted a stochastic loss which randomly choice at each iteration of the algorithm the specific function loss (text or visual) to use, helping thus to co-regularize the model. The regressor proposed learned to generate visual features into the same space as those obtained from using popular deep CNN like AlexNet and ReesNet-152 trained on ILSVRC12 and Places datasets. As the input dataset to train the model mapping textual queries into visual space vectors, we used McCOCO which consists in over 40,000 training images, where each one image is described using 5 different descriptions. We tested the effectiveness of our proposed method against a simple neural network regressor and a pair of relevant baselines. The results showed that our method remarkably improves all the other tested methods, providing also very interesting qualitative results in real images retrieval tasks.

8.2 Future research directions

As depicted in the introduction of this thesis, in the next years we can expect a tremendous growing demand of integrated text processing tools easily adaptable to all big data scenarios. The word "integration" is probably the key concept to explore in order to provide effective scalable text mining tools merging NLP capabilities with classic, or deep learning ML algorithms, and data mining methods. The increasing amount of available processable textual data is determining a gradual but inexorable movement from sequential software solutions to high performance distributed and parallel ones. For these last running environments, it would be thus important to invest time in designing software tools like libraries or frameworks helping programmers to be more productive, reducing at minimum the efforts spent on repetitive and common tasks typical of such scenarios.

JATECS is a great example for sequential code of the type of work that we should do also in distributed or parallel environments. It covers all the steps and components that are usually part of a complete text mining pipeline, providing a framework including pluggable components, which indicates clearly to developers how to build a complete text processing solution. In addition, the library provides a lot of ready to use components which can be used to resolve quickly a lot of typical text mining tasks. In this optic, the programmers need only to choose the right modules in order to solve a specific problem, e.g. feature extraction with chars n-grams, feature selection with ChiSquare, and so on. An additional advantage of JATECS was the proposal of the Index data structure. Such high level structure guarantees a great flexibility in data fruition and it simplifies the development.

When moving to distributed and parallel environments, it would be desirable to have solutions similar in spirit to JATECS, including all the aspects discussed above plus the ability to scale over data in a fault-tolerant way. In distributed contexts, the library

NLP4Spark proposed in Chapter 5 is a first solution into that direction, showing that such type of integration is feasible without building complex to use tools. The library is currently partially incomplete in terms of available features but it provides a solid architectural and code base which could be extended in the future by adding many of the methods and algorithms found on JATECS. Moreover, to promote interoperability between different software, additional efforts could be spent for data integration with other existing consolidated tools, e.g. adding to library the possibility to import or export data to tools like SparkML library, Apache Mahoout, etc. .

A similar type of work should be done also for parallel environments, in particular in deep learning contexts. Almost all currently existing deep learning libraries provide a bunch of methods and techniques specific for those type of algorithms (e.g. neural network architectures like CNN or LSTM, activation functions, loss functions, etc.) but for example, they marginally support the process of data ingestion, specially from text, i.e. how the feature are extracted and preprocessed from raw input data. Indeed the assumption of these libraries is to have access to a vector representation of the input data, how to obtain it is a task demanded to programmers with the consequent integration issues which need to be manually solved case by case without a systematic approach valid once and for all.

Another interesting research topic linked to software integration could be related to the design and implementation of hybrid systems which mix traditional big data processing tools with deep learning techniques. Merging these types of technologies and approaches could be very interesting because an hybrid software could provide solutions to real problems, requiring the capacity to analyze a lot of data in near-real time with very complex learning models. In this application scenario the technologies involved have very different requirements. Traditional big data tools use software infrastructures running on cheap commodity machines, while deep learning software require the usage of expensive machines equipped with powerful GPUs. Finding an optimal way to merge these two approaches to have benefit from both sides is not a trivial task, as demonstrated by some recent works on this topic [8, 134, 158]. On the other hand, the research on this topic is yet very limited, so there is a lot of space for improvements and innovation about the current state of the art.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016.

[2] Dimitris Achlioptas. Database-friendly random projections. In *Proceedings of the 20th ACM Symposium on Principles of Database Systems (PODS 2001)*, pages 274–281, Santa Barbara, US, 2001.

[3] Charu C Aggarwal and ChengXiang Zhai. *Mining text data*. Springer Science & Business Media, 2012.

[4] Charu C Aggarwal and ChengXiang Zhai. A survey of text clustering algorithms. In *Mining text data*, pages 77–128. Springer, 2012.

[5] Eugene Agichtein and Luis Gravano. Snowball: Extracting relations from large plain-text collections. In *Proceedings of the Fifth ACM Conference on Digital Libraries*, DL '00, pages 85–94, New York, NY, USA, 2000. ACM.

[6] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[7] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015.

[8] Mohammad Abu Alsheikh, Dusit Niyato, Shaowei Lin, Hwee-Pink Tan, and Zhu Han. Mobile big data analytics using deep learning and apache spark. *IEEE network*, 30(3):22–29, 2016.

[9] Robert W. Kennard Arthur E. Hoerl. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67, 1970.

[10] Marco Avvenuti, Salvatore Bellomo, Stefano Cresci, Mariantonietta Noemi La Polla, and Maurizio Tesconi. Hybrid crowdsensing: A novel paradigm to combine the strengths of opportunistic and participatory crowdsensing. In *Proceedings of the 26th International Conference on World Wide Web Companion*, pages 1413–1421. International World Wide Web Conferences Steering Committee, 2017.

[11] Marco Avvenuti, Mario GCA Cimino, Stefano Cresci, Andrea Marchetti, and Maurizio Tesconi. A framework for detecting unfolding emergencies using humans as sensors. *SpringerPlus*, 5(1):43, 2016.

[12] Marco Avvenuti, Stefano Cresci, Fabio Del Vigna, and Maurizio Tesconi. Impromptu crisis mapping to prioritize emergency response. *Computer*, 49(5):28–37, 2016.

[13] Marco Avvenuti, Stefano Cresci, Mariantonietta N La Polla, Andrea Marchetti, and Maurizio Tesconi. Earthquake emergency management by social sensing. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2014 IEEE International Conference on*, pages 587–592. IEEE, 2014.

[14] Marco Avvenuti, Stefano Cresci, Andrea Marchetti, Carlo Meletti, and Maurizio Tesconi. EARS (Earthquake Alert and Report System): a real time decision support system for earthquake crisis management. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1749–1758. ACM, 2014.

[15] Marco Avvenuti, Stefano Cresci, Andrea Marchetti, Carlo Meletti, and Maurizio Tesconi. Predictability or early warning: Using social media in modern emergency response. *IEEE Internet Computing*, 20(6):4–6, 2016.

[16] Stefano Baccianella, Andrea Esuli, and Fabrizio Sebastiani. Multi-facet rating of product reviews. In *European Conference on Information Retrieval*, pages 461–472. Springer, 2009.

[17] Stefano Baccianella, Andrea Esuli, and Fabrizio Sebastiani. Sentiwordnet 3.0: An enhanced lexical resource for sentiment analysis and opinion mining. In *LREC*, volume 10, pages 2200–2204, 2010.

[18] Yalong Bai, Wei Yu, Tianjun Xiao, Chang Xu, Kuiyuan Yang, Wei-Ying Ma, and Tiejun Zhao. Bag-of-words based deep neural network for image retrieval. In *Proceedings of the ACM International Conference on Multimedia*, pages 229–232. ACM, 2014.

[19] Gustavo E. Batista, Ronaldo C. Prati, and Maria C. Monard. A study of the behavior of several methods for balancing machine learning training data. *ACM SIGKDD Explorations*, 6(1):20–29, 2004.

[20] Jennifer Bauduy. Mapping a crisis, one text message at a time. *Social Education*, 74(3):142–143, 2010.

[21] Antonio Bella, Cesar Ferri, José Hernández-Orallo, and Maria Jose Ramirez-Quintana. Quantification via probability estimators. In *Data Mining (ICDM), 2010 IEEE 10th International Conference on*, pages 737–742. IEEE, 2010.

[22] Shai Ben-David, John Blitzer, Koby Crammer, and Fernando Pereira. Analysis of representations for domain adaptation. In *Proceedings of the 20th Annual Conference on Neural Information Processing Systems (NIPS 2006)*, pages 137–144, Vancouver, CA, 2006.

[23] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE Trans. Pattern Anal. Mach. Intell.*, 35(8):1798–1828, August 2013.

[24] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *J. Mach. Learn. Res.*, 3:1137–1155, March 2003.

[25] Giacomo Berardi, Andrea Esuli, and Fabrizio Sebastiani. A utility-theoretic ranking method for semi-automated text classification. In *Proceedings of the 35th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2012)*, pages 961–970, Portland, US, 2012.

[26] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13:281–305, February 2012.

[27] Upasna Bhandari, Kazunari Sugiyama, Anindya Datta, and Rajni Jindal. Serendipitous recommendation for mobile apps using item-item similarity graph. In *Proceedings of the 9th Conference of Asian Information Retrieval Societies (AIRS 2013)*, pages 440–451, Singapore, SN, 2013.

[28] Steven Bird, Ewan Klein, and Edward Loper. *Natural Language Processing with Python*. O'Reilly Media, Inc., 1st edition, 2009.

[29] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[30] John Blitzer, Mark Dredze, and Fernando Pereira. Biographies, Bollywood, boom-boxes and blenders: Domain adaptation for sentiment classification. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics (ACL 2007)*, pages 440–447, Prague, CZ, 2007.

[31] Kenneth Bloom, Navendu Garg, and Shlomo Argamon. Extracting appraisal expressions. In *HLT-NAACL*, volume 2007, pages 308–315, 2007.

[32] L Burks, M Miller, and R Zadeh. Rapid estimate of ground shaking intensity by combining simple earthquake characteristics with tweets. In *10th US National Conference on Earthquake Engineering*, 2014.

[33] Erik Cambria and Bebo White. Jumping nlp curves: A review of natural language processing research. *IEEE Computational intelligence magazine*, 9(2):48–57, 2014.

[34] Spencer Cappallo, Thomas Mensink, and Cees G.M. Snoek. Image2emoji: Zero-shot emoji prediction for visual media. In *Proceedings of the 23rd ACM International Conference on Multimedia*, MM '15, pages 1311–1314, New York, NY, USA, 2015. ACM.

[35] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[36] Claudio Carpineto, Stanisiaw Osiński, Giovanni Romano, and Dawid Weiss. A survey of Web clustering engines. *ACM Computing Surveys*, 41(3), 2009.

[37] Soumen Chakrabarti, Byron Dom, Rakesh Agrawal, and Prabhakar Raghavan. Scalable feature selection, classification and signature generation for organizing large text databases into hierarchical topic taxonomies. *The VLDB Journal*, 7(3):163–178, August 1998.

[38] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *ACM Sigplan Notices*, volume 45, pages 363–375. ACM, 2010.

[39] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[40] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. SMOTE: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16(1):321–357, 2002.

[41] Enhong Chen, Yanggang Lin, Hui Xiong, Qiming Luo, and Haiping Ma. Exploiting probabilistic topic models to improve text categorization under class imbalance. *Information Processing & Management*, 47(2):202–214, 2011.

[42] Hsinchun Chen, Roger HL Chiang, and Veda C Storey. Business intelligence and analytics: From big data to big impact. *MIS quarterly*, 36(4), 2012.

[43] Ning Chen, Jialiu Lin, Steven C. Hoi, Xiaokui Xiao, and Boshen Zhang. AR-miner: Mining informative reviews for developers from mobile app marketplace. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*, pages 767–778, Hyderabad, IN, 2014.

[44] Xinlei Chen, Hao Fang, Tsung-Yi Lin, Ramakrishna Vedantam, Saurabh Gupta, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco captions: Data collection and evaluation server. *arXiv preprint arXiv:1504.00325*, 2015.

[45] Zhiyuan Cheng, James Caverlee, and Kyumin Lee. You are where you tweet: a content-based approach to geo-locating twitter users. In *Proceedings of the 19th ACM international conference on Information and knowledge management*, pages 759–768. ACM, 2010.

[46] François Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[47] Wei Chu and S Sathiya Keerthi. Support vector ordinal regression. *Neural computation*, 19(3):792–815, 2007.

[48] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM, 2008.

[49] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537, 2011.

[50] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Mach. Learn.*, 20(3):273–297, September 1995.

[51] Jose Costa Pereira, Emanuele Coviello, Gabriel Doyle, Nikhil Rasiwasia, Gert RG Lanckriet, Roger Levy, and Nuno Vasconcelos. On the role of correlation and abstraction in cross-modal multimedia retrieval. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 36(3):521–535, 2014.

[52] Stefano Cresci, Marco Avvenuti, Mariantonoietta La Polla, Carlo Meletti, and Maurizio Tesconi. Nowcasting of earthquake consequences using big social data. *IEEE Internet Computing (forthcoming)*, 2016.

[53] Stefano Cresci, Andrea Cimino, Felice Dell'Orletta, and Maurizio Tesconi. Crisis mapping during natural disasters via text analysis of social media messages. In *Web Information Systems Engineering–WISE 2015*, pages 250–258, 2015.

[54] Stefano Cresci, Maurizio Tesconi, Andrea Cimino, and Felice Dell'Orletta. A linguistically-driven approach to cross-event damage assessment of natural disasters from social media messages. In *Proceedings of the 24th International Conference on World Wide Web Companion*, pages 1195–1200. International World Wide Web Conferences Steering Committee, 2015.

[55] Dimitrios Damopoulos, Sofia A. Menesidou, Georgios Kambourakis, Maria Papadaki, Nathan Clarke, and Stefanos Gritzalis. Evaluation of anomaly-based IDS for mobile devices using machine learning classifiers. *Security and Communication Networks*, 5(1):3–14, 2012.

[56] Maxwell Guimarães de Oliveira, Cláudio de Souza Baptista, Cláudio EC Campelo, José Amilton Moura Acioli Filho, and Ana Gabrielle Ramos Falcão. Producing Volunteered Geographic Information from social media for LBSN improvement. *Journal of Information and Data Management*, 6(1):81, 2015.

[57] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[58] Scott C. Deerwester, Susan T Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

[59] Li Deng. A tutorial survey of architectures, algorithms, and applications for deep learning. *APSIPA Transactions on Signal and Information Processing*, 3, 2014.

[60] Li Deng, Dong Yu, et al. Deep learning: methods and applications. *Foundations and Trends® in Signal Processing*, 7(3–4):197–387, 2014.

[61] J. Dinan, S. Krishnamoorthy, D.B. Larkins, Jarek Nieplocha, and P. Sadayappan. Scioto: A framework for global-view task parallelism. In *Parallel Processing, 2008. ICPP '08. 37th International Conference on*, pages 586–593, Sept 2008.

[62] Gianluca Dini, Fabio Martinelli, Andrea Saracino, and Daniele Sgandurra. MADAM: A multi-level anomaly detector for Android malware. In *Proceedings of the 6th International Conference on Mathematical Methods, Models and Architectures for Computer Network Security (MMM-ACNS 2012)*, pages 240–253. St. Petersburg, RU, 2012.

[63] Jeff Donahue, Yangqing Jia, Oriol Vinyals, Judy Hoffman, Ning Zhang, Eric Tzeng, and Trevor Darrell. Decaf: A deep convolutional activation feature for generic visual recognition. *arXiv preprint arXiv:1310.1531*, 2013.

[64] J. Dong, X. Li, and C. G. M. Snoek. Word2VisualVec: Cross-Media Retrieval by Visual Feature Prediction. *ArXiv e-prints*, April 2016.

[65] Paul S Earle, Daniel C Bowden, and Michelle Guy. Twitter earthquake detection: earthquake monitoring in a social world. *Annals of Geophysics*, 54(6), 2012.

[66] Andrea Esuli, Tiziano Fagni, and Fabrizio Sebastiani. MP-Boost: A multiple-pivot boosting algorithm and its application to text categorization. In *Proceedings of the 13th International Symposium on String Processing and Information Retrieval (SPIRE 2006)*, pages 1–12, Glasgow, UK, 2006.

[67] Andrea Esuli, Tiziano Fagni, and Fabrizio Sebastiani. MP-Boost: A Multiple-Pivot Boosting Algorithm and Its Application to Text Categorization. In Fabio Crestani, Paolo Ferragina, and Mark Sanderson, editors, *String Processing and Information Retrieval*, number 4209 in Lecture Notes in Computer Science, pages 1–12. Springer Berlin Heidelberg, October 2006.

[68] Andrea Esuli, Tiziano Fagni, and Fabrizio Sebastiani. *TreeBoost.MH: A Boosting Algorithm for Multi-label Hierarchical Text Categorization*, pages 13–24. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[69] Andrea Esuli and Alejandro Moreo Fernández. Distributional correspondence indexing for cross-language text categorization. In *Proceedings of the 37th European Conference on Information Retrieval (ECIR 2015)*, pages 104–109, Wien, AT, 2015.

[70] Andrea Esuli and Fabrizio Sebastiani. Training data cleaning for text classification. *ACM Transactions on Information Systems*, 31(4), 2013.

[71] Tiziano Fagni and Fabrizio Sebastiani. Selecting negative examples for hierarchical text classification: An experimental comparison. *Journal of the American Society for Information Science and Technology*, 61(11):2256–2265, 2010.

[72] Tiziano Fagni and Fabrizio Sebastiani. Selecting negative examples for hierarchical text classification: An experimental comparison. *Journal of the American Society for Information Science and Technologies*, 61(11):2256–2265, 2010.

[73] Hao Fang, Saurabh Gupta, Forrest Iandola, Rupesh K Srivastava, Li Deng, Piotr Dollár, Jianfeng Gao, Xiaodong He, Margaret Mitchell, John C Platt, et al. From captions to visual concepts and back. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1473–1482, 2015.

[74] Ronen Feldman and James Sanger. *The text mining handbook: advanced approaches in analyzing unstructured data*. Cambridge university press, 2007.

[75] Fangxiang Feng, Xiaojie Wang, and Ruifan Li. Cross-modal retrieval with correspondence autoencoder. In *Proceedings of the ACM International Conference on Multimedia*, pages 7–16. ACM, 2014.

[76] Paolo Ferragina and Ugo Scaiella. Tagme: on-the-fly annotation of short text fragments (by wikipedia entities). In *Proceedings of the 19th ACM international conference on Information and knowledge management*, pages 1625–1628. ACM, 2010.

[77] Anthony Finkelstein, Mark Harman, Yue Jia, Federica Sarro, and Yuanyuan Zhang. Mining app stores: Extracting technical, business and customer rating information for analysis and prediction. Technical Report RN/13/21, Department of Computer Sciences, University College London, London, UK, 2013.

[78] George Forman. A pitfall and solution in multi-class feature selection for text classification. In *Proceedings of the twenty-first international conference on Machine learning*, ICML '04, pages 38–, New York, NY, USA, 2004. ACM.

[79] George Forman. A pitfall and solution in multi-class feature selection for text classification. In *Proceedings of the 21st International Conference on Machine Learning (ICML 2004)*, pages 38–45, Banff, CA, 2004.

[80] George Forman. Quantifying counts and costs via classification. *Data Min. Knowl. Discov.*, 17(2):164–206, October 2008.

[81] Mario Frank, Ben Dong, Adrienne Porter Felt, and Dawn Song. Mining permission request patterns from Android and Facebook applications. In *Proceedings of the 13th IEEE International Conference on Data Mining (ICDM 2012)*, pages 870–875, Brussels, BE, 2012.

[82] Andrea Frome, Greg S Corrado, Jon Shlens, Samy Bengio, Jeff Dean, Tomas Mikolov, et al. Devise: A deep visual-semantic embedding model. In *Advances in Neural Information Processing Systems*, pages 2121–2129, 2013.

[83] John Gantz and David Reinsel. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the future*, 2007(2012):1–16, 2012.

[84] Huiji Gao, Geoffrey Barbier, and Rebecca Goolsby. Harnessing the crowdsourcing power of social media for disaster relief. *IEEE Intelligent Systems*, 26(3):10–14, 2011.

[85] Juan José García Adeva, Rafael A. Calvo, and Diego López de Ipińa. Multilingual approaches to text categorisation. *European Journal for the Informatics Professional*, 5(3):43–51, 2005.

[86] Judith Gelernter and Shilpa Balaji. An algorithm for local geoparsing of microtext. *GeoInformatica*, 17(4):635–667, 2013.

[87] Judith Gelernter and Nikolai Mushegian. Geo-parsing messages from microtext. *Transactions in GIS*, 15(6):753–773, 2011.

[88] Rayid Ghani, Rosie Jones, Dunja Mladenić, Kamal Nigam, and Seán Slattery. Data mining on symbolic knowledge extracted from the web. In *Proceedings of the KDD Workshop on Text Mining*, Boston, US, 2000.

[89] Alfio Gliozzo and Carlo Strapparava. Cross-language text categorization by acquiring multilingual domain models from comparable corpora. In *Proceedings of the ACL Workshop on Building and Using Parallel Texts*, pages 9–16, Ann Arbor, US, 2005.

[90] Rebecca Goolsby. Social media as crisis platform: The future of community maps/crisis maps. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 1(1):7, 2010.

[91] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*, pages 1025–1035, Hyderabad, IN, 2014.

[92] Clinton Gormley and Zachary Tong. *Elasticsearch: The Definitive Guide: A Distributed Real-Time Search and Analytics Engine*. " O'Reilly Media, Inc.", 2015.

[93] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7):1645–1660, 2013.

[94] Vishal Gupta and Gurpreet Singh Lehal. A survey of text summarization extractive techniques. *Journal of emerging technologies in web intelligence*, 2(3):258–268, 2010.

[95] Michelle Guy, Paul Earle, Scott Horvatch, Jessica Turner, Douglas Bausch, and Greg Smoczyk. Social media based earthquake detection and characterization. In *KDD-LESI 2014: Proceedings of the 1st KDD Workshop on Learning about Emergencies from Social Information at KDD2014*, pages 9–10, 2014.

[96] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, 2003.

[97] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[98] Hui Han, Wen-Yuan Wang, and Bing-Huan Mao. Borderline-SMOTE: A new over-sampling method in imbalanced data sets learning. In *Proceedings of the 1st International Conference on Intelligent Computing (ICIC 2005)*, pages 878–887, Hefei, CN, 2005.

[99] Zirije Hasani, Margita Kon-Popovska, and Goran Velinov. Lambda architecture for real time big data analytic. *ICT Innovations*, 2014.

[100] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[101] Pieter Hintjens. *ZeroMQ: messaging for many applications*. " O'Reilly Media, Inc.", 2013.

[102] Steven C. Hoi, Rong Jin, and Michael R. Lyu. Large-scale text categorization by batch mode active learning. In *Proceedings of the 15th International Conference on World Wide Web (WWW 2006)*, pages 633–642, Edinburgh, UK, 2006.

[103] Xian-Sheng Hua, Linjun Yang, Jingdong Wang, Jing Wang, Ming Ye, Kuansan Wang, Yong Rui, and Jin Li. Clickage: Towards bridging semantic and intent gaps via mining click logs of search engines. In *Proceedings of the 21st ACM international conference on Multimedia*, pages 243–252. ACM, 2013.

[104] Muhammad Imran, Carlos Castillo, Fernando Diaz, and Sarah Vieweg. Processing social media messages in mass emergency: a survey. *ACM Computing Surveys*, 47(4):67, 2015.

[105] Muhammad Imran, Shady Mamoon Elbassuoni, Carlos Castillo, Fernando Diaz, and Patrick Meier. Extracting information nuggets from disaster-related messages in social media. In *Proceedings of the 10th International ISCRAM Conference*, pages 791–801, 2013.

[106] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, March 2007.

[107] Peter Jackson and Isabelle Moulinier. *Natural language processing for online applications: Text retrieval, extraction and categorization*, volume 5. John Benjamins Publishing, 2007.

[108] Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Trans. Inf. Syst.*, 20(4):422–446, October 2002.

[109] Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Transactions on Information Systems (TOIS)*, 20(4):422–446, 2002.

[110] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. *ArXiv e-prints*, June 2014.

[111] Thorsten Joachims. Advances in kernel methods. In Bernhard Schölkopf, Christopher J. C. Burges, and Alexander J. Smola, editors, *Advances in kernel methods*, chapter Making Large-scale Support Vector Machine Learning Practical, pages 169–184. MIT Press, Cambridge, MA, USA, 1999.

[112] Thorsten Joachims. Advances in kernel methods. In Bernhard Schölkopf, Christopher J. C. Burges, and Alexander J. Smola, editors, *Advances in kernel methods*, chapter Making Large-scale Support Vector Machine Learning Practical, pages 169–184. MIT Press, Cambridge, MA, USA, 1999.

[113] Thorsten Joachims. Making large-scale SVM learning practical. In Bernhard Schölkopf, Christopher J. Burges, and Alexander J. Smola, editors, *Advances in Kernel Methods – Support Vector Learning*, chapter 11, pages 169–184. The MIT Press, Cambridge, US, 1999.

[114] Thorsten Joachims. A support vector method for multivariate performance measures. In *Proceedings of the 22nd International Conference on Machine Learning (ICML 2005)*, pages 377–384, Bonn, DE, 2005.

[115] Thorsten Joachims. Training linear SVMs in linear time. In *Proceedings of the 12th ACM International Conference on Knowledge Discovery and Data Mining (KDD 2006)*, pages 217–226, Philadelphia, US, 2006.

[116] George H. John, Ron Kohavi, and Karl Pfleger. Irrelevant features and the subset selection problem. In *MACHINE LEARNING: PROCEEDINGS OF THE ELEVENTH INTERNATIONAL*, pages 121–129. Morgan Kaufmann, 1994.

[117] Saint John Walker. Big data: A revolution that will transform how we live, work, and think, 2014.

[118] Stephen Kaisler, Frank Armour, J Alberto Espinosa, and William Money. Big data: Issues and challenges moving forward. In *System Sciences (HICSS), 2013 46th Hawaii International Conference on*, pages 995–1004. IEEE, 2013.

[119] Min-Yen Kan and Hoang O. Nguyen Thi. Fast webpage classification using URL features. In *Proceedings of the 14th ACM International Conference on Information and Knowledge Management (CIKM 2005)*, pages 325–326, Bremen, DE, 2005.

[120] Pentti Kanerva, Jan Kristofersson, and Anders Holst. Random indexing of text samples for latent semantic analysis. In *Proceedings of the 22nd Annual Conference of the Cognitive Science Society*, pages 1036–1037, Philadelphia, US, 2000.

[121] Alexandros Karatzoglou, Linas Baltrunas, Karen Church, and Matthias Böhmer. Climbing the app wall: Enabling mobile app discovery through context-aware recommendations. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management (CIKM 2012)*, pages 2527–2530, Maui, US, 2012.

[122] Andrej Karpathy and Li Fei-Fei. Deep visual-semantic alignments for generating image descriptions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3128–3137, 2015.

[123] Samuel Kaski. Dimensionality reduction by random mapping: Fast similarity computation for clustering. In *Proceedings of the IEEE International Joint Conference on Neural Networks (IJCNN 1998)*, pages 413–418, Anchorage, US, 1998.

[124] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[125] Mariam Kiran, Peter Murphy, Inder Monga, Jon Dugan, and Sartaj Singh Baveja. Lambda architecture for cost-effective batch and speed big data processing. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 2785–2792. IEEE, 2015.

[126] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, pages 1137–1143, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[127] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7, 2011.

[128] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[129] Yelena Kropivnitskaya, Kristy F Tiampo, Jinhui Qin, and Michael A Bauer. The predictive relationship between earthquake intensity and tweets rate for real-time ground-motion estimation. *Seismological Research Letters*, 2017.

[130] Yury Kryvasheyeu, Haohui Chen, Nick Obradovich, Esteban Moro, Pascal Van Hentenryck, James Fowler, and Manuel Cebrian. Rapid assessment of disaster damage using social media activity. *Science advances*, 2(3):e1500779, 2016.

[131] Quoc V. Le and Tomas Mikolov. Distributed representations of sentences and documents. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*, pages 1188–1196, 2014.

[132] David D. Lewis. Naive (bayes) at forty: The independence assumption in information retrieval. In *Proceedings of the 10th European Conference on Machine Learning*, ECML '98, pages 4–15, London, UK, UK, 1998. Springer-Verlag.

[133] David D. Lewis, Yiming Yang, Tony G. Rose, and Fan Li. Rcv1: A new benchmark collection for text categorization research. *J. Mach. Learn. Res.*, 5:361–397, December 2004.

[134] Peilong Li, Yan Luo, Ning Zhang, and Yu Cao. Heterospark: A heterogeneous cpu/gpu spark platform for machine learning algorithms. In *Networking, Architecture and Storage (NAS), 2015 IEEE International Conference on*, pages 347–348. IEEE, 2015.

[135] Xin Li and Yuhong Guo. Active learning with multi-label svm classification. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pages 1479–1485. AAAI Press, 2013.

[136] Yuan Liang, James Caverlee, and John Mander. Text vs. images: On the viability of social media to assess earthquake damage. In *Proceedings of the 22nd international conference on World Wide Web companion*, pages 1003–1006. International World Wide Web Conferences Steering Committee, 2013.

[137] Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In Stan Szpakowicz Marie-Francine Moens, editor, *Text Summarization Branches Out: Proceedings of the ACL-04 Workshop*, pages 74–81, Barcelona, Spain, July 2004. Association for Computational Linguistics.

[138] Jovian Lin, Kazunari Sugiyama, Min-Yen Kan, and Tat-Seng Chua. Addressing cold-start in app recommendation: Latent user models constructed from Twitter followers. In *Proceedings of the 36th ACM Conference on Research and Development in Information Retrieval (SIGIR 2013)*, pages 283–292, Dublin, IE, 2013.

[139] Jovian Lin, Kazunari Sugiyama, Min-Yen Kan, and Tat-Seng Chua. New and improved: Modeling versions to improve app recommendation. In *Proceedings of the 37th ACM Conference on Research and Development in Information Retrieval (SIGIR 2014)*, pages 647–656, Gold Coast, AU, 2014.

[140] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *Computer Vision–ECCV 2014*, pages 740–755. Springer, 2014.

[141] Bing Liu. Sentiment analysis and opinion mining. *Synthesis Lectures on Human Language Technologies*, 5(1):1–167, 2012.

[142] Bing Liu and Lei Zhang. A survey of opinion mining and sentiment analysis. In *Mining text data*, pages 415–463. Springer, 2012.

[143] Huan Liu and Hiroshi Motoda, editors. *Computational Methods of Feature Selection*. CRC Press/Taylor and Francis Group, London, UK, 2007.

[144] Steve Lohr. The age of big data. *New York Times*, 11(2012), 2012.

[145] Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. The stanford corenlp natural language processing toolkit. In *ACL (System Demonstrations)*, pages 55–60, 2014.

[146] Miguel Martinez-Alvarez, Alejandro Bellogin, and Thomas Roelleke. Document difficulty framework for semi-automatic text classification. In *Data Warehousing and Knowledge Discovery*, pages 110–121. Springer Berlin Heidelberg, 2013.

[147] Andrew K. McCallum, Ronald Rosenfeld, Tom M. Mitchell, and Andrew Y. Ng. Improving text classification by shrinkage in a hierarchy of classes. In *Proceedings of the 15th International Conference on Machine Learning (ICML 1998)*, pages 359–367, Madison, US, 1998.

[148] Patrick Meier. Crisis mapping in action: How open source software and global volunteer networks are changing the world, one map at a time. *Journal of Map & Geography Libraries*, 8(2):89–100, 2012.

[149] Pablo N Mendes, Max Jakob, Andrés García-Silva, and Christian Bizer. Dbpedia spotlight: shedding light on the web of documents. In *Proceedings of the 7th international conference on semantic systems*, pages 1–8. ACM, 2011.

[150] Stuart E Middleton, Lee Middleton, and Stefano Modafferi. Real-time crisis mapping of natural disasters using social media. *IEEE Intelligent Systems*, 29(2):9–17, 2014.

[151] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 3111–3119. Curran Associates, Inc., 2013.

[152] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[153] Gary Miner. *Practical text mining and statistical analysis for non-structured text data applications*. Academic Press, 2012.

[154] Stefano Mizzaro, Marco Pavan, Ivan Scagnetto, and Ivano Zanello. A context-aware retrieval system for mobile applications. In *Proceedings of the 4th Workshop on Context-Awareness in Retrieval and Recommendation (CARR 2014)*, pages 18–25, Amsterdam, NL, 2014.

[155] Gianmarco De Francisci Morales and Albert Bifet. Samoa: Scalable advanced massive online analysis. *Journal of Machine Learning Research*, 16:149–153, 2015.

[156] Alejandro Moreo, Andrea Esuli, and Fabrizio Sebastiani. Distributional random oversampling for imbalanced text classification. In *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval*, pages 805–808. ACM, 2016.

[157] Alejandro Moreo Fernández, Andrea Esuli, and Fabrizio Sebastiani. Lightweight random indexing for polylingual text classification. *Journal of Artificial Intelligence Research*, 57:151–185, 2016.

[158] Philipp Moritz, Robert Nishihara, Ion Stoica, and Michael I Jordan. Sparknet: Training deep networks in spark. *arXiv preprint arXiv:1511.06051*, 2015.

[159] T. Morton, J. Kottmann, J. Baldridge, and G. Bierner. Opennlp: A java-based nlp toolkit, 2005.

[160] David Nadeau and Satoshi Sekine. A survey of named entity recognition and classification. *Lingvisticae Investigationes*, 30(1):3–26, 2007.

[161] Tetsuji Nakagawa and Yuji Matsumoto. Detecting errors in corpora using support vector machines. In *Proceedings of the 19th international conference on Computational linguistics-Volume 1*, pages 1–7. Association for Computational Linguistics, 2002.

[162] Jiquan Ngiam, Aditya Khosla, Mingyu Kim, Juhan Nam, Honglak Lee, and Andrew Y Ng. Multimodal deep learning. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 689–696, 2011.

[163] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. Samza: Stateful scalable stream processing at linkedin. *Proc. VLDB Endow.*, 10(12):1634–1645, August 2017.

[164] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. Samza: Stateful scalable stream processing at linkedin. *Proc. VLDB Endow.*, 10(12):1634–1645, August 2017.

[165] Mohammad Norouzi, Tomas Mikolov, Samy Bengio, Yoram Singer, Jonathon Shlens, Andrea Frome, Greg S Corrado, and Jeffrey Dean. Zero-shot learning by convex combination of semantic embeddings. *arXiv preprint arXiv:1312.5650*, 2013.

[166] Douglas W Oard and Bonnie J Dorr. A survey of multilingual text retrieval. Technical report, University of Maryland at College Park, 1998.

[167] Dennis Pagano and Walid Maalej. User feedback in the appstore: An empirical study. In *Proceedings of the 21st IEEE International Requirements Engineering Conference (RE 2013)*, pages 125–134, Rio de Janeiro, BR, 2013.

[168] Sinno J. Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, 2010.

[169] Bo Pang, Lillian Lee, et al. Opinion mining and sentiment analysis. *Foundations and Trends® in Information Retrieval*, 2(1–2):1–135, 2008.

[170] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[171] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *EMNLP*, volume 14, pages 1532–1543, 2014.

[172] John C Platt, Nello Cristianini, and John Shawe-Taylor. Large margin dags for multiclass classification. In *nips*, volume 12, pages 547–553, 1999.

[173] Peter Prettenhofer and Benno Stein. Cross-language text classification using structural correspondence learning. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL 2010)*, pages 1118–1127, Uppsala, SE, 2010.

[174] Xiaoguang Qi and Brian D. Davison. Web page classification: Features and algorithms. *ACM Computing Surveys*, 41(2), 2009.

[175] Ali Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. Cnn features off-the-shelf: an astounding baseline for recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 806–813, 2014.

[176] David Reinsel, John Gantz, and John Rydning. *Data Age 2025: The Evolution of Data to Life-Critical, Don't Focus on Big Data; Focus on the Data That's Big*. IDC, 2017. Accessed: 2017-09-14.

[177] Ryan Rifkin and Aldebaro Klautau. In defense of one-vs-all classification. *J. Mach. Learn. Res.*, 5:101–141, December 2004.

[178] Stephen Robertson. Understanding inverse document frequency: On theoretical arguments for idf. *Journal of Documentation*, 60:503–520, January 2004.

[179] Stephen E. Robertson and Hugo Zaragoza. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends in Information Retrieval*, 3(4):333–389, 2009.

[180] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curino. Apache tez: A unifying framework for modeling and building data processing applications. In *Proceedings of the 2015 ACM SIGMOD international conference on Management of Data*, pages 1357–1369. ACM, 2015.

[181] Magnus Sahlgren. An introduction to random indexing. In *Proceedings of the Workshop on Methods and Applications of Semantic Indexing*, Copenhagen, DK, 2005.

[182] Takeshi Sakaki, Makoto Okazaki, and Yutaka Matsuo. Tweet analysis for real-time event detection and earthquake reporting system development. *IEEE Transactions on Knowledge and Data Engineering*, 25(4):919–931, 2013.

[183] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Inf. Process. Manage.*, 24(5):513–523, August 1988.

[184] Gerard Salton, Anita Wong, and Chung-Shu Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[185] Borja Sanza, Igor Santos, Carlos Laorden, Xabier Ugarte-Pedrero, and Pablo G. Bringas:. On the automatic categorisation of Android applications. In *Procedings of the 1st IEEE Consumer Communications and Networking Conference (CCNC 2012)*, pages 149–153, Las Vegas, US, 2012.

[186] Robert E. Schapire and Yoram Singer. Improved boosting algorithms using confidence-rated predictions. In *Proceedings of the Eleventh Annual Conference on Computational Learning Theory*, COLT' 98, pages 80–91, New York, NY, USA, 1998. ACM.

[187] Robert E. Schapire and Yoram Singer. Boostexter: A boosting-based system for text categorization. *Machine Learning*, 39(2):135–168, 2000.

[188] Robert E. Schapire, Yoram Singer, and Amit Singhal. Boosting and rocchio applied to text filtering. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '98, pages 215–223, New York, NY, USA, 1998. ACM.

[189] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[190] Sam Scott and Stan Matwin. Feature engineering for text classification. In *ICML*, volume 99, pages 379–388, 1999.

[191] Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM Comput. Surv.*, 34(1):1–47, March 2002.

[192] Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM Comput. Surv.*, 34(1):1–47, March 2002.

[193] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[194] Sören Sonnenburg, Gunnar Rätsch, Sebastian Henschel, Christian Widmer, Jonas Behr, Alexander Zien, Fabio de Bona, Alexander Binder, Christian Gehl, and Vojtěch Franc. The shogun machine learning toolbox. *J. Mach. Learn. Res.*, 11:1799–1802, August 2010.

[195] spaCy GmbH. spacy: Industrial-strength nlp. `https://spacy.io/`, 2015-2016.

[196] Ralf Steinberger. A survey of methods to ease the development of highly multilingual text mining applications. *Language Resources and Evaluation*, 46(2):155–176, 2012.

[197] Ralf Steinberger, Bruno Pouliquen, Anna Widiger, Camelia Ignat, Tomaz Erjavec, Dan Tufis, and Dániel Varga. The jrc-acquis: A multilingual aligned parallel corpus with 20+ languages. *arXiv preprint cs/0609058*, 2006.

[198] Philip J Stone, Dexter C Dunphy, and Marshall S Smith. The general inquirer: A computer approach to content analysis. *American Journal of Sociology*, 1966.

[199] L. Tassiulas and A. Ephremides. Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks. *IEEE Transactions on Automatic Control*, 37(12):1936–1948, Dec 1992.

[200] The Theano Development Team, Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, et al. Theano: A python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688*, 2016.

[201] Simon Tong and Daphne Koller. Support vector machine active learning with applications to text classification. *Journal of Machine Learning Research*, 2:45–66, 2001.

[202] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM.

[203] Salvatore Trani, Diego Ceccarelli, Claudio Lucchese, Salvatore Orlando, and Raffaele Perego. Dexter 2.0: an open source tool for semantically enriching data. In *Proceedings of the 2014 International Conference on Posters & Demonstrations Track-Volume 1272*, pages 417–420. CEUR-WS. org, 2014.

[204] Ricardo Usbeck, Michael Röder, Axel-Cyrille Ngonga Ngomo, Ciro Baron, Andreas Both, Martin Brümmer, Diego Ceccarelli, Marco Cornolti, Didier Cherix, Bernd Eickmann, et al. Gerbil: General entity annotator benchmarking framework. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1133–1143. ACM, 2015.

[205] Mark van Rijmenam. Understanding the various sources of big data - infographic. `https://datafloq.com/read/understanding-sources-big-data-infographic/338`. Accessed: 2017-09-13.

[206] Sudha Verma, Sarah Vieweg, William J Corvey, Leysia Palen, James H Martin, Martha Palmer, Aaron Schram, and Kenneth Mark Anderson. Natural language processing to the rescue? extracting "situational awareness" tweets during mass emergency. In *Proceedings of the 5th International AAAI Conference on Web and Social Media (ICWSM)*. AAAI, 2011.

[207] Sarah Vieweg and Adam Hodges. Rethinking context: Leveraging human and machine computation in disaster response. *Computer*, 47(4):22–27, 2014.

[208] Lei Wang and Krishna Kant. Special issue on computational sustainability. *IEEE Transactions on Emerging Topics in Computing*, 2(2):119–121, 2014.

[209] Shuohang Wang and Jing Jiang. Learning natural language inference with lstm. *arXiv preprint arXiv:1512.08849*, 2015.

[210] Ingmar Weber and Venkata Rama Kiran Garimella. Visualizing user-defined, discriminative geo-temporal twitter activity. In *ICWSM*, 2014.

[211] Tom White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[212] Yiming Yang and Christopher G. Chute. An example-based mapping method for text categorization and retrieval. *ACM Trans. Inf. Syst.*, 12(3):252–277, July 1994.

[213] Yiming Yang and Jan O Pedersen. A comparative study on feature selection in text categorization. In *Icml*, volume 97, pages 412–420, 1997.

[214] Yiming Yang and Jan O. Pedersen. A comparative study on feature selection in text categorization. In *Proceedings of the 14th International Conference on Machine Learning (ICML 1997)*, pages 412–420, Nashville, US, 1997.

[215] Yiming Yang, Seán Slattery, and Rayid Ghani. A study of approaches to hypertext categorization. *Journal of Intelligent Information Systems*, 18(2/3):219–241, 2002.

[216] Yiming Yang, Jian Zhang, and Bryan Kisiel. A scalability analysis of classifiers in text categorization. In *Proceedings of the 26th Annual International ACM SIGIR Conference on Research and Development in Informaion Retrieval*, SIGIR '03, pages 96–103, New York, NY, USA, 2003. ACM.

[217] Dragomir Yankov, Pavel Berkhin, and Rajen Subba. Interoperability ranking for mobile applications. In *Proceedings of the 36th International ACM Conference on Research and Development in Information Retrieval (SIGIR 2013)*, pages 857–860, Dublin, IE, 2013.

[218] Peifeng Yin, Ping Luo, Wang-Chien Lee, and Min Wang. App recommendation: A contest between satisfaction and temptation. In *Proceedings of the 6th ACM Conference on Web Search and Data Mining (WSDM 2013)*, pages 395–404, Rome, IT, 2013.

[219] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[220] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

[221] Xiang Zhang, Junbo Zhao, and Yann LeCun. Character-level convolutional networks for text classification. In *Advances in neural information processing systems*, pages 649–657, 2015.

[222] Bolei Zhou, Agata Lapedriza, Jianxiong Xiao, Antonio Torralba, and Aude Oliva. Learning deep features for scene recognition using places database. In *Advances in neural information processing systems*, pages 487–495, 2014.

[223] Hengshu Zhu, Enhong Chen, Hui Xiong, Huanhuan Cao, and Jilei Tian. Mobile app classification with enriched contextual information. *IEEE Transactions on Mobile Computing*, 13(7):1550–1563, 2014.

[224] Hengshu Zhu, Chuanren Liu, Yong Ge, Hui Xiong, and Enhong Chen. Popularity modeling for mobile apps: A sequential approach. *IEEE Transactions on Cybernetics*, 2014. Forthcoming.