MINISTRY OF EDUCATION AND SCIENTIFIC RESEARCH

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

ATHENA - Aproach for Twitter Harvesting, Enhancement, Normalisation & Analysis

MASTER THESIS

Graduate: Ioana-Călina TUTUNARU
Supervisor: prof. dr. ing. Ioan Salomie

2016

MINISTRY OF EDUCATION AND SCIENTIFIC RESEARCH

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

DEAN,	HEAD OF DEPARTMENT,
Prof. dr. eng. Liviu MICLEA	**Prof. dr. eng. Rodica POTOLEA**

Graduate: **Ioana-Călina TUTUNARU**

ATHENA - Aproach for Twitter Harvesting, Enhancement, Normalisation & Analysis

1. **Project proposal:** *Short description of the license thesis and initial data*
2. **Project contents:** *(enumerate the main component parts) Presentation page, advisor's evaluation, title of chapter 1, title of chapter 2, ..., title of chapter n, bibliography, appendices.*
3. **Place of documentation:** *Example*: Technical University of Cluj-Napoca, Computer Science Department
4. **Consultants:**
5. **Date of issue of the proposal:** November 1, 2014
6. **Date of delivery:** June 18, 2015 *(the date when the document is submitted)*

Graduate: ______________________________

Supervisor: ______________________________

MINISTRY OF EDUCATION AND SCIENTIFIC RESEARCH

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

Declaraţie pe proprie răspundere privind
autenticitatea lucrării de licenţă

Subsemnatul(a) ____, legitimat(ă) cu ____________ seria ____________ nr. ____________
CNP ______________________________, autorul lucrării ____________
____
____
elaborată în vederea susţinerii examenului de finalizare a studiilor de licenţă la Facultatea de Automatică şi Calculatoare, Specializarea ____________________
din cadrul Universităţii Tehnice din Cluj-Napoca, sesiunea ____________ a anului universitar ____________, declar pe proprie răspundere, că această lucrare este rezultatul propriei activităţi intelectuale, pe baza cercetărilor mele şi pe baza informaţiilor obţinute din surse care au fost citate, în textul lucrării şi în bibliografie.

Declar, că această lucrare nu conţine porţiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislaţiei române şi a convenţiilor internaţionale privind drepturile de autor.

Declar, de asemenea, că această lucrare nu a mai fost prezentată în faţa unei alte comisii de examen de licenţă.

În cazul constatării ulterioare a unor declaraţii false, voi suporta sancţiunile administrative, respectiv, *anularea examenului de licenţă.*

Data ____________

Nume, Prenume ____________________

Semnătura

De citit înainte (această pagină se va elimina din versiunea finală):

1. Cele trei pagini anterioare (foaie de capăt, foaie sumar, declaraţie) se vor lista pe foi separate (nu faţă-verso), fiind incluse în lucrarea listată. Foaia de sumar (a doua) necesită semnătura absolventului, respectiv a coordonatorului. Pe declaraţie se trece data când se predă lucrarea la secretarii de comisie.

2. Pe foaia de capăt, se va trece corect titulatura cadrului didactic îndrumător, în engleză (consultaţi pagina de unde aţi descărcat acest document pentru lista cadrelor didactice cu titulaturile lor).

3. Documentul curent **nu** a fost creat în MS Office. E posibil sa fie mici diferenţe de formatare.

4. Cuprinsul începe pe pagina nouă, impară (dacă se face listare faţă-verso), prima pagină din capitolul *Introducere* tot aşa, fiind numerotată cu 1.

5. E recomandat să vizualizaţi acest document şi în timpul editării lucrării.

6. Fiecare capitol începe pe pagină nouă.

7. Folosiţi stilurile predefinite (Headings, Figure, Table, Normal, etc.)

8. Marginile la pagini nu se modifică.

9. Respectaţi restul instrucţiunilor din fiecare capitol.

List of Figures

Contents

Chapter 1

Introduction - Project Context

Not long time ago, big data and the need for content analysis were idealistic, intangible concepts. However, recent developments in computer performance and the growing popularity of social media has enhanced the need for novel academic approaches in this field.

The biggest challenge with processing these huge amounts of data lies in its sheer volume. This is highly problematic in many different ways:

- storing and managing data is costly and sometimes needs special structuring attention
- irregularities across data streams often indicate the need for extra transformations
- filtering out noise and the extraction of meaningful data from thousands of entries requires specialised algorithms, architectures and platforms which are either inaccessible or difficult to use by regular users

1.1 Project context

Social media platforms these days have a strong presence in ”the Cloud”, which allows more and more developers to solve the problem of storage space and scalability in a relatively easy way. The high adoption of platforms like Facebook, Twitter, Instagram and many others have resulted in the generation of hundreds of thousands of online documents, text and media alike. Such a deluge of information is highly applicable for business purposes, but in is often stuck as a ”diamond in the rough”. An array of new and exciting job titles has also appeared, such as ”Social Media Marketer”, ”Social Media Manager”, ”Social Media Expert” and so on. Not only do they represent career alternatives for a new generation, but they are also still outliers for decision support software.

1.1.1 Social Media and why people care about what it says

While the concept of Social Media is not a new one to be precise, it is clearly just starting to engulf us. The spread of Social Media alongside the spread of worldwide Internet access, the emergence of specialised applications and the recent shift to mobile online presence are objective realities. Subjectively as well, people have become more engaged to consuming, creating and sharing content online, which is evident in newspaper sales drops, app sales increase, events "going viral" within groups and even the online presence of a political critical mass. Companies offer barter possibilities for a variety of bloggers and vloggers to push products to their followers (which is not only more effective[1], but also cheaper).

No wonder social media has a stake in public decisions! But for the first time in decades, people's choices are also quite open and public. Which means that some parties can get clues about society's preferences by using public data available on platforms such as Facebook, Youtube and Twitter. Consider two basic use cases, which I will emphasise throughout the course of this thesis:

- What does social media say about X?
- What does social media say about X vs. Y?

where X and Y can be public people, events, brands, companies etc.

1.1.2 Data Analysis tools today

There are a number of current platforms for data analysis, in the large sense including chart generators, format converters and importers and data visualisation plug-ins of sorts. Crossing into the realm of powerful analysis applications, most of them are complex and confusing to non-engineers, examples including Tableau[2], RapidMiner[3] and WolframAlpha[4]. Figure 1.1 shows Tableau's interface, which may seem quite intimidating.

1.1.3 Crossroads: Social Media in Business Intelligence

The Computer Science field is very much branched nowadays in clearly-delimited subjects. But it is such a case where the large-scale processing capacities of data analytics should merge with approaches for computer-assisted business decisions. Consider that data analytics software often requires extensive training and a specialised knowledge, while most users desire only simple, straightforward functionalities for basic data analysis. One

[1]http://www.techrepublic.com/article/election-tech-why-social-media-is-more-powerful-than-advertising/

[2]https://public.tableau.com/s/

[3]http://rapidminer.com

[4]http://www.wolframalpha.com

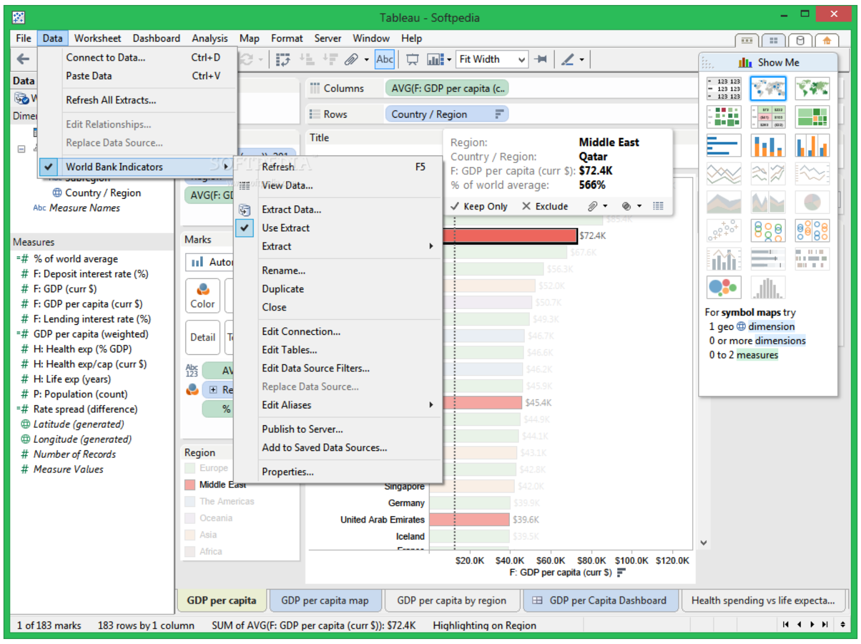


Figure 1.1: Tableau's interface

of Object Oriented Programming's fundamental principles is that of abstraction, in which end users need not (and should not) be aware of what is under the hood. Yet in data analytics, this is still the case on a

The following Master's Thesis is the result of such an endeavour, to combine solid concepts from big data analysis, web application architecture and design, as well as enterprise software and economics for a better understanding of social media trends and opinions.

Thesis structure

This document describes the general context, theoretical foundations and implementation details of a proof of concept application called ”Approach for Twitter Harvesting, Enhancement, Normalisation & Analysis” or, in short, ATHENA. Chapter 2 presents the objectives and requirements of our proposed system. Chapter 3 presents scientific approaches already validated by the community, some of which are generally similar to the entire ATHENA pipeline, while some are concerned with parts of the proposed implementation. Chapter 4 and 5 present a theoratical analysis and a detailed implementation of ATHENA respectively. Chapter 6 presents the testing and validation methods used as per ensuring application quality and usefulness. The final chapters contain technical instructions for installing the project, as well as the conclusions drawn from ATHENA's development endeavour.

Chapter 2

Project Objectives and Specifications

The project's purpose is to help non-expert users interested in social media analysis to get relevant and reliable information regarding specific concepts.

2.1 Web architecture

A web architecture is preferable as a modern-day alternative to installable applications. The fast evolution of mobile devices has pushed web developers to segregate User Interfaces from Backend implementations, permitting devices to seamlessly connect to the same under the hood implementation without much development cost and overhead. Communication via REST services is crucial in this concern. Beyond user preference towards such applications, a web architecture presents even more advantages:

- virtually infinite space extensible with storage backends such as AWS3 or Google Cloud
- scalability and outsourcing of time-costly services such as getting stream histories
- database distribution over nodes
- easily available documentation and integration throughout the development process

Using a web framework eases development even more so and presents security advantages such as stable features, packages and quick patches in the odd case of a security breach. Plugging in services is also considered in the project's context, with handling of specific use cases done in separate managers, loosely coupled with the Model-View-Controller architecture.

2.1.1 Data storage

Since scalability and distribution are a must for a large-scale stream analysis project, data storage should be handled using a method which supports such breakdown while still remaining query-efficient. NoSQL databases are a popular choice for such implementation. The correspondence between the database storage backend and the model part of the application should also be loosely coupled, allowing for the possibility of switching between database backends if the need arises.

Big players in the field of non-relational databases are obvious possible choices: HBase, Cassandra, MongoDB, CouchDB, etc.

2.2 ATHENA breakdown

This thesis is based on the ATHENA original project, which comprises different pipeline steps in acquiring and analysing Twitter feeds:

- Harvesting: acquisition of collections of related Twitter statuses, by hashtag and dates (start and end).
- Enhancement: basic analysis of a single harvest with classical unsupervised algorithms
- Normalisation and Analysis: comparative analysis of harvests

2.2.1 Twitter as data source

For the development of this project, I chose to only implement one social media backend as data source. Twitter was the choice, since the character limit they impose is a great asset in regards of data processing: First of all, the classical data analysis pipeline deals with special cases such as documents of different lengths providing unusual skewing to the data set. Another advantage of Twitter is the prevalence of the hashtag model, which has failed to catch on with Facebook to the same degree. This means that statuses basically come out of the box already tagged for content.

Twitter is arguably the second most popular social media platform at the time, gaining on a large number of users even before the mobile device explosion, by integrating with telephone service providers to facilitate posting through SMS. Its 140 character limit makes for a good spin, which has pushed Twitter into a more textual realm than its more successful counterpart, Facebook. The platform is largely popular in the United States and especially in the entertainment domain[1], with the most popular Twitter accounts belonging to celebrities.

[1]https://en.wikipedia.org/wiki/Twitter

It notably also has a history of being the ”go to” source for data analysis, after the United States’ current president Barrack Obama purportedly used it in his 2008 campaign as one of the principal media outlets, choosing to have a strong Twitter presence, akin to advertisement presence of past candidates.

In short, although this approach is social platform-independent, Twitter was chosen even from the specifications step of this project for a clear start towards the data mining part.

Fetching data through asynchronous jobs

Since most social media platforms communicate with external applications via REST APIs with rate limits, the project’s structure should include asynchronous modules. In this way, the user will be able to send an API-intensive job for queueing and asynchronous processing, without hindering their overall experience of using the application. This becomes apparent in the Harvesting part.

2.2.2 Harvesting

If we interpret ATHENA as a classical pipeline, then it becomes clear that the Harvesting part is the acquisition part. In order to perform analysis on data, it must first be fetched according to predefined rules and user options. Being time-intensive, these jobs must be designed asynchronously, following a FIFO model.

Figure 2.1 represents the conceptual model of data acquisition, starting from a collection of grouped documents (in our case they will be collections of statuses with a common hashtag). The Harvest manager handles sending asynchronous jobs via a FIFO queue towards a Consumer-type Harvesting job, represented as black box and further detailed in Figure 2.2. It is important to note that this pipeline is not in any way adjusted to the domain or the particular API platform; the only constraint we impose is that of periodically checking for compliance to rate limits. Social APIs usually limit requests either by number of requests or requested data size per unit of time. It is therefore a good idea to consider if any limitations alter our pipelines before starting on implementations.

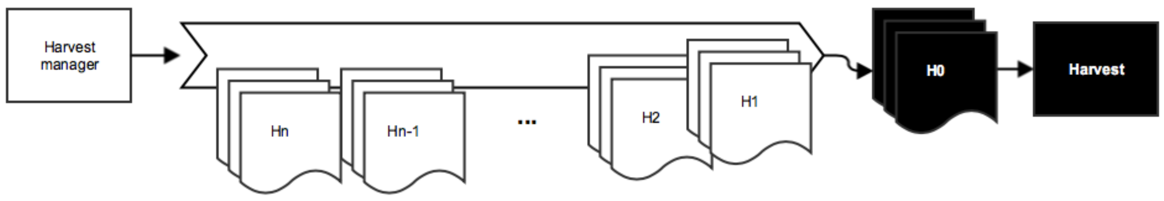


Figure 2.1: Harvesting pipeline

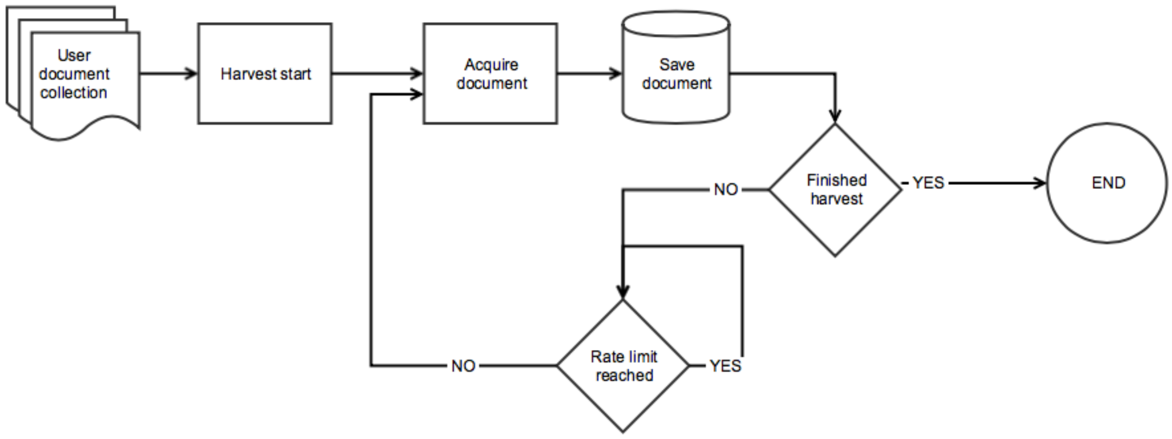


Figure 2.2: Harvesting job sub-pipeline

2.2.3 Enhancement

As previously stated, it is not of great importance to have the data, but the goal is to obtain meaningful information from it. In this thesis I will refer to ”harvests” as per the following definition:

Definition A Harvest is a collection of documents related by contant, containing a token T and spanning from a start date and to an end date, which have gone through the harvesting pipeline and are stored for further anaylsis.

After completing the Harvesting pipeline, resulting harvests are processed using various data extraction algorithms and presented to the user in the Enhancement step. These algorithms each have their own sub-pipelines of transformation, which will be detailed in Chapter 5.

User contribution modeling

Online document collections are difficult to understand in terms of user contribution. Here it is useful to apply a variant of the Pareto laws, which state that most of the time 80% of a document set will be created by 20% of the users. This is in fact a variant of more general power laws, which means we need to detect whether a dataset corresponds to a power distribution, in order to model users' contribution to the document set.

Influencers that produce and engage to content are those targets most suitable for marketing strategies. The user using ATHENA to perform analysis on datasets should thus be able to identify these outliers.

2.2.4 Normalisation

As previously stated, the user has the option of detecting outliers. A corollary of this is that such outliers also need to be eliminated for some analysis types. For example, we might need to filter out frequent posting users because their documents may have a promotional edge, and enjoy popularity in exchange for prizes. This skews the dataset too much, which means a normalisation step is needed to flatten the data to its core users and posts only.

2.2.5 Analysis

The Analysis step should be understood as a comparative analysis between two document sets, in fact restricted only to those document sets which have passed the Normalisation step. By eliminating the outliers, the user can compare the two by vocabulary and popularity. This step is, just like those before, highly optimised in regards to user experience.

2.3 Non-functional requirements

I am much concerned with the user's perspective in developing this application. As explained previously, one of the main objectives of this application is undoubtedly to be as simple to use as possible, to benefit non-expert users.

Firstly, I believe that a familiar and streamlined user experience should be implemented. The UI should have componenents found in most web application nowadays, in order to seem aproachable. The customisation options should also be loaded without input from the user, so that things like algorithm choices and parameters should stay hidden. Non-expert users do not know what *Tfidf Vectorizer* or *KMeans clustering* means, all they care about is to have some paplable results to their queries. In short, it is in the benefit of the non-expert users, which I target through this application, to have as little inputs, panels and buttons to figure out.

Besided the look and feel, users will also be interested in having a fast and reliable application. Exceptions and possible bugs should be properly filtered out and time-consuming jobs moved as per Figure 2.2 to asynchronous jobs.

Chapter 3

Bibliographic research

Although the ATHENA application itself is original work, it is common practice to reuse validated approaches and combine well-known algorithms to achieve a goal such ambitious as this one, i.e. to provide social media analytics capabilities in a familiar, web-based application, to non-expert users. A number of approaches related to this field are in existence, with some having overall structural similarities (e.g. pipeline approaches or web-based applications), while others being proven solutions for just part of the ATHENA pipeline (e.g. K-Means clustering, Power law fitting etc.).

3.1 Social media analytics

Social media analytics has received much attention in recent years. Since the explosion of social networks such as Facebook, Twitter, Instagram and others, the number of sentiment-loaded posts authored by individual users has increased exponentially. This raises the problem of filtering out useless data and noise, and compiling the large number of posts into useful, concentrated information snippets social media marketers can use for advertising and barter choice purposes.

3.2 In-house social media

Exploiting on the trend for social media data mining, some social networks have created their own analysis tools. Much of these are internal, but examples such as Facebook's Graph Search, which is free to use by application developers and individual users, are great tools for analytics purposes. Approaches such as the 2014 study [1] found that using Facebook Graph Search can facilitate people search along demographics and profiles.

Besides in-house tools, social media networks provide APIs for developers interested in researching user behaviour, brand popularity etc. Examples include Facebook, Twitter, Instagram, LinkedIn, Pinterest, Foursquare, Meetup, Slack, Flickr and Google Plus[1]. Web mining approaches such as [2] try to understand virtual social communities by using these APIs. This latter study uses Twitter API. It is not surprising that the reusability of certain API queries were exploited in numerous API wrappers, in different programming languages[2].

3.2.1 Social media analytics in companies

Enterprise resource planning (ERP) systems appeared in the 1990s as a counter wave to company siloing. The phenomenon is described as the specialisation and lack of streamlined cooperation between company departments. Marketing is one such department, in which operations should take into account results and goals from other departments and external sources combined. Unfortunately, since the social media explosion took place in recent years only, and ERP systems are generally large-scale and quite resistant to change, it is yet not implemented that social media data mining tools are well-integrated in ERPs. This is in spite of the clear connection between social media and Web 2.0 concepts to retailing power, as proven by numerous studies [3].

3.3 Structurally-similar works

The following section presents the advantages and drawbacks of using meta-frameworks such as approaches to application development and natural language processing.

3.3.1 Advantages of web applications

Web applications enjoy a high popularity since it is not necessary to install any software on computers in order to run such applications. Almost all modern computer users use web browsers, and so the execution intermediary is already installed in most cases. Ease of usage without downloading anything, availability of updates and patches without any distribution chain necessary and other advantages as mentioned in [4] suggest the appropriateness of developing such an application as a web one. Another important advantage is the fact that in case users develop a need to run this application from mobile devices, a REST architecture to expose the modules would suffice.

Since web application frameworks are ubiquitous these days, it is also the case that a scientifically-rich Python application can easily be integrated to a web application using one of Python's frameworks such as Django, Flask or CherryPy. Indeed the number of wrappers and tools available in Python's scientific libraries have prompted researchers to make various natural language processing applications in this language. The 2009 book

[1]http://www.programmableweb.com/news/top-10-social-apis-facebook-twitter-and-google-plus/analysis/2015/02/17

[2]https://dev.twitter.com/overview/api/twitter-libraries

”Natural language processing with Python” [5] encourages developers to use Python as a programming language most suitable for text processing, complete with support from external libraries such as NLTK (Natural Language ToolKit), Sci-Kit Learn (for machine learning), SciPy, Numpy, matplotlib and so on.

We conclude that the prevalence of the Python recommendation in literature is significant in making a decision related to the project's development. Based on numerous similar approaches, ATHENA also employs Python and its web application components in creating an easy to use and relevant social media analysis tool.

3.3.2 Pipeline model

Pipeline model-based applications are mostly common in the text processing domain of research. The approach works well for large data sets since it is difficult to transform the data end-to-end in one go. The specialised modules which compose the pipeline are indeed fitting with the general concepts of SOLID Object-Oriented Programming, since pipelines respect:

- **S**ingle Responsibility - each module handles one type of simple transformation
- **O**pen Closed - each module collaborates with its neighbours in the pipeline using interfaces
- **L**iskov Substitution - modules are interchangeable, in the sense that you can code to an interface and change each particular type of module with a similar one
- **I**nterface Seregation - pipeline steps are loosely coupled and generally do not depend on specific methods from unrelated steps
- **D**ependency Inversion - the building blocks are usually stable components

Futhermore, the usage of pipelines in natural language processing, data mining and other text-related endeavours, is time-proven, with renowned research groups adopting it as a general principle. One of such groups is the Stanford NLP Research Group [6]. The wide-spread practice became even more popular when text feature extraction (e.g. GATE, LaSIE, IXA, LONI etc.) tools adopted the same principle. Figure 3.1 presents a pipeline implementation used by some applications which handle text processing.

ATHENA's pipeline is similar in the sense that it prepares the document set during the Harvesting step, then it Enhances the dataset with useful data. The third pipeline step is normalising (or flattening) the data set to its most useful and representative core, while the final step takes as input the normalised document sets and runs a comparative Analysis.

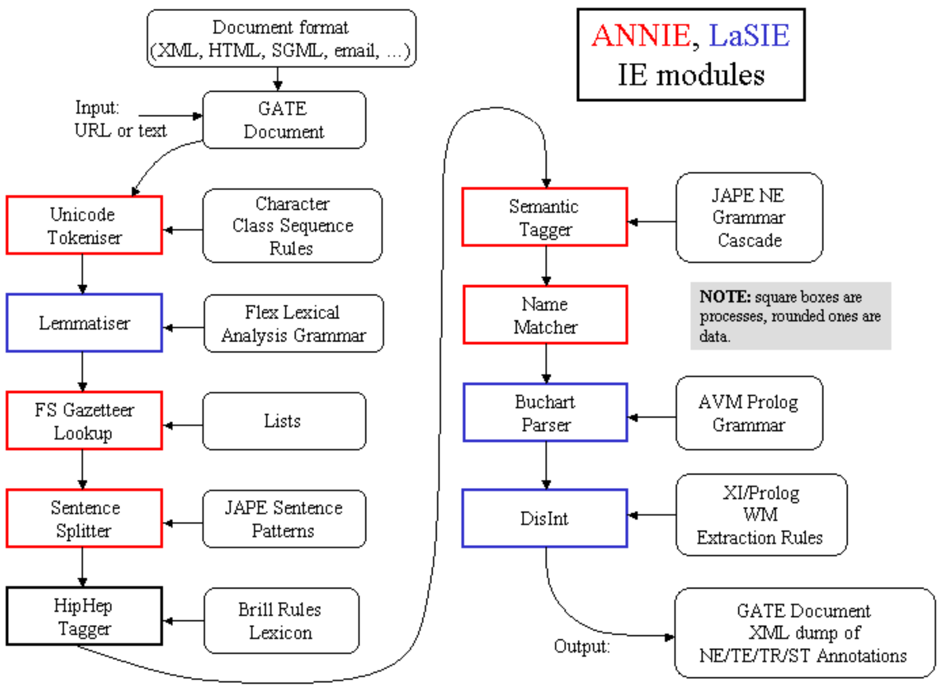


Figure 3.1: Classical NLP Pipeline as applied in ANNIE and LaSIE text processing tools

3.4 Data modeling and extraction approaches

The vast domain of text feature extraction and text mining is a very diverse one. A variety of approaches can be applied, as discovered by many branches of research. One of the most important issues to discuss is the advantages and drawbacks of the domain-related and domain-independent categories of machine learning, and what they mean for natural text processing. Towards the end of this section, the discussion will move to scientifically sound and time-proven methods which achieve parts of the necessary functionalities in ATHENA.

3.4.1 Supervised vs. unsupervised learning. Domain vs Domain-independent approaches

As previously discussed, one of the biggest discussions to be had at the start of a text processing application's development, is whether the application will fall into either one of the categories: supervised vs. unsupervised machine learning, domain-related or domain-independent. Usually the categories are coupled in the sense that domain-independent data can not be correctly processed within a supervised approach, which means the two are incompatible.

Supervised learning starts from the premise that we can classify some existing, annotated data, and once the classifier is correctly trained, it can be launched onto the real, full-scale dataset and perform well. Indeed with supervised learning, considering the domain-related approach, an impressive accuracy rate can be obtained. Factors such as the training set size, completeness and correctness influence, along with domain-specific tweaks, can combine to create a powerful classifier. However, there are some drawbacks:

- the approach can not be generalised to other domains without existing annotated data for those domains
- the training set is often difficult to find and/or manually annotate. Such a training set requires a lot of resources and a dedicated staff, while for the best final accuracy the number of documents correctly labeled in the training set should be very large.

Applications such as ATHENA seem to not have a choice but use the domain-independent approach, with specifically domain-independent algorithms to build the analysis results. But the results in the field are not discouraging. In 2002, a study by Turney [7] found an accuracy rate of 74% for an unsupervised learning approach, applied to text processing and sentiment analysis in particular. ATHENA does not restrict limits on the hashtags users can use to build harvests, and is in the impossibility of building training sets for all such hashtags, so it falls into the area of unsupervised learning. It is important to note that subsequent algorithms for calculating various application analysis results should therefore also be unsupervised ones.

3.4.2 Clustering

One of the most important unsupervised learning concepts that ATHENA uses, both in the Enhancement and Analysis steps, is that of clustering. Clustering is the grouping of related document features and/or documents, based on common features. It does not require any training data as input and does not have specific domain applicability, which means it is safe to use throughout our application.

A Google Scholar search of the terms "document clustering" produces almost 1.5 million results in books, papers and conference proceedings. The prevalence of clustering algorithms is due to it's easy to understand process and applicability over a large range of possible datasets. A 2000 study by Steinbach et. al [8] compares the accuracy of some clustering algorithms, including:

- K-Means
- bisecting K-Means
- UPGMA (Unweighted Pair Group Method with Arithmetic Mean)

with different centroid calculation methods. It demonstrates that bisecting K-Means performs best for hierarchical clustering, with some dataset particularities constituting exceptions.

KMeans

The clustering algorithm used in ATHENA is K-Means. Initially proposed by Stuart Lloyd in 1957 and applied to research in pulse-code modulation[3], K-Means has become popular in cluster analysis and data mining after its republication in Fortran by Hartigan and Wong in 1975/1979, which featured better efficiency. The algorithm starts from a number of K centroids and then alternates between two types of steps:

- assignment: assign each data point to the nearest centroid (in Euclidean distance)
- update: calculate new centroids at the mean of points assigned to each centroid

KMeans runs these steps alternatively until convergence, which is achieved when centroids no longer move from iteration to iteration, or at least do not move over a distance greater than a predefined minimum. In Chapter 4 I will refer to the NP-hard problem of centroid initialisation for encouraging fast convergence, which is based on a 2013 Celebi et. al study [9] which compares the performance of different centroid seeding methods.

ATHENA uses a variant of KMeans which translates the term occurrence and frequency in the document set to Euclidean distance, i.e. the presence of common terms inside a pair of documents defines the closeness of those documents.

3.4.3 Power law dataset analysis

Another important issue we need to correctly handle is the modelling of users behaviour in authoring documents. We are mostly interested in the degree of fitting of post numbers per user to the existence of a power law. Perhaps one of the best-known example of a power law in effect is that proposed by the Italian economist Vilfredo Pareto, that 80% of the land in Italy was owned by 20% of the population. The extension to other fields soon became apparent, with Pareto's law being generalised as: "For many events, roughly 80% of the effects come from 20% of the causes[4]. When applying Pareto's law to our dataset of document numbers, it might be the case that unusually high numbers of documents are authored by unusually low numbers of users.

In Nassim Nicolas Taleb's 2007 book "The Black Swan"[10], the principles of power laws are given great focus. The author maintains that the existence of power laws in economic contexts is of such nature, that the modeling of economic events in Gaussian distributions is entirely wrong, with Mandelbrotian (fractal) modeling being far more suitable. The argument is that, in such datasets, the outliers are so powerful, that they skew the results of an otherwise correct calculation. These long tail measures apply mostly to data which is not a physical property such as height or weight of people, but to economic

[3]https://en.wikipedia.org/wiki/K-means_clustering
[4]https://en.wikipedia.org/wiki/Pareto_principle

and social aspects such as popularity, wealth, career-wise success. It is probably safe to admit that post numbers and post popularity in social media fall into the same non-physical category which can many a times be modeled as a power law distribution.

As to calculation methods of this fitting degree, authors Clauset et. al built a Matlab implementation of an algorithm doing these estimations. The formal proof of algorithm correctness, as well as arguments for the algorithm, are collected in a 2009 study [11]. The same algorithmic approach was later translated to various programming languages either by the authors themselves, or by third party developers interested in using the library. These endeavours saw R, Python, C++ and Java libraries created on the same basic ideas.

Summary

In this chapter I discussed the similarities between the proposed application of ATHENA and other scientific endeavours to create feature extraction systems. The subjects covered started with arguments for designing ATHENA as a web application and using the pipeline architecture, as validated and widely-used throughout the scientific community concerned with text processing.

I then discussed the advantages and drawbacks of domain-related vs. domain-independent text processing, as well as the supervised and unsupervised approaches, motivating why only the latter is suitable for the ATHENA application and explaining the consequences of this categorisation. In the last part of this section, I covered a couple of algorithms used in the analysis the application performs, with KMeans and Plfit as algorithms suitable to unsupervised learning and thoroughly researched and validated by the scientific community.

More in-depth analysis of existing approaches, their suitability to this project and particular implementations needed will be discussed in the next chapter

Chapter 4

Analysis and Theoretical Foundation

While the previous chapter covered existing theoretical approaches which intersect somewhat to the ATHENA approach itself, the following will go into more detail related to scientific methods used throughout the application. The theoretical foundations of ATHENA will be covered, with the conceptual behaviour of such an application being formally described. Please note that the following set of methods apply to a variety of text feature extraction use cases and are not in any way bound to the application itself, but rather belong to the general approach. For details on the implementation of the ATHENA approach as implemented in a Django/Python web application, consult Chapter 5 of this thesis.

4.1 Pipeline architecture

Pipeline architectures are an idea emerged at the dawn of the industrial era to better manage the production cycle of factories. The basic idea is that workers are specialised and only perform small parts of the total work needed to complete a task or project, with the semi-finished goods being transported between specialised stations throughout the process.

These concepts were first applied in computer sciences in the context of pipelined CPUs[1] with the aim of increasing instruction number per unit of time. This meant executing the micro-instructions separately, storing the result and reusing that stored result as message exchange between the pipeline steps. Figure 4.1 presents a classical RISC pipeline with 5 steps: Instruction Fetch, Instruction Decode, Execute, Memory access and Register write back.

As discussed in the previous chapter, the pipeline architecture has subsequently gained momentum in high-level computing as well. The pipeline architecture is suitable for projects which feature large transformation efforts with multiple failure points. By separating the set of transformations into separate steps, the black boxes composing the broken down architecture are separate entities with manageable numbers of failure points.

[1]Central Processing Unit

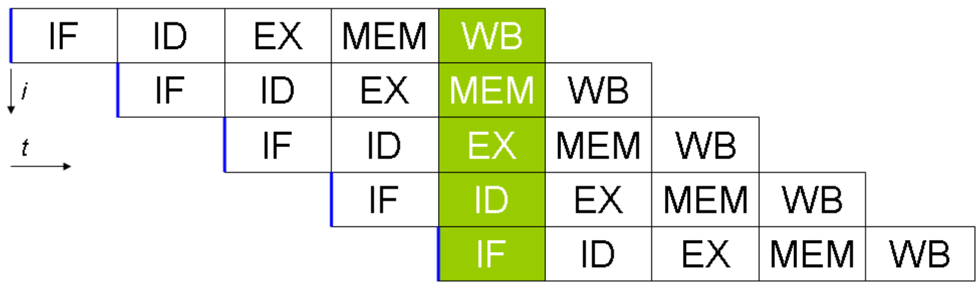


Figure 4.1: Five-stage pipeline in a RISC machine

Furthermore, the communication between the black boxes takes place by using stored semi-finished data as a method of message exchange.

Pipeline architectures are well suited for Object-Oriented Programming best practices as well, since the concept itself is SOLID (Single Responsibility, Open/Closed, Liskov Substitutional, Interface Segregated and Dependency Inverted, as per previous chapter discussion). The architecture is also encouraging of loose coupling and scaling to distributed applications. Considering the task of developing applications for fetching and analysing data from social media, it is self-understood that scaling towards distribution might be mandatory from some point on.

4.1.1 Black boxes

In the context of this approach, the black boxes which correspond to different pipeline stages are:

- the Harvesting module: methodology and system of fetching data from external APIs provided by different social networks.
- the Enhancement module: methodology for enhancing the fetched and stored data with specific details which model its structure and particularities
- the Normalisation module: handles the flattening of fetched and stored harvests using different configurations. This module helps filter outliers, noisy and/or erroneous data points, in the end storing the result.
- the Analysis module: performs comparative analysis of differences and commonalities between already normalised data sets, with focus on differences and commonalities between the two

4.1.2 Communication between modules

The execution of the Harvesting module results in the fetching and storage of related document sets. These collections of documents are stored using a persistent and manageable data source. For the time being, the choice of the data source backend is unimportant, especially from a formal point of view. However, it must be noted that the data source to be chosen is required to be fast, scalable and easily manageable.

The communication from Harvest to Enhancement and Normalisation respectively is one much similar to the classical approach of pipelined CPUs, i.e. the data is stored and then recollected inside the destination modules. As it will be seen in further sections, the nature of asynchronous jobs (employed in order to fetch the data in rate-limited APIs) adds an extra buffer to this ad-hoc message exchange channel, by not permitting yet incomplete harvests to enter processes such as Enhancement and Normalisation.

Enhancement is a module without side-effects, which means the data it uses as part of calculations remains unchanged. However, it is another case when discussing the Normalisation module. The calculations of data flattening done by the Normalisation step are recorded in the persistence layer, for further use in the Analysis module.

Last but not least, the final step considers the data previously saved by the Normalisation step to make comparative calculations of similarity and differences between two document collections which have already passed through the Harvest and Normalisation steps and are now both:

- correspondent to real-life documents authored by users using the queried social network

  and

- flattened in order to eliminate outliers and noise.

Data is generally considered as output from source boxes (steps) and input for destination ones. This is easily enforceable using step-wise validation.

4.2 Asynchronous application behaviour

The following section covers the particularities of the Harvesting module, which necessitates special attention in regards to the execution of processes. I will cover the particularities of asynchronous jobs and why it is necessary to place part of this module inside such a job.

4.2.1 Asynchronous jobs

An important classification in regards of job execution in web applications (and not only) is whether the execution is synchronous or asynchronous. A synchronous job receives its input from the end user via the application, it executes and returns a result instantly, which the user waits for and gets immediately. In the case of asynchronous jobs, users send their input to the application, but instead of waiting for a response to come, they continue to browse, interact with the application and maybe even send more and more jobs

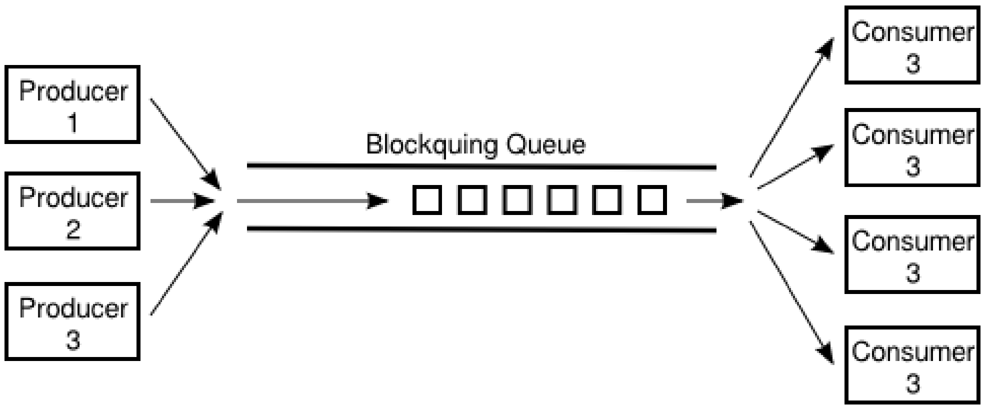


Figure 4.2: Producer/Consumer architecture example

to execute. On the backend side, said jobs are executed and might or might not notify the user when they have finished.

In dealing with large response times, it is generally better to use asynchronous execution, for a variety of reasons which culminate with the user's general experience. If a user is forced to wait for a result and prevented from doing anything else, they get easily frustrated. The ATHENA methodology, in particular the Harvest module, requires extensive querying of external data sources. Furthermore, the external APIs provided by social networks enforce rate limits, which means the client application can only get a maximum number or total size of documents before needing to sleep until the limit is replenished. It is impossible in this context to make the user wait for the result.

4.2.2 Job handling with the Producer/Consumer architecture

Up to now, we established the need for an asynchronous job execution as part of the Harvest module. To clarify on that, it is needed to establish a general architecture in which these jobs are created, executed and finalised.

The Producer/Consumer concept subsumes problems, architectures and design pattern suggestions throughout the computer science field, starting from the didactic example of the need to introduce semaphores and atomic operations in computer architecture. As a job execution framework, this setup is formalised as such:

Two processes called Producer and Consumer are connected using a buffer, which usually takes the form of a Blocking Queue. The Producer is tasked to generate data, add it into the buffer and start over. Meanwhile, the Consumer is removing data from the buffer (piecewise and in the order it was produced). In the context of asynchronous job execution, the Producer is the one that creates the jobs, while the Consumer handles each job, in order. Figure 4.2 contains a graphical representation of these details.

ATHENA includes a Producer-Consumer architecture inside its Harvest pipeline, as part of the asynchronous job execution whose necessity was explained above. While the task of getting the options from an end user and creating the job setup itself should be part of a core system, the jobs are then sent via a buffer towards a Consumer. The Consumer is instructed to fetch the related document sets as per the read options from the buffer. As part of the Consumer, a social network's REST API will be used as a source for the document collection.

To clarify, the asynchronous functionality is provided by the Consumer. Indeed the buffer is also outside what we consider core synchronous functionality, but its just a passive data source.

4.3 REST API communication

As discussed in the previous chapter, social networks have not only developed in-house analysis applications, but moreover exposed APIs (Application Programming Interfaces) which developers can use in order to query for documents authored by users of that social network. Previous sections have described *how* Harvesting works, but not exactly *what* it does, i.e. the executable content of the Consumer.

Most APIs follow the REST architecture, which stands for Representational State Transfer. The method is widely-used as opposed to other Application service implementations, because of its compatibility between applications written in different programming languages and its general ease of use, which is based on calling URIs with one of the classical HTTP verbs:

- GET: to read data
- POST: to modify data
- PUT: to update by replacing data
- PATCH: to update/modify data
- DELETE: to remove data

REST services are called by a *client* and affect a *server*. In order to acquire data about users and posts from an API exposed by a social network, ATHENA assumes the role of a client in the exchange with the social network's servers. Usually since this approach works by fetching data and not modifying it, the primarily used verb is GET, along with different parameters.

4.3.1 Authentication

External APIs usually provide users with a variety of access tokens and/or secret codes which serve both to protect the API from illegal usage by identifying the client application and cutting off its access in case any irregularities are found. On the other hand, the client can make sure that, if they haven't distributed the access tokens to other parties, their account is safe from outside penetration, i.e. other clients can not impersonate the application to piggypack any requests.

These authentication details are usually sent in every request or otherwise the credentials are established once in a certain interval of time. In any case, for the examples below detailing the usage of HTTP methods and parameters in fetching social data, we oversee authentication description for simplicity purposes.

4.3.2 Filtering with parameters

When filtering the set of posts we are interested in, we consider input from the end user of ATHENA. They are not interested in getting all the posts of all the users in every time. However, the details provided by the user when creating the Harvesting job can be used in order to filter the request to only contain conformant data.

GET parameters can be directly added to the URL called on the server, by adding a question mark character at the end of the base URI (?) and adding key and value pairs of the desired parameters separated by ampersands (&). The pairs themselves take the form:

$$key = value$$

ATHENA filters content by its label (content or, in the specific case of Twitter, hashtag) and date limits (start and end dates), which means the query URL will look something similar to:

$$https : //api.twitter.com/1.1/search/tweets.json?$$
$$q = \#python\&since = 2016-06-06\&until = 2016-06-07\&lang = en$$

4.3.3 API wrapper libraries

It is worth mentioning that in often cases, the same type of requests are used by many developers. The repetitive nature of the task to be implemented creates the business value for API wrappers, which can seamlessly be used by developers, based on some input access keys and wrapper functions.

It was previously discussed throughout this thesis that external APIs usually enforce rate limits. These can usually take the form of maximum number or size of responses per unit of time. The solution is sometimes to halt the processes fetching the data until the rate limits are replenished. These is another very common use case with repetitive nature, which makes it a candidate for wrapper functions as well. In Chapter 5 we describe the activity of chosing an implementation method for these concepts, factoring in the benefits of API wrapper libraries and functions.

4.4 Vectorising the data space

As part of the Enhancement step, ATHENA is mostly concerned with structural particularities of the document sets. In other words, we are interested in modeling the most common occurrences of content, the most popular labels and categories assigned by authors and the general correspondence between users, their posted documents and their creative content.

A first step in achieving content analysis is to perform term ocurrence analysis. This step is often called *vectorising* the collection of text documents, since it takes as an input the set of documents (as arrays of terms) and outputs a $m \times n$ matrix of term occurrences, where:

- m is the number of rows i.e. text documents in the collection
- n is the number of columns i.e. tokens found (also known as terms, however the notion of token is used since the terms are intended as single-piece elements, with eventual corelations being studied later)

But what do we consider to be a token? Well, for the first iteration of this approach, the decision to use hashtags was a result of their prevalence throughout various social networks. This particular type of labeling is done directly by the user with special characters and has gained momentum ever since its introduction on Twitter, Instagram and later Facebook. Hashtags are therefore informal labels and/or categories, which transforms this otherwise unsupervised task of feature extraction into one of less difficulty, since users themselves have performed a sort of annotation.

A larger discussion about the usage of hashtags in the particular context of Twitter can be consulted in the next chapter. For now, suffice it to say that the vectorisation can consider only particular forms as tokens and can ignore non-hashtag content if instructed so.

4.4.1 Sparse matrices

Special attention needs to be dedicated to the result of the vectorisation step. Indeed the result is a matrix but considering the large number of documents and the little number of re-occurring terms one document may contain, the matrix will undoubtedly have many 0 values. Such a matrix is called *sparse* and it requires special data structures and algorithms for proper processing. For example, consider the following document set:

1. The #cat in the #hat.
2. The #hat in the #flat.
3. The #flat #data set is better for #analysis.

Considering hashtag-only vectorisation i.e. only for tokens starting with the character "#", it produces the following result, with columns from top to bottom: #cat, #hat, #flat, #data, #analysis and documents from left to right 1, 2, 3:

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

With only 3 documents the resulting matrix is already somewhat sparse (containing 8 zero values to 7 non-zero ones), but considering a much larger document number, it becomes clear that a high proportion of the resulting matrix will contain zero values.

Generally a sparse matrix can be stored in different ways, including the classical two-dimensional array. However, traversing and modifying such a large data structure is considered cost-ineffective. Other formats[2] include those optimised for modifications:

- dictionary of keys: maps $(row, column)$ pairs to the value of elements
- list of lists: stores one list per row
- coordinate list: stores $(row, column, value)$ tuples

While other formats support efficient access and matrix operations:

- compressed sparse row: similar to coordinate list, but compresses the row indices. Indicated for row access and vector multiplication
- compressed sparse column: similar to compressed sparse row but with columns compressed. Recommended for arithmetic operations and vector products

The exact type of storage used depends on the implementation itself. Some processing methods may produce one type of representation or another, but the underlying data structure will always be a sparse matrix. Having performed vectorisation, it is then a matter of calculations and choosing the correct algorithms towards the goal.

Vectorising the space and calculating clusters (as discussed below) are important parts of the Enhancement step in ATHENA.

4.5 Clustering with KMeans

While mere presence or absence of terms inside a document collection is interesting to consult, it is not a detail which gives users much information about the composition of

[2]https://en.wikipedia.org/wiki/Sparse_matrix

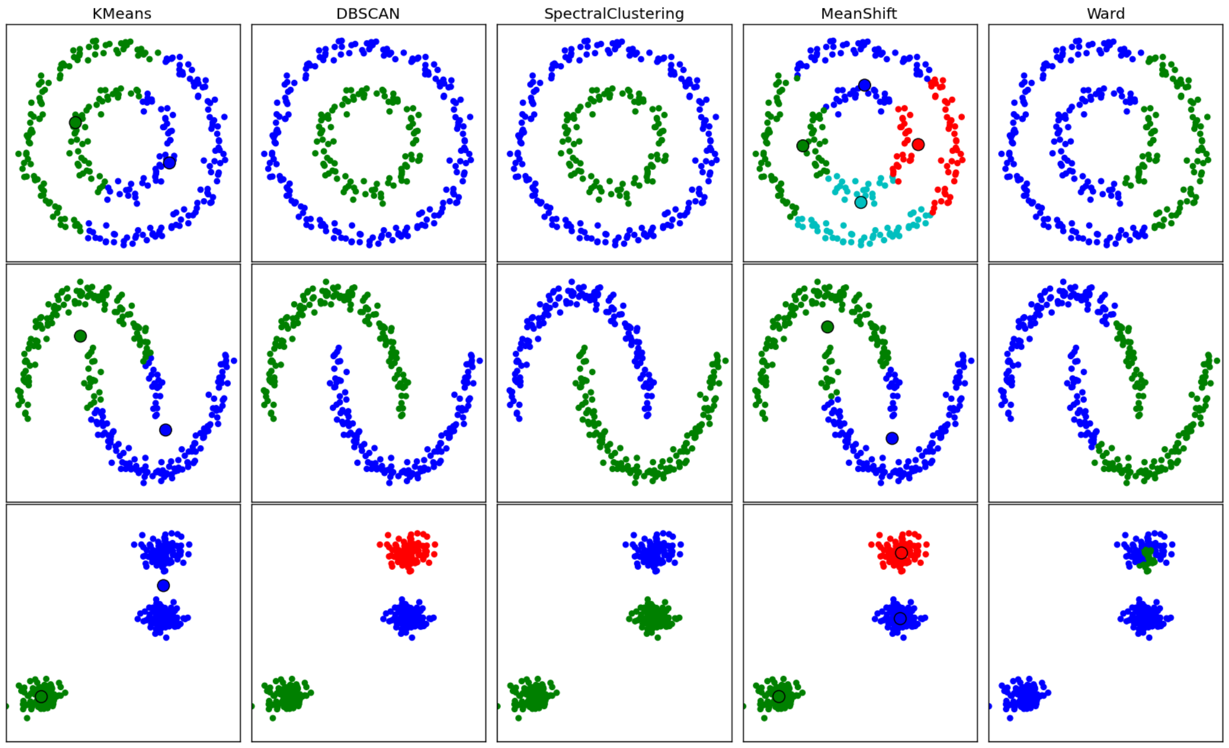


Figure 4.3: Clustering algorithms in effect on various data sets

said document collection. Vectorisation of terms is in this case a precursor for the clustering step, which uses the previously generated sparse matrix for identifying commonalities between some documents.

By clustering we generally refer to an approach, algorithm or general endeavour which assigns data points to agglomerations called clusters. The basic example follows points on a plane with given coordinates. Clusters of will be formed based on data points agglomerations, considered by the Euclidean distance measurement.

A variety of clustering algorithms exist, all with specific advantages and disadvantages. Some algorithms perform better on pattern-full data, while others handle unexpected metrics better. The efficiency of a clustering algorithm depends on the data particularities, initialisation and running options. Figure 4.3 exemplifies the performance of various clustering algorithms on different data sets, with data points marked as belonging to certain clusters by coloring on the graph. Note that some algorithms also place *centroids* (represented in the figure as small circles) which represent the center of a cluster, although it doesn't necessarily coincide with a data point.

When considering term occurrence, clearly the distance is not the Euclidean classical one. However the distance in use here is the commonality of terms used throughout the document space. E.g. documents which are used in many documents together are considered *close* while terms which never intersect inside the same document are *far* from one another. Considering the general lack of irregularities and patterns possible inside a textual dataset, the K-Means algorithm is a good fit to our approach.

Essentially, the algorithm works by initialising a number of k centroids and subsequently changing the cluster location and the data point assignment based on mean distances, i.e. at every execution step, a point is assigned to the closest centroid and a centroid's position is recalculated at the mean of its components. I will not go into much detail regarding the steps here, but rather into the problem of centroid initialisation and KMeans variants available.

4.5.1 Centroid initialisation

Centroid initialisation is a sensitive problem when considering K-Means algorithm implementation due to the propagation of errors and convergence time costliness an inappropriate initial setup could produce. The goal of centroid initialisation algorithms is to minimise intra-class distance (sum of Euclidean distances between each data point and its centroid). Some common centroid initialisation methods are:

- Forgy: randomly chooses k data points and uses them as initial means
- Random partition: starts by randomly assigning a cluster to each data point
- KMeans++: works by selecting an i centroid using a probability function based on the minimum distance from an initial x. This algorithm usually prevents accidental selection of close centroids, which can propagate and affect the algorithm's working. Using a KMeans++ initialisation, it is guaranteed that the algorithm will converge in O(log k)

In order to ensure fast convergence to a good solution, the KMeans++ initialisation algorithm is a good fit for the approach presented in this thesis.

4.6 Power law fitting

In Chapter 3 I discussed the particularities of data sets which correspond to power functions. In this section I will ellaborate on the usage of these modeling techniques to identify potential pages, bots and fake accounts on social media. In concernes related to power functions and fractalic modeling, it is not problematic to output a data set corresponding to a power function, but I will present an algorithm handling the reverse problem: how, given a data set, we can calculate the degree to which it corresponds to a power law.

4.6.1 Power functions and well-known examples

In Chapter 3 I briefly mentioned Vilfredo Pareto's now famous law which stated that 80% of the effects are generally the results of 20% of actions, as well as other scientists' (such as Mandelbrot and Taleb) concerns towards understanding and modeling general power law principles. Pareto's law indeed seems to be applicable to many domains and fields of economics and computer science[3]

- (in business) 80% of a company's profits come from 20% of its customers
- (in sales) 80% of a company's sales are made by 20% of its sale staff
- (in load testing, as a general recommendation for testing) 80% of the traffic occurs in 20% of the time
- (in artficial intelligence) some toy-worlds have demonstrated a natural convergence towards wealth distribution of 80%/20%

The proportion is of course not always 80/20 as suggested in numerous studies finding even larger discrepancies in such behaviour. For example, a 2013 study [12] found that 80% of content on the online user-editable encyclopedia Wikipedia is authored by a mere 5% of Wikipedia users. This suggest that there is merit in studying the user's post numbers as being indicative of some special pattern. There is not a study formally demonstrating that abnormally frequent posters are indeed pages, bots or fake accounts, and even in ATHENA we offer the raw modeling data to end users. However, empirical evidence seems to suggest a degree of truth in this matter, at least for some users. I do believe the modern trend in analysing fat tails and power functions, as well as non-Gaussian probability distributions will continue and further investigations can be made into the subject.

4.7 HashMap-based duplicate removal

During the Normalisation step, a classical duplication removal algorithm is used. HashMaps as a general concept are data structures that behave as a dictionary of pairs (key, value). However, the interesting part is that the constraint of uniqueness can be set on such a structure, which enforces that the collection of all keys is a set. This means that no duplicate keys may be found in the same HashMap. HashMaps have different interpretations and names in different programming languages, e.g. in Ruby they are called Hashes, in Java HashSets and in Python Dictionaries.

The duplication removal algorithm loops through the existing dataset and appends the corresponding (key, value) pairs to a HashSet. In the simplest implementation, collision is simply a matter of overwriting the existent data, but other options include checking

[3]https://en.wikipedia.org/wiki/Pareto_principle

whether the key already exists, which is of $O(1)$ complexity. This means the final algorithm complexity will be $O(n)$, corresponding to the looping part.

It is important to note that the implementation itself may vary per programming language particularities. In 5.7.3 I go into more detail on how the Python implementation can be summarised to a concise declaration instead of a list of instructions.

Summary

In this Chapter, I discussed several architectures and algorithms which are combined under the approach presented in this thesis. Firstly, I covered the suitability of pipelined architectures in text processing endeavours. Secondly, in regards to asynchronous jobs and the Producer/Consumer concept, I discussed the way document collection works with third party services which enforce rate limits.

REST API Communication is also an important concept discussed in this Chapter, as its prevalence in modern applications, platform-independence and the existence of wrapper libraries makes it easy to use when collecting documents which have been posted online, especially in social media.

As part of the pipeline's black boxes, a number of algorithms were implemented. Their importance and suitability was discussed. One such algorithm is the Vectorisation of the document collection. Not only is it used as a standalone information source, but it serves as input for document clustering as well. In regards of clustering, KMeans and its centroid-initialisation variants were covered. Power law fitting of the data set was also an important part of the ATHENA general approach, with power functions being ubiquitous in economics, mathematics and computer science. I ended with the HashMap algorithm for duplication detection and removal.

Throughout the course of this Chapter I did not go into much detail about the Analysis step of the approach, as it produces simple calculations and reuses some of the algorithms in previous steps. These functionalities will be presented in the next chapter, which is concerned with the implementation of the proposed approach into a user-friendly web application.

Chapter 5

Detailed Design and Implementation

The following chapter presents the architecture, components and implementation particularities of the ATHENA application.

5.1 Python and Django

The choice of Python as main programming language for this application, alongside the choice of Django as a framework, were based on the industry's long-time endorsement of these technologies. According to the TIOBE programming language index, Python ranks as the 6th most popular programming language, the index being calculated according to the number of engineers world-wide, courses and third-party vendors concerned with this programming language [13]. Its higher ranked competitors are Java and the C family (C, C++ and C#), which are not as well diversified in the field of research and string processing as Python.

Python for scientists

Besides its prevalence in the web application realm, Python is also one of the preferred languages for research, as argued in [14], with a plethora of scientific-oriented third-part libraries including NumPy, SciPy and Matplotlib. String processing is also generally considered faster and more cohesive than in other programming languages.

Python libraries

One of the main libraries used here is Django, arguably the best known Python web framework. Django was chosen in the detriment of others such as Flask and CherryPy, due to its active and dynamic open source community, its good documentation resources, high number of compatible third party libraries and generally available support with installation, development and running.

For the purpose of this application, a number of libraries were installed via `pip`, the package manager preferred by Python applications. The entire application was build using `virtualenv`, a virtual environment creator that helps Python developers manage different Python versions in different environments, to avoid unnecessary creation of virtual machines. Running the command `pip freeze` inside the activated virtual environment we get the list of installed third party libraries:

```
amqp==1.4.9
anyjson==0.3.3
appnope==0.1.0
backports-abc==0.4
backports.shutil-get-terminal-size==1.0.0
backports.ssl-match-hostname==3.5.0.1
beautifulsoup4==4.4.1
billiard==3.3.0.23
cassandra-driver==3.4.1
celery==3.1.18
certifi==2016.2.28
cssselect==0.9.1
cycler==0.10.0
Cython==0.24
dask==0.9.0
decorator==4.0.9
DistributedLock==1.2
Django==1.8.13
django-annoying==0.9.0
django-m2m-history==0.3.5
django-oauth-tokens==0.6.3
django-picklefield==0.3.2
django-taggit==0.19.1
functools32==3.2.3.post2
futures==3.0.5
gnureadline==6.3.3
ipython==4.2.0
ipython-genutils==0.1.0
Jinja2==2.8
jsonschema==2.5.1
kombu==3.0.35
lxml==3.6.0
MarkupSafe==0.23
matplotlib==1.5.1
mpmath==0.19
networkx==1.11
nltk==3.2.1
numpy==1.11.0
oauthlib==1.1.1
pandas==0.18.1
pathlib2==2.1.0
pexpect==4.1.0
pickleshare==0.7.2
Pillow==3.2.0
plfit==1.0.2
ptyprocess==0.5.1
pylab==0.1.3
pyparsing==2.1.4
pyquery==1.2.13
python-dateutil==2.5.3
python-memcached==1.58
pytz==2016.4
pyzmq==15.2.0
redis==2.10.3
```

```
requests==2.10.0
requests-oauthlib==0.6.1
scikit-image==0.12.3
scikit-learn==0.17.1
scipy==0.17.1
seaborn==0.7.1
simplegeneric==0.8.1
simplejson==3.8.2
singledispatch==3.4.0.3
six==1.10.0
sklearn==0.0
sympy==1.0
toolz==0.8.0
tornado==4.3
traitlets==4.2.1
tweepy==3.5.0
```

The advantage of using the `pip` package manager is that upon installing either library, its dependencies are installed as well. Throughout this thesis I will refer to the list of installed libraries when explaining the choice and particular implementation where each library is used. For now, please note the Django 1.8 installation, which is the current long-term release, chosen for its stability and support.

5.2 Storage with Cassandra

The Cassandra NoSQL database was chosen for this project's non-relational database requirement, due to its large flexibility and scalability to more clusters when the need arises.

A Cassandra server was installed locally and needs to be up for all queries run from the aplication. The Python library `cassandra-driver` allows for easy connection to Cassandra clusters and execution of queries, similarly to SQL ones. Empyrical debugging is easy via the `cqlsh` command line utility, which acts like a Cassandra interactive console. Figure 5.1 presents a demo of these functionalities, including starting up the console, connecting to a cluster and submitting a query.

The same functionality can be mirrored in Python using the following set of instructions:

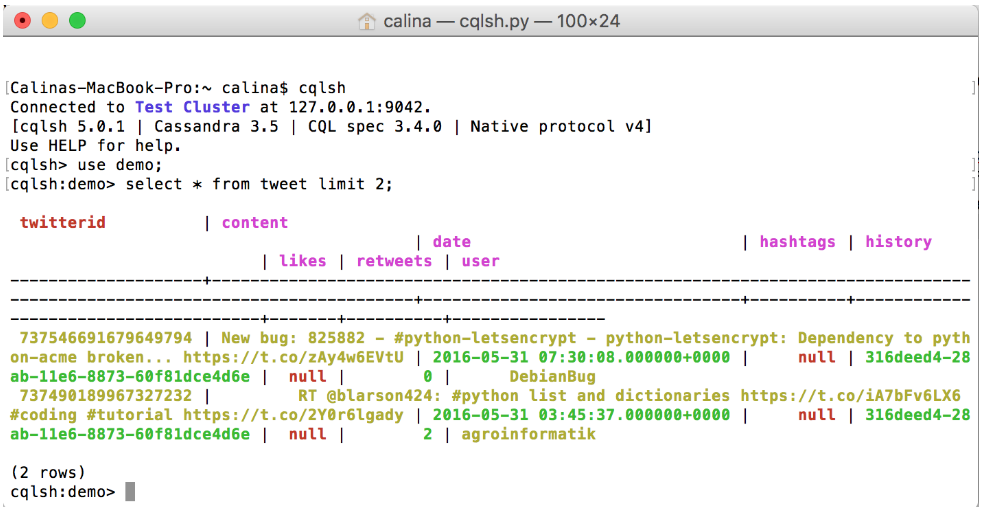
```
from cassandra.cluster import Cluster

cluster = Cluster()
session = cluster.connect('demo')

 harvests = session.execute(
    """
    select * from tweets limit 2
    """
)
```

with the result being a generator which can be further consumed by the application. This means that it is easy to perform queries programatically, without much overhead, by employing the usage of this driver.

Cassandra queries are used throughout the application, both in synchronous and asynchronous application steps. Its speedy retrieval and updating of records is not a



```
Calinas-MacBook-Pro:~ calina$ cqlsh
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 3.5 | CQL spec 3.4.0 | Native protocol v4]
Use HELP for help.
cqlsh> use demo;
cqlsh:demo> select * from tweet limit 2;

 twitterid          | content
                                          | date                            | hashtags | history
                          | likes | retweets | user
--------------------+-------------------------------------------------------------------------------
------------------------------------------+---------------------------------+----------+------------
--------------------------+-------+----------+-----------------
 737546691679649794 | New bug: 825882 - #python-letsencrypt - python-letsencrypt: Dependency to pyth
on-acme broken... https://t.co/zAy4w6EVtU | 2016-05-31 07:30:08.000000+0000 |     null | 316deed4-28
ab-11e6-8873-60f81dce4d6e |  null |        0 |      DebianBug
 737490189967327232 |          RT @blarson424: #python list and dictionaries https://t.co/iA7bFv6LX6
#coding #tutorial https://t.co/2Y0r6lgady | 2016-05-31 03:45:37.000000+0000 |     null | 316deed4-28
ab-11e6-8873-60f81dce4d6e |  null |        2 | agroinformatik

(2 rows)
cqlsh:demo>
```

Figure 5.1: Demo of the cqlsh utility

bottleneck, as API rate limits are.

5.2.1 Database structure

Two tables are used in our application. The `harvest` table containing columns:

- uuid (primary key, generated for each user-submitted harvest form)
- start date (from which we harvest Tweets)
- end date (up to which we harvest Tweets)
- hashtag (used to query Twitter's API for statuses containing this particular hashtag)
- done (boolean fag indicated whether the harvest has finished downloading Tweets)

The `tweet` table is used for storage of tweets belonging to histories and contains:

- twitterId (the id of that status on Twitter)
- user (username of the author on Twitter)
- content (textual content including hashtags)
- date (of postage)
- retweets (number of retweets, indicating popularity)
- history (uuid of harvest that Tweet belongs to inside ATHENA)

5.3 Harvesting tools

The harvesting module as presented conceptually in 2.2 was implemented in ATHENA using asynchronous job queueing. The user is presented with a form for submitting the harvesting job, consisting of a content field, start and end dates. An asynchronous job is launched via Python's Celery task library, which uses the tweepy library to connect to the Twitter Search API and Cassandra driver to save tweets and harvests.

5.3.1 Celery for asynchronous jobs

The Python library Celery[1] is designed for the fairly frequent development case where asynchronous jobs need to be managed. The Producer-Consumer architecture is implemented with Celery, as used in its real-time mode, while another option would be using Celery to run scheduled jobs. The job granularity in this case is indeed Harvest-level, with each job inside the Producer-Consumer queue is defined as the job of downloading one single, separate Harvest.

As previously stated, a huge advantage of Python is that third party libraries are highly compatible and easily configurable. Such is the case with Celery as well, setting up an asynchronous jobs taking very little effort:

1. install the `celery` library using `pip`
2. add configuration information to a celery.py file inside the application directory, including the task body and its decorator `@app.task(bind=True)`
3. import the function and use it. In the ATHENA Harvesting module context, the function was added as part of the Form validation customisation in Django's FormView class
4. install and configure a service broker such as Redis

5.3.2 Using Redis as a Celery broker

Celery offers a variety of possible brokers for message transport through the job queue. Stable brokers are RabbitMQ and Redis, while others are in Experimental stages or offered by third parties[2]. Redis is an open source data structure store which can perform as a database or cache system, but in our case we are interested in its functionality as a message broker.

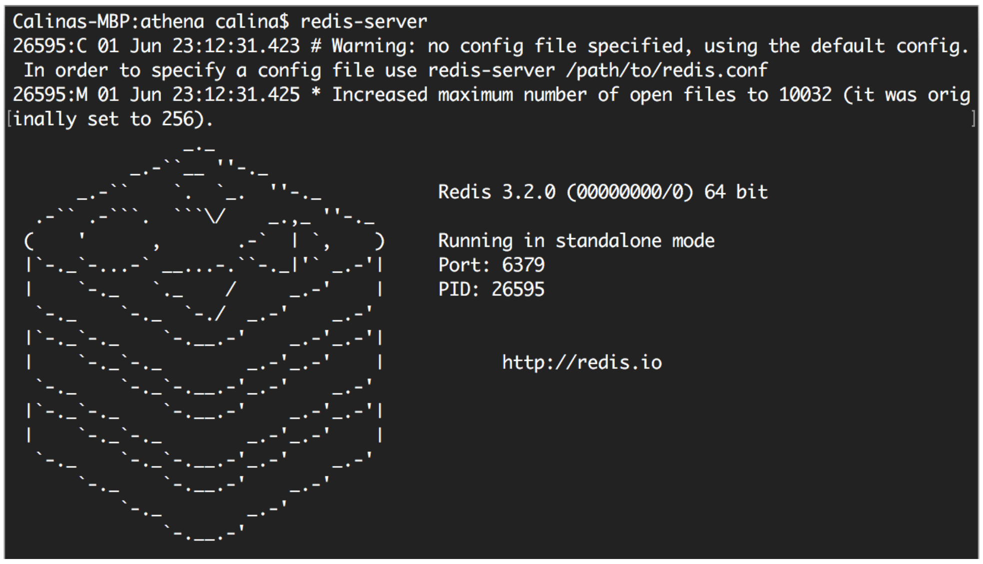
Installing Redis is straightforward using a downloaded package and even some available package managers such as Mac's `brew`. After the installation is complete, the Redis server can be fired up using the command `redis-server`. A splash screen with Redis' logo

[1]http://www.celeryproject.org

[2]http://docs.celeryproject.org/en/latest/getting-started/brokers/

as ASCII art, such as the example in 5.2 should appear, but connection to the running Redis server can also be tested by pinging:

```
$ redis-cli ping
PONG
```



```
Calinas-MBP:athena calina$ redis-server
26595:C 01 Jun 23:12:31.423 # Warning: no config file specified, using the default config.
 In order to specify a config file use redis-server /path/to/redis.conf
26595:M 01 Jun 23:12:31.425 * Increased maximum number of open files to 10032 (it was orig
inally set to 256).
                _._
           _.-``__ ''-._
      _.-``    `.  `_.  ''-._           Redis 3.2.0 (00000000/0) 64 bit
  .-`` .-```.  ```\/    _.,_ ''-._
 (    '      ,       .-`  | `,    )     Running in standalone mode
 |`-._`-...-` __...-.``-._|'` _.-'|     Port: 6379
 |    `-._   `._    /     _.-'    |     PID: 26595
  `-._    `-._  `-./  _.-'    _.-'
 |`-._`-._    `-.__.-'    _.-'_.-'|
 |    `-._`-._        _.-'_.-'    |           http://redis.io
  `-._    `-._`-.__.-'_.-'    _.-'
 |`-._`-._    `-.__.-'    _.-'_.-'|
 |    `-._`-._        _.-'_.-'    |
  `-._    `-._`-.__.-'_.-'    _.-'
      `-._    `-.__.-'    _.-'
          `-._        _.-'
              `-.__.-'
```

Figure 5.2: Redis splash screen

After the Redis server is properly installed and running, there are some extra steps for hooking it up: adding the Redis configuration settings in our project's `settings.py` file and installing the `redis` library using pip. After all these steps are completed, running:

```
celery -A athena_app worker -l info
```

should confirm Celery's connection to Redis. Any tasks that were previously set up will now go through this job queue.

5.3.3 Fetching data from Twitter using the Search API and Tweepy

Once the general configuration of the asynchronous job is done, we can use the Twitter Search API to Harvest tweets per the specifications.

Twitter belongs to a series of web applications which fully understands the developers' need to hook into some of their features. Creating a Twitter application is easy from their developer support pages, with the creator receiving a set of OAuth access keys:

- a Consumer Key (API Key)
- a Consumer Secret (API Secret)

- an Access Token
- an Access Token secret

For harvesting tweets, my approach uses a wrapper to Twitter's Search API called Tweepy[3]. It is a library that handles connection and customised requests to the API, in a Pythonic fashion. Tweepy needs to be installed using pip, and then configured with the proper access keys (here in the code sample replaced with placeholders for security purposes):

```
from tweepy import OAuthHandler

consumer_key="XXXX"
consumer_secret="XXXXXXXX"

access_token="XXXX"
access_token_secret= "XXXXXXXX"

auth = OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_token, access_token_secret)
```

The way I am using this is by first setting up an `api` object which I will further query for statuses. I will initialise it using the `auth` object and a few flags:

- `wait_on_rate_limit=True`. Flagging this indicates to the api object that whenever it reaches the API rate limits, it should not stop, but rather sleep for the required amount of time until new data can be acquired. This approach is preferable, since our harvesting takes place asynchronously and we do not mind the process sleeping for a while
- `wait_on_rate_limit_notify=True`. Setting this flag tells the api object to print out a warning string to the console, indicating when it has reached the rate limit and at the beginning and end of the sleep cycle. E.g.:

  ```
  [2016-06-02 10:17:46,804: WARNING/Worker-2]
  Rate limit reached. Sleeping for:
  627
  ```

After setting up the API, a Tweepy Cursor class is used for querying it. This is initialised with the api object and parameters indicating the query we send to the server. This uses parameters defined by the user upon submitting the harvest form, refined by our application's specifications. The hashtag field is prefixed with a "#" sign, start and end dates are formatted to "yyyy-mm-dd" and the language set to English, since extension to other languages is beyond the scope of this thesis.

[3]http://tweepy.readthedocs.io/en/v3.5.0/

The cursor will return a Pyhton generator, which means that the contents of the tweet list is rather consumed one by one, rather than having the whole list present at either point in time. Here, using Cassandra, we save the contents to the database.

```
tweets = tweepy.Cursor(api.search, q='#' + hashtag, since=start_date,
    until=end_date, lang='en').items()

for tweet in tweets:
    session.execute(
     """
     insert into tweet (twitterId, user, content, date, retweets,
        history) values (%s, %s, %s, %s, %s, %s)
     """,
     (str(tweet.id), tweet.author.screen_name.encode('utf8'), tweet.
        text, tweet.created_at, tweet.retweet_count, key)
     )
```

The stored and completed harvests are now ready for the Enhancement step.

5.4 Enhancement tools

Enhancement tools are concentrated in the enhancement_manager.py file/module where basic numerical calculations on the data is performed, alongside more complex transformations such as related hashtag collection and hashtag clustering. The process of enhancing data starts on the Enhancement tab, with the user being presented the list of available Harvests. The user decides on the harvest they want to enhance and display by clicking an icon next to the harvest's title.

5.4.1 Simple numerical data

A number of easily calculated, relevant data is calculated for the user and displayed on the enhancement results page. The number of tweets is calculated using a database query and displayed in a separate `<div>` on the results page.

User-related data

Upon consuming the tweet generator which resulted from the query, the data related to users is restructured as a Pyhton dictionary having the usernames of keys and the number of documents authored by those users, in the contex of the present harvest. The following numerical data is displayed related to users:

- the number of unique users contributing tweets to the enhanced harvest

- the maximum number of tweets per user. The author's username is also presented in parantheses, and will usually represent a bot or a domain-related page, e.g. for the `#pyhton` hashtag, the highest number of posts in a given day is posted by the user `@pyhtonbot_`
- the average number of posts per user, which is a metric indicating the general activity level of the harvest's generator hashtag

Modeling the post numbers

Since many a times the number of posts per user tends to skew towards pages and bots, it may be the case that the number of posts indicate a certain pattern. To analyse the eventuality of post numbers as an estimation of a power function, the `plfit` library was used. First the data was prepared as a list, from the users/posts dictionary described above.

The `plfit` library was originally written in Matlab by Clauset et al [11] and later converted to R, Python and C++ modules by various developers. It handles calculations regarding data pattern fitting to a power law as described in 4.6.1. The library is suitable for non-domain-related data, being based on a combined maximum-likelihood fitting and goodness-of-fit tests. It was tested on twenty-four real-world data sets from unrelated fields of study and has found good results with power law fitting.

After importing the `plfit` library into the enhancement module, a plfit object is created. The tweet numbers are then fitted using the plfit method on the list described above. The minimum value and alpha are printed on the enhancement results page.

5.4.2 Sci-kit learn

Sci-kit Learn[4] is proof of the fact that Python has recently gained much momentum in science and research. Its architecture is build on other stable and related Python science libraries, namely NumPy, SciPy and matplotlib. In fact, upon installation with the `pip` command line utility, sci-kit also installs these libraries as dependencies. According to a large number of scientists [15] and as displayed on their website, sci-kit learn is used for supervised and unsupervised data mining and data analysis endeavours, including:

- Classification
- Regression
- Clustering
- Dimensionality reduction
- Model selection

[4]http://scikit-learn.org/stable/

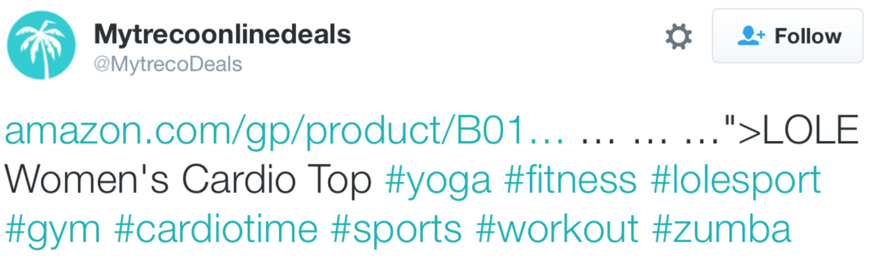
- Preprocessing

Other important advantage of this library is its open souce BSD licensing. This means that, even tough it is a highly specialised tool, it is free to use for developers worldwide. The following presents *sci-kit learn*'s employment in ATHENA.

5.4.3 Related hashtag calculation

Twitter uses a hashtag system to encourage users to tag data for proper search and visibility. Hashtags are words prefixed by a "#" character, which represent categories of statuses. The hashtag system has been used on Twitter ever since its start, as opposed to other social networks such as Facebook, who only employed tagging much after being launched. Such other social networks have a hashtag disadvantage, since the system did not "catch on" with their users as naturally as it has on early-adopters such as Twitter.

On going into the realm of more complexly calculated data, as opposed to the numerical values extracted directly from the list of tweets, calculating a harvest's related hashtags is a more difficult task and is achieved using sci-kit learn. Figure 5.3 presents a sample tweet (the screenshot was taken on 7th June 2016), and it is clear empirically from such status messages that hashtags can be interpreted as category labels.


Mytrecoonlinedeals
@MytrecoDeals

amazon.com/gp/product/B01… … … …">LOLE Women's Cardio Top #yoga #fitness #lolesport #gym #cardiotime #sports #workout #zumba

Figure 5.3: Sample status message with hashtags

The tweet in Figure 5.3 also empirically demonstrates that the same post belongs to a number of related categories, in this case *yoga*, *fitness*, *lolesport*, *gym*, *cardiotime*, *sports*, *workout* and *zumba*. Indded these hashtags seem much related to one another, in this case, all belonging to a domain of sport and fitness.

Analysing each status' component hashtags will reveal related hashtags. As part of the enhancement process, we consider all the related collected tweets which compose a harvest and go trough the `get_vocabulary` function, much like in the grammatical process of collecting a word's semantic field.

Getting the hashtag vocabulary

For the purpose of getting the list of related hashtags to our own harvest's base hashtag, we use a few customised components from the *sci-kit learn* toolbelt. I start by vectorizing the list of tweets using the `TfidfVectorizer` with the following options:

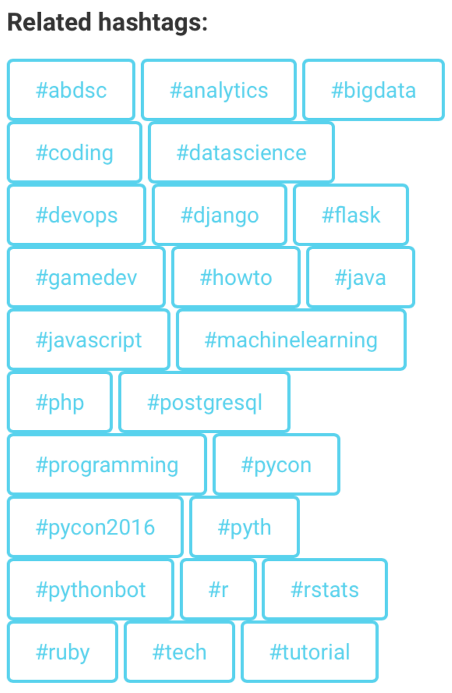
- `min_df=10`, to prevent hashtags who appear in less than 10 tweets appearing in our final list of results. This prevents noise generated by use-once hashtags through unusual, parody or disparate hashtags.
- `max_df=0.8`, to prevent flooding with most-common hashtags appearing in more than 80% of the corpus
- `sublinear_tf=True` to use a sublinear function. A linear idf function may boost the document scores sometimes, but since the frequency of a term to relevance is usually a sublinear function, we use this option to ensure higher-precision results.
- `use_idf=True` to enable inverse-document-frequency re-weighting
- `max_features=25` to limit our search of most relevant related hashtags to a number of 25. We do this so that only the best most relevant hashtags are displayed for the user and prevent them being distracted by too much information.
- `token_pattern='#[a-zA-Z0-9][a-zA-Z0-9]*'`. This option takes a regular expression (RegEx) argument which it uses to identify relevant tokens inside the corpus. Since we are only interested in hashtags and not other content words, the RegExp introduced forces the vectorizer to only consider "words" that start with a "#' symbol and has one or more alphanumerical characters afterwards.

The sparse matrix indicating term frequency is obtained using the initialized vectorizer to `fit_transform` the list of tweets' texts. Although a sparse matrix would overwhelm a non-expert user, and I have chosen not to display it as part of the enhancement results, the structure's content can be printed on the console and used for debugging and bird's eye correctness checking.

We do however return the `vocabulary` variable. We calculate this by having the vectorizer perform the `get_feature_names()` method. This enables us to print a list of related hashtags on the enhancement results page in a separate `div`, as exemplified in Figure 5.4. Indeed the empyrical data obtained in the example for the #python hashtags showcases some of Python's most popular features (#bigdata, #machinelearning), tools (#django, #flask) and related technologies (#javascript, #ruby, #r).

5.4.4 Related hashtag clustering using KMeans

Another interesting aspect the hashtags indicate is that there are some general directions (or common features) of our text. In a supervised approach, this could be tackled with an adnotated training dataset which would help classify the data. However, since ATHENA is based on user input and analysis of non-domain-related documents, such an approach is not appropriate.


Related hashtags:

#abdsc #analytics #bigdata
#coding #datascience
#devops #django #flask
#gamedev #howto #java
#javascript #machinelearning
#php #postgresql
#programming #pycon
#pycon2016 #pyth
#pythonbot #r #rstats
#ruby #tech #tutorial

Figure 5.4: Example of related hashtags in a #Python harvest over one day

One of the most classic approaches in classifying non-adnotated data is the K-Means clustering algorithm. Luckily *Sci-kit learn* also contains tools for clustering approaches, including various implementations of K-Means variants.

Calculations start similarly to the approach described above, for related hashtags, but features an extra step for clustering. The concern for choosing a number of cluster centroids was similar to the one choosing the number of related hashtags to be displayed: A number of 5 centroids was chosen with the goal of having enough diversification, but not too many clusters to display, so as not to confuse the user. In a more domain-tweaked approach, the number of clusters would have been chosen according to the data's structure, but here it is a trade-off between the various factors explained.

A new `TfifVectorizer` object is instantiated with similar options, except for the `max_df`, in order to also consider highly-used hashtags. After reproducing the `fit_transform` step and obtraining the fitted set X, we apply the KMeans algorithm as such:

```
from sklearn.cluster import KMeans
[...]

model = KMeans(n_clusters=true_k, init='k-means++', max_iter=100,
    n_init=1)
model.fit(X)
```

For the centroid initialisation, I chose the "K-Means++" algorithm described in

4.5.1, a maximum number of iterations towards the solution of 100. The `n_init` option instructs the K-Means algorithm to run just once with different centroid seeds. In case a larger number would have been chosen, the output of the algorithm would have been the best output during these runs.

Ordering and displaying the hashtag clusters

The main issue with clustering generation, as directly resulted from the *Sci-kit* K-Means algorithm, is that the clusters are not ordered. Items are assigned with some probability to each cluster, but there is no clear representation and collection of those clusters. In order to properly display these clusters on the results page, though, they need to be sorted and formatted.

```
order_centroids = model.cluster_centers_.argsort()[:, ::-1]
terms = vectorizer.get_feature_names()
clusters = {}

for i in range(true_k):
    cluster_name = 'Cluster ' + str(i+1) + ':'
    cluster = {}
    for ind in order_centroids[i, :10]:
        cluster[ind] = str(terms[ind])
    clusters[cluster_name] = cluster

    return clusters
```

Note the similarity to the previously described feature. The terms are fetched just the same, and then their assignment to the clusters is checked. The resulting structure is a dictionary consisting of cluster names as keys and hashtags (from the same cluster) as values.

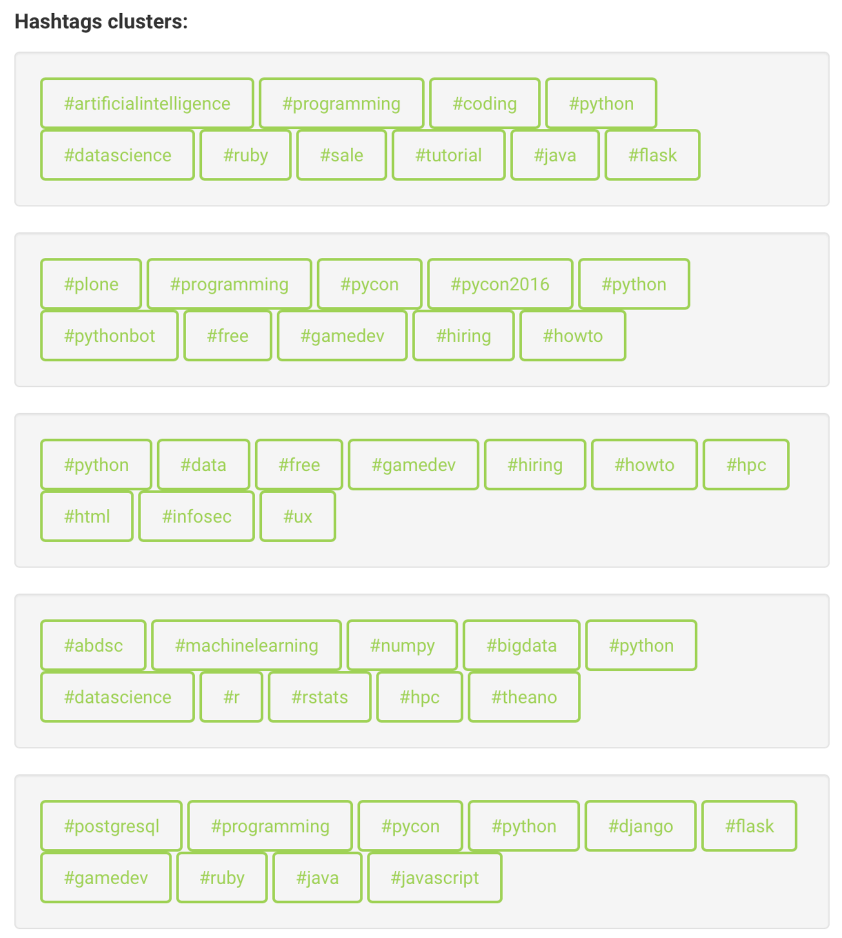
They are returned to the controller, then to the front-end part and displayed via Django's templating engine. For a better understanding of their separation, each cluster is placed into its own div with Bootstrap's `well` CSS class, which adds a separate wrapper and colouring to its contents. Figure 5.5 presents an example of such a clustering.

It can be seen in the example that the clusters follow a structure similar to Python's main fields of development.

1. the first cluster is correspondent to some genric Python fields: #flask, #coding, #artificialintelligence

2. the second cluster is correspondent to Python conventions, promotions and opportunities for development: #free, #gamedev, #hiring, #pycon

3. the third cluster relates to front-end technologies and their prevalence in Python conventions: #html, #ux



Hashtags clusters:

#artificialintelligence #programming #coding #python #datascience #ruby #sale #tutorial #java #flask

#plone #programming #pycon #pycon2016 #python #pythonbot #free #gamedev #hiring #howto

#python #data #free #gamedev #hiring #howto #hpc #html #infosec #ux

#abdsc #machinelearning #numpy #bigdata #python #datascience #r #rstats #hpc #theano

#postgresql #programming #pycon #python #django #flask #gamedev #ruby #java #javascript

Figure 5.5: Example of clustered hashtags in a #Python harvest over one day

4. the fourth cluster relates to Python's endeavours in machine learning: #machine-learning, #bigdata, #numpy, #datascience, #r, #rstats

5. the fifth cluster is a collection of technologies related to Python or generally used alogngside it: #postgresql, #django, #flask, #javascript

New clustering is calculated with every refresh, and the user can observe a high degree of convergence for rich hashtags such as our #python example. For one-fold or loosely related hashtags, the number of centroids chosen might indeed be too large.

5.5 Normalisation

5.5.1 Reasons and options

As previously discussed, many a times the dataset is skewed due to spikes and outliers such as pages and bots. The normalisation step helps with the process of eliminating outliers and possible fakers, by flattening the dataset to a single representative post per each user. This is done mostly because professional pages intentionally use repetitive hashtags to promote certain events. On the other hand, bots scan for and retweet statuses which feature specific content, which results in the flooding of the dataset with promotional messages. Both these series of events lead to skewed dataset wich is easily recognisable by its fitting to a power function.

Remember that the Enhancement step only takes regular harvests as inputs, as it calculates power set fitting, likely pages and outliers etc. In a similar fashion, the Analysis step will only consider normalised harvests as permitted input. This means that output of the Normalisation step should be properly stored, in a separate table.

It must also be noted that the Normalisation step is still in its early proof of concept ages, since the main focus of this application was to develop and validate the Harvesting, Enhancement and Analysis steps, and only then add more functionality to the Normalisation module. Proper Normalisation should indeed take into consideration various factors:

- flattening process:
  - one post per user, to remove outliers such as pages and bots
  - multiple posts-only, to analyse only regular users, which use the hashtag more times, which would filter out accidental ones with opinions that might be unrepresentative for the community
  - popular posts-only, considering only posts with retweet count larger than a threshold r
  - liked posts-only, considering only posts with like count larger than a threshold l.
  - combinations of the above options
- time period considered
  - short term options: n-days with $n \leq 7$, m-hour normalisation
  - long term options: n-days with $n > 7$, which would also imply the possibility of streaming harvests on a longer period, per Twitter API's rate limits.

Each normalisation method has its very own advantages and drawbacks, so in order to choose a viable normalisation method to implement, the following was considered: Unfortunately with promotional messages such as those posted by pages and bots, the retweet and like counts is often tied to some form of prize or gratification a user gets for sharing the content. This means that the popularity of the tweet is many a times artificially boasted using a variety of marketing methods. Also, it might be the case that few users actually post multiple tweets in a short-term amount of time, which rules out the second option of flattening as well.

Current implementation restricts normalisation options to one post per user and one-day date limits. However, the code is designed in such a fashion that new options should be added easily, when the need arises.

5.5.2 One post per user, one-day limit normalisation process

The process I describe below is currently the only allowed Normalisation process, with more to come after testing and user validation of the ATHENA proof of concept app. This method of normalisation intends to remove outlier influence of pages, bots and fake accounts.

The first interface presented to the user in the Normalisation step is a form containing the list of harvests in a dropdown. After submitting the form, the selected harvest is sent to the `normalisation_manager.py` file through the controller. This manager file contains the process necessary to fetch the harvest's tweets from the database and flattening the result into a list.

The method used was the classical approach of duplicate removal using HashMaps (as implemented in the Python dictionary data type). Using the username as the dictionary's key, the last occurrence in the list of tweets for each user is considered. The value of this result dictionary is a tuple of tweet id and tweet content, which are later used in the Analysis module. Of course, the tweet's post date is also considered, in order to filter out tweets outside the date limit.

Results are stored in the `normal` database table, which has a structure consisting of the following columns:

- original harvest uuid, which facilitates comparison between corresponding normalised and regular harvests
- name of the normalised harvest (formed using the original harvest's name and the normalisation type details)
- JSON-serialised normalised dictionary resulting from the normalisation step

Or, as CQL script:

```
CREATE TABLE normal (uuid uuid, name text, content text, PRIMARY KEY(
    uuid));
```

Multiple formats would have been suitable for serialisation if persistence only was considered. But in the idea of adding a RESTful module to the application, enabling its use from mobile and web applications seamlessly, the normalisation result is stored in a JSON format. JSON is a widely-spread and widely-used format for REST service communication and is easily integrated with Python and Django via the `json` library. The contents of the Python data structure is encoded using `json.dumps()`, while the loading from a string and into a Python data structure can be achieved using `json.loads()`.

5.6 Analysis

The Analysis module features a lot of similarity to both Enhancement and Normalisation, except that in both cases, the pages contain double the harvests. Firstly, the form step of Analysis contains two dropdowns which are populated with normalised harvests. After choosing a pair of harvests for comparison, the application redirects to a page displaying the analysis results.

As per previous modules, this one also has a dedicated manager file, called `analysis_manager`. Analysis jobs are therefore passed from the controller to the manager and the results are sent back after calculation. A number of analysis types are run on the normalised harvests, including common vocabulary, common users and post proportions.

5.6.1 Retrieving the normalised results

The task of retrieving the previously saved normalised results is a fairly simple one, consisting of connecting to the database, running a query to select the two normalised entries as selected in the form (using their respective uuids) and deserialising the JSON string in the content using the `json.loads()` function. The deserialised result is exactly as the one calculated in the Normalisation step, i.e. the keys are the users and the values are tuples of significant data, including tweet content.

5.6.2 Calculating the common vocabulary

The common vocabulary of hashtags is a type of analysis inspired from the previous Enhancement step. We are interested, in case of hashtag pairs belonging to the same fields or domains, in their very own, particular characteristics (handled here with Enhancement), but also in some common values which the share. One such important question is whether there are any common labels between the two.

Common vocabulary generation is very much similar to vocabulary generation, as done in the Enhancement module. Hence the calculation consists of both respective vocabularies intersected. This means that the vocabulary function from enhancement can be reused in a DRY (Do not Repeat Yourself) fashion. The operation of intersection is done using the & operator between the two sets, which is an efficient way of getting the result without much code written.

Figure 5.6 shows an example of results for the comparison between one-day normalisations of the #python and #php hashtags, indicating two general categories the two

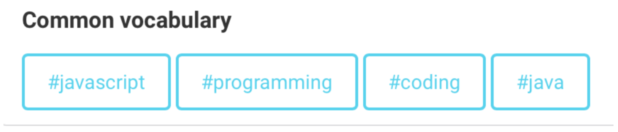


Figure 5.6: Example of common vocabulary in #Python vs #PHP comparison

belong to (#programming and #coding), and two technologies usually associated to them, one on the front-end side (#javascript) and one on back-end (#java)

5.6.3 Calculating common users

Common users are calculated in a similar fashion to that described above. Since the result deserialised from the database is a dictionary with unique keys (Python dictionaries are based on an underlying HashMap structure), the only task is to combine the two key arrays using an intersection operator, as previously done for the vocabulary part, e.g.:

$$common\_users = set(h1\_users)\&set(h2\_users)$$

5.6.4 Calculating post numbers and proportions

Another important metric is the prevalence of posts belonging to a harvest as part of the domain. In comparing two normalised harvests, we are interested in the proportion of each exclusive tag and also the number of common posts. These metrics show whether one or the other hashtag is the most productive in number of posts, which may indicate number of supporters. Furthermore, by analysing the numbers and proportions of these posts, we can deduce whether the domain is clearly split between the two hashtags, or whether there is a lot of gray area, which contains posts belonging to both categories. This latter option would indicate that supporters of both concepts are fluid and have no clear preference, while a clear separation could indicate die-hard fans.

Of course this relation can be exploited in a number of ways. In a very hostile, polarised domain, the user can analyse the risks of appealing to one of the groups, while rejecting the other. However in a cohesive domain with much middle ground, a good option is to appeal to the general unifying concepts.

In order to calculate the domain breakdown between the two normalised harvests, we consider users as a point of reference. We have already calculated the set of common users during the previous steps, so now we calculate the difference sets using the lists of posting users in each harvest and excluding the common ones.

Of course displaying the raw numbers on the results page doesn't help the user that much. It is therefore necessary to perform some extra steps to indicate the polarisation of the domain between the two hashtags: displaying the numbers as a graphical pie chart.

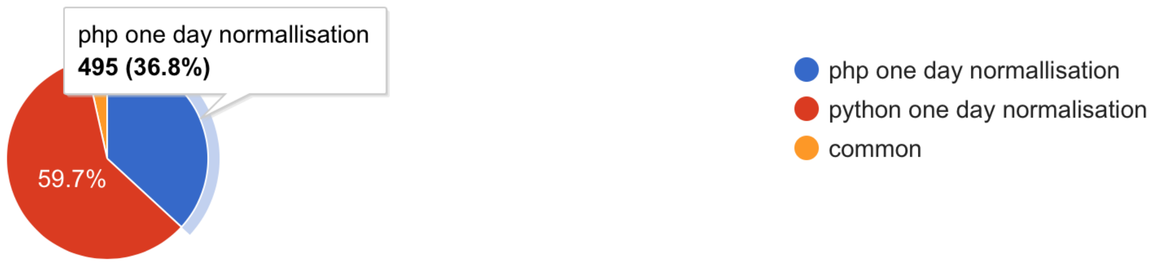


Figure 5.7: Proportions of users who post #Python vs #PHP tagged statuses

Such a chart indicates the proportions of users who post exclusively for each hashtag and common users, who represent the middle ground.

Figure 5.7 presents the results for the user proportions between two hashtags: #python and #php. The results indeed seem to match with empyrical real-life data, since the two are both back-end programming languages and it is rarely seen that both are used by the same people or in the same projects. Results also display a highly-polarised domain, with a slight better cult of Python, which can be confirmed from other sources such as the TIOBE index, which maintains Python is more popular than PHP.

Displaying the chart using Google Charts API

A number of approaches for HTML charting are available for free use. Some work primarily on the back-end with image generation, while some are exclusively front-end. Among the simplest and easiest to use approaches is using Google Chart API, a javascript library for front-end chart generation. The image in 5.7 is captured from the application and is indeed generated using Google Chart API. The code is fairly straightforward, using javascript on the front-end side, from the list of values.

```
var data = google.visualization.arrayToDataTable([
        ['Norm. Harvest', 'Number of posts'],
        ["{{ h1 }}",       {{ proportions.0 }}],
        ["{{ h2 }}",       {{ proportions.1 }}],
        ["common",       {{ proportions.2 }}],
    ]);
```

Note that the curly brackets used are from Django's templating engine, and are variables which are passed from the controller onto the view. When executing the javascript code, the values are already replaced with the corresponding values and will execute like normal strings or numbers within the code.

Hovering the mouse over one pie chart slice generates a dialog containing both the raw number and the proportion from the total, as shown in the figure as well, onto *php one day normalisation*.

5.7 Code organisation

Generally speaking, the code is organised in a classical MVC (Model-View-Controller) fashion, with specifics related to Django's recommentations. As per their recommended usage, the functionality is separated into an app called `athena_app` found in the `athena` workspace, under a virtual environment wrapper with Django 1.8 installed.

5.7.1 File and folder structure

As Django tends to encourage reusability, some of the scaffolding and file structure is generated via commands using the `manage.py` command line utility. E.g., upon creating a new app, a directory structure is also created containing:

- `urls.py`, the file which contains URL path definitions and directs received data to the corresponding views, as defined by the user. Our application currently defines a total of 8 URLs, for the index page, harvest, enhancement, normalisation and analysis page as well as some result pages. The local `urls.py` file is imported into the main `urls.py` which resides in the root of the project.

- `views.py`, the file which contains what other frameworks call "controller actions". These functions determine the behaviour of the application when encountering the data matched by respective entries in `urls.py`. As a general best practice, our functions in `views.py` either inherit and override significant parts of Django generic views[5], or use minimal code and delegate to manager files for the core functionality.

- `models.py`, which should contain model definition. This file was not used, since database queries were written manually in CQL, which is not that well supported by Django's migration creating structure for models and migrations.

- `forms.py` which contains form definitions, holding part of the functionality, field definitions and validation instructions

- other various folders such as the `static` folder where static files such as images, CSS and JS are stored, and the `templates` folder where html views are stored. Our application's static folder contains various Twitter Bootstrap CSS and JS elements for beautifying the front-end, while our templates directory holds a number of 6 individual views (mostly corresponding to the URLs enumerated above) and one `base.html` view which the others use as a general template. This enables us to only overwrite those sections which change and leave the menu and headers generally intact.

[5]Code templates for widely-used functionalites such as CRUD, forms or simple HTML rendering views

- some other files and folders which we have not used and are beyond the scope of our application, such as `admin.py` (which should define administrative actions with Django Admin) and others.

Python's import structure is defined using file names and classes/functions defined inside those files. This makes it easy to use functions from files such as managers, by simply importing those functions. The separation of concerns principle is very important for proper maintenance and good functionality of the code, so the controllers stay light and handle Request processing and Response sending, while the core work is done separately in managers. The application defines a number of four managers:

- `harvest_manager.py`, dealing with harvesting jobs and used primarily from the harvest parts of the `view.py` file
- `enhancement_manager.py`, dealing with the enhancement-concerned parts of the application
- `normalisation_manager.py`, dealing with flattening the datasets as instructed by the corresponding actions in `views.py`
- `analysis_manager.py`, containing analysis functions and used by the Analysis module

As previously stated, Python's simple structure make it easy to import functions from the managers inside the views using specific commands and then straightforwardly using the function, e.g.:

```
from athena_app.harvest_manager import get_harvests
[...]
result = get_harvests()
```

5.7.2 Using Django generic views

Django offers the possibility of reusing specific classes from their very own codebase, classes which were designed for repetitive tasks. As opposed to having programmers code their very own implementations of rendering simple HTML views, creating and validation forms or performing CRUD (Create, Update, Delete) funcionalities, these classes, called generic views, can be directly used, or efficiently customised by overwriting a few attributes or methods.

ATHENA uses Django's `FormView` and the basic `View`, imported from
`django.views.generic.edit` and
`django.views.generic.base`
respectively. The former is used for creating the Enhancement and Normalisation harvest selection page, and the Analysis pairwise normalised harvest selection page. FormView is also used when creating the harvest and is perhaps the most representative

example, combining fields of various types and a special success method performed on return.

The HarvestForm is defined in `forms.py` and used inside the FormView in `views.py`. Inside the class definition an extra method was added, called `create_harvest`, which is not an override of any parent method, but instead contains instructions on performing the asynchronous job of creating the harvest. Otherwise, the rest of the Form definition consists of field declarations with their corresponding types:

- `hashtag` as type `CharField`, the hashtag used for searching tweets on Twitter API
- `start_date` as type `DateField`, the beginning date for filtering tweets
- `end_date` similarly, represents the end date for tweet filtering

HarvestView inherits from Django's FormView and overwrites some fields and methods, which are somewhat instructive for understanding the other customised Forms and FormViews defined inside the application's codebase:

- template name: path of the HTML template to render
- form class: the HarvestForm we defined in `forms.py` and imported here
- success url: where the view should redirect upon form submission and success
- get context data(): defines extra variables to be passed to the templating engine. In our case, it is the list of available harvests.
- form valid(): extra validation and/or steps to be performed in case of success

In regards of the classic `View` generic, its only overwritten method was `get()` which instructs the view on the course it should consider when receiving a GET-type HTML request.

5.7.3 List comprehensions and other Python specifics

Python is a language of simplicity, but it is also a language difficult to understand from the outside. Its focus on simplicity starts from the building blocks: no visibility modifiers, easy import management, indentation-based execution blocks. However, some simplifications may seem obscure.

One of such shorthand becoming cipher is the declaration of arrays and lists, as in:

```
tweet_texts = []
tweet_users = {}
```

This snippet first declares an array called tweet_texts, while the second, tweet_users, is a dictionary. To complicate things even worse, tuples are declared using round braces. This is in opposition to the generally verbose approach of other programming languages and is important to note.

On a related note, list and dictionary comprehensions are an efficient way of creating or filtering a list or dictionary with as less code written as possible. Code such as:

```
[ ( x.uuid, x.name) for x in all_harvests ]
```

may be perceived as ambiguous for unfamiliar users. However for Python developers it becomes clear that this snippet produces a list of tuples from the iterable `all_harvests`, with tuples containing the iterable element's uuid and name. In English, what this does it that it filters out some unnecessary properties in all harvests and transforms the list of objects into a simple list of pairs(uuid, name).

Summary

In this section the details of implementation inside ATHENA were explained. The choice of Python and Django as programming language and frameowrk respectively were motivated by their large-scale use in developing scientific applications and web applications in general. The list of third party libraries was covered, with focus on particular libraries and their usage throughout the application.

The persistence model was chosen to be non-relational and scalable, in order to support scaling the system to a larger number of nodes and eventually moving towards a distributed web application. The task of fetching Twitter statuses was handled using asynchronous job running, using the Producer-Consumer architecture, via the Celery library with Redis as a message broker. This approach ensures the user doesn't have to wait for a job to finish before continuing to use the application independently. Inside the job, rate limits and concrete Twitter API communication were handled using Tweepy, a Python wrapper. The resulting, grouped information is called a Harvest.

The Enhancement step displays numerical data, a harvest's related hashtags and uses the acclaimed Sci-kit library to perform clustering on domain hashtags, using the K-Means algorithm. Another issue ATHENA tackles is the fitting of the post numbers dataset to a power function, investigating on outliers such as pages, bots and fake accounts producing significally more content than regular users.

The Normalisation step flattens the data set in a "one post per user, one day" fashion, with other normalisation options currently in development. The Analysis step uses the data flattened, filtered and stored by Normalisation to run a comparison between two harvests. Such data includes vocabulary commonalities and polarisation analysis.

The final part of this section approaches Python particularities which are used in the application but might seem unfamiliar to developers coming from different backgrounds, such as Java or PHP, where coding is usually more verbose.

Chapter 6

Testing and Validation

About 5% of the paper

6.1 Title

6.2 Other title

Chapter 7

User's manual

The following Chapter is a short description of the installation process needed to run the ATHENA app. The instructions are written for Mac OSX El Capitan, however most of the instructions work on other *nix systems as well, i.e. flavours of Linux including Ubuntu.

Prerequisites

For the installation and running of this project, you will need:

- XCode, the Mac toolkit for software development. It is generally a prerequisite for installing various programmming languages.
- Python (installed via the native package manager or using `brew`
- Pip, Python's installer application
- optional (but recommended) VirtualEnv, a wrapper to permit different Python and Django versions on the same host platform, without the need for virtual machines.

Getting the codebase

The project's codebase can be fetched from the CD attached to this physical thesis or via GitHub. For the GitHub version, run the command:

```
git clone https://github.com/calina-c/athena.git
```

Either option you chose, you should now have in your directory of choice a file structure containing: 2 folders: `athena` and `athena_app` and a few other files including:

- a `README.md` describing part of these installation steps
- a hidden `.gitignore` file used by `git` in order to know which files to upload to the GitHub servers

- the `manage.py` file which acts as a Django command line utility
- the `requirements.txt` file listing the third party libraries needed to be installed

and .

Installing dependencies

Install `redis` and `cassandra` on your systems per the recommended installation for your Operating System. Run `redis-server` and `cassandra` in separate terminal tabs. They need to stay up for the time the application is used.

As previously stated, the list of Python dependencies needed is listed in `requirements.txt`. Considering the `pip` installer utility is installed on your system or virtualenv, run the following command:

```
pip install -r requirements.txt
```

This command should fetch and install any necessary third-party Python libraries and prompt you if any errors or incompatibilities exist, as well as suggest solutions in these cases.

Creating the database structure

Considering your Cassandra server is already running, you can run the `cqlsh` command to enter the Cassandra console. From there, create the database structure by running the queries (also listed in README.md):

```
CREATE TABLE tweet (    twitterId text,    user text,    content text,
        date timestamp,    retweets int,    likes int, hashtags list<text
    >, history uuid, PRIMARY KEY (twitterId));

CREATE TABLE harvest ( uuid uuid, start_date timestamp, end_date
    timestamp, hashtag text, done boolean, PRIMARY KEY(uuid));

CREATE TABLE normal (uuid uuid, name text, content text, PRIMARY KEY(
    uuid));
```

Running the application

To run the application locally, run the command:

```
python manage.py runserver
```

from the root folder of the application, where the `manage.py` file resides. Then, start your modern browser of choice (Chrome, Safari or Firefox) and use the localhost URL to access the web application:

`http://localhost:8000/app/`

Use the navbar on the left to select a pipeline step from the application. If you need asynchronous jobs, start another process in a separate terminal for Celery, using:

```
celery -A athena worker -l info
```

The Celery worker also needs to run in the background constantly, in order to properly be able to function as a Consumer for asynchronous jobs, as described in previous Chapters.

Chapter 8

Conclusions

About. 5% of the whole
Here your write:

- a summary of your contributions/achievements,
- a critical analysis of the achieved results,
- a description of the possibilities of improving/further development.

8.1 Title

8.2 Other title

Bibliography

[1] N. V. Spirin, J. He, M. Develin, K. G. Karahalios, and M. Boucher, "People search within an online social network: Large scale analysis of facebook graph search query logs," in *Proceedings of the 23rd acm international conference on conference on information and knowledge management.* ACM, 2014, pp. 1009–1018.

[2] A. Java, X. Song, T. Finin, and B. Tseng, "Why we twitter: understanding microblogging usage and communities," in *Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 workshop on Web mining and social network analysis.* ACM, 2007, pp. 56–65.

[3] E. Constantinides, C. L. Romero, and M. A. G. Boria, "Social media: a new frontier for retailers?" in *European Retail Research.* Springer, 2008, pp. 1–28.

[4] C.-M. Tung and H.-Y. Wu, "An internet-connected world: Google's platform strategies to network industry." in *Intelligent Environments (Workshops)*, 2013, pp. 201–213.

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

[6] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky, "The stanford corenlp natural language processing toolkit." in *ACL (System Demonstrations)*, 2014, pp. 55–60.

[7] P. D. Turney, "Thumbs up or thumbs down?: semantic orientation applied to unsupervised classification of reviews," in *Proceedings of the 40th annual meeting on association for computational linguistics.* Association for Computational Linguistics, 2002, pp. 417–424.

[8] M. Steinbach, G. Karypis, V. Kumar *et al.*, "A comparison of document clustering techniques," in *KDD workshop on text mining*, vol. 400, no. 1. Boston, 2000, pp. 525–526.

[9] M. E. Celebi, H. A. Kingravi, and P. A. Vela, "A comparative study of efficient initialization methods for the k-means clustering algorithm," *Expert Systems with Applications*, vol. 40, no. 1, pp. 200–210, 2013.

[10] N. N. Taleb, *The black swan: The impact of the highly improbable.* Random house, 2007.

[11] A. Clauset, C. R. Shalizi, and M. E. Newman, “Power-law distributions in empirical data,” *SIAM review*, vol. 51, no. 4, pp. 661–703, 2009.

[12] L. Muchnik, S. Pei, L. C. Parra, S. D. Reis, J. S. Andrade Jr, S. Havlin, and H. A. Makse, “Origins of power-law degree distribution in the heterogeneity of human activity in social networks,” *arXiv preprint arXiv:1304.4523*, 2013.

[13] “TIOBE index May 2016,” http://www.tiobe.com/tiobe_index?page=index, 2016, [Online; accessed 5-June-2008].

[14] K. J. Millman and M. Aivazis, “Python for scientists and engineers,” *Computing in Science & Engineering*, vol. 13, no. 2, pp. 9–12, 2011.

[15] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, “Scikit-learn: Machine learning in python,” *The Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

Appendix A
Relevant code

```
 /** Maps are easy to use in Scala. */
object Maps {
  val colors = Map("red" -> 0xFF0000,
                   "turquoise" -> 0x00FFFF,
                   "black" -> 0x000000,
                   "orange" -> 0xFF8040,
                   "brown" -> 0x804000)
  def main(args: Array[String]) {
    for (name <- args) println(
      colors.get(name) match {
        case Some(code) =>
          name + " has code: " + code
        case None =>
          "Unknown color: " + name
      }
    )
  }
}
```

Appendix B

Other relevant information (demonstrations, etc.)

Appendix C

Published papers