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,
Prof. dr. eng. Liviu MICLEA

HEAD OF DEPARTMENT,
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

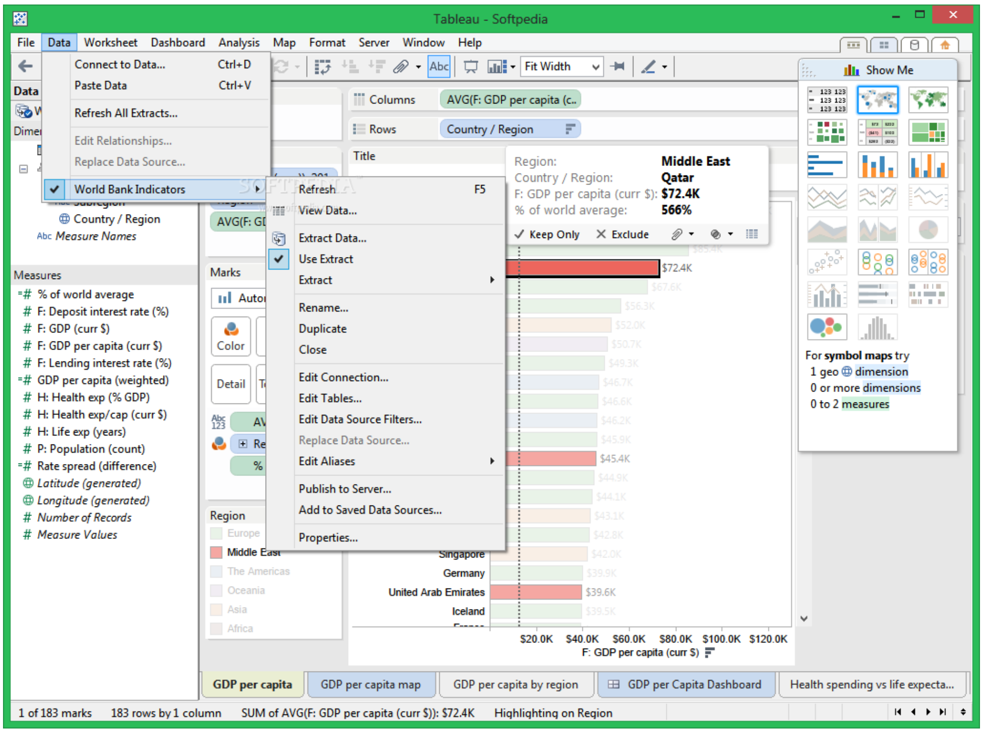
The Computer Science field is very much branched nowadays in clearly-delimited subjects. But it is such a case where the large-scale processing capacities of data analytics should merge with approaches for computer-assisted business decisions. Consider that data analytics software often requires extensive 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



Region: Middle East
Country / Region: Qatar
F: GDP per capita (curr $): $72.4K
% of world average: 566%

1 of 183 marks 183 rows by 1 column SUM of AVG(F: GDP per capita (curr $)): $72.4K Highlighting on Region

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

TODO

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.

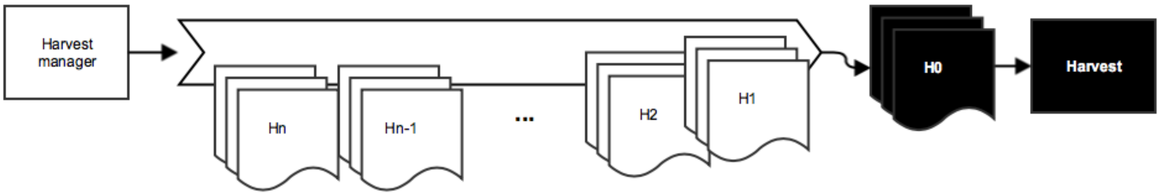


Figure 2.1: Harvesting pipeline

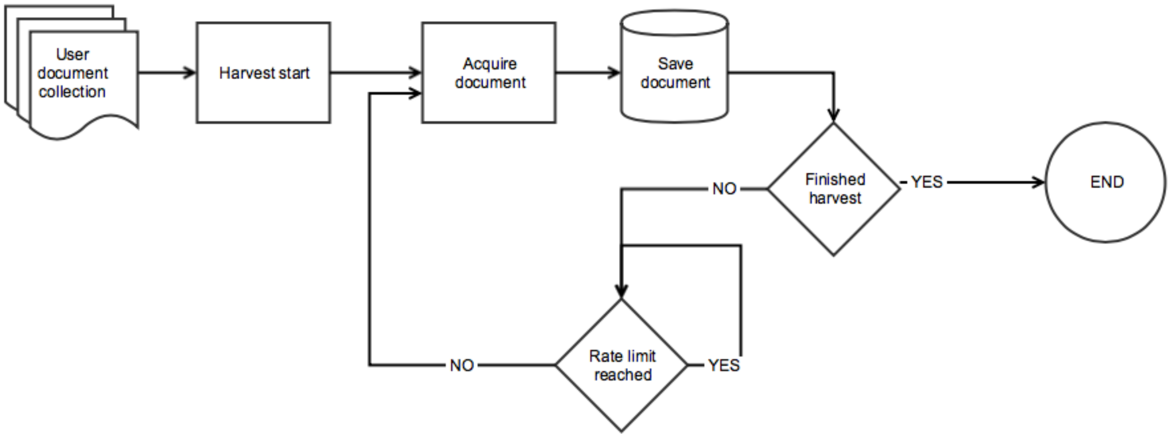


Figure 2.2: Harvesting job sub-pipeline

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 the Analysis step.

User contribution modeling

TODO

2.2.4 Normalisation and analysis

TODO

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

Bibliographic research has as an objective the establishment of the references for the project, within the project domain/thematic. While writing this chapter (in general the whole document), the author will consider the knowledge accumulated from several dedicated disciplines in the second semester, 4^{th} year (Project Elaboration Methodology, etc.), and other disciplines that are relevant to the project theme.

Represents about 15% of the paper.

Each reference must be cited within the document text, see example below (depending on the project theme, the presentation of a method/application can vary).

This section includes citations for conferences or workshop [?], journals [?], and books [?].

In paper [?] the authors present a detection system for moving obstacles based on stereovision and ego motion estimation. The method is ... *discus the algorithms, data structures, functionality, specific aspects related to the project theme, etc....* Discussion: *pros and cons.*

In chapter 4 of [?], the *similar-to-my-project-theme algorithm* is presented, with the following features ...

3.1 Title

3.2 Other title

Chapter 4

Analysis and Theoretical Foundation

TODO: Producer consumer architecture

TODO: Plfit theory and [1]

TODO: supervised vs unsupervised TODO: kmeans TODO: Talk about Kmeans++ algorithm

Together with the next chapter takes about 60% of the whole paper

The purpose of this chapter is to explain the operating principles of the implemented application. Here you write about your solution from a theory standpoint - i.e. you explain it and you demonstrate its theoretical properties/value, e.g.:

- used or proposed algorithms
- used protocols
- abstract models
- logic explanations/arguments concerning the chosen solution
- logic and functional structure of the application, etc.

YOU DO NOT write about implementation.

YOU DO NOT copy/paste info on technologies from various sources and others alike, which do not pertain to your project.

4.1 Title

4.2 Other title

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 [2]. 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 [3], 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.

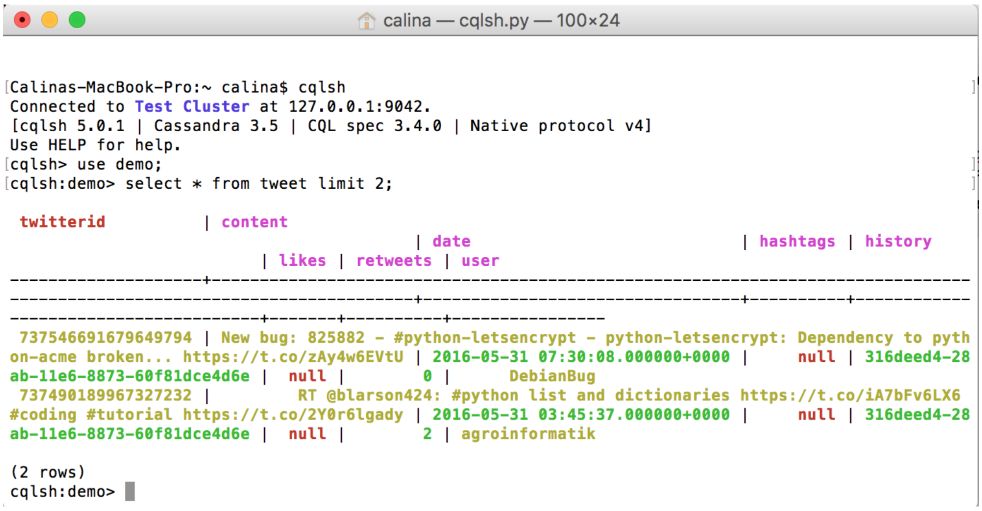
A Cassandra server was installed locally and needs to be up for all queries run from the aplication. The Python library `cassandra-driver` allows for easy connection to Cassandra clusters and execution of queries, similarly to SQL ones. Empyrical debugging is easy via the `cqlsh` command line utility, 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.

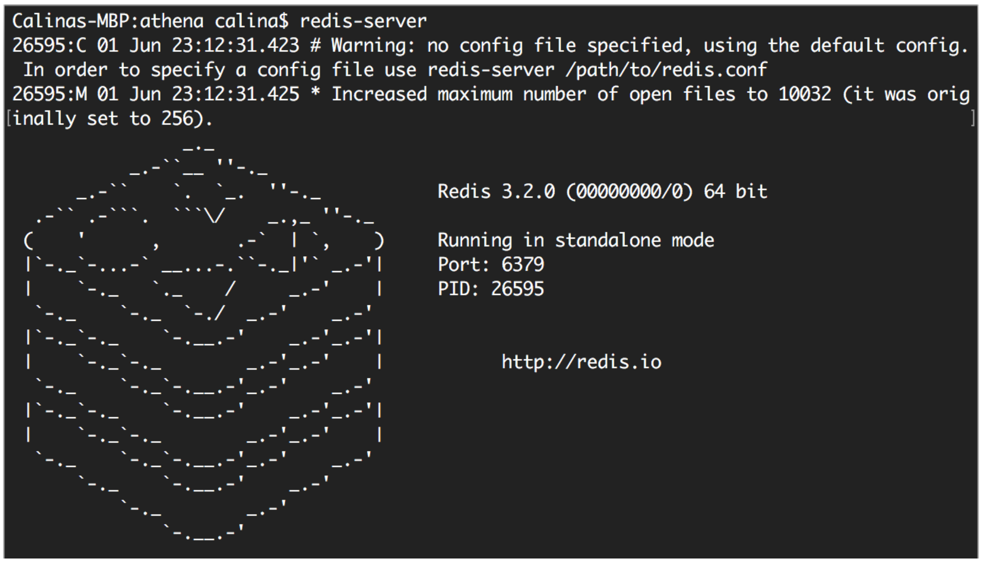
Installing Redis is straightforward using a downloaded package and even some available package managers such as Mac's `brew`. After the installation is complete, the Redis server can be fired up using the command `redis-server`. A splash screen with Redis' logo

[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 [1] 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. 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 [4] 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/

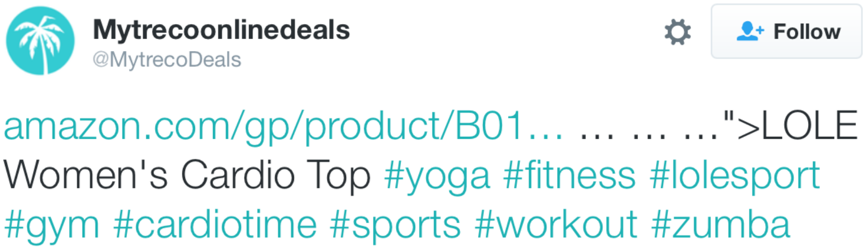
- Preprocessing

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.

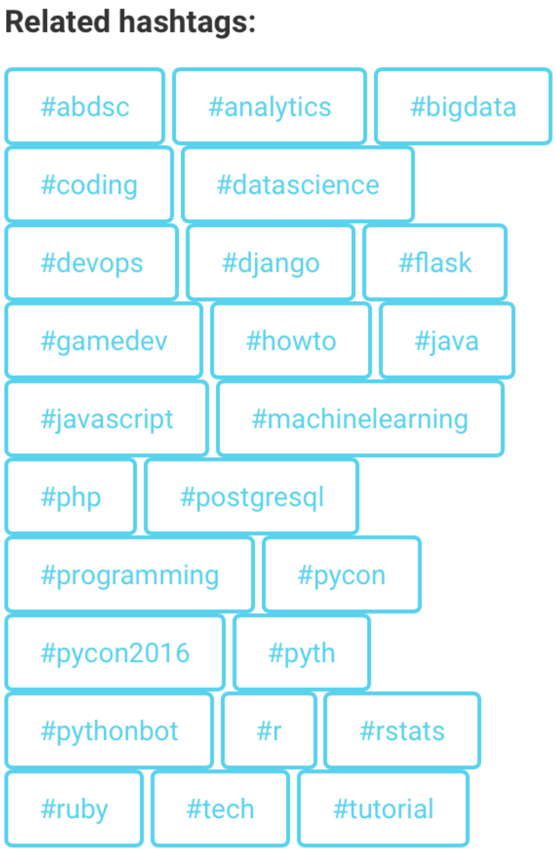


Figure 5.4: Example of related hashtags in a #Python harvest over one day

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, 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.

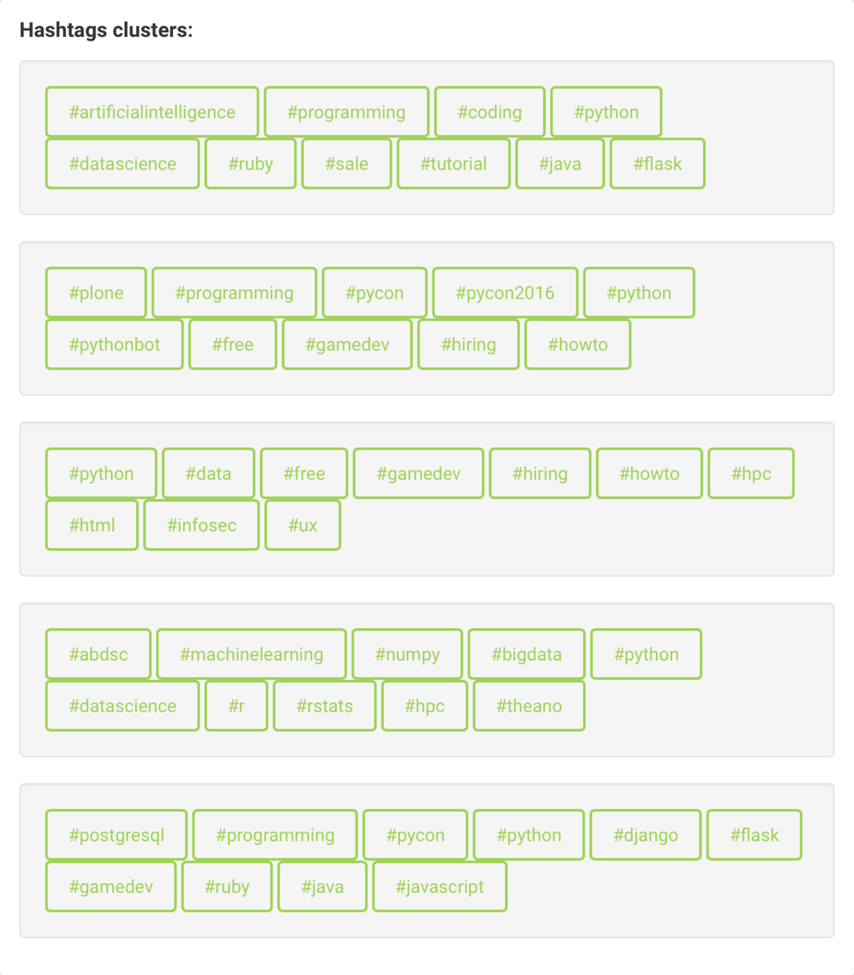
```
order_centroids = model.cluster_centers_.argsort()[:, ::-1]
terms = vectorizer.get_feature_names()
clusters = {}

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



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

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

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

5. the fifth cluster is a collection of technologies related to Python or generally used alognside 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

TODO

5.7 Code organisation

TODO until page 44

Chapter 6

Testing and Validation

About 5% of the paper

6.1 Title

6.2 Other title

Chapter 7

User's manual

In the installation description section your should detail the hardware and software resources needed for installing and running the application, and a step by step description of how your application can be deployed/installed. An administrator should be able to perform the installation/deployment based on your instructions.

In the user manual section you describe how to use the application from the point of view of a user with no inside technical information; this should be done with screen shots and a stepwise explanation of the interaction. Based on user's manual, a person should be able to use your product.

7.1 Title

7.2 Other title

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] 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.

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

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

[4] 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