



Table of contents

1	CONTEXT	4
1.1	PROJECT DESCRIPTION	4
1.2	PROJECT PLAN	5
1.3	HALF-WAY OBJECTIVES	5
1.4	HOW TO COLLABORATE ON A BIG DATA PROJECT?	6
2	DATA CHOICES	6
2.1	TWITTER API STRUCTURE	6
2.2	WHICH DATA DO WE NEED?	7
2.2.1	A FEW HINTS ABOUT OUR MACHINE LEARNING PLANS	7
2.2.2	DATA FILTERING PROCESS	7
2.2.3	WHICH DATA WE ACTUALLY TAKE	8
3	DATA CAPTURE	9
3.1	MONGODB	9
3.2	TIME CONSTRAINTS	10
4	TRANSFORMATION INTO A MACHINE LEARNING DATASET	11
4.1	WORD-BAGGING OF TWEETS	11
4.2	DATA CLEANING	13
4.3	DATA FEATURING	15
4.4	FINAL DATABASE DIAGRAM	16
5	MACHINE LEARNING – POLL PREDICTION	18
6	FINAL ESTIMATION OF THE US ELECTION SCORE	20
6.1	TEST PHASE: OUR TWEETER’S USER SAMPLE	20
6.2	ELECTION PREDICTION	21
6.3	POSTERIOR ANALYSIS	21
6.3.1	ANALYSIS OF THESE RESULTS	21
6.3.2	GEOGRAPHICAL ANALYSIS	21
7	TECHNICAL DIFFICULTIES	23

7.1	INITIAL BAG OF WORDS MUCH TOO LARGE	23
7.1.1	DESCRIPTION	23
7.1.2	SOLUTION	23
7.2	EXECUTION TIME	23
7.2.1	DESCRIPTION	23
7.2.2	SOLUTION	24
7.3	OTHERS	24
8	IMPLEMENTATION	26
8.1	DATA CAPTURE	26
8.2	FROM DATABASE TO DATASET	29
8.2.1	WORD-BAGGING AND CLEANING	29
8.2.2	DATA FEATURING	29
8.3	MACHINE LEARNING	29
8.4	ELECTION PREDICTION	29
8.4.1	VOTERS POLL SAMPLE	29
8.4.2	SAMPLE DATASET BUILDING	29
8.4.3	ELECTION ESTIMATION	29
9	CONCLUSIONS	29
9.1	REWIND IT!	29
9.2	PREDICTIONS CONCLUSIONS	31
10	BIBLIOGRAPHY	31

Note for the reader

Parts 1 to 4 were already handed-in at the halfway assignment. We still put it here again for a complete understanding of the project. We tried to make it exhaustive, that's why a lot of content is down here. Implementations are separated in part 9. We also advice you have a look at our GitHub repository [here](#) where all the pieces of code are available.

1 Context

2012 Obama's campaign has shown that big data can play a crucial role in US presidential elections. As a politician, listening to the people means analyzing data. And what better way of listening to people than using social networks? Twitter provides a great API that allows getting millions of tweets and a lot of information on users. Besides, 2016 US presidential campaign made Twitter the number one social media for talking about politics.

Improving polls accuracy can have two different meanings:

- Getting closer results to the actual votes of the people. Indeed, US presidential elections as well as the Brexit referendum appeared to be a massive failure for polls institutes.
- Getting more specific results about the kind of person who votes for one or the other candidates.

1.1 Project description

Our project will try to predict for which candidate one Tweeter user would have voted for, according to their tweets. The final goal is to build a social poll and estimating a result of the US Election.

To do so, we will first retrieve a lot of data from Twitter, and thus **constitute our own dataset from scratch**. Once this dataset built, we will work with machine learning models to predict one user's vote from the word-bagged text of his tweets. Finally, we will build a representative panel of Tweeter users and estimate the whole result.

Following this process, we will arrive to a complete dataset's size of **15 to 20 GB**, spread over our learning data and the poll sample.

This project does not claim to be able to better predict the output of the election (especially since the election has already happened) but tries to conduct some research on the possibility of using Twitter to get a better understanding people's opinions. Once this engine built, many different analysis about various topics could be run: which topics were the most shared on Twitter? Which topic Trump's and Hillary's voters evoke the most? Does the daily news have any influence on Twitter behaviour.

It could be useful both for a candidate to understand what subjects matter to people. It could also be useful to polls institute to conduct their survey and this study can be part of the survey methodology as an answer to the question: “What does the twittosphere think about this?”

As a purpose of this project we are going to make a meaning analysis on all tweets of a given sample of users, to estimate if they would be more likely to vote for Hillary Clinton or for Donald Trump. We are aware that there is a large probability that the estimation we get may be very different from the actual result of the presidential elections, but our output would be the vote according to Twitter random members. Hence the result may be biased towards voters enclined to new technologies, social medias, etc...

1.2 Project plan

Here is the detailed step-by-step plan we will go through:

- Dataset elaboration
 - Twitter data
 - Identification of **panels of Tweeter** users only voting for Trump or Clinton (our future training and test set)
 - Choice of the data we will need to achieve our predictions (cf. Twitter API information)
 - **Retrieval** of this data
 - Creation and filling of a **NoSQL MongoDB** database to store the retrieved data
 - Transformation into a **machine learning dataset**: data cleaning and featuring
- Poll predictions
 - Experimentation of **different machine learning models** on our data. Parameters optimization
 - Creation of a Tweeter sample of users and estimation of the US Election score
- Posterior analysis using our poll engine

1.3 Half-way objectives

Here are bolded our objectives for this midway hand-in:

- **Dataset elaboration**
 - **Twitter data**
 - Identification of panels of Tweeter users only voting for Trump or Clinton (our future training and test set)
 - Choice of the data we will need to achieve our predictions (cf. Twitter API information)
 - **Retrieval** of this data
 - **Creation and fulfilling of a NoSQL MongoDB database to store the retrieved data**
 - **Transformation into a machine learning dataset**

- Data cleaning
- **Word-bagging of tweets**
- Data featuring to reduce the number of features
- Poll predictions
 - Experimentation of different machine learning models on our data. Parameters optimization and adjustment
 - Creation of a Tweeter sample of users and estimation of the US Election score
- Possible posterior analysis using our poll engine

Among these steps, the non bolded bullet points are planned to be implemented for the final hand-in. These are the next points.

1.4 How to collaborate on a Big Data project?

As such a project requires a lot of data, group work, various pieces of implementations on several machines, the whole process needs some rigorous organization. To be able to modify common pieces at the same time and, we decided to work with a **GitHub Repository** to share our respective works easily and keep a track of the whole project.

2 Data choices

2.1 Twitter API Structure

Twitter API is very well documented and structured. We will here introduce the few elements we used:

- The class **users**, in which we can find all the information regarding one Tweeter subscriber: its description, its followers, its abonnements, its description, its tweets, etc...
- The class **tweets**, corresponding to one tweet issued anywhere of Tweeter. We can find as attributes the user which tweeted it, the text, the amount of likes and retweets, hashtags, URLs, and user mentions which are referenced as actual attributes in the class (the **#**'s or **@**'s utility!). The main thing is the **user_id** of the user who tweeted this tweet, extremely useful for us to link those the users and tweets entities

To be able to use Twitter API, we created an 'application' on Twitter Developers community, in order to have **keys and tokens** to access the data from Python. Moreover, **tweepy** Python library will be used to hit on Tweeter API. Here are the few lines used to initialize the process:

```
filename = "C:/token/access.json"
with open(filename) as file:
    token = json.load(file)

# Logging on the twitter API. This is mandatory to be able to use it
auth = tweepy.OAuthHandler(token["consumer_key"], token["consumer_secret"])
```

```
auth.set_access_token(token["access_key"], token["access_secret"])

# Main Twitter object to make the queries
# Note that the API also provides a sleep method if the Twitter Rate Limit is reached
# So if this limit is reached, the API is just going to wait until it is allowed to
load data again
api = tweepy.API(auth, wait_on_rate_limit=True)
```

The file `access.json` contains the API keys and tokens to connect to our profile. We put it in a file to avoid exposing them in the code, as well as being able to run the same code with different access tokens, on different machines.

More generally, any data available on a profile or on a tweet can be retrieved via this API. Technical aspects of this API are not very relevant to explain here, the documentation of Tweeter API is available [here](#). We'll detail the implementation later on.

2.2 Which data do we need?

Choosing the data from Twitter requires a lot of anticipation regarding the machine learning part. Indeed, data downloading and storage take a lot of time: we have to be sure that we have enough data to have a relevant learning process, but obviously the more data we have, the longer it takes to process it.

2.2.1 *A few hints about our machine learning plans*

We are here constituting our training set. Thus, our data needs to be complete, with all the features we need, but also **need to be labelled**. To be clear, we will train our algorithm on a bunch of Tweeter user knowing who they vote for. Hence, we have to download Tweeter user of a certain kind: Trump voters and Clinton voters. Let's detail how our user will be picked.

2.2.2 *Data filtering process*

To select Tweeter data, we will first consider all Donald Trump and Hillary Clinton followers. Hence we will randomly pick some of them (Tweeter allows us only to retrieve a random list of followers from one user, without filtering anything). Finally, we will match them to these filters before inserting them in our base:

- To be following only one candidate out of the two (for Trump's followers, not to be among Hillary's followers and vice versa): we base our whole machine learning process on this assessment: **somebody following Trump and not Clinton will vote for Trump**
- To have tweeted enough before (for the learning to be relevant): tweet number > 100
- To be English speaker

- *At the beginning we wanted to pick only American citizen. But this raises a lot of issues: first people don't necessarily fill the 'location' field of their profile. Then even if this field is completed, it is very complicated to link it to a country, unless the label country is exactly completed: e.g. locations can look like "the beach", "Kuala Lumpur", "New York, USA", "Chicao, Illinois", "Lyon", "my room" etc. To map these fields to an actual location on Earth is a little project itself. Finally, even if we make this mapping successful, finding enough users matching the filter "USA" would take too much time: we can't search by location, we can only pick random followers of Trump or Hillary and check if their location fits. Finally, even though those filters would be running, extraction time would blast and we wouldn't have enough data before the end of the project.*
- *Picking just English speakers is a way just to find meaning in tweets, for people which are biased towards Trump or Clinton. Even though they are not American, a meaning analysis of their tweets to discover whether they follow Donald Trump of Hillary Clinton still makes sense.*

This data will constitute our training data. However, we also need to build a sample set to evaluate our poll engine on, at the end of the project. Obviously, this sample set requires more filters to be accurate, and to match as much as possible the American population's behavior. Here are additional filters that we will apply for the sample set:

- Location: location of the user has to be filled up.
- Time zone US: As filtering locations is a hard task, we will filter on the time zone. Of course, users who didn't fill their time zone information won't be picked.

At the end of the process, we get back only English speakers from US, who filled up their location. Afterwards, we will try to narrow-down this sample by American state, if we have enough data. We will then be able to draw some geographical statistics about the election's prediction.

2.2.3 Which data we actually take

Once our user chosen and ready to be added to our database, here are the data we exactly retrieve about the user:

- ID: Tweeter references its users with unique ID. This ID will be extremely helpful to find that user's tweets.
- Location: if filled, it could be useful to draw geographic analysis afterwards
- Number of tweets: useless for the learning, but very interesting to qualify the user's activity intensity on tweeter: are its tweets really trustable?
- Number of followers: useful to weight one user's importance
- Description: The user's few lines self-introduction on his profile. Could also be useful afterwards
- Label: The most important data: is this user an exclusive follower of Donald Trump or Hillary Clinton. This is the target we will try to classify our data on.

Note: We didn't choose to keep user's tweets IDs (which are available in the class), because it would have been too much duplicated data which the Tweets base. Instead we will build an index on this `user_id` in the tweets collection (see MongoDB paragraph).

Next steps is to retrieve all the tweets of that user. To do so, we will use the user ID to look for its tweets. The function `api.user_timeline(user_id)` allows to retrieve the last tweets of a user, up to 3200. Implementation details will be shown afterwards.

Hence, we retrieve the more tweets we can from that user ID, with the following data:

- Tweet ID: Tweeter references its tweets with unique IDs
- User ID: Will be crucial to find one's tweets in our database
- Text: The text message of that tweet
- Hashtags: The hashtags mentioned in the tweet. Useless for our learning, but maybe useful for posterior analysis
- Date: By retrieving the date at which the tweet was posted, we could run some time analysis afterwards, for example by correlating tweets with current news

Among all these users and tweets data, some of it may be useful for posterior analysis but not for our learning. We still decided to keep them stored, to open great opportunities for interesting insights.

3 Data capture

Let's first note that we captured the data at the right moment: we retrieved our tweets and users very close to the date of the election. Indeed, tweets that we collected were mostly posted before the election, and are then very strongly linked to US politics. Our machine learning phase, without assuming about its success or not, will at least be based on strong and consistent training data.

3.1 MongoDB

This part aims at explaining how the data is structured. The two basic components are Users and Tweets.

However, **we want to strongly distinguish the data we use for our training from the data we use to make the actual poll**. Indeed, the data we use for the training does not filter by country (otherwise it would take too much time to retrieve all the data) and as a result a lot of users are not from US. The actual poll can be accurate even with less data. Indeed, to have 95% probability that the result lies between minus or plus 1% from what we get, we just need 9000 users (given by the calculator www.surveysystem).

So we are going to have four collections:

Users_Training	Tweets_Training	Users_Sample	Tweets_Sample
<ul style="list-style-type: none"> • <code>_id</code> • <code>description</code> • <code>label</code> • <code>location</code> • <code>nb_followers</code> • <code>nb_tweets</code> 	<ul style="list-style-type: none"> • <code>_id</code> • <code>user_id</code> • <code>date</code> • <code>hashtag</code> • <code>message</code> 	<ul style="list-style-type: none"> • <code>_id</code> • <code>description</code> • <code>location</code> • <code>nb_followers</code> • <code>nb_tweets</code> 	<ul style="list-style-type: none"> • <code>_id</code> • <code>user_id</code> • <code>date</code> • <code>hashtag</code> • <code>message</code>

Figure 1: MongoDB collections

Here note that setting the id of a tweet or a user as '`_id`' allows to avoid duplicates. Then we can simply insert a document in one of the collections by doing: `db.users.insert_one(user)`

We have acquired around 15GB of data for the training (Around 70M tweets and 60k users) and 5000 users for the sample. The 5000 sample users have taken as much time to download as the 60k users because the additional conditions.

3.2 Time constraints

This part is one of the trickiest part of our project. It turned out that it was not possible to collect users only in the US fast enough (We would still be loading the data). So to improve our data collection, we made two choices:

- We do not filter by country on the training set
- We parallelize the tasks between the different computers we can use.

We can actually use 3 computers. Hence two computers are devoted to get respectively Donald Trump's voters and Hillary Clinton's voters, and the last one is devoted to acquire the sample. The first two databases have to be merged together then, with this command:

```
mongodump --db mydb --out 'PATH'
mongorestore --db mydb 'PATH'
```

It is also possible to use the commands `mongoexport` and `mongoimport`.

4 Transformation into a machine learning dataset

4.1 Word-bagging of tweets

At this step, our MongoDB is full of well-structured data. We need now to **move from a simple database to a learning dataset**. For each user, we will use all its tweets to predict his label (voting Trump or Clinton). To do so, we will build a **bag of words representation** of the concatenation of all its tweets. In our global bag-of-words, one line represents one user (the text concatenation of all his tweets). This bag-of-words will be, afterwards, passed in our machine learning algorithm. We need to focus a lot on having a final dataset very clean, consistent, and representative of our raw data.

As MongoDB includes a Map Reduce engine, we decided to build our bag of words as a collection of our database, generated through Map Reduce process. This has one main advantage: our data doesn't go through Python live memory. Such an amount of data would have been very heavy to handle this way. We hence win a lot of efficiency by processing the data directly inside Mongo. Moreover, it allows us to learn a new skill with Mongo's MapReduce which works with JavaScript.

Counting words through a MapReduce process has been made in Week 7, but not with the MongoDB version of it. First, the MapReduce algorithm in itself is quite different, given the special structure of our data (tweets are separated in the database, but a lot of them will count for the same line (1 line = 1 user) in the database. Below, the Map Reduce algorithm.

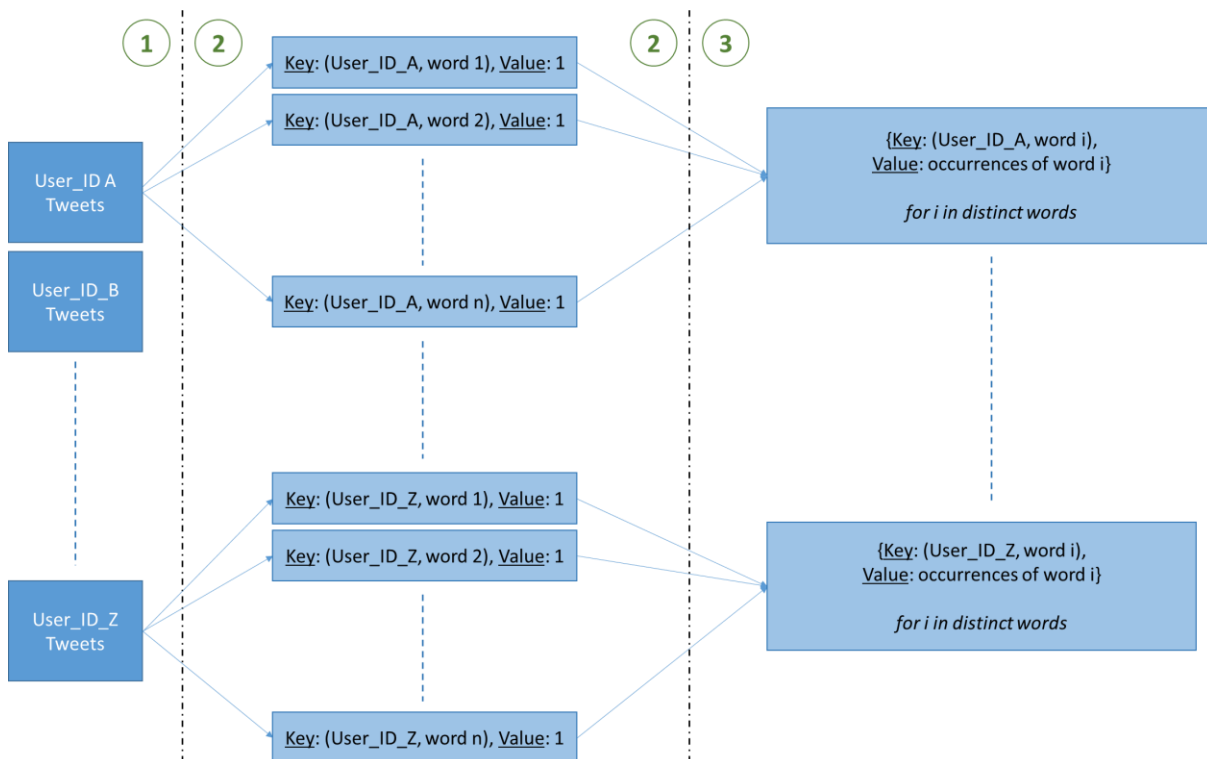


Figure 2: MapReduce structure

A few explanations about the MapReduce structure above:

- Map step 1 -> 2
 - We start from the text of each user (the concatenation of its tweets). We split this text, lower-case and remove punctuation, and then map, for every word in the splitted set, the couple (user_id, word) as a key, and 1 as a value
- Reduce step 2 -> 3:
 - The simple reduce part that we know: summing up all the values from the same key. This will simply end to count the occurrences of each word, by each user

Note: In practice, the data we start from is the Tweets collection. Hence we have separated tweets. By applying the same algorithm to the whole collection, we obtain exactly the same results, as tweets issued by the same user we will still end up summing up each other at the end. We didn't represent this version for more clarity here.

The implementation with Mongo framework and PyMongo [can be found here](#).

This is the critical operation of our whole data engineering process as it is very heavy to manipulate and time consuming. As soon as the data is represented as a bag of words, it's much lighter and easy to play with. You can found here a screenshot showing outputs of this operation.

```
> db.words_occurences_by_user.find()
{ "_id" : { "user_id" : 4521, "word" : "" }, "value" : 316 }
{ "_id" : { "user_id" : 4521, "word" : "a" }, "value" : 355 }
{ "_id" : { "user_id" : 4521, "word" : "abc" }, "value" : 2 }
{ "_id" : { "user_id" : 4521, "word" : "abed" }, "value" : 1 }
{ "_id" : { "user_id" : 4521, "word" : "able" }, "value" : 2 }
{ "_id" : { "user_id" : 4521, "word" : "about" }, "value" : 48 }
{ "_id" : { "user_id" : 4521, "word" : "abq" }, "value" : 1 }
{ "_id" : { "user_id" : 4521, "word" : "abraham" }, "value" : 1 }
{ "_id" : { "user_id" : 4521, "word" : "absence" }, "value" : 1 }
{ "_id" : { "user_id" : 4521, "word" : "absolute" }, "value" : 1 }
{ "_id" : { "user_id" : 4521, "word" : "absorb" }, "value" : 1 }
{ "_id" : { "user_id" : 4521, "word" : "abuse" }, "value" : 1 }
{ "_id" : { "user_id" : 4521, "word" : "ac" }, "value" : 1 }
{ "_id" : { "user_id" : 4521, "word" : "academyawards" }, "value" : 1 }
{ "_id" : { "user_id" : 4521, "word" : "accept" }, "value" : 3 }
{ "_id" : { "user_id" : 4521, "word" : "accessing" }, "value" : 2 }
{ "_id" : { "user_id" : 4521, "word" : "accident" }, "value" : 1 }
{ "_id" : { "user_id" : 4521, "word" : "accidentally" }, "value" : 2 }
{ "_id" : { "user_id" : 4521, "word" : "accomplished" }, "value" : 1 }
{ "_id" : { "user_id" : 4521, "word" : "according" }, "value" : 2 }
```

*Figure 3:
MapReduce
output*

This first version of our bag of words is way too large (much more words than users) and contains around 200 millions documents. It won't be relevant for the machine learning part. We will overfit our data. To reduce its width, we will need to clean it.

4.2 Data cleaning

A quick overview on the data in the bag of words shows that a lot of words are either proper names, unknown words, misspelled words, smileys, or just irrelevant strings with special characters etc. We will then need to clean this data.

To clean this data, we first thought about using an English dictionary (a file of 350k English words) to separate right words from misspelled ones. We would have then added the most used words of those tweets to still learn from words like “Trump”, “Clinton”, “Obama”, “Obamacare”, etc...

We tried to launch our Machine Learning algorithms with this cleaning process, but we realized that it was not appropriate: too many words were not really meaningful (‘the’, ‘a’, etc).

Here is then the solution that we implemented:

We used a “stop words” list (that you will be able to look at in our GitHub directory). These 670 words are all the English words that are used to structure the language, like articles, pronouns, adverbs, etc. We will eliminate those words from our text.

Then, we will just count occurrences of distinct words, and take the top 5,000. Our bag of words will be built just among those words, and not the others.

Here is a summary of our successive steps:

- Construct a raw bag of words of all the tweets
- Count the occurrences of all the words in the tweets, regardless of the user. This result will also be stored as a collection *all_words_count* in Mongo. This will also be implemented in MapReduce.
- Remove from this list all the ‘stop words’
- Sort the word list by occurrences
- Take the 5,000 most used words
- Sort the word list by occurrences

```
> db.stop_words.find()
{ "_id" : "a" }
{ "_id" : "able" }
{ "_id" : "about" }
{ "_id" : "above" }
{ "_id" : "abst" }
{ "_id" : "accordance" }
{ "_id" : "according" }
{ "_id" : "accordingly" }
{ "_id" : "across" }
{ "_id" : "act" }
{ "_id" : "actually" }
{ "_id" : "added" }
{ "_id" : "adj" }
{ "_id" : "affected" }
{ "_id" : "affecting" }
{ "_id" : "affects" }
{ "_id" : "after" }
{ "_id" : "afterwards" }
{ "_id" : "again" }
{ "_id" : "against" }
```

Figure 4: Stop words overview

- Build the new bag of words which counts occurrences of those 5,000 words in tweets

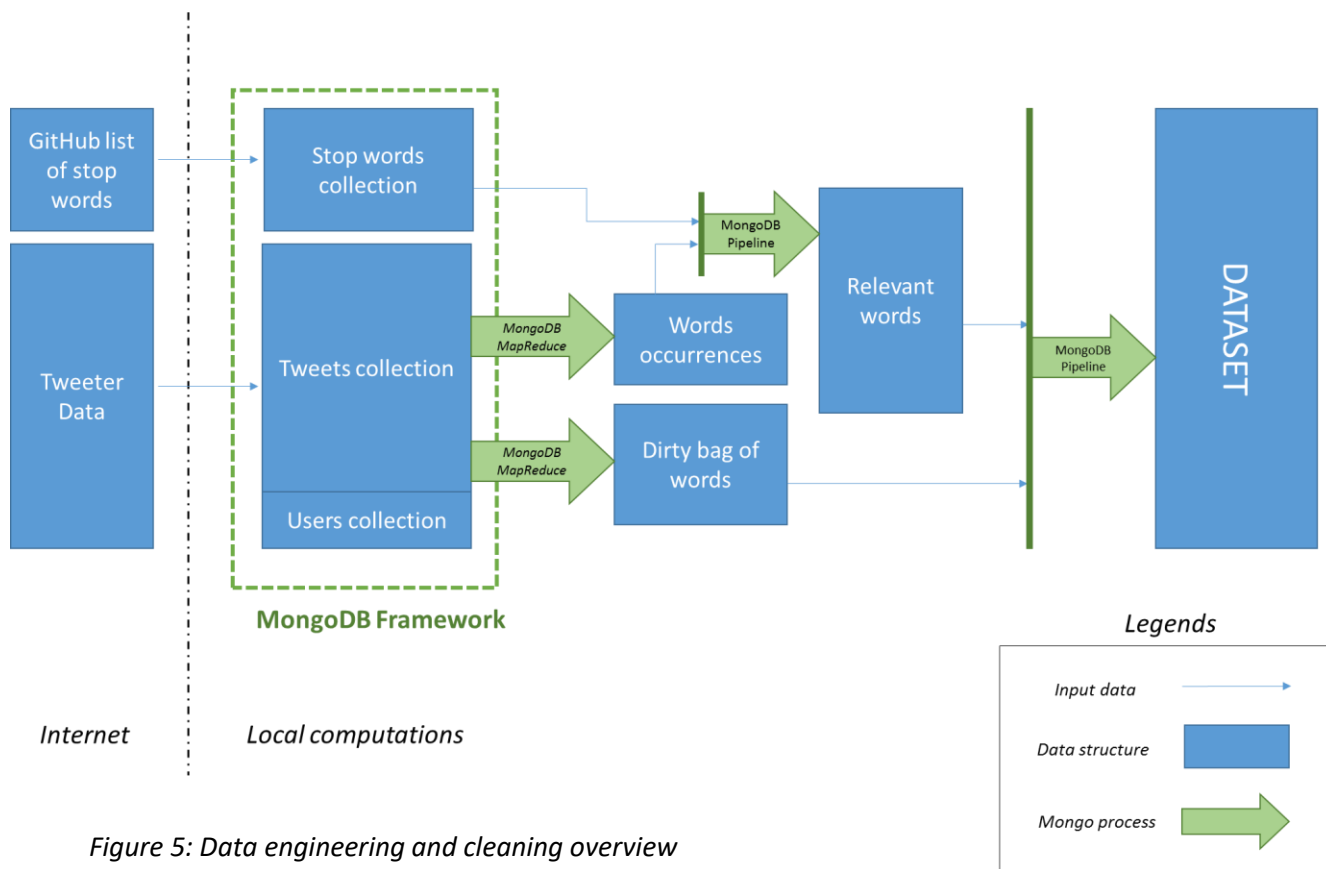


Figure 5: Data engineering and cleaning overview

Hence, **we isolate here the most used meaningful terms** out of our dozens of millions of tweets. It's very interesting to take a look at this list, which words are definitely meaningful. For instance, we can find 'Trump', 'Clinton', 'Obama', 'women', 'white' in the list, but also abbreviations like 'idk', 'btw', 'lol', etc. These pieces of English language are crucial as it's a meaningful part of our data. Here is an overview of this list:

```
> db.relevant_words.find()
{ "_id" : "dont", "value" : 2097287 }
{ "_id" : "love", "value" : 2067346 }
{ "_id" : "amp", "value" : 1889522 }
{ "_id" : "will", "value" : 1826738 }
{ "_id" : "day", "value" : 1690533 }
{ "_id" : "good", "value" : 1594297 }
{ "_id" : "time", "value" : 1522770 }
{ "_id" : "people", "value" : 1498710 }
{ "_id" : "today", "value" : 1389611 }
{ "_id" : "cant", "value" : 1163482 }
{ "_id" : "life", "value" : 1039077 }
{ "_id" : "happy", "value" : 1026954 }
{ "_id" : "great", "value" : 979163 }
{ "_id" : "going", "value" : 942609 }
{ "_id" : "best", "value" : 938945 }
{ "_id" : "lol", "value" : 846248 }
{ "_id" : "de", "value" : 749013 }
{ "_id" : "night", "value" : 733637 }
{ "_id" : "well", "value" : 704008 }
{ "_id" : "work", "value" : 703411 }
```

Figure 6: First words of our relevant_words list

The value is the occurrences of these words in all our database. This collection gathers the 5,000 most used words.

Let's insist of the fact that 'Trump', 'Clinton', 'women', 'white', 'Obamacare', and many other very meaningful words for our learning phase are in those 5,000 words.

Finally, [find here the implementation](#) of the MapReduce algorithm and various queries associated to this cleaning phase.

Obviously, with more than 15Go of data, all those operations are not straight-forward. [Here are a few hints about the way we handled the execution times.](#)

4.3 Data featuring

For now, we have a bag of words, with relevant and meaningful words that will influence the result. After a few machine learning tests, we realized that this was not enough and that we needed to improve the **representation of our data**.

We will then feature the data to enhance words that make it possible to discriminate the users with each other. For example, a very scarce word in the bag of words (very sparse columns) has to be enhanced as users using it are differentiating a lot from the others. Hence, we will use a featuring called **TF-IDF** (Term frequency – Inverse Document Frequency). The principle is the following: the more often a word appear compared to the total number of words the user has used, the more valuable it is for our representation (TF) and the more lines a word appears in, the less valuable that word is (IDF). We then enhance **only frequent and distinctive words**. Moreover, we balance the number of tweets that each user posted: a guy who posted 100 tweets is evaluated

the same way as another one with 3,000 tweets. This algorithm replaces the simple occurrence of words by the following value:

$$TFIDF_i = frequency_i * \log\left(\frac{N}{\text{number of lines containing } i}\right)$$

Where $frequency_i$ is the frequency of that word, that's to say its occurrences divided by the total number of words for this line. N is the total number of lines.

This representation gave much better accuracies for the machine learning part. This step will be implemented directly inside Python, after having loaded the bag of words from MongoDB. [Implementation details are here.](#)

Finally, we serialize our final objects to be able to get it back for the learning phase. The execution time is very long, we will then save a lot of time by saving locally our results.

4.4 Final database diagram

Our database is now our final dataset. We will apply our machine learning processes on it. As a summary, Figure 7 is the final structure of our Mongo dataset. We display the collections of the database 'Tweepoll'. Inside the boxes, the attributes of the collection:

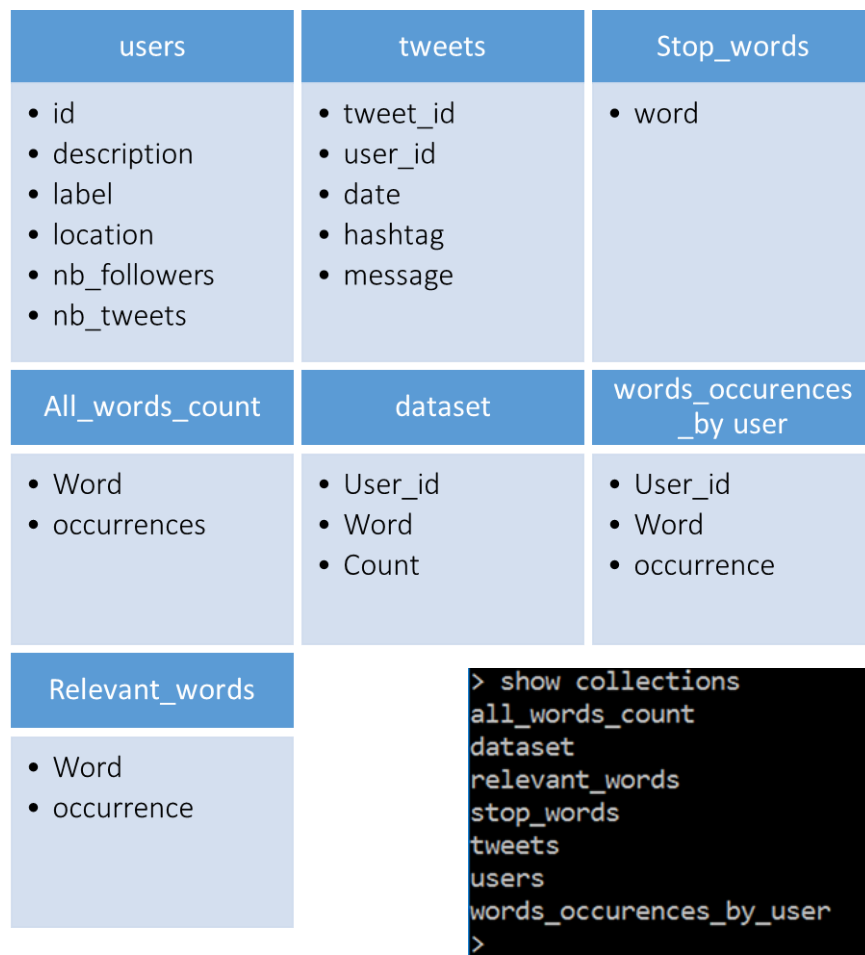


Figure 7: Final database diagram, aside with the MongoDB show collections query

5 Machine learning – Poll prediction

Our whole data engineering phase (data capture and creation of our dataset) was oriented to make this part as accurate as possible.

First, a few reminders after a lot of technical details.

- The final count of users that we could get at the end of the process was 33,860 potential voters (because of the filtering of data that has not been well downloaded). This is our training set, that means that we know for who they voted for: if they are following only Trump (and not Clinton) on Twitter, we assume that they voted for Trump. Same for Clinton.
- We will train a machine learning model to guess who they voted for, based on the content of their tweets. To do so, we will use a featured TF/IDF bag-of-words with selected words. For each line of this bag-of-words (1 user = 1 line), the associated text is the concatenation of all the tweets of this user.
- Once we find the right model (a lot of attempts before) with the right parameters, we will **predict** votes of an unknown sample of users. We will detail below how we picked those voters.

Let's have a few words about the challenges we faced to be able to apply the models to our dataset:

- We need to feed the *sklearn* classifiers with features and targets. **A crucial point** is that we need to create the targets accordingly to the dataset. There is an '**alignment**' task to do, that's to say associating to each line of the bag of words the right label (the vote of this particular user). This task will be outlined in the model implementations.
- More generally it's important to **keep the track of the user's id** in those operations, in order to run further analysis afterwards. For example, look at his location, his Twitter description, etc. We don't really know yet what we will need, but in order to keep all the doors open, we absolutely need to keep the track of these ids. This is not a complex task, but has to be thought in the implementation. Keeping track of the map words/columns may also be interesting for a future analysis.

[Here is the implementation of this model.](#)

	Wrong predictions	Right predictions	Error rate	Execution time
LDA	904	2234	71,19 %	580 s
Random Forest	1044	2094	66,73 %	37 s
Logistic Regression	921	2217	70,65 %	7 s
QDA	1010	2128	67.81%	578s

Figure 8: Machine learning error rates summary

Here is the output of our implementation:

```
----- Logistic Regression -----
0.706500956023
[[ 952  680]
 [ 241 1265]]
Success: 0.706501 out of 3138 points
Logistic Regression in 7.668461 s
----- LDA -----
[[1120  463]
 [ 441 1114]]
Success: 0.711918 out of 3138 points
LDA in  580.118840 s
----- QDA -----
[[1046  542]
 [ 468 1082]]
Success: 0.678139 out of 3138 points
QDA: 578.360345
----- RandomForest Classifier -----
0.667304015296
[[1162  410]
 [ 634  932]]
Success: 0.667304 out of 3138 points
Random Forest in 37.410005 s
```

Figure 9: Implementation output of the classifier

We can see our different models with their error rates, the execution time, and the confusion matrix. **We will keep LDA as our classifier**, as it has the best error rate on our training phase. Moreover, the confusion matrix is very symmetric, which is a very good sign (no bias for one candidate or the other). The logistic regression, although having a good error rates, is very biased towards the prediction 'Clinton'.

6 Final estimation of the US Election score

6.1 Test phase: our Tweeter's user sample

Regarding the results of our trainings, we will use the Linear Discriminant Analysis as the final estimator for our poll. We are now able to use this trained classifier on a sample base.

Note: Don't be confused between the test set in our training phase (10% of the data used to test if the classifier works well or not), and the sample data we will estimate the US Election results on.

To do so, the main challenge is: **how do we pick a good sample of American voters on Twitter?** Two mains methods are used when it comes to building a poll sample:

- Build a representative sample of the population on many criteria: age, occupation, social class, gender, origins, situation, etc. This method is very accurate, but is very difficult to set up in our case. Indeed, all that information is not always put in Tweeter profiles. For instance, guessing the age or the occupation of a Tweeter user could be a machine learning project in itself.
- Pick users randomly, such as they all have the same probability to be picked. **We will pick this method here**

To do so, we used the "Tweeter" profile. We are very likely to find different kind of people following this profile. Here is the process we followed to create our sample:

- Pick a random user from this "Tweeter" profile followers
- Check if he posted more than 100 tweets
- Check that his language is English
- Check that he fulfilled his location
- Check that its time zone includes 'US' (we will also grab Canadian people. This is an approximation we do to simplify our sampling)
- If everything is checked, add the user and all his tweets to the database
- If any of those condition is not fulfilled, pick another user

We could actually have done the same with other accounts like CNN, TED Talks, which would probably had changed the results we will obtain.

[Here is the implementation of this sampling process.](#)

Of course, this sample is not representative of US Society: Twitter's users age tends to young age ranges, and more generally this sample is made of active social network users, which is of course far from the real US population. But that is the mindset of our project: what if only Tweeter people had voted? We don't expect to get perfectly close to actual election results, but we can then estimate how far the tweetosphere is from the real world.

We need now to prepare our sample to apply our trained classifier. To do so, we will apply the transformations as the training set. We first calculate the raw bag of words *words_occurrences_by_user*.

Then **the process is a little bit different from the training phase**. We need to feed our classifier with exactly the same dataset structure, that's to say with the same words as the training phase. Hence, we will copy the *relevant_words* collection into the sample database, and extract from the raw bag of words the right columns, that's to say the columns matching the words used in the training. Furthermore, when retrieving the content from MongoDB to Python, we absolutely need to have exactly the same columns in the same order. Those implementations needed a few adjustments from the training implementation. As it's very close to the first version, we decided not to include it in this report, for more clarity.

We have now our final *dataset*, ready for the final estimation!

6.2 Election prediction

Here we are! All elements are gathered to predict the results:

- A poll sample
- A machine learning trained classifier
- The algorithms to create the dataset to give to the classifier

Here is the [global implementation of this phase](#).

By running our poll estimator on this sample, we obtain 2,252 voters for Hillary Clinton, and 2,435 voters for Donald Trump. This is an overall score of **48,05 %** for Hillary Clinton and **51,95 %** for Donald Trump.

6.3 Posterior analysis

6.3.1 Analysis of these results

These results are pretty close to the actual results that happened in the US Election. Indeed, a few weeks ago, according to the popular vote (that's to say the global estimation on all the states), Clinton got 47,70 % and Trump got 47,5 %. We didn't take into account the other candidates that represent the few percent's remaining in this estimation.

6.3.2 Geographical analysis

An important side of US Election analysis is geographical. Indeed, given the principle of "electors" of each state, the total amount of votes for one or the other candidates doesn't reflect the final result. Proof is that Hillary Clinton got more votes than Donald Trump when counting ballot papers. Yet, Trump got more 'electors'.

Let's try to include in our prediction this geographical aspect. The difficulty is that we don't have the same amount of users for every state. The predictions in states where few users are registered are of course not trustful.

The other **main difficulty is that we have to map the field 'location'** of our users, with actual USA states. This can be tough, as anyone can put anything in that field. From 'Chicago' to 'down on the beach', from 'Silicon Valley' to 'Delaware', those fields are far from being homogeneous.

Hence, **we built a mapping to link any location with one of the American states**. This mapping recognizes the name of a city or state in the field 'location', and associate the right state. To do so, we used an open dataset listing main cities of the USA with their state. Of course, this operation is not 100% efficient. We managed to identify almost 50% of the values.

Once this mapping done, we just counted the predictions for each state. Here are the results that we got, aside with the size of the sample for each state:

State	Clinton	Trump	Estimated winner	Real winner
California	52.48%	47.52%	Clinton	Clinton
New York	58.70%	41.30%	Clinton	Clinton
Texas	41.45%	58.55%	Trump	Trump
Florida	38.30%	61.70%	Trump	Trump
Illinois	52.50%	47.50%	Clinton	Clinton
Ohio	52.25%	47.75%	Clinton	Trump
Washington	62.11%	37.89%	Clinton	Clinton
Pennsylvania	48.24%	51.76%	Trump	Trump
Massachusetts	53.85%	46.15%	Clinton	Clinton
Kentucky	46.05%	53.95%	Trump	Trump
Colorado	43.10%	56.90%	Trump	Trump

Figure 10: Predictions by state

We managed to show extremely realistic results on our 10 most populated American states (these states are the ones in which we got the most results). We first wanted to do a whole estimation and estimating a realistic result with the 'electors' system. Though, we didn't have enough accurate data to perform this analysis.

Out of these 10 states, we predicted 9 right and 1 wrong. Moreover, 3 states are what is called 'swing states', that's to say where the results are crucial as they don't have a 'predetermined' wing: Florida, Pennsylvania and Ohio. Moreover, the scores that we got are pretty close from the actual scores that happened.

7 Technical difficulties

This part might be a little bit redundant with other parts of this report, but it looked crucial to us to **expose clearly all the issues we faced during this project**. This will help the reader getting aware of the technical depth we had to go through.

7.1 Initial bag of words much too large

7.1.1 *Description*

As Twitter's content is completely free and very diversified, the raw bag of words than we launched on more than 60 Millions of Tweets were much too large. We absolutely need to reduce this number of words, otherwise our models will largely overfit the data. An important part of our efforts on this second half was to run this reducing phase, which didn't appear this time-consuming at the very beginning.

7.1.2 *Solution*

Two steps to reduce our bag-of-words:

- **Cleaning the data:** a lot of words are irrelevant to us: wrong spellings, smileys, reactions (for instance 'aaaaaaaah' written with multiple occurrences of 'a'), etc. More generally, one English word could be found as a dozens of different versions. The successive steps to clean the data were:
 - o Isolating real English words: we found an English dictionary with all forms and tenses. We matched our words with this dictionary
 - o Finding the most used terms: except from real English words, a lot of proper names, expressions, hashtags, etc... were also found in the texts. To process this, we counted all the occurrences of all the words, whatever the user. Then, we isolated the 1,000 most used terms and added them to our dictionary. That way, words like "Clinton", "Obama", "Obamacare", "Trump" could be taken into account

Cleaning processes were implemented thanks to MapReduce algorithms. Details are available [here](#).

7.2 Execution time

7.2.1 *Description*

The execution time, either when collecting the data or when computing the data engineering tasks, were our main issue here. We focused on reducing this time and optimizing it in our implementation. Here are an overview of the most time consuming parts that we had to handle:

- Capturing data from Twitter
- Merging our two databases
- Creating data backups

- Running data transformation before the machine learning starts (bag of words, counting words)
- Training our machine learning algorithms

We also had to fight with Twitter's rate limit when capturing our data. Twitter allows us a finite amount of queries per 15 minutes, which definitely slowed down our progression.

7.2.2 *Solution*

Here are a few solutions we tried to implement to improve execution times:

- Avoiding large data transmission in Python kernel: we preferred, when it was possible, to work with Mongo engines (MapReduce, query aggregation, duplication, backups, etc) instead of Python data structures
- Working with a "fake" database much smaller in size, to try all our processes before. The various tasks were run only once in the main database
- Serializing our training dataset once built
- Creating Mongo indexes to find data by certain keys
- Parallelizing the tasks between our 3 PCs.
- Taking from Twitter just the very necessary data, and not more

Below, figure 10 is an overview of our timeline (both our programming steps, and our PCs schedule).

7.3 Others

- **Non homogeneousness of Twitter profiles.** Users don't fill their profiles the same way. For example, locations are not always labelled correctly. **See 6.3.2** for more details
- **Getting enough data fitting our filters:** given Twitter's rate limit and downloading times, it was difficult to get enough data while matching our requirements. We then had to widen our conditions and approximating our hypothesis.
- When implementing our **MapReduce algorithm with MongoDB**, we discovered that we needed to code our mapping and reducing functions in JavaScript. As we didn't know this language at all, it took us much more time than expected to run those instructions

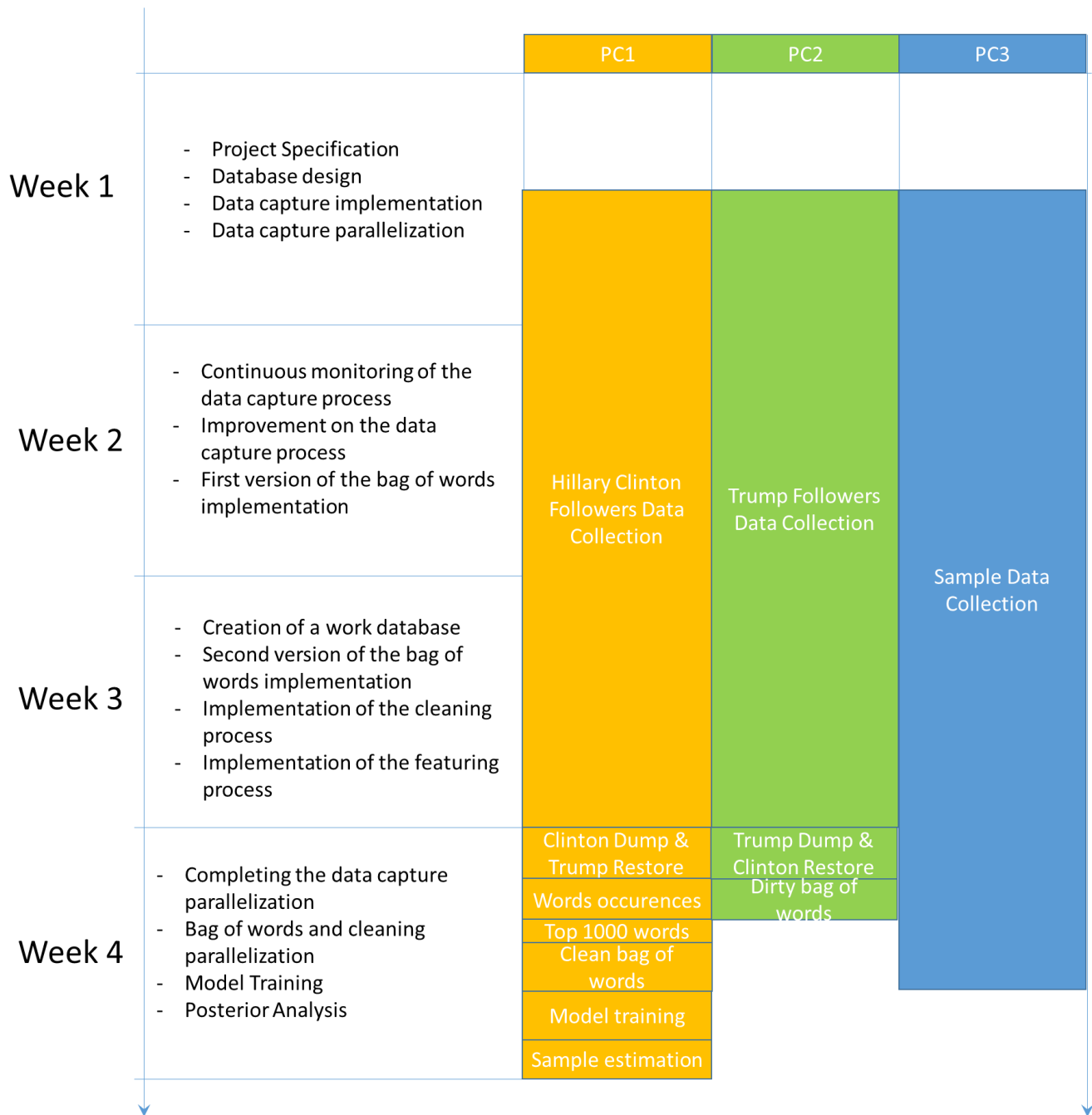


Figure 11: Project and execution timelines

8 Implementation

These pieces of code are also available directly in our GitHub directory [here](#).

8.1 Data capture

The purpose of this part is to summarize the code and help the reader understand it. Code is also [available here on GitHub](#).

There are actually **three pieces of code, each of them is run on a separate machine**. One piece gets **Hillary Clinton's voters**, another gets **Donald Trump's voters**, and the final one get **random users** only located in the US and who have indicated their state, in order to constitute the voters sample for the final estimation. The first two parts are exactly the same (Annexes 1) and the only thing to change from one to the other is to switch a Boolean from True to False. So we have decided to attach only one of them. The last one (Annexes 2) is very similar to the other but not exactly the same, and since it would be a bit confusing to only include the pieces of code which are different from the others, we chose to include it entirely.

Here are the different steps the program processes:

- Connecting to the twitter API
- Retrieving a bunch of users that follow the chosen candidate
- Checking if these users speaks English, has a minimum of 100 tweets and is a follower of only one of the two candidates (filters we mentioned earlier on)
- Inserting the relevant users and their tweets in the database

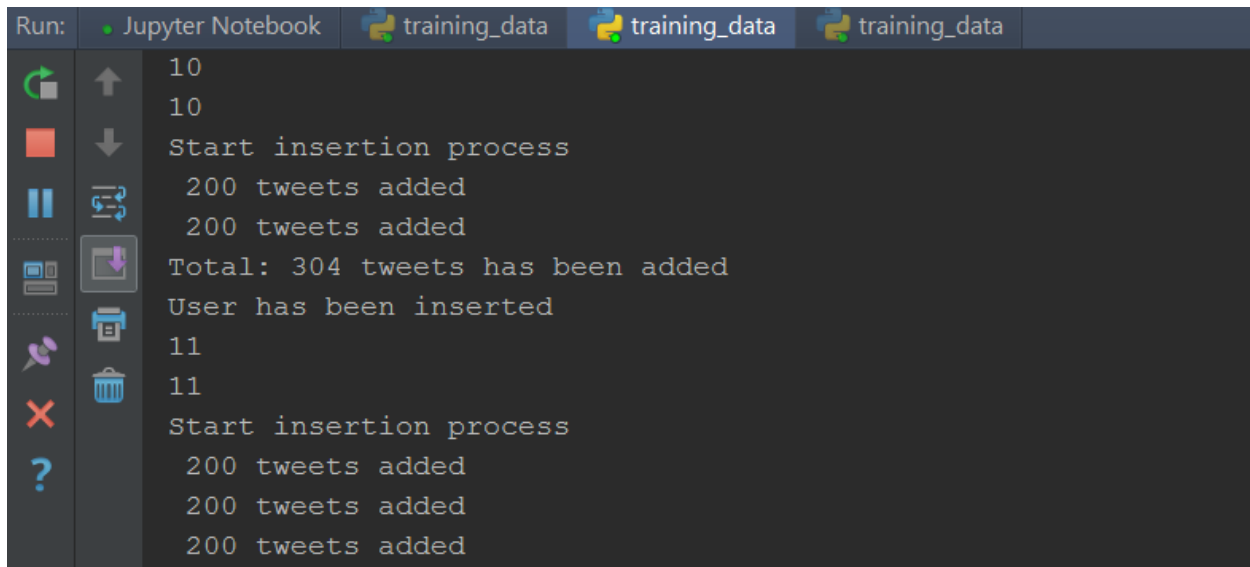
A crucial task in this program is to make sure that all errors are handled and that we can leave the computer running for hours.

Annexes 2's code takes people following the page "Twitter" as users and has more conditions on the users (Timezone must be US, and location must be filled).

Both Annexes contain the same functions:

- **Insert_tweet_information:** given one user, insert all his tweets (limited to 3200) in the database
- **Insert_user_information:** given one user, insert his profile in the database and call the function `insert_tweet_information`
- **Extract_information:** given a maximum number of people, call `insert_user_information` until this maximum is reached.

The output we get on the console is as follows for machine 1 & 2:



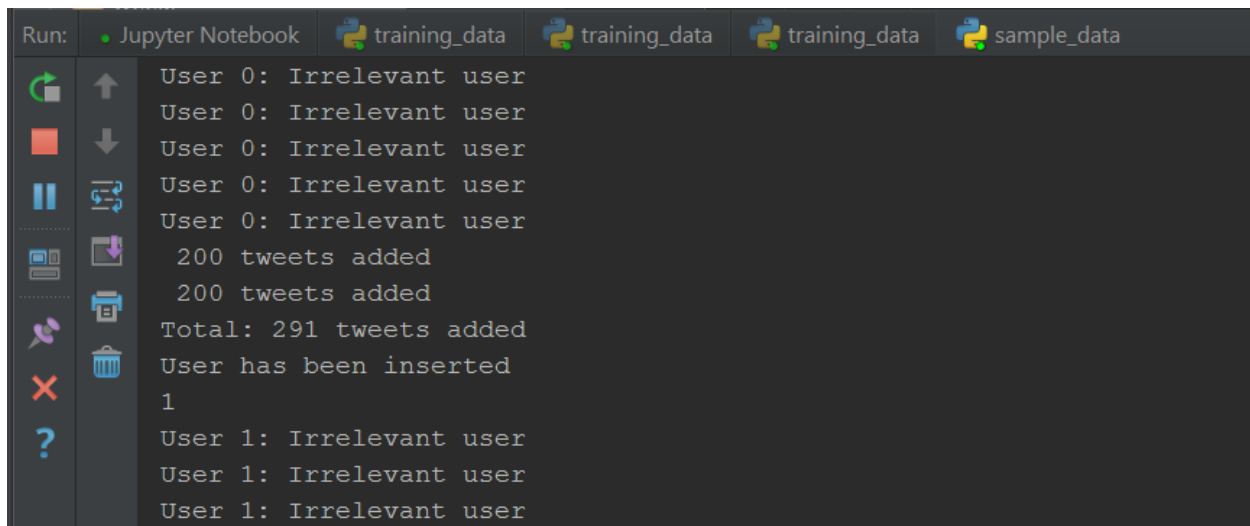
```

Run: Jupyter Notebook training_data training_data training_data
10
10
Start insertion process
200 tweets added
200 tweets added
Total: 304 tweets has been added
User has been inserted
11
11
Start insertion process
200 tweets added
200 tweets added
200 tweets added

```

Figure 12: Data training extraction – Printed information

And for machine 3, extracting random users for the sample, here are the outputs we have to follow the retrieval:



```

Run: Jupyter Notebook training_data training_data training_data sample_data
User 0: Irrelevant user
User 0: Irrelevant user
User 0: Irrelevant user
User 0: Irrelevant user
User 0: Irrelevant user
200 tweets added
200 tweets added
Total: 291 tweets added
User has been inserted
1
User 1: Irrelevant user
User 1: Irrelevant user
User 1: Irrelevant user

```

Figure 13: Sample data extraction – Printed information

Note: On machine 3, there are much more irrelevant users because the conditions are much stronger. This is why we have approximately ten times less sample_data than training_data.

Then we can check that the database is filled by doing simple queries that count the documents in each collection and that take the first element of each:

```
In [46]: print(db.users.find().count())
         print(db.tweets.find().count())

11008
14214178
```

Figure 14: Count items from the database

```
In [39]: tweets = [u for u in db.tweets.find({}).limit(1)]
         users = [u for u in db.users.find({}).limit(1)]

In [43]: users[0]

Out[43]: {'_id': ObjectId('5823c84c79b6e33560134123'),
         'description': 'Writer. Mother. Tea lover. Regular at
         e Daily, Mindfood magazine and others.',
         'id': 546445694,
         'label': True,
         'location': 'Sydney',
         'nb_followers': 2280,
         'nb_tweets': 12338}

In [47]: tweets[0]

Out[47]: {'_id': ObjectId('5823c84a79b6e33560133483'),
         'date': datetime.datetime(2016, 11, 10, 1, 6, 9),
         'hashtag': [],
         'message': 'RT @HillaryClinton: "To all the little gi
         powerful & deserving of every chance & opport.
         'tweet_id': 796519279277576193,
         'user_id': 546445694}
```

Figure 15: Overview of user and tweet object in MongoDB

Note: Here it is interesting to note that for the training set, we don't filter on location so it's possible to get people not from US such as this user. For the sample (where we actually do the poll) we make sure, that we take only American

8.2 From database to dataset

8.2.1 *Word-bagging and cleaning*

Here is the implementation of the various queries used to **build the bag of words in MongoDB** and the various collections involved in the cleaning process. [GitHub display here.](#)

8.2.2 *Data featurizing*

Here is the implementation of the **construction of the Python bag of words** (import from MongoDB bag of words), and the featurizing with TF/IDF process. [GitHub display here.](#)

8.3 Machine Learning

Here is the implementation of our whole classifier. The main point is that we serialize the trained version of our LDA classifier, to be able to use it on our sample afterwards. [GitHub display here.](#)

8.4 Election prediction

8.4.1 *Voters poll sample*

Here is the implementation of the sample capture. **This piece of code is almost the same as the one in 9.1, the only difference being in the way we filter our users.** We still put the whole code to be complete and make sure the reader is not confused. [GitHub display here.](#)

8.4.2 *Sample dataset building*

Here is the implementation of the sample dataset building. **This piece of code is almost the same as the one in 9.2.2.** We still put the code to be complete and make sure the reader is not confused. The main differences are that we do not care about labels and we use the words list from the training sample. This is very important to make sure the columns are representing the same words, so that coefficients of the model can be applied to the sample. [GitHub display here.](#)

8.4.3 *Election estimation*

Here is the implementation to **estimate the final score.** [GitHub display here.](#)

9 Conclusions

9.1 Rewind it!

For more clarity, and before heading to the actual implementation side, which you may have already seen parts of, let's sum up guidelines of our projects in a few points.

Project goal

Exploit a huge amount of Twitter data (tweets from Trump and Clinton followers) to run some machine learning predictions on US Elections.

Project main steps

- Definition of our training data (label choices)
- Extract Twitter data from Trump et Clinton followers
- Design a Mongo database to store 59 000 voters and their 68 millions tweets
- Transform this data to fit the machine learning requirements (bag of words, cleaning, overfitting)
- Train machine learning classifiers
- Draw the actual prediction

Project key points

- The final database reaches **21 GB**, which fulfills the project requirements.
 - A lot of data engineering as well as a learning phase
 - We built from scratch our own dataset from Twitter data
 - Our project doesn't pretend to be better than actual politics polls. It allows to understand how Twitter and politics are linked
 - Interesting conclusions and analysis come up at the very end of our timeline.
- This is it: this course brought us a very useful bunch of tools to **play with data in order to get some value from it**

```
> show dbs
local                0.000GB
tweepoll_sample      2.118GB
tweepoll_v2          19.151GB
work_db              0.384GB
>
```

Tools used from the course

- Week 1: GitHub synchronization
- Week 2: Python implementations for all we made
- Week 3: numpy and sklearn to handle the machine learning part
- Week 5: Strong gain of experience with MongoDB NoSQL Framework, for all our data storage and manipulation
- Week 7: MapReduce paradigm used for optimizing our data engineering processes

New tools discovered out of the course

- MongoDB MapReduce in Javascript framework
- TF/IDF featuring of a bag-of-words
- Serialization with *pickle* to keep a track of our classifiers and data structure used several times

9.2 Predictions conclusions

We finally got a result of 51,95 % to 48.05% for Donald Trump and a prediction state by state which has met the actual result 9 times out of 10.

We have built a classifier that estimates if a twitter user is more likely to be following one candidate or the other, with the content of his tweets. Here we would like to highlight that this has worked for US 2016 election but can be adjusted to other presidential elections or referendums with only a few changes. It is also interesting to notice that the approach can answer the question: Is this user likely to be following this twitter account? This twitter account could be a football club, as well as a brand or a university. Hence our project can be generalized to a lot different fields, especially in advertising.

It seems important to us to remind the reader that we did not aim at building the best classifier possible, but at showing evidence that it is possible to get a lot of information on a user thanks to his tweets with current big data technologies. Hence we focused a lot on the steps to acquire and analyze the data, and less on how to improve our classifier.

To improve our results a few steps could be helpful:

- Perform a stemming of the words prior to the bag of words construction. This means removing all suffixes, for example finishes, finish and finishing would be considered as the same word
- Improving the current stop_words list. Here if we spend some time on the final bag of words, it is clear that not all the current words add information
- Improving the location analysis to get more information on the user.
- Draw a sample which is more likely to be representing the actual population (instead of a random sample)

10 Bibliography

Here are a few links useful to consult with our code and explanations.

- GitHub repository: <https://github.com/Cerblo/projects>
- TF-IDF documentation on Wikipedia [here](#)
- Twitter API documentation: <https://dev.twitter.com/docs>
- A few twitter accounts to understand how our data is built:
 - [DTU's account](#)
 - [Hillary Clinton](#)
 - [Donald Trump](#)
 - [Barack Obama](#)