Universität Innsbruck

Department of Computer Science
Research Group Quality Engineering

MASTER THESIS

Sentiment analysis and linking of social media with open-source software repositories

Martin Ďurček

Supervisor: Prof. Dr. Michael Felderer

Innsbruck, 10. October 2018

Abstract

The main goal of this master thesis is to extract and analyse publicly available data from social media (Twitter, Reddit), Stack Overflow and Git and determine whether there is any relationship between the *sentiment of open-source tool users* and the release frequency and release size. Second part of the thesis is about automatic *pairing Git issues with the Stack Overflow discussions* based on the text similarity.

These tasks required a development of an algorithm based on *natural language processing* and *machine learning*. Thesis covers all important steps of developing such algorithm - from choosing a training dataset through preprocessing the data and evaluation and fine-tuning of classifiers to the presentation of the results. Performance of various algorithms from various development packages and with several datasets is studied. Final implementation is based on *Scikit-learn* Python module and utilizes *Frequency - Inverse Document Frequency (Tf-Idf)*.

In the last part of the thesis, the results are presented to the open-source project users and compared with their personal experience. Also, two extra real-world use cases where my classifier was applied are mentioned.

Contents

Chapter 1

Introduction

With the social media boom in the last decade, amount of generated data increased exponentially. Current average daily tweets count oscilates around 500 million[1]. These data contain valuable information of all kinds and that is the reason why with social media rise also natural language processing and sentiment analysis became hot research areas. They provide insight into data created by real people, users and consumers. More and more businesses start to take advantage of this and move from traditional means of data gathering and analysis to this new and still rising space. Reviews, blogs, statuses, tweets - these all provide rich environment to extract knowledge from. Sentiment analysis has several approaches such as classification, regression or ranking and each is applied to tackle different tasks. Probably the most intuitive one is classification which assigns one of defined classes to each analysed textual document. That is also an approach I have decided to use in this thesis.

I have analysed tweets of several open-source projects (OSS) and tried to find a relationship between their release frequency and sentiment. Analysing tweets differs to another sentiment analysis implementations and represents interesting challenge as tweets are special type of textual data limited to 280 characters per "document" while often using very specific language.

Another growing field of natural text processing is text semantics interpretation, topic modelling and its implementation within bug tracking systems either for filtering or flagging duplicate reported bugs. As second part of my thesis, I have done somewhat similar process - I have tried to pair social media discussions with their respective reported Git issues based on similarity of their textual features. Here I have used a common approach of transforming documents into vectors and computing their cosine similarity.

[1] https://www.omnicoreagency.com/twitter-statistics/

1.1 Motivation

Motivation behind this thesis lies in my interest in machine learning and the fact that I unfortunately did not manage to explore this field over the course of my studies. After taking some very basic Coursera[2] courses in this area, I felt I wanted to learn more and rather than finishing some online course exercises, I wanted to implement the whole workflow of such algorithm from the ground up and experience all the nitty-gritty myself. Machine learning, sentiment analysis and data mining in general are fields on the rise and it is always good to have as wide knowledge as possible. That is also the reason why my thesis is relatively wide-spread and targets many areas of data mining. I can very easily get drawn from one interest to another and that is also a case in this thesis, just in a smaller scale. I am also a fan of open-source movement and mindset and I liked the idea of combining these two into my thesis.

[2]https://www.coursera.org/learn/machine-learning

Chapter 2

Goal and proposed framework

Despite Open-Source Software being popular, widely used and implemented in production, there's no definite consent regarding the ideal and most favoured release frequency and size. Some projects release several times per month while the others only once per quarter or even less. Also releases differ in size, number of commits and fixed issues they address. My goal is to explore some potential hidden patterns which could be generalized and used in the future release planning and policy in Open-Source space.
Another issue and not only with Open-Source projects are bugs and finding workarounds and temporary solutions for them. We all know the scenario when a correct code doesn't function properly. We head to Stack Overflow or Reddit and start to look for similar problems. If lucky enough, we find a question with a solution presented in the comment section. But that's not always the case. Especially questions about faults caused by library/framework bugs have often empty and not really helpful comment sections because there should not be such problem in the first place. Such situations can eat up a lot of developers' time and kill productivity. Therefore I hoped to create an algorithm which could link corresponding GitHub and Stack Overflow items and make these frustrating situations easier.

This thesis is divided into 2 parts. I aimed to create a framework which could be potentially used as a base of tool used to recognize level of user satisfaction and its sudden changes. Second part of the framework should be able to find and link pairs of issues reported in bug tracking systems and their respective social media entries. Both parts of the framework use various text processing methods, sentiment analysis concentrating more on machine learning while bug linking utilizes topic modelling and text similarity concepts.

As can be seen in Figure 2.1, two final products of the framework are sentiment classifier and linking algorithm. Green elements are the ones used in

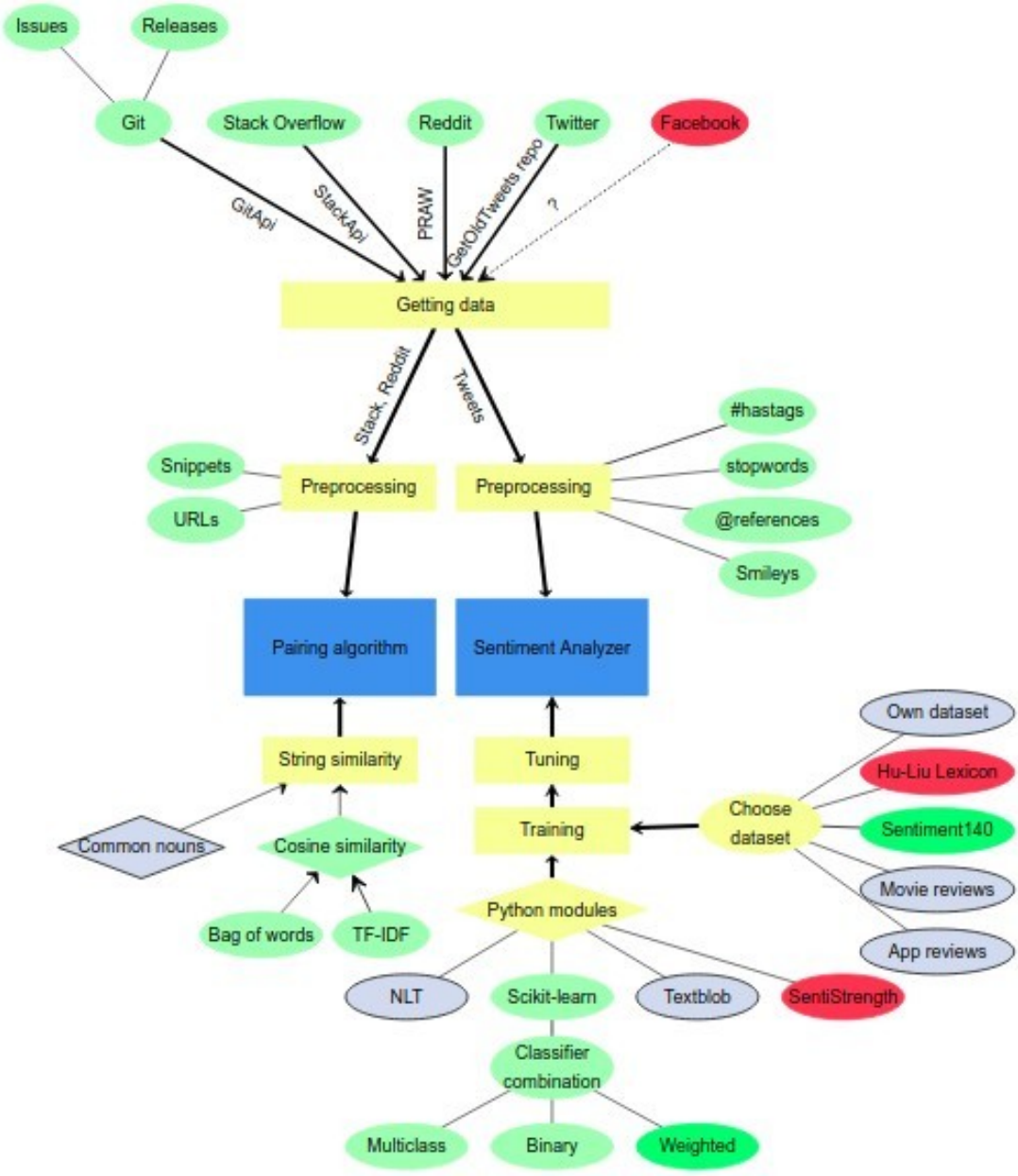


Figure 2.1: Brief sketch of dataflow in the proposed framework

final solution, grey ones were tested and considered and red ones were left out or not possible to implement. Work on the framework could be divided into several steps:

1. **Getting the data** - there is no data science, machine learning or natural language processing without the data. That is why the very first step is to create a mechanism to easily mine data from various sources. Git mining is required to get information about OSS projects and is described in subsections number 5.4.1 and 5.4.2 while Twitter, Reddit and SO were mined to provide data which are the target of analysis 4.2.1.

2. **Data preprocessing** - all textual information, especially those which originate on social media contain noise. To get rid of all this extra information, which might cause inaccuracies and faults, data preprocessing is a necessary step.

3. **Finding a right training dataset** - since the performance of any ML algorithm is very tightly bound to the training data used, this step should not be neglected and be considered as important step as any other. It is described in the Section 6.2.

4. **Choosing a tool/module/library and particular classification algorithm** - there are many algorithms used for classification but environment for their usage and best performance differs greatly. To choose a correct algorithm and find the best combination of parameter values is often a long process. Things do not get any easier when we take into account that there are not only many algorithms but they are actually also implemented in several programming languages and libraries. Inconsistencies between results of several SE modules were pointed out by Bin Li at al. in his paper *How far can we go*. My approach to problems and challenges of this step are described in great detail in sections number 6.3 and 6.5

5. **Topic modeling and text similarity** - this is the crucial step for the second part of the framework. Once the data are downloaded from Git as well as from social media, the last step to do is to find the matches. This part is described in Chapter 7.

After all the steps are implemented, proposed framework will be completed. But that does not guarantee that the output data are easy to interpret. That's why one extra additional step is needed. Using some statistics and data science methods, I am interpreting the results in Subsection 6.7.3 and Subsection 6.7.4.

This framework targets following research questions:

- RQ_1: Do the OSS projects which release more often get general better sentiment score on social media?
- RQ_2: Does a release have an immediate effect on sentiment?
- RQ_3: Is there a correlation between sentiment change and size of the release (number of commits) ?
- RQ_4: Is it possible to link social media entries to their respective bugs which they talk about?

Chapter 3

Related work

Thesis itself combines many fields and areas. Its main topics are Git mining, Twitter mining, sentiment analysis and text similarity.

Nice example to Git mining is work done by Russel and Mathew [1]. They discovered 9 interesting perils and according to their paper, most projects are inactive and have low count of commits. Two thirds of repositories are personal and large portion of these are not software development projects. Also, pull-requests are not very popular feature and many projects do not even conduct all their software to the Git. Tsay et al. [2] developed two measures of project success - Developer Attention and Work Contribution. They found out that projects with highly socially connected developers are not necessarily the most active or popular projects and projects with a high level of developer multitasking surprisingly tend to receive less Developer Attention, but greater Work Contribution. Little evidence that usage of specific social media features in GitHub directly correlates with software project success was found. Jarczyk et al. [3] created in their work 2 own metrics - one of them being Quality of Support. They surprisingly shown that higher number of developers has positive effect only on fixing bugs in the long term, not on the quick response within 3 days. Takhteyev et al. [4] explored developers' geographic locations by examining self-reported information available within GitHub profiles. They confirmed the hypothesis that largest share of users are registered from United States - 39 %. A research has also been done regarding availability and retrieval improvement of Git data ([5] and [6]). Sentiment analysis of Git commits was described in Guzman et al [7]. They found that language used in the project has an affect on sentiment. Java projects showed the worse sentiment. Also, commit messages on Monday tend to show worse emotions. Christley and Scott [8] extracted 29 open-source related activity types based on social positions and activity patters. Instead of GitHub data, they used Sourceforge.net dataset. Sadly, they didnt continue in their work to analyse the effect of the

quantitative population for these positions within various projects.

Last mentioned paper brings me to another part of my thesis, which was sentiment analysis of social media, mostly Twitter data. A continuous series of Twitter evaluation exercise called SemEval organized by SIGLEX show how big and complex task Twitter sentiment analysis is as several teams continue working on couple of shared tasks. This project historically started already in 1998 and after 4 conferences once every 3 years, it takes place yearly from 2012. As stated in Rosenthal et al. [9]) summary, in 2017 was the count of participating teams 48. The most common technologies used across all the teams were Python (numpy and sklearn libraries), Java, Weka or NLTK and the most common external lexicon was was Sentiment140. This justifies my selection of training data in Section 6.2, as well as used language and the particular library in Section 6.3. Like I mention in Section 4.2.1, Twitter is a default choice for social media sentiment analysis. That means lot of work has already been done in this area. Agarwal et al. [10] acquired 11,875 tweets from commercial source, translated them to English using Google Translate and after deleting bad translations and stratified sampling to get balanced dataset, their dataset's size was 5127 tweets. After preprocessing, they calculate 50 various features for each document (tweet). They used unigram as their baseline model and offered a comprehensive set of experiments using Tree kernel and Senti-features models. Compared to the baseline, they reported 4% gain for both binary and 3-way classification. Koloumpis et al. [11] used various feature types (n-gram, lexicon, POS and microblogging) and showed that using POS tagger might not be useful for sentiment analysis of Twitter and therefore microblogging sphere in general. Pak and Paroubek [12] worked with considerably bigger dataset of 300,000 tweets evenly distributed among 3 main sentiments. Their approach is using Twitter API and exact same steps as Go et al. [13] and Read in 2015, but because of its latest changes it might not be usable anymore. My workaround for this issue is described in Subsection 4.2.1. They queried for positive and negative emoticons to build an arbitrarily large training dataset which is one of their contributions. This is an interesting approach as emoticons do carry lot of emotions but in case of irony can be misleading. They trained 2 classifiers based on Naive Bayes and experimented with different features as bigrams, trigrams or unigrams and also part-of-speech (POS) distribution information. The best performance was achieved with bigrams and a very high accuracy with a low decision value was obtained. As shown in the previous references, the consensus usually is that preprocessing should get rid of as much noise as possible. But Saif et al. [14] are on the other side of the spectrum. They demonstrated that not removing stopwords can actually increase accuracy. Apart from this, they have also defined new so called "semantic features" which are assigned to each entity within a tweet. In their paper, they shown that using these

features increased a performance of two-class Naive Bayes classifier. While textual features are analysed to the biggest detail, emoticons and emojis are mostly overlooked or preprocessed (also my case in Section 6.5, Table 6.5). Hogenboom et al.[15] created sentiment framework which does text-based and emoticon-based analysis separately and they merge the results later. Novak et al. [16] went even step further and decided to analyse the new generation of emoticons - emojis. They drew a sentiment map of the 751 emojis, compare the differences between the tweets with and without emojis, the differences between the more and less frequent emojis and their positions in tweets.

Much less saturated is the topic of linking social media bug entries and git repositories. I have not found any paper tackling this problem. Bachman et al. [17] pointed out that not all known bugs are reported through one channel only (e.g GitHub's issue system) what is important to keep in mind during my work described in Chapter 7. The closest topic to this is probably duplicate bug detection as it also requires extracting and understanding of bug descriptions. Bug analysis most similar to my work was done by Runeson et al. [21]. They implement and test a technique of detecting bug duplicates using similar techniques to mine in Section 7.1. Bug reports are turned into stream of stemmed tokens and stopwords are removed. On top of these procedures their work shares with mine, they replace synonyms. For similarity calculation, they used 3 measures - cosine, dice and Jaccard but eventually decided to continue only with cosine as their main choice. I'm also using cosine what makes our work even more similar. Results of this study showed that about 2/3 of the duplicates can possibly be found. Similar task was tackled by Jalbert and Weimer [22] who introduced an algorithm which flags duplicate bug reports on their arrival. They claim to reduce the development costs by 8% while still allowing at least one reported issue per bug to reach developers. They pointed out that inverse document frequency was not useful for their task. Algorithm by Anvik et al. [18] assigns incoming bug reports to correct developers based on the textual features of the report with 57% precision for Eclipse project. Using bugs reported for Firefox, unique bugs are identified 90% of time and their duplicates correctly assigned 28% of times. This shows that my results presented in Section 7.3 are reasonable. Weiss et al. [19] predict time required to fix a bug based on the description similarity with bugs from the past. For similarity calculation, they used Lucene engine developed by Apache and worked only with bug title and description. I've chosen the same approach in Subsubsection 5.4.2.

Thesis has 2 separate parts, each tackling its own unique task. Both parts utilize simple Term Frequency Inverse Document Frequency (TF-IDF) to extract word relevance from text. It has been discussed and used for this

task in many papers over the years. Lan et al. [23] came up with their own comparative scheme called *tf.rf* where $IDF = log(N/n_i)$ was replaced by $rf = log(1 - n_i/n_i-)$. Using this scheme with linear SVM on McNemar's significance tests [24] shown better performance than 4 other TF-IDF variants. As described in Section 6.5, I'm also using and testing SVM with TF-IDF. The tests were executed on Newsgroup and Reuters corpus.

Chapter 4

Social Media

Social media is a term referring to online communication channels meant for social interaction, content-sharing and collaboration. Over time, this term became widely used, its exact definition somewhat blurred and most of the today's websites could be labeled as social media to some extent. In context of this thesis, I refer to social media in its original meaning. To be considered a social media platform, most of following features usually need to be fulfilled:

- **User accounts** - platform allows users to create and run their own accounts that they can log into. These are online representations of their owners and serve as a tool to reach and interact with other users.
- **Profile pages** - pages which represent an individual - be a real person, group of people or company. It should contain personal information about the user like bio, profile picture or some other personal data.
- **Friends, followers, groups** - list of accounts whose owners have some form of a relationship or common interest with the user.
- **News feeds** - Area where all new content from other connected entities appears.

Even if a platform fulfills these requirements, it doesn't necessarily have to be classified as social networking platform as pointed out in Haewoon Kwak's paper[27].

4.1 Potential of social media data mining

Social media are changing the way that information is passed across societies and around the world.[28] Among many other potentials of social media, there is a huge amount of data generated on daily basis. These data carry lots of real world data and if used correctly, can offer deep insight

into almost any area. The process of analysing these data and searching for repetitive reoccurring patterns with goal of predicting future trends is also called social media mining. Successful mining can not only save money and time spent on getting the data in more traditional way like surveys but can also provide crucial factor in planning or decision making of businesses. Although internet is one big hole and contains lot of false facts and disinformation, there is indication that social networks tend to favour valid information over rumours.[29]

4.2 Data

4.2.1 Getting data

As you will have a chance to see, big social networking platforms started realizing that data they own are a golden egg and getting raw full data from them got much more difficult than it was in the early years of social media age.

Twitter: When talking about sentiment analysis of social media, analysis of Twitter data has in last couple of years became almost a default choice. This change can be nicely seen in the Mantyla, Graziotin and Kuutila's [30] wordcloud of SE papers before and after 2013.4.1.

Figure 4.1: Wordcloud comparison of pre and post 2013 SE papers 4.1

To get the data from Twitter, I first tried to use the Twitter API but I very early got to know that Twitter is well aware of the worth of their data. They do not provide tweets older than 2 weeks what basically made their API inapplicable for my purposes. The only way how to get historical Twitter data is actually:

- collect them over time
- buy them from Twitter
- buy them from other companies who collect Twitter dumps over time

Therefore to obtain my data, I had to use the Twitter Search API. To do this I used the GetOldTweets[1] repository from Jefferson-Henrique which provides an extra layer on top of Search API and simplifies working with it. Using this technique, I managed to get the significant amount of data needed for all my tasks in this thesis.
This repository offers following collection of search parameters which can be used to filter specific tweets according:

- setUsername - Twitter account without "@"
- setSince - lower date bound with format "yyyy-mm-dd"
- setUntil - upper date bound with format "yyyy-mm-dd"
- setQuerySearch - search text to be matched
- setTopTweets - boolean flag whether to return only top tweets
- setNear - location are reference
- setWithin - radius from "near" location
- setMaxTweets - max amount of tweets to be retrieved

I got historical Twitter data by looping all projects and their release dates and requested data:

- on the release dates (Listing 4.1)
- in the interval of 2 days before and after release date (Listing 4.2)
- weekly

[1] https://github.com/Jefferson-Henrique/GetOldTweets-python

```
miningConsoleCommand = "python Exporter.py
        --querysearch '" + frameworkName + " AND " + version + "
    '    --since " + str(releaseDate) + "
        --until " + str(afterRelease) + "
        --output='" + frameworkName + "_" + str(releaseDate) + "
    .csv" + "'"
```

Listing 4.1: Creating command to get Tweets about a project version on release dates

```
miningConsoleCommand = "python Exporter.py
        --querysearch '" + frameworkName + "'
        --since " + str(fromDate) + "
        --until " + str(toDate)  + " --lang " + "en" + "
        --maxtweets " + str(TWEETS_PER_RELEASE) + "
        --output='" + frameworkName + ".csv" + "'"
```

Listing 4.2: Creating command to get Tweets about a project version in particular time interval around release date

Facebook: I originally planned to use Facebook statuses as a big part of analyzed data. Sadly, this was not possible since Facebook Graph API does not allow post searching feature. There was this option till early 2014 with Facebook API 1.x versions but since Graph API has been introduced, there is no way how to make Facebook application send requests to 1.x versions of API. At first, application created before 2014 were still working on top of the early API versions and it has been maintained but over time, all the applications were migrated to 2.x API versions. There is currently no public way how to freely get the Facebook posts data. The other types provided by the API are for example user, page, event, group or place.

Reddit: Despite that Reddit does not offer big amount of user data in OSS projects subreddits, I thought getting and working with Reddit could increase the variety of users and the data which I will be working with. To get the data I used Python Reddit API Wrapper (PRAW) used to directly work with Reddit API via HTTP requests.
Class Reddit provides a convenient way how to access Reddit API. Instance of this class can be seen as a gateway to interact with API through PRAW. To instantiate this class, user first has to register his application. This gives user unique *useragent* key which identifies the application. This is so that if a program misbehaves for some reason, it can be more easily identified, rather than look like a browser. All mandatory arguments are shown in Listing 4.3.

```
reddit = praw.Reddit(
                clientid='CLIENTID',
                clientsecret="CLIENTSECRET",
                password='PASSWORD',
                useragent='USERAGENT',
                username='USERNAME'
                )
```

Listing 4.3: Instantiating Reddit class object

After connection to API is successful, sending requests with PRAW is straightforward. To get the submissions from a particular subreddit in a specific time interval, just a basic loop shown in Listing 4.4 is enough.

```
for submission in reddit.subreddit(redditName).submissions(FROM,
    TO):
```

Listing 4.4: Getting posts from subreddit *

**During the writing of this thesis, new Reddit API deprecated the submission class and all endpoints using this class.*

Each post (submission) contains among other information also array-like member variable of all comments. Simply concatenating all those gives the whole textual representation of the discussion.

Stack overflow: To extract the questions about the OSS projects of my interest I once again used provided API. Python module called StackApi offers a way how to communicate with various Stack Exchange API endpoints - answers, badges, comments, posts, questions, tags and users. To initiate communication with StackAPi, one needs only to specify Stack webpage and choose an endpoint, tag, time interval and if needed also some other non-mandatory parameters. Afterwards, calling *fetch* method starts returning questions. Snippet of how I got the data can be seen in Listing 4.5.

```
 SITE = StackAPI('stackoverflow')
     SITE.max_pages = 1;
     while True:
       questions = SITE.fetch(
         'questions',
         fromdate=date(2012, 5, 8),  # year,month,day
           todate=date(2016, 4, 15),
           tagged=project,
           filter='withbody',
           sort='creation',
           page=page
        )
```

Listing 4.5: Getting Stackoverflow questions with StackApi

At first I intended to use the class Posts, which returns both, questions and answers, but later I have realized how huge amount of data SO contains. The average count of questions using one of the examined projects names as a tag was around 150 000. Because of this, I had to find out how to filter just the questions with higher probability of talking about bugs. Since the questions on SO do not have any labels which I could use to my advantage like I did with Git issues, I decided to keep just the questions which mention a word bug. This still left me with considerable big dataset of 8143 questions to work with. Out of all properties of questions retrieved from questions endpoint, I decided to store and further work just with several of them mainly its title and body since these two provide most of the semantic meaning.

Chapter 5

Open Source Projects Analysis

In recent years, there has been a substantial increase in interest of open source projects. Open source software is typically developed by community of people interested in the particular area who do not necessarily know each other. These communities are usually web-based. Open-source phenomenon raises many interesting questions. Its proponents regard it as a paradigmatic change whereby the economics of private goods, built on the scarcity of resources, is replaced by the economics of public goods, where scarcity is not an issue. [31] There are many projects which are on top of their game and yet still being open source. For example, Apache, a free server program is often a go-to decision when web server implementation is needed. In 2002 it was used on 56% of web servers worldwide [32].

5.1 History of Open-Source Software

When talking about open source, most people immediately think of Linux. But there's so much more than than. The origin of open-source software can be traced back to the 1950s and 1960s, when software was sold together with hardware, and macros and utilities were freely exchanged in user forums. [31] In late 1983, GNU project has been announced and the next year a Free Software Foundation has been founded - both by an MIT employee Richard M. Stallman. All software written and released under Free Software Foundation had zero licences. The Linux kernel was released as freely modifiable source code in 1991 and like Unix, it attracted attention from many volunteer programmers. Another big milestone was a year 1998 when Netscape released a source code for Mozilla. In 1999 it was obvious that more and more corporate money will be invested into open source space as IBM announced their support for Linux by investing $1 billion in its development.

5.2 Types of Open-Source Software analysis

Open-Source space is very interesting space to analyse. Not only because it was not that long ago, when a thought of decentralized team working on software for free seemed unlikely and now it is a common scenario, but also because of its nature. It does not hide anything and therefore all the data for the deeper analysis of requirements needed to success are up there for grabs. To determine whether an OSS project is thriving and sustainable, various analysis types can be done. The most obvious metric to look at is code activity and release history. This can be easily investigated by looking at projects' revision control system and pattern of contributions. Number of contributions usually differs as projects go through cycles of releases and cooldown periods. It's also hard to generalize between projects as they all have different approach towards releasing frequency. This is exactly the topic I'm studying in chapters 6.7.3 and 6.7.4. Another important and also easier to interpret aspect is user community developed around the project/framework. Just a straightforward use of search engine statistics can show how popular various repositories are. This next point is little bit counter-intuitive, but healthy project should also have a reasonable number of known issues as it shows there is a real community behind the project using it. Therefore are bug tracking systems great source of information as well. Last but not least indicator and area to analyse project's success is the ecosystem as a whole. Seeing consultancy and customisation services or and companies that bundle the software with other products as part of solutions is a strong indicator the project is doing well.

5.3 Choosing projects of interest

There are loads of OSS projects nowadays and it turned out to be pretty interesting process to choose the best ones for my thesis. To make sure chosen projects fit into my work and fit all my needs I defined several requirements which needed to be fulfilled at least to some extent:

- Project needs to be widely used and well-known. This ensures there will be enough data on social media about it what will result in the less biased final results.
- Project has to have accessible bug tracking system or Git repository with list of known issues. This will provide the data for linking of social media with their corresponding GitHub issues.
- Ideally, projects could be from the same area to avoid coincidentally choosing an outlier project from some either popular or unpopular field.

After considering these three points, I ended up choosing several open source web development frameworks as my projects of interest.
Projects of my choice are NodeJS, AngularJS, EmberJS, VueJS for front-end technologies and Laravel, Symfony and CakePhp for backend PHP technologies. Some frameworks like React, Meteor or Django have been left out because of their misleading names would require lot of additional work to filter out data unrelated to the actual frameworks. For example, when tested Django framework, most of the twitter data referred to movie "Django Unchained".

5.4 GitHub mining

GitHub as a most-popular version control system is often a go-to choice for many OSS projects. With over 10 million repositories, it is becoming one of the most important source of software artifacts on the Internet [1]. Git providing valuable information and insight into projects happened to be the case in this thesis.
Thanks to GitHub Rest API v3[1] is data mining with GitHub very easy and straightforward. To send request to this API, OAuth2 token needs to be present in the request header. There are several ways how to acquire this token. I have decided to register an OAuth2 App under my GitHub account and that way I got constant token. Another way is to request a token programmatically, but I thought it would be an unnecessary overhead. OAuth2 applications can be registered under *Account settings >Developer settings >Personal access tokens >Generate new token.*

5.4.1 Release dates

To get the project release dates, I've sent one request to the endpoint *git/refs/tags* (Listing 5.1)

```
request = Request(projectUri + "/git/refs/tags")
request.add_header('Authorization', 'token \%s' \% token)
project = urlopen(request).read()
tags = json.loads(project)
```

Listing 5.1: Requesting all project tags git API tags endpoint

To get the release dates, after obtaining all tags, one more request for every one of them was needed (Listing 5.2). Path to release date field within response body as JSON is tag >object >url >Send new request >author >date.

[1] https://developer.github.com/v3/?

```
for tag in tags:
  version = tag['ref']
  got_object = tag['object']
  detailedUrl = got_object['url']

  #request details of particular release
  request = Request(detailedUrl)
  request.add_header('Authorization', 'token %s' % token)

  #get the person responsible for the release
  repoReleaseDetails = json.loads(urlopen(request).read())
  tagger = repoReleaseDetails['author']

  #get the date of the particular release
  releaseDate = tagger['date']
```

Listing 5.2: Requesting tag details and accessing release date

I saved the release dates in simple text file, one per line. This code was later on changed (in subsection 6.7.4) to include also number of commits per release.

After executing this step, I needed to divide frameworks into groups based on their releasing frequency. I ended up with 3 groups:

- Seldom releasing frequency - less than once per month
- Medium releasing frequency - between one and 3.5 times per month
- Often releasing frequency - on average more than 3.5 releases per month

Using these values as bounds for groups, projects were divided into their respective groups. Projects, release counts and average release frequency are in Table 5.1.

Group	**Project**	**Release count**	**Months**	**Frequency**
Often	EmberJS	266	76	3.5
	VueJS	209	45	4.64
	NodeJS	440	100	4.4
	Symfony	291	76	3.82
Medium	AngularJS	190	82	2.34
	CakePHP	289	104	2.77
	Bower	102	60	1.7
Seldom	Gulp	16	25	0.64
	Yii	41	59	0.69
	Bootstrap	42	72	0.58

Table 5.1: OSS projects grouped according to their releasing frequency

5.4.2 Issues

To get the project issues, I have used the endpoint *issues*. It offers various parameters like state, labels, sort, direction or since date. I have decided to work just with closed issues and snippet where I am sending the request can be seen in Listing 5.3).

```
request = Request(projectUri + '/issues?state=closed&perpage
    =100&page=' + str(pageNum))
```

Listing 5.3: Requesting 100 closed issues

It is also worth noting here that GitHub's REST API v3 considers every pull request an issue so I had to identify and filter them out using their *pullRequest* key. I'm saving all the data API provides but later work only with bug description and title.
During my later work, I have realized the amount of issues is just too big and broad and not every issue is a bug or even remotely similar to bug. That was when I have decided to filter the issues and keep only real bugs. For this I have used Git labels. Labels on GitHub help organize and prioritize work. They can be applied to issues and pull requests to signify priority, category, or any other information you find useful. There are two types of labels - default and custom. GitHub provides default ones in every new repository. All default labels can be seen in Table 5.1 and can be used to create a standard workflow in a repository:

Label	Description
`bug`	Indicates an unexpected problem or unintended behavior
`duplicate`	Indicates similar issues or pull requests
`enhancement`	Indicates new feature requests
`good first issue`	Indicates a good issue for first-time contributors
`help wanted`	Indicates that a maintainer wants help on an issue or pull request
`invalid`	Indicates that an issue or pull request is no longer relevant
`question`	Indicates that an issue or pull request needs more information
`wontfix`	Indicates that work won't continue on an issue or pull request

Figure 5.1: Default Git labels provided for every repository

From there I have chosen the label *bug*. Then I have checked all custom tags of all repositories and chosen just those which were semantically similar to bug. All chosen labels can be seen in the Table 5.2.

Repository	Chosen custom labels
NodeJS	confirmed-bug, errors
AngularJS	type: bug
VueJS	browser-quirks, 1.x, 2.x
Aurelia	enhancement
EmberJS	Bug

Table 5.2: Reddit submissions counts

Chapter 6

Sentiment analysis

Sentiment analysis and opinion mining is the field of study that analyses people's opinions, sentiments, evaluations, attitudes and emotions from written language.[33] It is also known as emotion AI or opinion mining. The main and basic task of this field is to correctly classify the polarity of a particular text and evaluate whether is it positive, negative or in some cases neutral. It can be used in almost any situation where data need to be analysed for its sentiment aspect, what means the application options are almost endless. Some examples are product reviews, online discussions or social media content.
SE is in demand mostly because of its efficiency. Tens of thousands of documents can be analysed within seconds and despite results are not always as exact as human workers would produce, the efficiency boost is often too big not to take advantage of it. SE can be divided into several steps:

1. **Data collection** - involves all the substeps required to gather user-generated content from any source. Surveys, blogs, various forums and social media. These all contains huge amount of real people from real world with real experiences. These data are always expressed completely different - using slang, shortcuts, internet language or generally just being used in different context.

2. **Data preprocessing** - this step represents cleaning the data. Data preprocessing often significantly impacts generalization performance of a supervised ML algorithms [34] and if there is much irrelevant information and data are unreliable, then knowledge discovery during the training phase is more difficult. It removes irrelevant content which could potentially lead to bad and incorrect results. This is a very delicate step because it manipulates the raw data and if not done correctly, it can easily change results.

3. **Sentiment detection** - this step basically stands for training of the classifier. Subjective feelings and expressions are highlighted, empha-

sized and retained while objective information like facts are discarded and ignored.

4. **Sentiment classification** - subjective expressions are assigned to the most fitting class.

5. **Output interpretation** - graphical presentation of obtained results. Time and sentiment can be analysed to construct various charts, graphs, timelines and many quantitative numerical metrices.

6.1 History

Interest in opinion of other individuals is probably as old as the communication itself. There is evidence that already in ancient Greece, generals were trying to detect dissent among their subordinates using various "primitive" approaches [35]. Another approach to measure and evaluate a public opinion coming from ancient Greece and used to these date is voting. In the first decades of twentieth century, efforts in capturing public opinion started utilize questionaires and in 1937, first scientific journal on public opinion was founded.

In the last 10 years, SE and ML in general experienced big boom. According to data collected by Mantyla, Graziotin and Kuutila [30], nearly 7000 papers about sentiment analysis have already been written and not surprisingly, 99% of them were published after 2004 - making sentiment analysis one of the fastest growing research areas. The increasing interest about this area can be seen in the Figure 6.1.

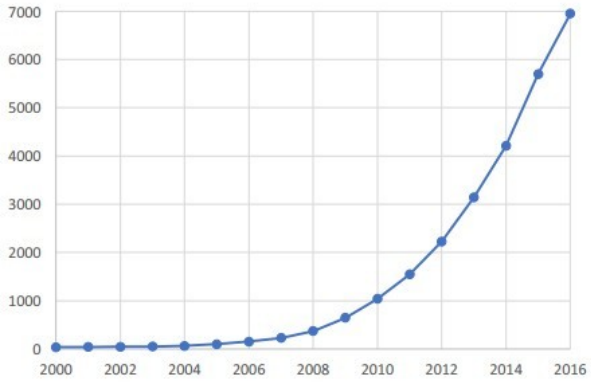


Figure 6.1: Cumulative count of papers about sentiment analysis [30]

6.2 Training datasets

One of the main building blocks of any correctly and accurately functioning ML projects is a training phase. It can actually be seen as a base for the whole project. One can have the best fine-tuned optimized classifier, but if the training data he used do not fit the domain where the classifier is intended to be used, results of the classifier can be (surprisingly) bad. It is the same as house and its base. If the base is not done correctly, however cool architectural solution have other storeys used, house is still going to fall in the next big storm.
That is why choosing a sufficient and fitting dataset to train my classifiers was a very important task. The datasets I have considered were:

- **Dataset140**[1] - it is currently the biggest dataset with tweets labeled by their sentiment. What is interesting and makes the dataset special is that opposed to other datasets being manually annotated by humans, this one was created by a program. It contains 1.6 million tweets with their polarity score (0 = negative, 2 = neutral, 4 = positive), tweet id, date of tweet publication, author of the tweet and the text of the tweet. More about how this dataset was created can be found in Go et al. paper [13]

- **Movie review data**[2] - Thousand positive and thousand negative labeled movie reviews. This dataset was introduced in Pang/Lee ACL 2004 [36]

- **Hu-Liu lexicon**[3] - plain list of 6800 common English words labeled as positive and negative

- **Warriner et al lexicon**[4] - This list of words was collected with Amazon Mechanical Turk. Three components of emotions are distinguished: valence (the pleasantness of a stimulus), arousal (the intensity of emotion provoked by a stimulus), and dominance (the degree of control exerted by a stimulus) [37]. Warriner and Kuperman extended ANEW norms collected by Bradley and Lang from 1034 words to 13,915 words (lemmas).

- **Stack overflow dataset**[5] - Later into the thesis I have decided to test dataset of 1500 manually labeled Stack overflow sentences created by Bin Lin et al. in their late paper on negative results in SA called "How far can we go".

[1] http://cs.stanford.edu/people/alecmgo/trainingandtestdata.zip
[2] http://www.cs.cornell.edu/people/pabo/movie-review-data/
[3] https://github.com/woodrad/Twitter-Sentiment-Mining/tree/master/Hu%20and%20Liu%20Sentiment%20Lexicon
[4] http://crr.ugent.be/archives/1003
[5] https://sentiment-se.github.io/replication.zip

- **My own cryptocurrency tweets dataset** - As already said before, performance of the classifier depends on how close the training data are to the real use-case data. That is why I considered and even started to create my own dataset targeting specifically only cryptocurrency tweets, which I have intended to analyze as well.

Surprisingly, I didn't find any big and widely used (standard) Reddit dataset.

6.3 Language processing tools

From the very beginning, I knew I wanted to use Python. I had some slight background knowledge in ML from online courses and most of them were done in Python. Therefore while searching and deciding which library should I use, I have always given a slight edge to the Python options. I have considered (and tested) these:

- **NLTK** - probably the best-known Python module for NLP. It provides easy-to-use interfaces for more than 50 corpora and lexical resources. It also offers a rich palette of processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning.
- **Textblob: Simplified Text Processing** - as a name says, Textblob provides easy processing and is actually built on top of NLTK. It provides a simple API for diving into common natural language processing tasks such as part-of-speech tagging, noun phrase extraction, sentiment analysis (Naive Bayes, Decision Tree), classification, translation and more.
- **Scikit-learn** - Python module for general ML, data mining and data analysis. It is built on NumPy, SciPy and Matplotlib modules.

Also, sentiment analysis is just one part of the task. To evaluate the data and find pattern, basic data science algorithms will be needed. With data science, R is very often listed as a default choice. Therefore were the results of google trends query shown in Figure pretty surprising. This definitely helped my decision with sticking to Python.

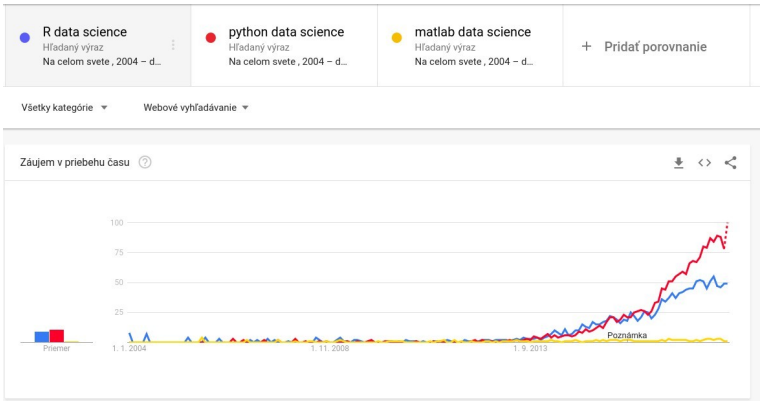


Figure 6.2: Google trend of searches regarding data science with various programming languages

6.4 Performance metrics

Terminology Before diving into talk about the metrics, there are 4 crucial terms which need to be explained:

- **True Positives (TP)** - instances correctly labeled as positive
- **True Negatives (TN)**- instances correctly labeled as negative
- **False Positives (FP)** - instances incorrectly labeled as positive
- **False Negatives (FN)**- instances incorrectly labeled as negative

The metrics that you choose to evaluate your machine learning algorithms are very important and not all are suitable for every situation. Choice of metrics influences how the performance of machine learning algorithms is measured and choosing a wrong evaluation metric for particular use could potentially lead towards eliminating the best performing algorithm in favour of the worse one. Here are some of the most often used metrics used to evaluate classification algorithms. It is also useful to choose the metric before doing the analysis, so you will not get distracted by already having the results in case of doing the decision later.

- **Classification accuracy** - this is the most intuitive and common evaluation metric for classification problems but it is also the most misinterpreted one. It is really only suitable when there are an equal number of observations in each class (which is rarely the case) and that all predictions and prediction errors are equally important, which is often not the case. In case of imbalanced dataset with 90% of instances in one class and only 10% in the other, predicting every instance as a

majority class without even considering its features would lead to high accuracy of 90%. This is called **accuracy paradox.**

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

- **Confusion matrix** - clean and unambiguous way to present the prediction results of a classifier. If the classification is binary (there are only 2 classes), this matrix has 2 rows and 2 columns - therefore altogether 4 cells which are filled with true/false positives/negatives count. Such scenario is demonstrated in Table 6.1. Although the confusion matrix shows all of the information about the classifier's performance, more meaningful measures can be extracted from it to illustrate certain performance criteria.[38].

Confusion matrix		
	Predicted positive	Predicted negative
Real positive	TP	FN
Real negative	FP	TN

Table 6.1: Confusion matrix

- **Precision** - Precision can be seen as a representation of a classifier;s exactness. A low precision can also indicate a large number of False Positives. If the precision is high, it says that there is a high probability of positive label being True Positive. It cannot be tricked but it also hides a lot.

$$Precision = \frac{TP}{TP + FP}$$

- **Recall** - also called *sensitivity* or the *True Positive Rate*, it is a number of True Positives divided by the number of True Positives and the number of False Negatives. In other words, it is a ratio of how many of all positive instances have been identified. Recall can be tricked (labeling all as majority class) but if used next to precision, it gives extra information

$$Recall = \frac{TP}{TP + FN}$$

Nice example to demonstrate difference between precision and recall is the concept of Indian Jurisprudence, where "100 culprits may be let go free but no innocent should be punished". If we let go so many culprits in order to ensure no innocent is punished, recall will be pretty low, but precision very high. There are cases when we want to maximize recall and situation when we want to maximize precision. As with most concepts in data science, there is a trade-off in the metrics we choose to prioritize. In my case, none

of these two has higher importance and in such scenarios F1 score offers a compromise between the two.

- **F1 score** - as already mentioned, precision hides some facts and recall can be tricked. To give the full story, they need to be used together. That is what F1 measure (F measure) is for. It is the harmonic average of the precision and recall, where its best value is at 1.

$$F = 2 * \frac{precision * recall}{precision + recall}$$

6.5 Classifier evaluation

All the testing has been done on the testing data supplied in Sentiment140 dataset.

Textblob: As a first option, I executed the analysis with Python TextBlob. Sentiment polarity returned by Textblob is a float value in range from -1 to 1 where positive values stand for positive sentiment and negative values for negative sentiment. Bigger the absolute value of output, stronger the sentiment is. This solution didn't need any training data or labeled dataset of positive and negative words because it already comes with trained classifiers. This alone was the reason why I felt that it might not be the best performing classifier. Because returned scores are floats and Sentiment140 tweets are labeled with just positive/negative, I had to execute a small transformation of the output. Interesting point in the transformation was handling of the zero(neutral) score. In the following Table 6.2 are shown several considered transformation options and from them resulting accuracy of the model.

Score = 0	**Accuracy**
Label as positive	0.608944
Label as negative	0.622649
Ignore	0.679612

Table 6.2: Textblob accuracy with various handling of neutral tweets

Obviously, none of these primitive approaches is the correct one. How to handle neutral class in sentiment analysis is still pretty open question and some possible approaches will be discussed later.
As mentioned earlier, accuracy is not the best metric to evaluate classifiers on, but with such low values, it is apparent that Textblob is not performing very well. There might be several reasons for this:

- Tweets are in general difficult to analyse because they are limited to 160 characters and therefore display sparse and noisy behavior typical

for short texts. They also contain lot of various special entities like hashtags or emoticons (I will get rid of these in my final classifier implementation). Classification is therefore not as accurate as it would be with other (longer) texts which provide more textual features.

- Textblob sentiment analysis comes with already pre-trained classifier. Golden rule of any ML algorithm says that training data should always be as similar as possible to the actual data the model is intended to be used on (and Textblob classifiers are not trained on the tweets about open source web development frameworks).

NLT: After getting discouraging results with Textblob, I have decided to implement my own classifier from the ground up using Natural Language Toolkit Python module. After reading several forum discussions, I have decided to use Naive Bayes classifier as people repeatedly claimed it to have the best performance for sentiment recognition of short texts like tweets. I have evaluated it using k-fold cross-validation with k-value of 4. In every iteration, new instance of Naive Bayes was trained on the 25% of preprocessed Sentiment140 tweets and evaluated on the rest 75%. Core of the whole training and cross-evaluation is demonstrated in listings number 6.1 and 6.2.

```
1 word_features = get_word_features(get_words_in_tweets(tweets))
2 training_set = nltk.classify.apply_features(extract_features,
      tweets)
3 classifier = nltk.NaiveBayesClassifier.train(training_set)
```

Listing 6.1: Feature extraction and NLT classifier training

```
1  def get_words_in_tweets(tweets):
2        all_words = []
3        for (words, sentiment) in tweets: all_words.extend(words)
4        return all_words
5
6  def get_word_features(wordlist):
7        wordlist = nltk.FreqDist(wordlist)
8        word_features = wordlist.keys()
9        return word_features
10
11 def extract_features(document):
12       document_words = set(document)
13       features = {}
14       for word in word_features:
15             features['contains(%s)' % word] = (word in
                                                  document_words)
16       return features
```

Listing 6.2: Helper methods for text features extraction

The results of the 4-fold cross-validation can be seen in Table 6.3. On the listed values, it is visible that there is something not right. As written earlier, recall 100% very likely means the classifier is labeling everything as one class. That is also exactly what is happening here.

Metric	Acore
Accuracy	0.5069
Recall as negative	1
Precision	0.5069
F1 score	0.672

Table 6.3: Textblob accuracy with various handling of neutral tweets

Sci-kit Learn

Choosing classifiers Third and final module I have tested was Sci-kit. It offers lot of various classifiers as well as optimization options. To come up with the best performing classifier to my abilities, I have followed a workflow shown in the Figure 6.3.

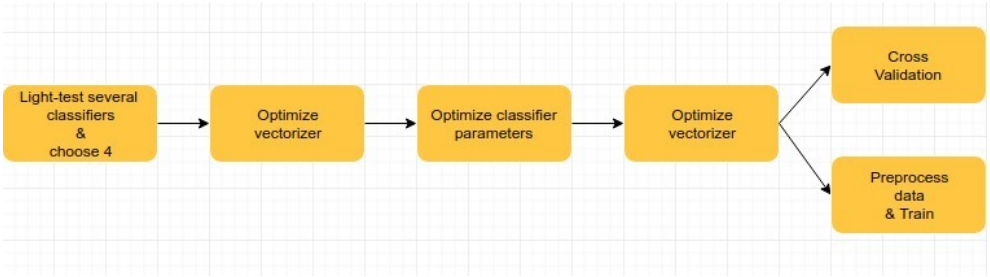


Figure 6.3: Implementation workflow of sentiment analysis using Scikit module

After manually testing and playing around with all sci-kit classifiers which are mostly used for analysis of textual data, I have decided to continue working with four of them - **Logistic Regression, Linear support vector machine and 2 Naive Bayes classifiers for multinomial models.** I had high expectations especially from Bernoulli NB as it should perform better on shorter texts. I have measured their accuracy on Sentiment140 as well as movie reviews dataset. Just out of pure curiosity I have tried to train the classifiers on movie reviews and test them on tweets contained in Sentiment140. Obviously, such approach is not correct but I wanted to see how much worse will the classifiers perform. All metrics will be shown later in Table 6.4.

Vectorizer and its optimizations: Goal of this step is to find best performing vectorizer for feature extraction. Vectorization is a step when "words are turned into numbers". While words can be transformed into numbers, an entire document can be translated into a vector. Text data vectors are usually high-dimensional because each dimension of your feature data represents one word from the document corpus.
The most common and simplest vectorizer approach is **bag of words**. In this approach, union of all the words from all documents in the corpus is the dimension of the vector which is created. That means that 800 documents with 1000 words would result in 1000-dimensional vector where every unique word has its own index in that vector. Every document is processed with so called "one-hot" encoding where basically every word in the corpus is looked up in the document and zero/one is assigned accordingly. It results to every document being represented by zeros and ones. This approach has been used in my classifier built with NLT python module described in previous section. Bag of words approach is naive in the sense that it does not distinguish the context of how a sentence or paragraph is structured. It pays attention to frequency of words but completely ignores things like position of the word in the sentence.
The solution for this could be N-grams. Using N-grams, vector is encoded for various combinations of words rather than just single words. This helps to preserve semantic meaning better but also causes the vector dimensionality increase. Bag of words also cannot recognize whether or not is a particular word unique - fact that a word is showing up repeatedly within a certain type of document can hint at its importance.
Solution for this problem is to employ **Term Frequency-Inverse Document Frequency (TF-IDF) Matrix** as a vectorization approach. That is also the vectorizer I have used for my classifier. TF-IDF allows to place more emphasis on infrequent words by assigning a weight to each word instead of a binary value. The weight is determined through a combination of the word's frequency in a document, and how rare the word is in the entire corpus. More about TF-IDF can be found in section 7.1.

Once I have decided which vectorizer class to use, there is also a question which parameters should I use it with. Sci-kit module offers GridSearchCV class which is a way how to conveniently find the best combination of all specified vectorizer parameter values. Altogether were tested and evaluated 92 vectorizers. Testing method and values examined can be seen in the following Listing 6.3.

```
    def tuneVectorizerParameters(corpus,labels):
      pipeline = Pipeline([
          ('tfidf', TfidfVectorizer(stop_words='english')),
          ('clf', LinearSVC()),
      ])
      parameters = {
        'tfidf__max_df': (0.75, 0.9),
        'tfidf__ngram_range': [(1, 1), (1, 2), (1, 3)],
        'tfidf__sublinear_tf': (True, False),
        'tfidf__stop_words':['english']
    }

    grid_search_tune = GridSearchCV(pipeline, parameters, cv=2,
    n_jobs=2, verbose=3)
    print("Searching best parameters combination:")
    grid_search_tune.fit(corpus, labels)

    print("Best parameters set:")
    print grid_search_tune.best_estimator_.steps
```

Listing 6.3: Tuning of vectorizer using GridSearchCV class

Classifier optimizations All classification algorithms have several parameters which can adjust and possibly improve their performance. Choosing the correct parameters for machine learning algorithms or so called tuning process is a field in itself and separate thesis could be written just about this. I have again used GridSearchCV class for this. For all algorithms, I have specified various values for several parameters, but even the performance of returned best parameter combination performed worse than the default parameters which are used in default constructor. Optimization of classifiers using GridSearchCV is briefly shown in the Listing 6.4.

```
scores = ['accuracy']
# Set the parameters to combine
SVC_parameters = [{'C': [1,10,100,1000],
                   'loss': ['hinge','squared_hinge'],
                   'multi_class': ['ovr','crammer_singer'],
                   'fit_intercept': [True,False]
                  }]
MultiNB_parameters = [{'alpha': [1.0, 2.0, 5.0, 10.0],
                       'fit_prior': [True, False]
                      }]
BernoulliNB_parameters = [{'alpha': [1.0, 2.0, 5.0, 10.0],
                           'binarize': [0.0, 2.0, 5.0, 10.0],
                           'fit_prior': [True, False]
                          }]

for score in scores:
    print("Tuning hyper-parameters for %s" % score)
```

```
18
19      clf = GridSearchCV(LinearSVC(), SVC_parameters, cv=5,scoring
        ='%s' % score)
20      clf.fit(train_corpus_tf_idf, y_train)
21
22      print("Best parameters set found:")
23      print(clf.best_params_)
24      print("Performance for all combinations:")
25      means = clf.cv_results_['mean_test_score']
26      stds = clf.cv_results_['std_test_score']
27      for mean, std, params in zip(means, stds, clf.cv_results_['
        params']):
28                  print("%0.3f (+/-%0.03f) for %r"
29                        % (mean, std * 2, params))
30
31      print("Detailed classification report of model trained and
        evaluated on full dev/eval sets:")
32      y_true, y_pred = y_test, clf.predict(test_corpus_tf_idf)
33      print(classification_report(y_true, y_pred))
```

Listing 6.4: Tuning of classifiers using GridSearchCV class

Training data	**Test data**	**Classifier**	**Accuracy**
Sentiment140	Sentiment140	Linear SVM	0.785
		Multinomial Naive Bayes	0.761
		Bernoulli Naive Bayes	0.773
		Logistic Regression	0.797
Movie reviews	Movie reviews	Linear SVM	0.849
		Multinomial Naive Bayes	0.807
		Bernoulli Naive Bayes	0.792
		Logistic Regression	0.823
Movie reviews	Sentiment140	Linear SVM	0.560
		Multinomial Naive Bayes	0.562
		Bernoulli Naive Bayes	0.5
		Logistic Regression	0.515

Table 6.4: Scikit classifiers accuracy on Sentiment140 dataset

As we can see, performance of these classifiers is much better than the previous one using Textblob. Despite 78% accuracy might still sound pretty low, it is important to realize that Tweets are very short text pieces which often do not offer much sentiment input to work with. Therefore is 78% quite promising performance for future work. For example, other paid services like MonkeyLearn [6] offer 81% accuracy on Tweets sentiment classification.

[6] https://monkeylearn.com/

To increase the classifiers' accuracy I also tried preprocessing. First I have kept only words which contained only letters and then I have decided to get rid also of URLs, punctuation, usernames and hashtags. Accuracy metrics can be seen in Table 6.5. Training and testing data were from Sentiment140. Performance boost was smaller that expected.

Preprocessing type	**Classifier**	**Accuracy**
Just words with letters	Linear SVM	0.785
	Multinomial Naive Bayes	0.76
	Bernoulli Naive Bayes	0.772
	Logistic Regression	0.796
Stripped URLs, punctuation, usernames, alphanumetric characters, hashtags	Linear SVM	0.786
	Multinomial Naive Bayes	0.766
	Bernoulli Naive Bayes	0.774
	Logistic Regression	0.795

Table 6.5: Scikit classifiers accuracy on Sentiment140 dataset

Neutral Tweets : At this point, after all the tuning, optimizing and cross-validation, I could finally run my analyser on the test data also provided by Sentiment140. Compared to the training data which contain just positive and negative tweets, test file contains also neutral ones. Obviously, this is a problem, since my classifiers are not familiar with the concept of neutral tweets. In current state, accuracy on a test set with neutral tweets is just 58.4% whereas on the same test set with excluded neutral tweets, accuracy was more than 81% - which is expected because it is basically the same set just with different tweets which has also been used during cross-validation.
In general, third neutral class in sentiment analysis is still causing big problems. E.g Go, Huang and Bhayani [39] considered any tweet without an emoticon to be part of the neutral class, which they themselves admitted to be a flawed approach. Kouloumpi at al. [11] trained their classifier just on hashtags and emoticons and also had to build their own neutral training dataset. Agarwal at al [10] annotated their own training dataset to contain neutral tweets and achieved very nice accuracy of 60% which is much higher than the base line of 33%. Although it is a nice result, they had training data which contained neutral data. Saif et al. [14] as well as Go, Huang and Bhayani [13] in their second article that year just stated that identifying neutral tweets is part of their future work plan. Afer realizing that doing a tree way analysis rather than just binary or qualitative (output is a score value) analysis is worth a separate thesis I have decided to train my classifier for only weighted qualitative output in interval from 0 to 1, where 1 is the most positive. Just out of curiosity, I have defined confidence thresholds for positive and negative tweets. Everything which would be between these 2

threshold levels could be considered a neutral tweet. I manually tried several values and accuracy scores from these measurements are recorded in Table 6.6. I have achieved accuracy just 7% above the baseline at most, so the lack of optimization for neutral tweets classification is obvious.

Negative threshold	Positive threshold	Accuracy
0.1	0.9	0.357
0.2	0.8	0.385
0.3	0.7	0.393
0.4	0.6	0.401

Table 6.6: Sci-kit accuracy with various handling of neutral tweets

Abstracting several classifiers under one custom classifier: I later came up with an idea of abstracting all classification algorithms all under one custom classifier. My custom classifier is able to be trained in 3 modes - as a binary classifier, multiclass classifier or a classifier outputting a float confidence score of text being positive. This approach has of course both, advantages and disadvantages as well. Advantage for sure is that possibility of False Positives or False Negatives is much lower as this requires 3 incorrect classifications instead of 1. Trade-off for this is that the confidence score of being positive is in general lower because the possibility of at least one of 3 algorithms being wrong is (obviously) higher than if there is only one single classifier.

All performance matrices of the final classifier for various datasets are displayed in Figure 6.4.

Figure 6.4: Final classifier performance for several datasets

It is clear that the best results are achieved with movie or application reviews because these texts offer more features to draw the sentiment from because they are longer texts. We can also see that training datasets built from shorter texts (My twitter dataset or SO labeled dataset created and used in Bin Li's paper) have worse performance, especially recall. With my own dataset, its size might be causing this as it contains only 84 labeled entries. I concluded that using Sentiment140 as a definitive training dataset is a good choice as its performance is just slightly worse compared to reviews but it is still authentic Twitter data which I am going to analyse the most.

6.6 Data to analyze

I found Twitter data very suitable for sentiment analysis and therefore decided to analyse only tweets and use SO and Reddit later in Chapter 7 about recognizing bugs in discussions.

Using the steps described in Section 4.2.1, I have downloaded tweets with following time conditions/patterns:

- days prior and after release
- weekly

6.7 Results

6.7.1 Results for weekly collected data

First and the most simple analysis was to compare sentiment of tweets collected with period one week. I have also added the sentiment of some projects' Reddit discussions (keep in mind that classifier is trained on tweets, so this is not the best approach). Results can be seen in Table 6.7 and Figure 6.5.

Project	Twitter average sentiment	Reddit average sentiment
NodeJS	0.697	0.633
EmberJS	0.709	0.69
VueJS	0.715	0.60
Symfony	0.727	*
AngularJS	0.727	0.618
CakePhp	0.703	*
Bower	0.641	No subreddit
Laravel	0.701	*
Gulp	0.578	No subreddit
Yii	0.658	*
Bootstrap	0.713	0.635

* Reddit changed its API and submissions endpoint is not supported anymore

Table 6.7: Average sentiment of tweets collected weekly

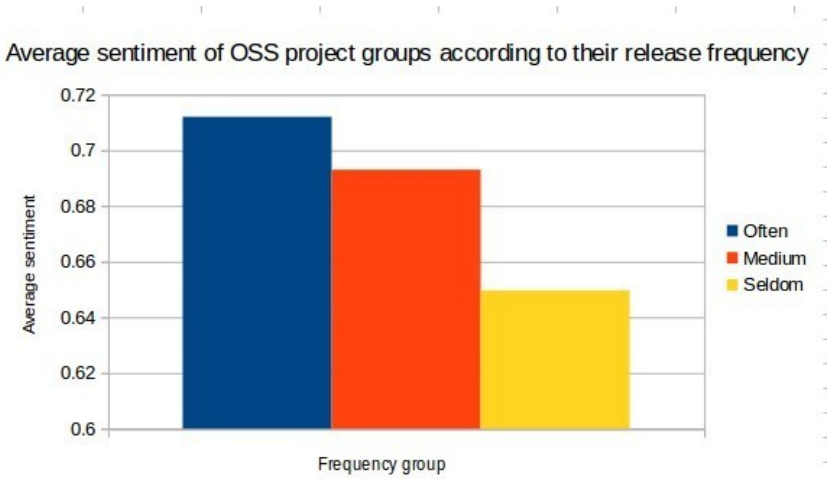


Figure 6.5: Average sentiment of OSS project groups according to their release frequency

In both of previous analysis, I have used the simplest aggregation operation which is average. Replacing this one with median, modus or some more advanced aggregations could potentially yield different results but because each group is represented just by 3 or 4 OSS projects, losing any data would actually be a significant part of the overall analysed dataset. Using more projects could be potentially a part of future work.

6.7.2 Immediate sentiment change with releases

Next step was to do the analysis on the smaller scale and see if sentiment changes between tweets collected 3 days prior and post-release. Results are shown in Table 6.8.

Project	Before	After
NodeJS	0.707	0.709
EmberJS	0.719	0.719
VueJS	0.694	0.708
Symfony	0.73	0.729
AngularJS	0.718	0.719
CakePhp	0.71	0.702
Bower	0.634	0.638
Laravel	0.683	0.696
Gulp	0.608	0.573
Yii	0.643	0.606
Bootstrap	0.713	0.693

Table 6.8: Average sentiment of tweets 3 days before and after releases

From the data shown in table is visible, that there is no big sentiment shift caused by releases on the small time frame. The biggest noticeable pattern (nicely displayed in Figure 6.6) is 4% decrease in sentiment of seldom releasing projects. In groups which release often or normally, sentiment change was very small.

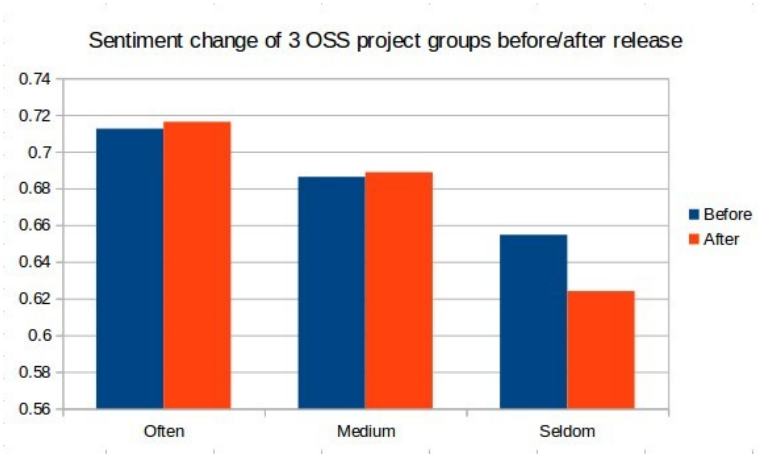
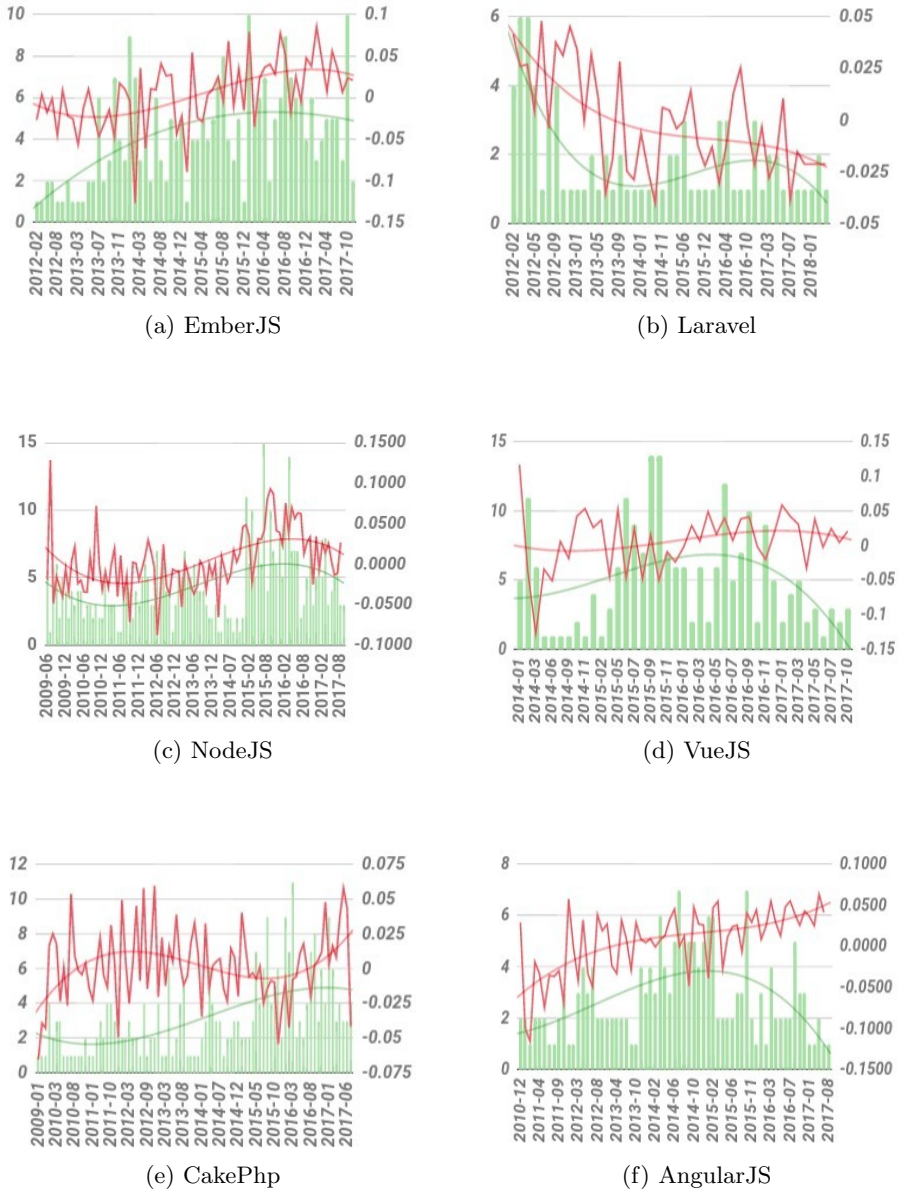


Figure 6.6: Change in sentiment for all 3 OSS groups before/after release

A clear pattern to notice in both figures (6.5 and 6.6) is, that more often projects release, better their sentiment gets.

6.7.3 Cross-correlation between releases and sentiment change

Monitoring sentiment change in regards to releases can be done in bigger scale as well. In this section, instead of using tweets from several days before and after release, I have used the data collected weekly and analysed their relationship with number of releases per month. In the Figure 6.7 are plotted sentiment difference from the average and the release count per month (red line represents sentiment while green stands for release count).



(a) EmberJS

(b) Laravel

(c) NodeJS

(d) VueJS

(e) CakePhp

(f) AngularJS

Figure 6.7: Sentiment difference from average and release counts per month

Looking at linecharts gives some feeling and intuition whether and what type of relation there is. When working with time series, it often needs to be determined whether one series causes changes in another or vice versa and also measure this relationship quantitatively. To find this relationship, measuring a cross-correlation and finding a lag is one way how to do it. Lag represents the effect of change in one data series on the other several periods later.

To ensure a cross-correlation calculation makes sense, first I have to determine, whether the data are stationary. A stationary time series is one whose properties do not depend on the time at which the series is observed[40]. More precisely, if y_t is a stationary time series, then for all s, the distribution of $(y_t,,y_{t+s})$ does not depend on t.

To determine whether my data are stationary, I have used the Dickey-Fuller test method of tseries package in R. Results can be seen in the Table 6.9 and 6.10.

Stationarity test of web frameworks sentiment data		
Framework	Dickey-Fuller	p-value
NodeJS	-2.6775	0.2964
AngularJS	-3.883	0.0199
EmberJS	-4.0783	0.0199
VueJS	-3.438	0.0646
CakePHP	-3.480	0.04847
Laravel	-2.57	0.3431
Symfony	-4.3979	0.01

Table 6.9: Stationarity test of sentiment

Stationarity test of web frameworks release count		
Framework	Dickey-Fuller	p-value
NodeJS	-2.896	0.205
AngularJS	-2.547	0.353
EmberJS	-3.297	0.0802
VueJS	-2.158	0.511
CakePHP	-3.224	0.08915
Laravel	-2.368	0.425
Symfony	-2.218	0.488

Table 6.10: Stationarity test of release counts

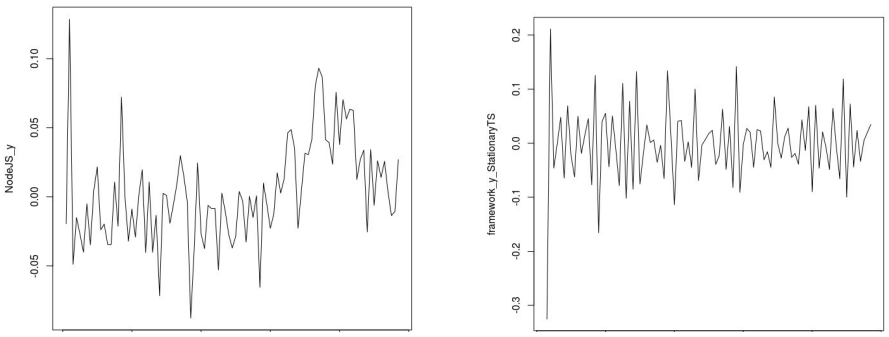
As we can see, p-values are always higher than 0.05 what indicates non-

stationarity of the data, therefore I cannot calculate the cross-correlation on them in this state. To transform non-stationary data into stationary, 2 approaches can be used. These are differencing and transforming. I have taken data series and differenced the values in Listing 6.5. I have executed both, seasonal differencing and stationary differencing.

```
Differencing <- function(x,y)
{
 framework_x_seasdiff <- diff(x, differences=1)  # seasonal
    differencing
 framework_x_Stationary <- diff(framework_x_seasdiff,
    differences= 1)
 framework_y_seasdiff <- diff(y, differences=1)
 framework_y_Stationary <- diff(framework_y_seasdiff,
    differences= 1)
 return(list(framework_x_Stationary,framework_y_Stationary))
}
```

Listing 6.5: Used differencing method in R

New differenced values do appear to be stationary in mean and variance, as the level and the variance of the series stays roughly constant over time. Sentiment for NodeJS before and after differencing can be seen in Figure 6.8.



(a) Before differencing (b) After differencing

Figure 6.8: NodeJS monthly sentiment values

Same procedure needed to be done with the "number of releases per month" data and afterwards. Then, cross-correlation could be executed. For this task I have used ccf method in R which implements Pearson's correlation calculation method. Results for all 7 OSS projects can be seen in Figure 6.9.

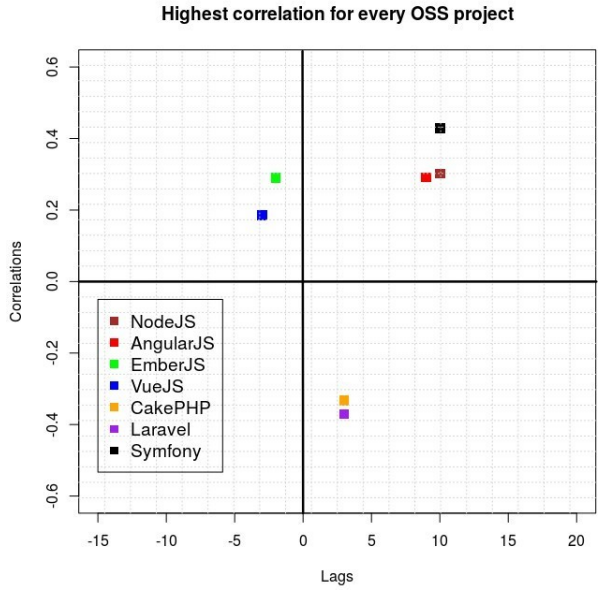


Figure 6.9: Highest correlations for every OSS project

I think there could be made a counter-point about whether the transformation to stationary data is really needed here, I have also included Figure 6.10 where are the results with the raw data.

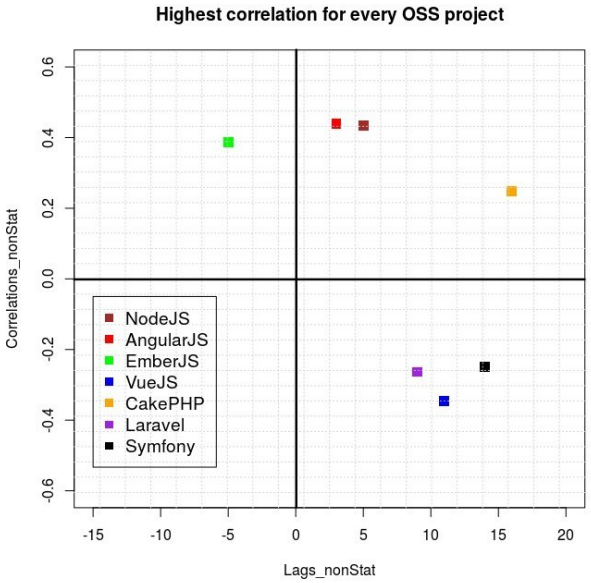


Figure 6.10: Highest correlations for every OSS project (non-stationary)

Results interpretation: Maximal project correlations happen to occupy 3 of 4 possible quadrants. Each quadrant represents a different relationship

between number of releases and sentiment change. 3 out of 7 projects stayed in the same quadrant for both - stationary and non-stationary data.

- **I. Quadrant**(Positive correlation + positive lag) - Increase of release count increases a sentiment in the upcoming months
- **II. Quadrant**(Positive correlation + negative lag) - Increase of sentiment increases a release count in the upcoming months
- **III. Quadrant**(Negative correlation + positive lag) - Increase of release count decreases a sentiment in the upcoming months
- **IV. Quadrant**(Negative correlation + Negative lag) - Increase of sentiment decreases a release count in the upcoming months

No project ended up in quadrant 3 so the potential result is that increasing release count definitely doesn't hinder project's sentiment. It can boost it but it is also not guaranteed.

6.7.4 Commits count within releases

Initially, I thought that to modify project to take into account a size of the release (amount of commits) will be a straightforward task but I've eventually encountered several unexpected problems on the way.
I intended to extend my previously used method from Section 5.4.1 which uses Git API tags endpoint to get the release dates. Unfortunately I wasn't able to find number of commits in the returned objects. JSON object returned from API has following structure:

```
{
  "url": X,
  "assets_url": X,
  "upload_url": X,
  "html_url": X,
  "id": X,
  "tag_name": X,
  "target_commitish":X,
  "name": X,
  "draft": X,
  "author":{},
  "prerelease": X,
  "created_at": X,
  "published_at": X,
  "assets":[],
  "tarball_url": X,
  "zipball_url":X,
  }
```

I have done some extra searching but did not want to spend extra time so I decided to go the way I knew will work. Instead of using API to get the commit counts, I crawled GitHub UI page of each release and extracted information directly from page source code. Each release details page provides information how many commits behind the current HEAD the release is. The difference in this number between two following releases represents count of new commits for a release. Results of simple tabular substraction with spreadsheet formula needed to be manually corrected because projects often release several branches parallel and therefore substraction from the previous release was not always the correct one.

Eventually, I got correct number of commits for every release and could execute the same cross-correlation analysis described in the previous chapter, but this time instead of releases count, I have explored relationship between sentiment and commits count. One possible flaw in the commit count data are the pre-releases. I treated them as normal releases because they do offer new features but those very same commits are then counted in the official releases later on.

After getting the data ready I performed a stationarity test for commit counts. Sentiment values are the same as before with count of releases. Table 6.11 shows the results.

Stationarity test of web frameworks commit counts		
Framework	Dickey-Fuller	p-value
NodeJS	-7.0239	0.01
AngularJS	-2.547	0.3531
EmberJS	-3.2764	0.0831
VueJS	-2.9748	0.1886
CakePHP	-3.655	0.03283
Laravel	-2.919	0.2084
Symfony	-4.8461	0.01

Table 6.11: Stationarity test of commit counts

I see that there are again several data series (AngularJS, EmberJS, VueJS, Laravel + NodeJS because of unstationarity of sentiment data) which are not stationary so exactly as before with release counts, I had to transform the data. After that, Pearson's cross correlation was calculated. Results for all 7 OSS projects can be seen in Figure 6.11

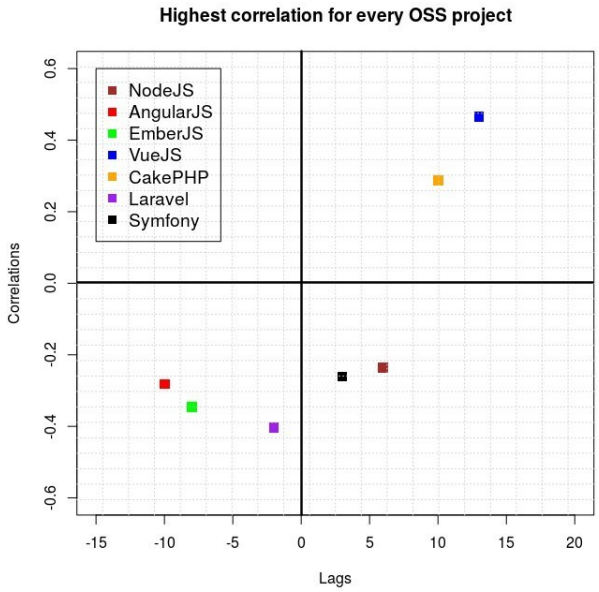


Figure 6.11: Highest correlations for every OSS project

If I would skip the step of making the data stationary, results would again look completely different 6.12.

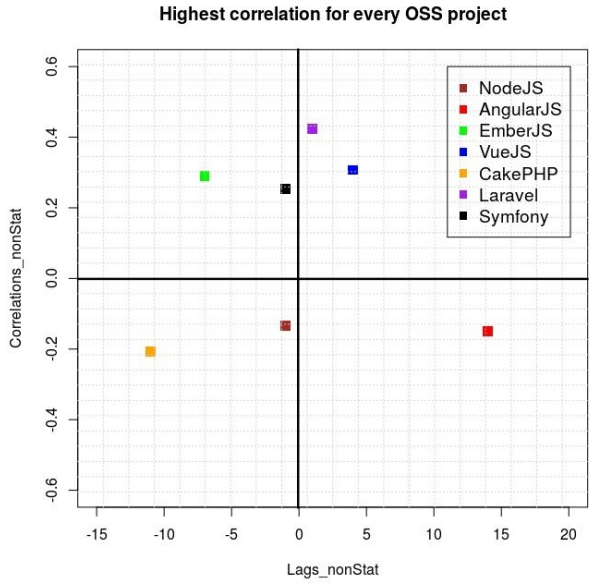


Figure 6.12: Highest correlations for every OSS project (non-stationary)

Chapter 7

Linking bug repositories and social media

My goal of linking bugs with social media might be somewhat similar to nowadays very active field of bug report duplicate discovery but has also a lot common with topic modelling and general text similarity algorithms.
To link any two texts based on their content, there is an obvious need for understanding what the text features are. To do this, there are several ways how to calculate text (string) similarity value or one can even execute so called "Topic modelling" algorithm and try to connect documents based on their matching topics.
Topic modelling is a method used to organize and summarize large textual information. It is used to discover hidden topical patterns and annotate documents according to these topics. It can also be described as a method of finding group of words (i.e topic) from in a text that best represents the information in the collection.
The obstacle of using a topic modelling for my case is that output is not granular enough to differentiate among similar texts which are all from the very same domain. Because the output provided by topic modelling was not enough to link particular items, I searched for, found and considered several alternative approaches.

7.1 Approaches

There are many ways and approaches to find out whether 2 texts share some common topic. Most of them are to some extent very similar as the general rule is to extract textual features and compare them using statistical approaches. Common way to do this is to transform documents into vectors and then compute cosine similarity between them. These text transformations are implemented in several Python packages.

I have tried following:

1. String similarity using NLTK
2. String similarity using Scikit-learn

NLTK: First step in calculating similarity was to tokenize the text. NLTK offers several types of tokenizers with various outputs. Text tokenization can operate on various levels and the structure of input has to be considered. As I am comparing the whole documents and do not want to consider sentences as standalone objects, I have chosen to use *nltk.tokenize.wordtokenize* method instead of e.g. *nltk.tokenize.senttokenize*.
Next step after the document is tokenized is to stem the words. Stemmers remove morphological affixes from words, leaving only the word stem. Once again as with tokenizers, there are several stemmers implemented within NLTK. After doing a short research I have come to conclusion that as far as I use the same stemmer for both, Git issues and SO/Reddit entries, it should not play any major role in results.
Next step is getting rid of stop words. These usually refer to the most common words in a language, but there is no single universal list of stop words used by all natural language processing tools. The set of stop words defined for my NLTK version had a size of 153.
Listing 7.1 illustrates the implementation of the described similarity calculation procedure.

```
1     tokens = word_tokenize(text)
2     words = [w.lower() for w in tokens]
3
4     porter = nltk.PorterStemmer()
5     stemmed_tokens = [porter.stem(t) for t in words]
6
7     # removing stop words
8     stop_words = set(stopwords.words('english'))
9     filtered_tokens = [w for w in stemmed_tokens if not w in
      stop_words]
10
11    # count words
12    count = nltk.defaultdict(int)
13    for word in filtered_tokens:
14      count[word] += 1
15    return count;
```

Listing 7.1: Text similarity implementation with NLTK

After the previous 3 steps are executed on both documents, 2 vectors from all words from both documents are created. Each documents then sets the counts of words it contains and a cosine similarity of these two "count vectors" is calculated. This similarity calculation is a basic similarity calculation and could definitely be optimized. For example, it does not

analyse and consider role and position of word in a sentence (POS tagger would be required here).

Scikit-learn and TF-IDF: There are several ways to assess the importance of each feature by attaching a certain weight in the text. The most popular ones are: feature frequency (FF), Term Frequency Inverse Document Frequency (TF-IDF), and feature presence (FP) [41]. My next similarity checker I have implemented was using Scikit-learn module and TF-IDF vectorizer. While BoG only takes into consideration the frequency of words in a document TF-IDF reflects how important a word is for the particular document. For a word to have high TF-IDF in a document, it must appear a lot of times in said document and must be absent in the other documents. It must be a signature word of the document.

Term frequency represents how often is the word present in the said document. Simplest approach is the raw count.

$$tf_{count}(t, d) = f(t, d)$$

Other options include term frequency adjusted for document length, logarithmically scaled frequency or augmented frequency to handle bias towards longer documents.
Inverse document frequency is a counterweight factor which diminished importance of terms that appear in the set very often in the document set and therefore increases the weight of terms that occur rarely.

$$idf(t) = log\frac{N}{df(t)}$$

The final weighing scheme combines term frequency and inverse document frequency.

$$tfidf(t, d) = tf(td)xidf(t)$$

Listing 7.2 shows my very simple implementation of TF-IDF similarity checker using Scikit-learn module.

```
1     def getSimilarity(self,text1, text2):
2       tfidf = self.vect.fit-transform([text1, text2])
3       return (tfidf * tfidf.T).A
```

Listing 7.2: Text similarity implementation with Scikit using Tf-Idf model

Stack Overflow questions preprocessing: Online texts usually contain lot of noise and uninformative parts such as HTML tags, scripts and advertisements [41]. Before running implemented similarity algorithms, I have decided to preprocess the stack questions and get rid of code snippets within *<pre><code>* tags and hypertext links. Especially snippets could potentially effect (increase) the similarity score if kept in the text.

7.2 Available data

Git issue reports: Using the approach described in the section 5.4.2, I have downloaded 96,651 issue reports from Git. 25,978 of those are labeled as bug or similar. If I would use all the data, the calculations would take too much time. That is the reason why I decided not to work with all the data and rather just picked several projects of interest. The bug counts among this projects is displayed in the Table 7.1.

Framework	Bug count
NodeJS	1615
AngularJS	2225
EmberJS	1284
VueJS	353
Aurelia	73
Bower	155

Table 7.1: Bug count per project

Stack overflow questions: SO mining has been described in Section 4.2.1 and I have downloaded 5,847 questions. There are thousand questions for AngularJS, NodeJS, Bower, Ruby on Rails and VueJS each and EmberJS has only 847 questions. Downside is, that despite having a lot of questions, it does not necessarily mean that each and every one of them talks about some known bug. Actually, opposite is true as out of all those questions only very tiny percentage does (can be seen in Table 7.2). Because the projects I worked with till now had just so few questions linked to its own issues, I had to increase the number of bugs in the dataset. Since StackApi is limited to 10 thousand requests per day, I decided to from opposite end and crawled through all closed git issues of projects and checked if there is a SO question linked to them. If that was the case I have saved both, git issue and SO question and that way increased the size of my dataset (can be seen in Table 7.3)

Framework	Question count	Bug count
NodeJS	1000	2
AngularJS	1000	0
EmberJS	847	2
VueJS	1000	0
Aurelia	646	4
Bower	1000	5

Table 7.2: Bug count per project

Framework	Bug count
NodeJS	2
AngularJS	46
EmberJS	20
VueJS	1
Aurelia	3
Bower	4
Django	13

Table 7.3: Extra bugs added to the dataset

Reddit dialogues: Reddit subreddits mining has been described in the same subsection as SO mining. After I have learnt that there is too much data online, with Reddit I downloaded only submissions with any Git issue link (not necessarily own issue). Results of this process are shown in the Table 7.4.

Framework	Submissions count
NodeJS	108
AngularJS	43
VueJS	20
EmberJS	13

Table 7.4: Reddit submissions counts

7.3 Similarity results

Linking items and in general finding similarity among these already area-specific texts proved to be a problematic task. To test my algorithm I decided to compare 89 SO questions with its matching Git issue and 4x one random git issue of the project. Results are plotted in Figure 7.1.

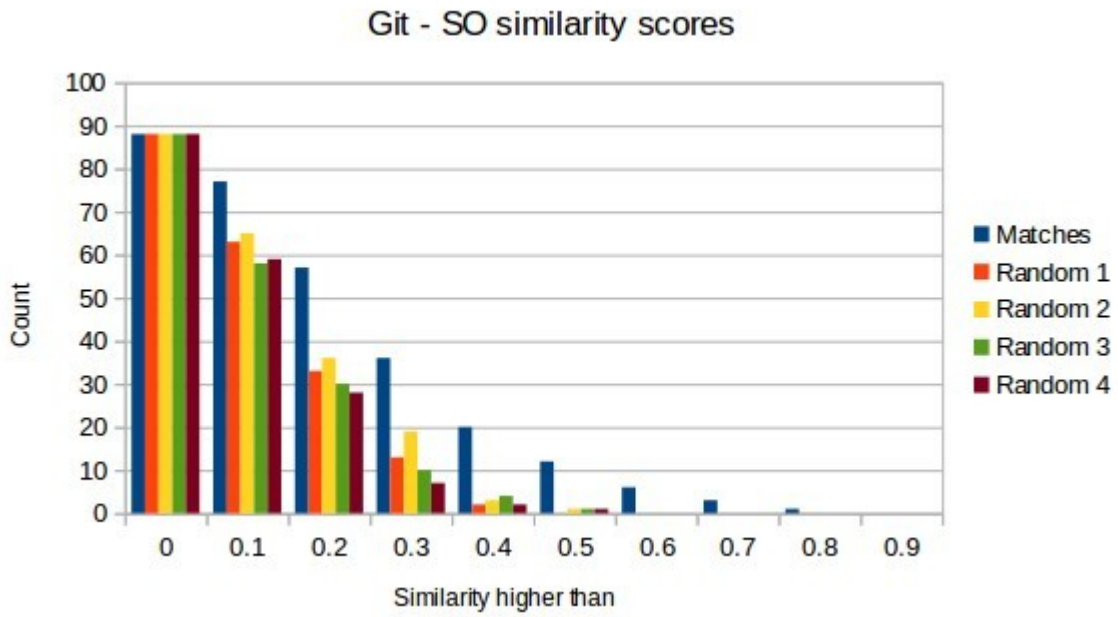


Figure 7.1: Similarity distribution of 89 matches and four data series of 89 random pairs

Results very clearly show that there is a particular similarity score, which is very hard to pass for two unrelated items. This score is obviously not constant and it depends on implementation of the similarity calculation. For my NLTK BoW algorithm used in Figure 7.1, the threshold is 0.4 or 0.5. Choosing the value of threshold would also depend on the desired characteristics of a classifier and the prioritized metric. For example if a precision would be more important than recall, the ideal threshold would be 0.7 as it is the value which has never been passed by any of 356 non-matching pairs. The detailed table with the values for a Figure 7.1 can be seen in appendix.

I have executed the same steps using the TF-IDF approach and although the values are different (as expected), they follow the same pattern. Figure 7.2 demonstrates these results and detailed values can again be found in appendix.

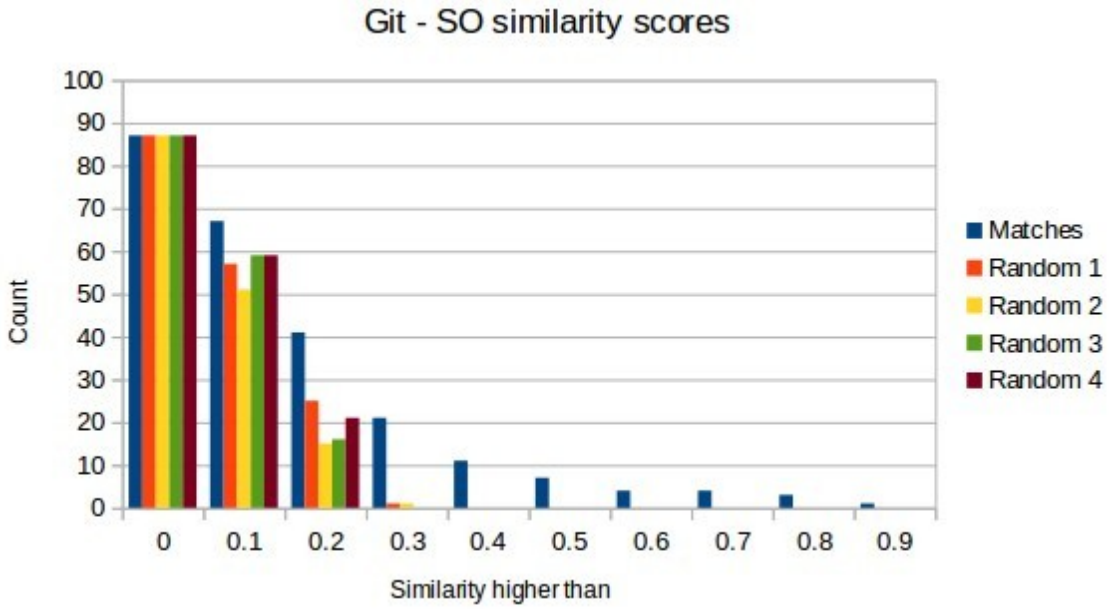


Figure 7.2: Similarity distribution of 89 matches and four data series of 89 random pairs

Despite clearly seeing possible threshold values, an algorithm which would label all the pairs above this threshold as matches would achieve very bad recall around 30%. On the other side, precision would be 100%. The potential problem is that the real-world ratio between matching and non-matching pairs rises exponentially with every new item on any side (Git issue, SO question). Testing the algorithm on bigger amount of data in the future could answer the question whether the threshold around 0.3 really proves as a unbreakable resistance for unrelated Git-SO pairs.

Although I was not successful to such extent that the output are Git-StackOverflow or Git-Reddit True Positive pairs, I have pointed out some interesting results and data relationships.

7.3.1 Stack Overflow

The average similarity (using NLTK approach) between SO questions talking about particular issue and that particular issue description is 0.316 without body preprocessing and 0.292 with body preprocessing. The distribution of similarities in buckets by increased by 0.05 can be seen in histogram in Figure 7.3

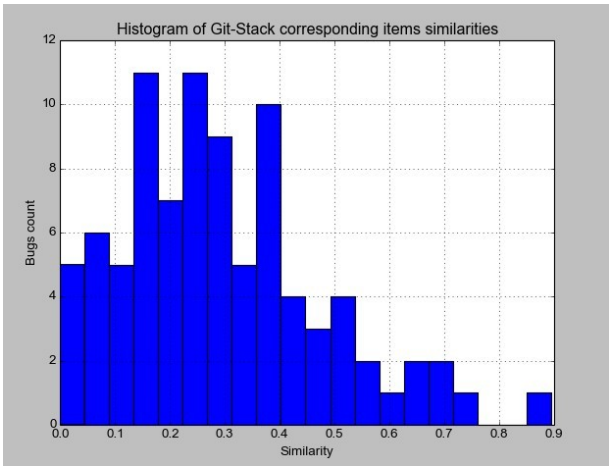


Figure 7.3: Histogram of similarities distribution among git issues and their matching SO questions

For random SO questions, amount of comparisons to Git issues needed to be limited. If every SO question would be compared to every Git issue, time of computation would exceed timeframe of this thesis. Every SO question was therefore compared to 20 random Git issues and resulting average similarity scores are in following figures.

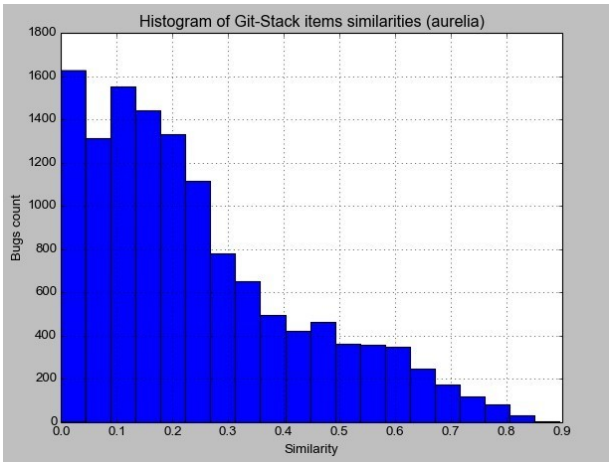
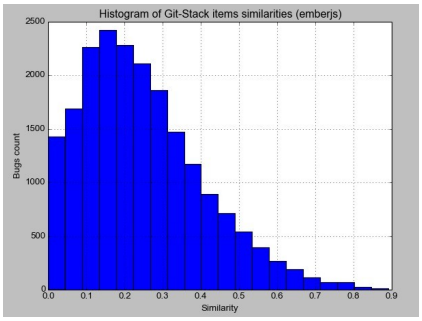
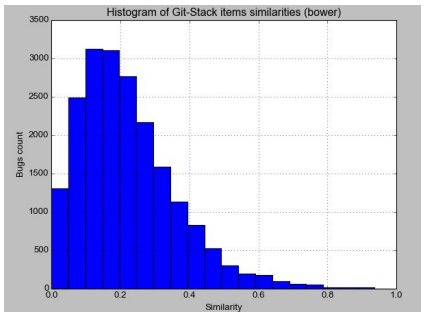


Figure 7.4: Histogram of Aurelia SO questions and random git issues. Average similarity was 0.244

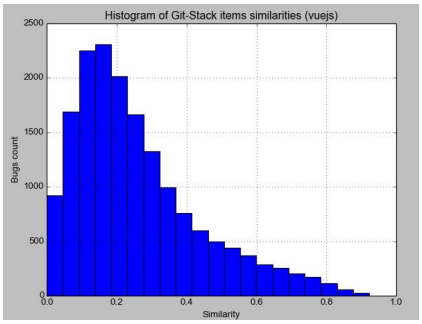


(a) EmberJS average similarity - 0.247

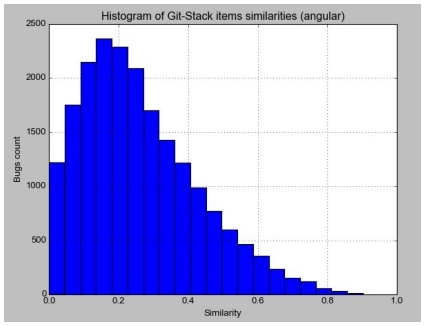


(b) Bower average similarity - 0.217

Figure 7.5: EmberJS and Bower similarity histogram



(a) VueJS average similarity - 0.255



(b) AngularJS average similarity - 0.258

Figure 7.6: VueJS and AngularJS similarity histogram

Table 7.5 illustrates results of comparing Git bugs descriptions with SO questions talking about the same project issues and general issues. From values displayed, it is apparent that the difference between matches and random pairs is not very big. This is probably because all the texts about a particular project are already very specific and similar in their nature anyway.

Framework	**Own issues similarity**	**20 random issues similarity**
NodeJS	0.265	0.243
AngularJS	0.241	0.260
EmberJS	0.282	0.246
VueJS	0.261	0.258

Table 7.5: NLTK similarity values for SO questions

7.3.2 Reddit

Here I have calculated the similarity between the bug description and either particular comment in the Reddit discussion which mentioned the bug or the whole discussion itself. Using NLTK BoW approach, average similarity score for all considered projects (NodeJS, AngularJS, VueJS and EmberJS) was 0.481 for the whole discussion and 0.368 for the comment itself. Scikit If-Idf values were 0.263 and 0.207 respectively. Detailed scores for each project can be found in Table 7.6 for NLTK implementation and Table 7.7 for Sci-kit. Subreddit for EmberJS did not reference any of its own bugs.

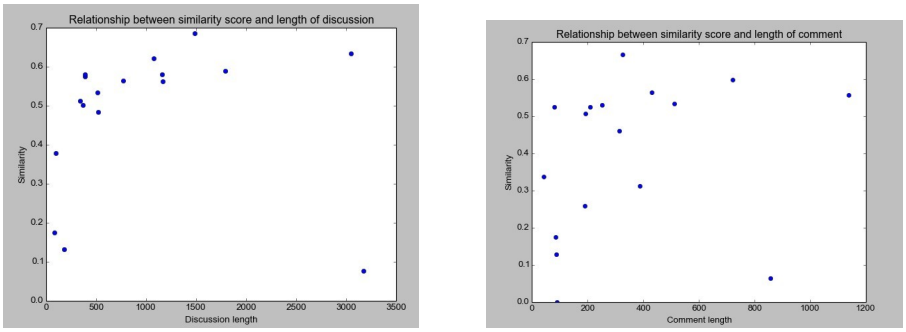
Framework	**Bug comment**	**Whole discussion**
NodeJS	0.447	0.507
AngularJS	0.306	0.57
VueJS	0.359	0.380

Table 7.6: Reddit NLTK similarity values

Framework	**Bug comment**	**Whole discussion**
NodeJS	0.255	0.328
AngularJS	0.168	0.278
VueJS	0.209	0.208

Table 7.7: Reddit Scikit similarity values

Both similarity calculations indicate that the semantic meaning of the bug is better expressed in the whole discussion rather than just the particular comment which referenced the bug. This made me question if it could be generalized that longer the text is, more similar it is to actual bug description. I have plotted a relationship between similarity score and length for Reddit in Figure 7.7 and the same for Stack Overflow discussion can be seen in Figure 7.8.



(a) Discussion lengths (b) Comment lengths

Figure 7.7: Text lengths and similarity scores with matching issues

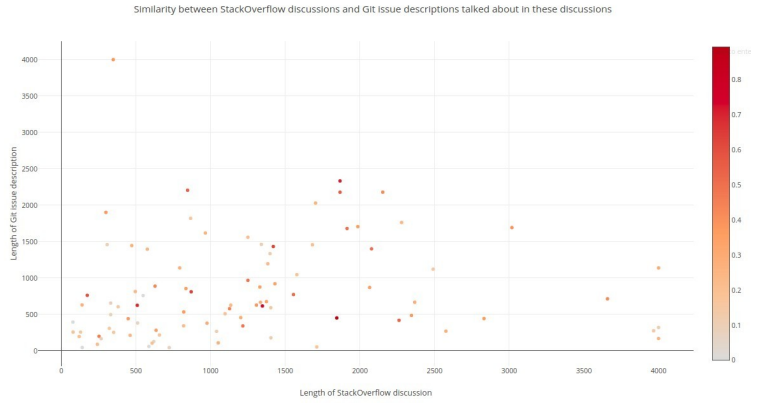


Figure 7.8: Git and SO discussion lengths and similarity scores with the issue

7.4 Using GIT labels

One more considered approach to recognize an issue's topic was using the GIT labels.
These default labels come as a big help in directing the project and targeting the most important issues, but they do not say much about the nature of the issue itself.
The custom tags tell are used to specify the part of the project, where the issue is located but they still do not give any semantic information about the issue itself. That is the reason why this approach was rejected.

Chapter 8

Answers and Discussion

8.1 Research questions

In the thesis, I have tried to answer following questions:

- RQ_1: Do the OSS projects which release more often get general better sentiment score on social media?
- RQ_2: Does a release have an immediate effect on sentiment?
- RQ_3: Is there a correlation between sentiment change and size of the release (number of commits) ?
- RQ_4: Is it possible to link social media entries to their respective bugs which they talk about?

After I examined, analysed and researched every question I have sent my results to 200 field experts. To ensure that their insights are as applicable as possible, these experts are actually OSS developers and late commit authors in projects analysed in the thesis. I extracted their email addresses from commit details from GitHub. This process was done manually as I figured out that creating script for this would cause unnecessary overhead. Sadly, only 10 of these developers replied, but even they offered very insightful remarks.

8.2 Discussion

8.2.1 Release frequency and sentiment (RQ_1)

Anwser on this question was provided in Subsection 6.7.1. Results obtained in this section are clearly in favour of this hypothesis. There is a significant sentiment difference between all 3 sentiment groups I have worked with. Hypothesis also found some support among OSS experts as two thirds of

them answered the question they would expect higher sentiment for more releasing frameworks. On the other side, one third is big enough not to be ignored. Some answers questioned the release frequencies of buckets and some pointed out that it just comes to quality of code and not the release frequency:

- *"I don't necessarily agree that this is a split yes or no answer. A project could release constantly, but if they're constantly shipping buggy code, it'll have a negative sentiment overall."*
- *"Releasing less then once per month doesn't seem negative to me. Think you'd need a bigger range like once every 1-3 months, once per 6-12 months etc."*
- *"More often releases generally means that lots of patches for bugs are coming out, which leads to happier devs and a happier community."*

8.2.2 Release effect on sentiment (RQ_2)

What I have hoped for while formulating this question was an immediate sentiment change among users in the very first days after a release. This case was examined and partially validated by the results presented in Subsection 6.7.2. As always, it depends on the way how results are interpreted. One of 3 release frequency groups registered a significant sentiment change of sentiment between tweets prior and past a release. On the other side, only 2 of 11 projects did (both were the part of seldom releasing group). This obviously leads to question whether the chosen projects were just the overall outliers or the seldom releasing projects do really experience disappointment after releases (as the users expect more from them, when they do not happen as often). Also field experts were not able to agree on the expected outcome. 55% said it is a surprise that sentiment got worse while 45% said it does not surprise them at all. Interesting point here was the comment about releasing fixes which actually destroy the workarounds around bugs:

- *"Projects that don't update frequently are giving more time for buggy behavior to cement into the community and have code baked around it. This leads to breakage when the bugs are patched."*
- *"Generally speaking I think it's difficult for most projects to keep every developer happy, and typically after a release people are reminded about the project and therefore will be more vocal. I would expect sentiment and volume of tweets to reduce as time passes from the release date."*

8.2.3 Relation between release and sentiment (RQ_3)

This question was studied in subsections number 6.7.3 and 6.7.4. The results have shown that there is definitely a difference between counting releases and

counting commits numbers within releases. The real question is if there is any pattern in the commits/sentiment relationship. Sadly, I have not found any strong relationship here, as the projects ended up in all 4 quadrants - commits positively/negatively effecting sentiment, sentiment positively/negatively effecting commit count in the next months. 80% experts agreed that the size and type of the release play a role in user sentiment while the rest did not notice any sentiment changes:

- *"I generally see higher sentiment on minor releases (every 6 weeks, include new features) than with patch releases (irregular, frequent, only fix bugs) because people are excited for new features."*
- *"A high number of commits around a release could indicate that there are regressions (especially recently with ember-data :( ). But generally, I'm surprised that more commits for ember decreases sentiment. "*

8.2.4 Links between social media and version-control or bug-tracking system (RQ_4)

This task proved to be very difficult and I did not achieve the results I hoped for in regards of linking particular matches together. Whole work is described in Chapter 7. But while working on this task I found some interesting relationships in the data. During the analysis, I have noticed that the similarity calculated for the whole Stack Overflow or Reddit discussion is higher than just a similarity of the particular comment referring the bug. Response in the survey approved of this as more than 77% would expect this. Respondents agreed that in the whole discussion, the knowledge of a group is shared and the likelihood of particular comment diverging is higher than the whole "averaged" discussion. Experts also provided several interesting answers:

- *Overall discussion of a problem will lead to group knowledge being shared, whereas the initial comment is just the knowledge of one person. There is a higher likelihood that any individual comment will diverge from any other comment about a particular topic, but the average evens out because all information is shared over time."*
- *"The shorter the SO discussion, the more likely it is clearly an issue recorded on git(hub - I assume)... maybe?"*
- *"Seems like longer descriptions are more likely to be similar, maybe because they are copy pasted? I would check for a bias in the underlying similarity finder though "*
- *"Seems like there's a pretty high correlation between similarity of short length GitHub issue discussions and longer SO discussions. Could be*

that bugs are reported on SO first, then discussed, then re-reported and opened on GitHub once the underlying cause is found."

I've plotted similarities of Stack Questions with 20 random Git issues for every project in Subsection 7.3. Among these histograms I mixed one which showed just similarities of real Git-Stack matches talking about the very same issue. More than 55% of people have chosen the correct histogram when asked to choose the one which represents matches. Compared to statistical average of 25%, this is a huge increase so there definitely is a textual similarity between Git-Stack matches.

Also some extra general questions were in my survey and got very nice response regarding accuracy of my analysis. 55% of respondents said that community sentiment in their particular projects is/was accurate to my findings. 45% responses stated they have not been contributing to the project long enough to say. Noone expressed direct disagreement with my results.

Chapter 9

Future work

The future work of probably any machine learning related implementations consists mostly of performance tuning. I would like and certainly will look into emoticon analysis as Hogenboon at al. [15] showed very promising results. Sentiment analyser could be improved by considering more values during GridSearchCV tuning and also the POS tagger could considerably increase the performance. Also the process of choosing the best performing classification algorithms comparison to decide which 3 I will end up choosing for my custom analyser could be improved. I have compared their accuracy without any prior fine-tuning so it is possible some would possibly perform better with specific combination of parameters rather than the ones I have chosen. Another option is to build a custom training dataset from tweets talking just about web development. All these mentioned potential are still just a basic level of sentiment analysis.

The doubt of choosing correct programming language/module/algorithm/-training dataset and so on was ever-present during working on the thesis. Possible approach to tackle so many changing variables could be to create a variation points model and examine which parts of an algorithm are currently not using the best performing technology for this use-case.

There is also another future work which could be done on top of already done work in this thesis. As the cryptocurrencies saw a big boom in the last couple of years, I thought my sentiment analyser could be used in this area as well. Without almost any tweaks, I have taken it and applied it on tweets about Bitcoin. It is very interesting to see how the sentiment changes in regards to price action. Figure 9.1 shows the sentiment change over the course of last year. Price change and couple of my remarks I found interesting are also present in the graph.

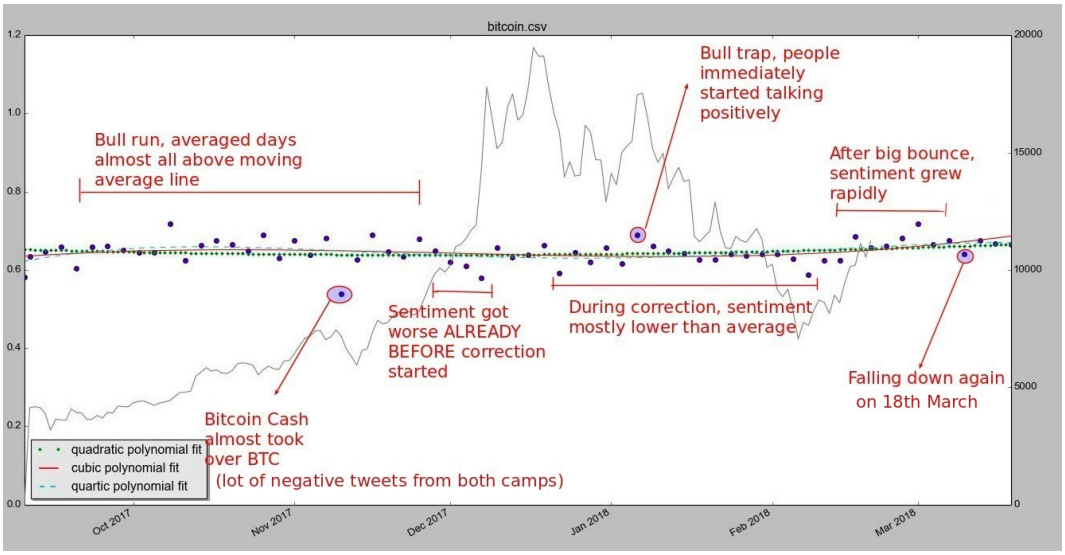
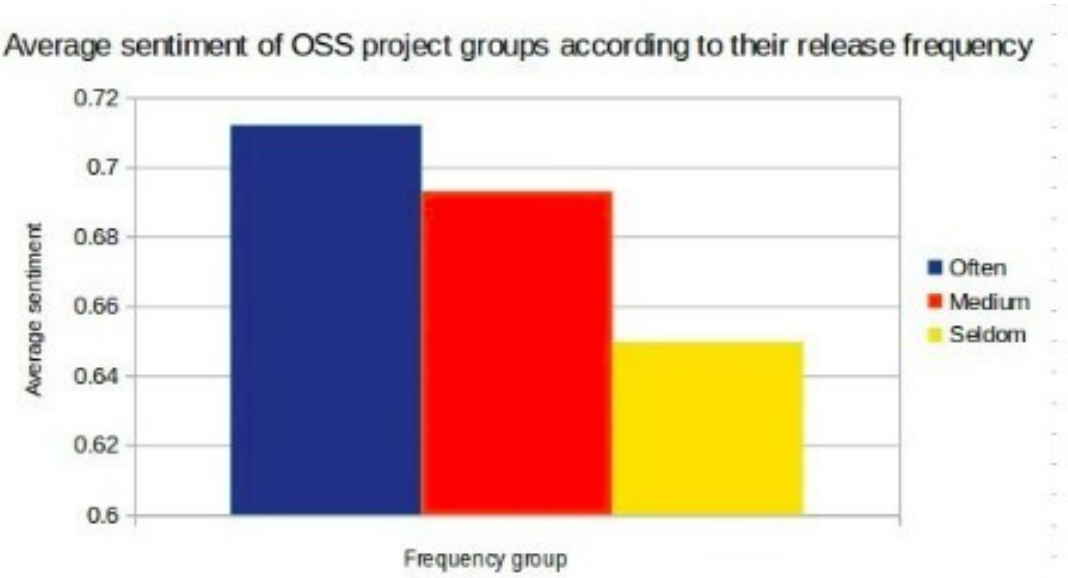


Figure 9.1: Change of sentiment in bitcoin tweets

Another use I have found for my classifier during writing of this thesis was once again in cryptocurrency area. Me and my friends noticed that some small market cap coins react to good news published on their subreddits by huge value gain in just several hours. We have crawled coinmarketcap.com website which contains list of all coins with links to their respective subreddits. Then using PRAW library, we are getting all new submissions and send them to my analyser to classify them as one of 5 classes - partnership, article, exchange, roadmap and noise. These has been then continuously posted to our Slack channel as can be seen in Figure 9.2. The whole process is happening in real time. For this use case, I had to change my classifier from binary to multiclass. That meant getting rid of some classifiers and adding some new ones which are able to do so, but in general, the structure of the analyser stayed the same.

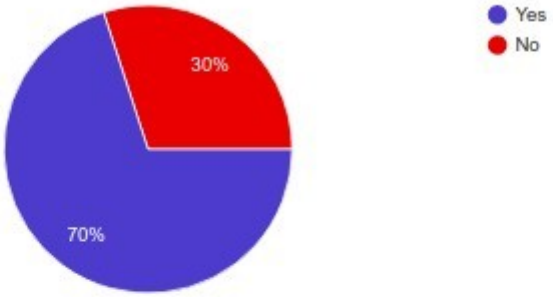
Figure 9.2: Real-time classification of cryptocurrency subreddits

Appendix



Question 1: Would you expect following sentiment values?

10 odpovedí

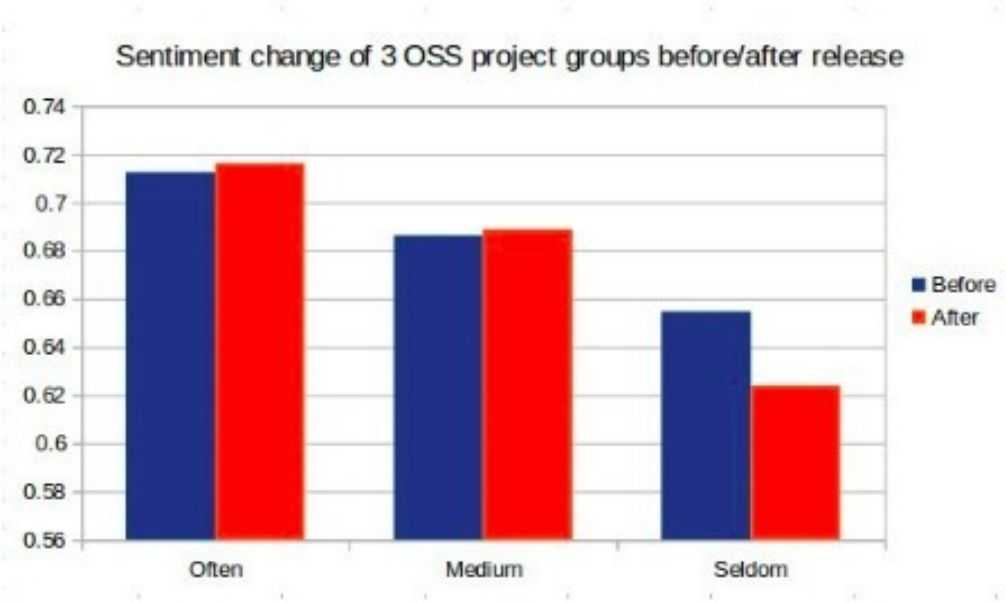


Additional remarks (if any):

4 odpovede

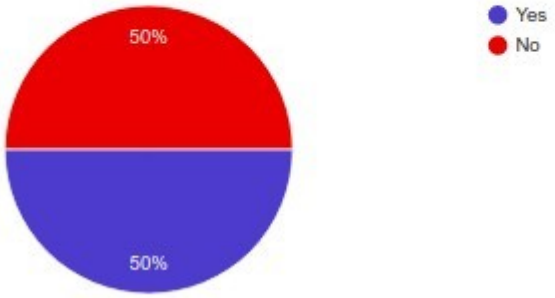
Releasing less then once per month doesn't seem negative to me. Think you'd need a bigger range like once every 1-3 months, once per 6-12 months etc.
I don't necessarily agree that this is a split yes or no answer. A project could release constantly, but if they're constantly shipping buggy code, it'll have a negative sentiment overall.
Idk how you're gonna normalize your data. React has many releases and I have no idea how people *feel* about it (other than obsessing over it for being easy, but having no clue how to use it in a big app.....). But Angular and Ember have fewer releases, but have great backwards compatibility and Ember has LTS releases which makes the sentiment very high / positive.
More often releases generally means that lots of patches for bugs are coming out, which leads to happier devs and a happier community

Question 2: I've measured sentiment 2 days before and 2 days after releases.



Question 2: I've measured sentiment 2 days before and 2 days after releases. Does is surprise you that it got significantly worse for projects with low(seldom) releasing frequency?

10 odpovedi



Additional remarks (if any):

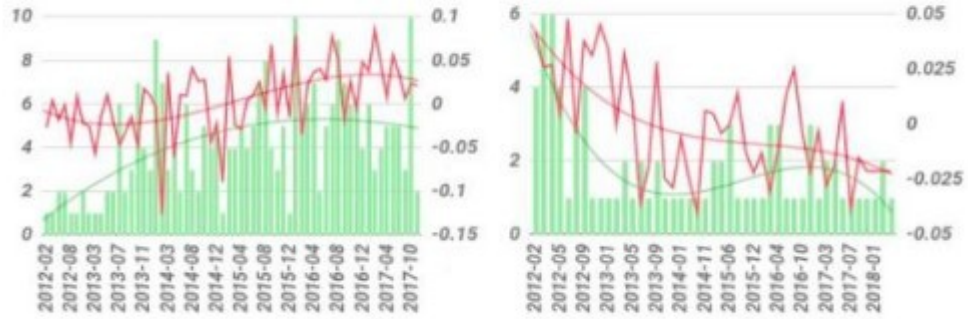
3 odpovede

Generally speaking I think it's difficult for most projects to keep every developer happy, and typically after a release people are reminded about the project and therefore will be more vocal. I would expect sentiment and volume of tweets to reduce as time passes from the release date.

Again, I don't know how this is normalized. But personally, for an upcoming ember release, there is some hype, and then once it comes out, people either get busy using it, or... idk. Also, with semver (something few projects do correctly), *major* releases remove deprecations, and don't add features.. it's intentionally boring.

it doesn't really, projects that don't update frequently are giving more time for buggy behavior to cement into the community and have code baked around it. This leads to breakage when the bugs are patched.

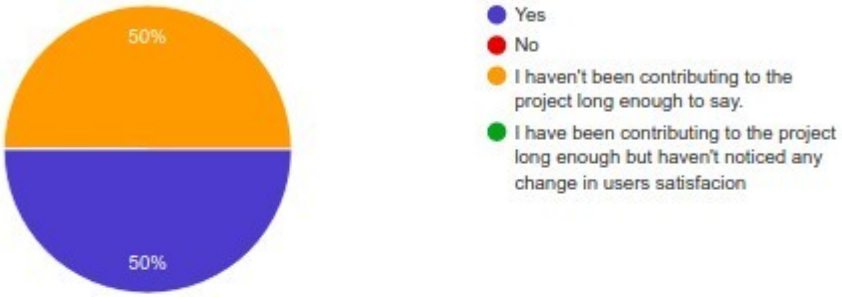
Question 3: Red line represents sentiment, green line represents release count



(a) EmberJS (b) Laravel

Question 3: Red line represents sentiment, green line represents release count per month. Find a project you've been contributing to lately. If you've been involved in the project over longer period of time, can you confirm the sentiment change over the years (red line)? Graphs start with the first release of the project and end around beginning of 2018.

10 odpovedí



Additional remarks (if any):

3 odpovede

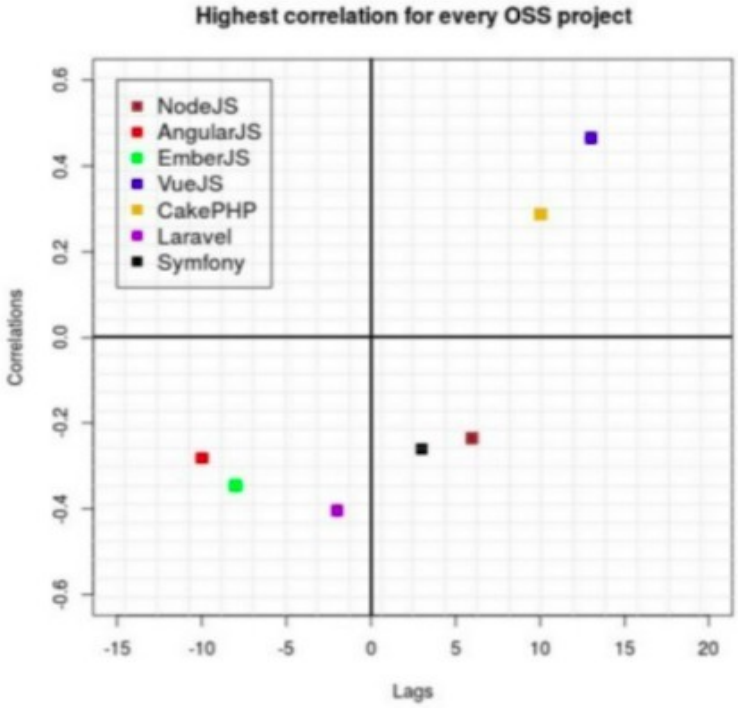
Is the higher value for sentiment good or bad? Can't really tell from the graph. That said, Laravel has been growing in popularity over the years and with that will have had an increased exposure to developers.

This seems kinda accurate for ember, I think. I'd be _very_ interested to see how this trend continues for 2018.

I haven't contributed to AngularJS (v1) / Angular (v4+), but the chart surprises me. A *lot* of people hated Angular 2.

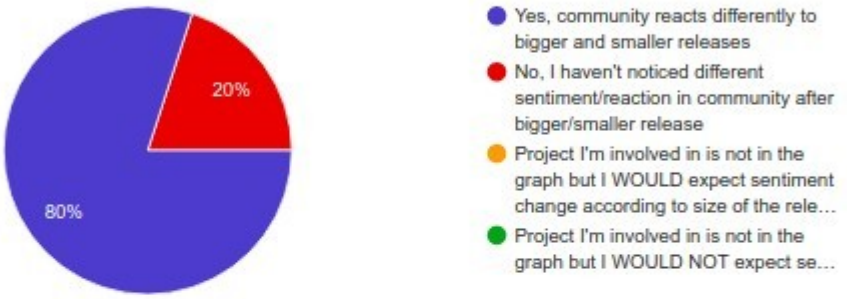
This is definitely roughly following the sentiment of the Ember community. The community dipped years back as Angular and others got more popular, then climbed as Angular and others went through many breaking changes. Recently sentiment has gotten worse again due to Ember "lagging" behind other frameworks in terms of features, though that seems to be changing again.

Commits:



Question 4: Calculating max cross-correlation points on the previous data series (sentiment + release count per month) gave me following results. Then I've changed the basic release count to commits count within releases (to see whether the size of releases plays a factor in sentiment) and the output changed significantly. Would you agree that your community reacts to bigger and smaller releases differently?

10 odpovedi



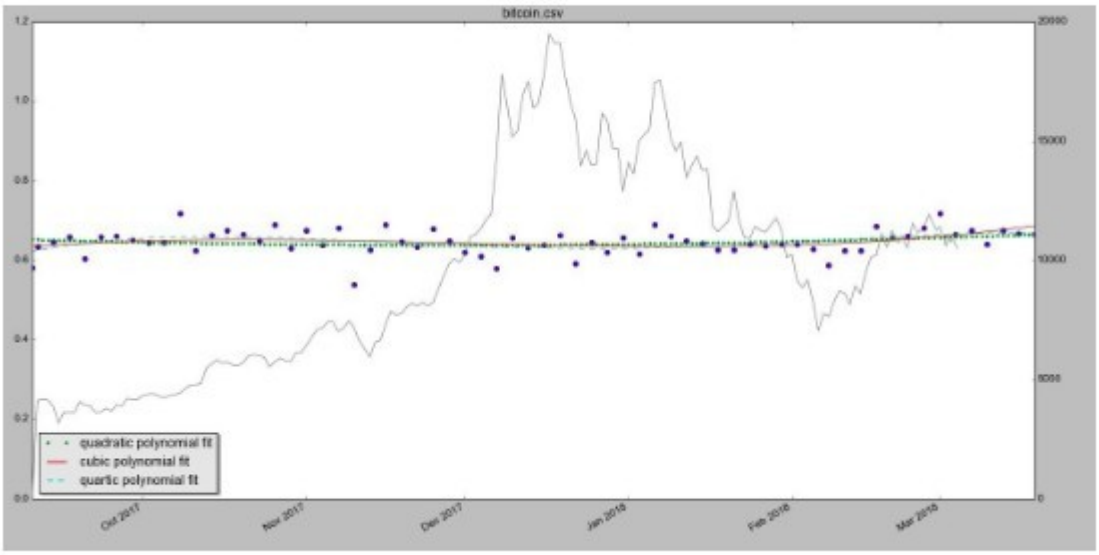
Additional remarks (if any):

2 odpovede

A high number of commits around a release could indicate that there are regressions (especially recently with ember-data :( ). But generally, I'm surprised that more commits for ember decreases sentiment. There has been amazing work done on ember recently, and a lot of positivity in general (at least that I've seen) -- esp with #EmberJS2018 - https://www.google.com/search?q=emberjs2018+call+for+blog+posts&oq=emberjs2018+call+for+blog+posts&aqs=chrome..69i57.5334j0j7&sourceid=chrome&ie=UTF-8

I generally see higher sentiment on minor releases (every 6 weeks, include new features) than with patch releases (irregular, frequent, only fix bugs) because people are excited for new features.

Bonus question: Figure shows weekly sentiment measured on bitcoin tweets



Bonus question: Figure shows weekly sentiment measured on bitcoin tweets (blue data points). Describe in couple of sentences if and where you see the correlation between price development (grey line) and sentiment. You can point exact /approximate times.

4 odpovede

People seem to be more positive when bitcoin is stable and dislike drastic price changes in any direction

There was a huge amount of press speculation about the value of bitcoin which is where the spikes of sentiment for bitcoin appear. However when it started to drop in value the sentiment reduced.

it looks like people have no idea what they are talking about regardless of what happens with the price.

price goes up, sentiment goes down in one place and up in another – the inverse is also true.

Can't see any correlation

REDDIT: The table shows similarity score between git issue description and it's

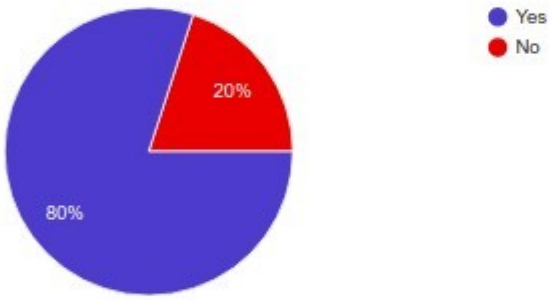
Framework	Bug comment	Whole discussion
NodeJS	0.447	0.507
AngularJS	0.306	0.57
VueJS	0.359	0.380

Table 0.19: Reddit NLTK similarity values

Part 2: Pairing reported Git issues with their related Stack/Reddit discussions

REDDIT: The table shows similarity score between git issue description and it's matching reddit discussion / particular comment which referenced the issue. Would you expect that the whole discussion shows often much higher similarity compared to the comment?

10 odpovedi



Additional remarks (if any):

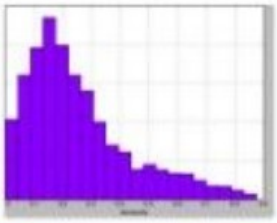
2 odpovede

Bug reports are always kind of more negative in nature, imo

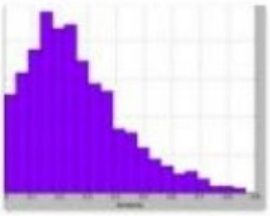
Overall discussion of a problem will lead to group knowledge being shared, whereas the initial comment is just the knowledge of one person. There is a higher likelihood that any individual comment will diverge from any other comment about a particular topic, but the average evens out because all information is shared over time.

STACK OVERFLOW: There are 4 histograms total. Each shows text similarity

Histogram 1

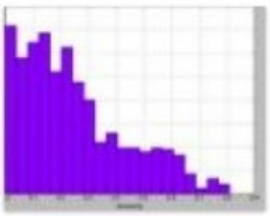


Histogram 3



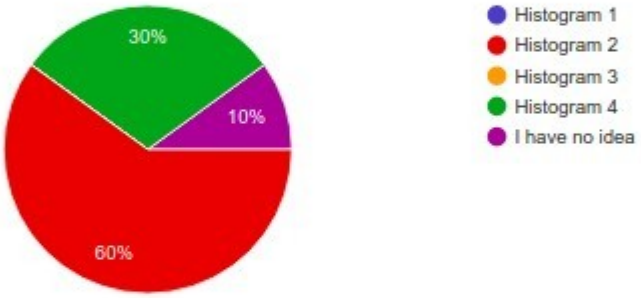
Histogram 2

Histogram 4



STACK OVERFLOW: There are 4 histograms total. Each shows text similarity distribution between 1000 Git issues and StackOverflow discussions (similarity increases from left to right). 3 histograms show similarity for one specific project and similarities have been calculated for related but also unrelated Git-Stack item pairs. One histogram though contains just the similarities of SO discussions and their matching Git issues which they talk about. Can you identify that histogram?

10 odpovedi



Additional remarks (if any):

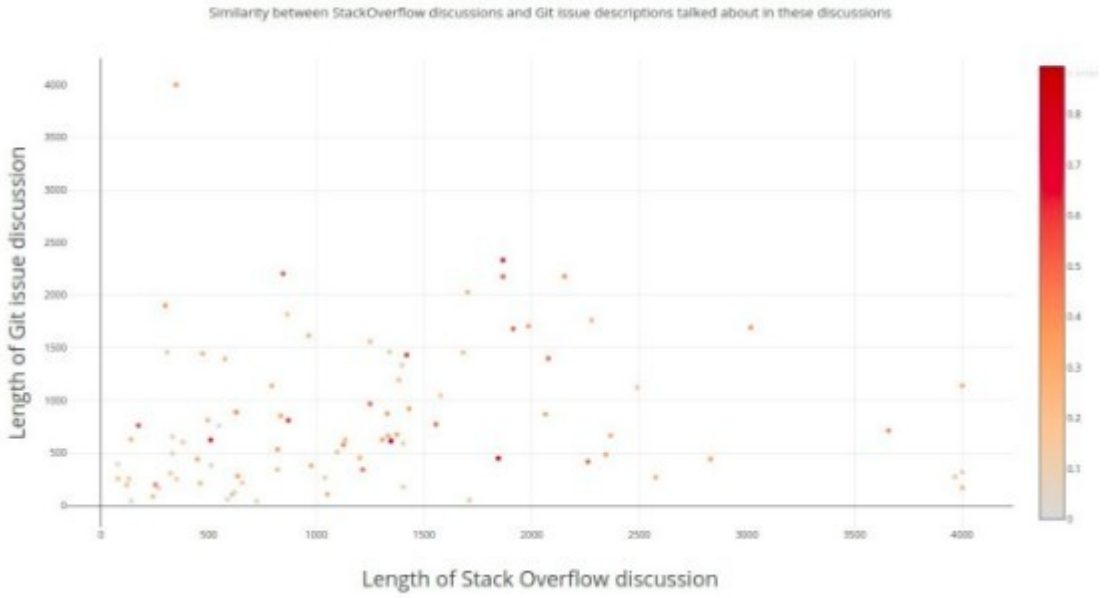
3 odpovede

This question is really opaque and hard to understand.

I have no idea what these are measuring.

I feel like comparisons across the entire set would even out more often, so I'm going with the irregular graph.

STACK OVERFLOW: The following scatter plot shows three dimensions - length



STACK OVERFLOW: The following scatter plot shows three dimensions - length of StackOverflow question regarding a git issue, length of Git description of that very same issue, similarity of these two texts. Each dot represents an issue and it's color represents the similarity of issue description and Stack Overflow discussion. Explain whether you see a relationship between the three data series or not! Interactive 3D Scatter plot can be found here -> https://plot.ly/~DurkoMatko/3

6 odpovedi

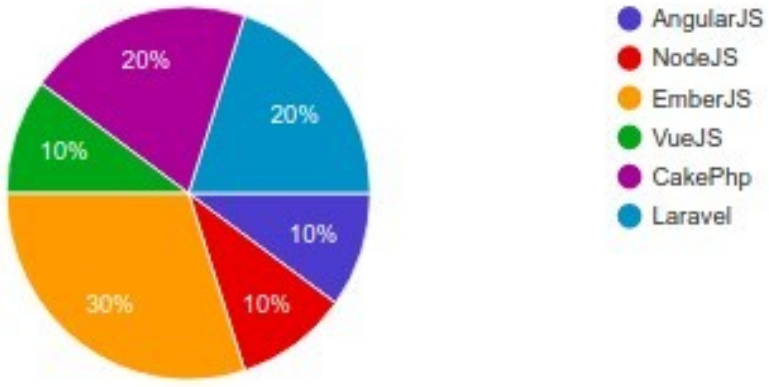
The shorter the SO discussion, the more likely it is clearly an issue recorded on git(hub - I assume)... maybe?
Not really, SO/GitHub are basically a preferred communication styles.
Seems like longer descriptions are more likely to be similar, maybe because they are copy pasted? I would check for a bias in the underlying similarity finder though
it looks like the longer both things are, the more similar they are. though... what are examples of this? why would git issues also be discussed on stack overflow? I need details!
I don't see any relationship
Seems like there's a pretty high correlation between similarity of short length github issue discussions and longer SO discussions. Could be that bugs are reported on SO first, then discussed, then re-reported and opened on GitHub once the underlying cause is found.

LAST question: Which project have you lately contributed to?

- ○ AngularJS
- ○ NodeJS
- ○ EmberJS
- ○ VueJS
- ○ CakePhp
- ○ Laravel

LAST question: Which project have you lately contributed to?

10 odpovedi



Bibliography

[1] Matthew A Russell. *Mining the Social Web: Data Mining Facebook, Twitter, LinkedIn, Google+, GitHub, and More.* " O'Reilly Media, Inc.", 2013.

[2] Jason T Tsay, Laura Dabbish, and James Herbsleb. Social media and success in open source projects. In *Proceedings of the ACM 2012 conference on computer supported cooperative work companion*, pages 223–226. ACM, 2012.

[3] Oskar Jarczyk, Błażej Gruszka, Szymon Jaroszewicz, Leszek Bukowski, and Adam Wierzbicki. Github projects. quality analysis of open-source software. In *International Conference on Social Informatics*, pages 80–94. Springer, 2014.

[4] Yuri Takhteyev and Andrew Hilts. Investigating the geography of open source software through github. *Manuscript submitted for publication*, 2010.

[5] Patrick Wagstrom, Corey Jergensen, and Anita Sarma. A network of rails: a graph dataset of ruby on rails and associated projects. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 229–232. IEEE Press, 2013.

[6] Georgios Gousios and Diomidis Spinellis. Ghtorrent: Github's data from a firehose. In *Mining software repositories (msr), 2012 9th ieee working conference on*, pages 12–21. IEEE, 2012.

[7] Emitza Guzman, David Azócar, and Yang Li. Sentiment analysis of commit comments in github: an empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 352–355. ACM, 2014.

[8] Scott Christley and Greg Madey. Analysis of activity in the open source software development community. In *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, pages 166b–166b. IEEE, 2007.

[9] Sara Rosenthal, Noura Farra, and Preslav Nakov. Semeval-2017 task 4: Sentiment analysis in twitter. In *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)*, pages 502–518, 2017.

[10] Apoorv Agarwal, Boyi Xie, Ilia Vovsha, Owen Rambow, and Rebecca Passonneau. Sentiment analysis of twitter data. In *Proceedings of the workshop on languages in social media*, pages 30–38. Association for Computational Linguistics, 2011.

[11] Efthymios Kouloumpis, Theresa Wilson, and Johanna D Moore. Twitter sentiment analysis: The good the bad and the omg! *Icwsm*, 11(538-541):164, 2011.

[12] Alexander Pak and Patrick Paroubek. Twitter as a corpus for sentiment analysis and opinion mining. In *LREc*, volume 10, pages 1320–1326, 2010.

[13] Alec Go, Richa Bhayani, and Lei Huang. Twitter sentiment classification using distant supervision. *CS224N Project Report, Stanford*, 1(12), 2009.

[14] Hassan Saif, Yulan He, and Harith Alani. Semantic sentiment analysis of twitter. In *International semantic web conference*, pages 508–524. Springer, 2012.

[15] Alexander Hogenboom, Daniella Bal, Flavius Frasincar, Malissa Bal, Franciska de Jong, and Uzay Kaymak. Exploiting emoticons in sentiment analysis. In *Proceedings of the 28th annual ACM symposium on applied computing*, pages 703–710. ACM, 2013.

[16] Petra Kralj Novak, Jasmina Smailović, Borut Sluban, and Igor Mozetič. Sentiment of emojis. *PloS one*, 10(12):e0144296, 2015.

[17] Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar Devanbu, and Abraham Bernstein. The missing links: bugs and bug-fix commits. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 97–106. ACM, 2010.

[18] John Anvik, Lyndon Hiew, and Gail C Murphy. Coping with an open bug repository. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 35–39. ACM, 2005.

[19] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *Mining Software Repositories, 2007. ICSE Workshops MSR'07. Fourth International Workshop on*, pages 1–1. IEEE, 2007.

[20] Emanuel Giger, Martin Pinzger, and Harald Gall. Predicting the fix time of bugs. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, pages 52–56. ACM, 2010.

[21] Per Runeson, Magnus Alexandersson, and Oskar Nyholm. Detection of duplicate defect reports using natural language processing. In *Proceedings of the 29th international conference on Software Engineering*, pages 499–510. IEEE Computer Society, 2007.

[22] Nicholas Jalbert and Westley Weimer. Automated duplicate detection for bug tracking systems. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 52–61. IEEE, 2008.

[23] Man Lan, Chew-Lim Tan, Hwee-Boon Low, and Sam-Yuan Sung. A comprehensive comparative study on term weighting schemes for text categorization with support vector machines. In *Special interest tracks and posters of the 14th international conference on World Wide Web*, pages 1032–1033. ACM, 2005.

[24] Thomas G Dietterich. Approximate statistical tests for comparing supervised classification learning algorithms. *Neural computation*, 10(7):1895–1923, 1998.

[25] Juan Ramos et al. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, volume 242, pages 133–142, 2003.

[26] Michael Gamon, Anthony Aue, Simon Corston-Oliver, and Eric Ringger. Pulse: Mining customer opinions from free text. In *international symposium on intelligent data analysis*, pages 121–132. Springer, 2005.

[27] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600. ACM, 2010.

[28] Antony Mayfield. What is social media. 2008.

[29] Carlos Castillo, Marcelo Mendoza, and Barbara Poblete. Information credibility on twitter. In *Proceedings of the 20th international conference on World wide web*, pages 675–684. ACM, 2011.

[30] Mika V Mäntylä, Daniel Graziotin, and Miikka Kuutila. The evolution of sentiment analysisa review of research topics, venues, and top cited papers. *Computer Science Review*, 27:16–32, 2018.

[31] Shaosong Ou Alexander Hars. Working for free? motivations for participating in open-source projects. *International Journal of Electronic Commerce*, 6(3):25–39, 2002.

[32] Josh Lerner and Jean Tirole. The open source movement: Key research questions. *European economic review*, 45(4-6):819–826, 2001.

[33] Bing Liu. Sentiment analysis and opinion mining. *Synthesis lectures on human language technologies*, 5(1):1–167, 2012.

[34] SB Kotsiantis, D Kanellopoulos, and PE Pintelas. Data preprocessing for supervised leaning. *International Journal of Computer Science*, 1(2):111–117, 2006.

[35] JA Richmond. Spies in ancient greece. *Greece & Rome*, 45(1):1–18, 1998.

[36] Bo Pang and Lillian Lee. A sentimental education: Sentiment analysis using subjectivity summarization based on minimum cuts. In *Proceedings of the 42nd annual meeting on Association for Computational Linguistics*, page 271. Association for Computational Linguistics, 2004.

[37] Amy Beth Warriner, Victor Kuperman, and Marc Brysbaert. Norms of valence, arousal, and dominance for 13,915 english lemmas. *Behavior research methods*, 45(4):1191–1207, 2013.

[38] Andrew P Bradley. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern recognition*, 30(7):1145–1159, 1997.

[39] Alec Go, Lei Huang, and Richa Bhayani. Twitter sentiment analysis. *Entropy*, 17:252, 2009.

[40] Rob J Hyndman, George Athanasopoulos, Slava Razbash, Drew Schmidt, Zhenyu Zhou, Yousaf Khan, Christoph Bergmeir, and Earo Wang. forecast: Forecasting functions for time series and linear models, 2013. *R package version*, 5.

[41] Emma Haddi, Xiaohui Liu, and Yong Shi. The role of text preprocessing in sentiment analysis. *Procedia Computer Science*, 17:26–32, 2013.

[42] Joseph Feller, Brian Fitzgerald, et al. *Understanding open source software development*. Addison-Wesley London, 2002.

[43] Alec Go, Lei Huang, and Richa Bhayani. Twitter sentiment analysis. *Entropy*, 17:252, 2009.

[44] Yingnong Dang, Rongxin Wu, Hongyu Zhang, Dongmei Zhang, and Peter Nobel. Rebucket: a method for clustering duplicate crash reports based on call stack similarity. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 1084–1093. IEEE, 2012.

[45] Alina Lazar, Sarah Ritchey, and Bonita Sharif. Improving the accuracy of duplicate bug report detection using textual similarity measures. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 308–311. ACM, 2014.

[46] Preslav Nakov, Alan Ritter, Sara Rosenthal, Fabrizio Sebastiani, and Veselin Stoyanov. Semeval-2016 task 4: Sentiment analysis in twitter. In *Proceedings of the 10th international workshop on semantic evaluation (semeval-2016)*, pages 1–18, 2016.