Universität Innsbruck

Department of Computer Science
Research Group Quality Engineering

MASTER THESIS

The Title

Martin urek

Supervisor: Ass.-Prof. Dr. Michael Felderer

Innsbruck, July 22, 2018

Contents

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. 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've analysed tweets of several open-source projects 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 as second part of my thesis, I've done somewhat similar process - I've 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 Motivation

Motivation behind this thesis lies in my interest in machine learning and the fact that I unfortunately didn't manage to explore this field over the course of my studies. After taking some very basic Coursera[1] 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. Machine learning, sentiment analysis and data mining in general are fields on the rise and it's always good to have as wide

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

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'm also a fan of open-source movement and mindset and I liked the idea of combining these two into my thesis.

Proposed framework

The main goal of this thesis can be 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 matching pairs of issues reported in bug tracking systems and their respective social media entries. Both parts of the framework utilize modern text processing approach, sentiment analysis concentrating more on machine learning approach while bug pairing utilizing topic modelling and text similarity principles.

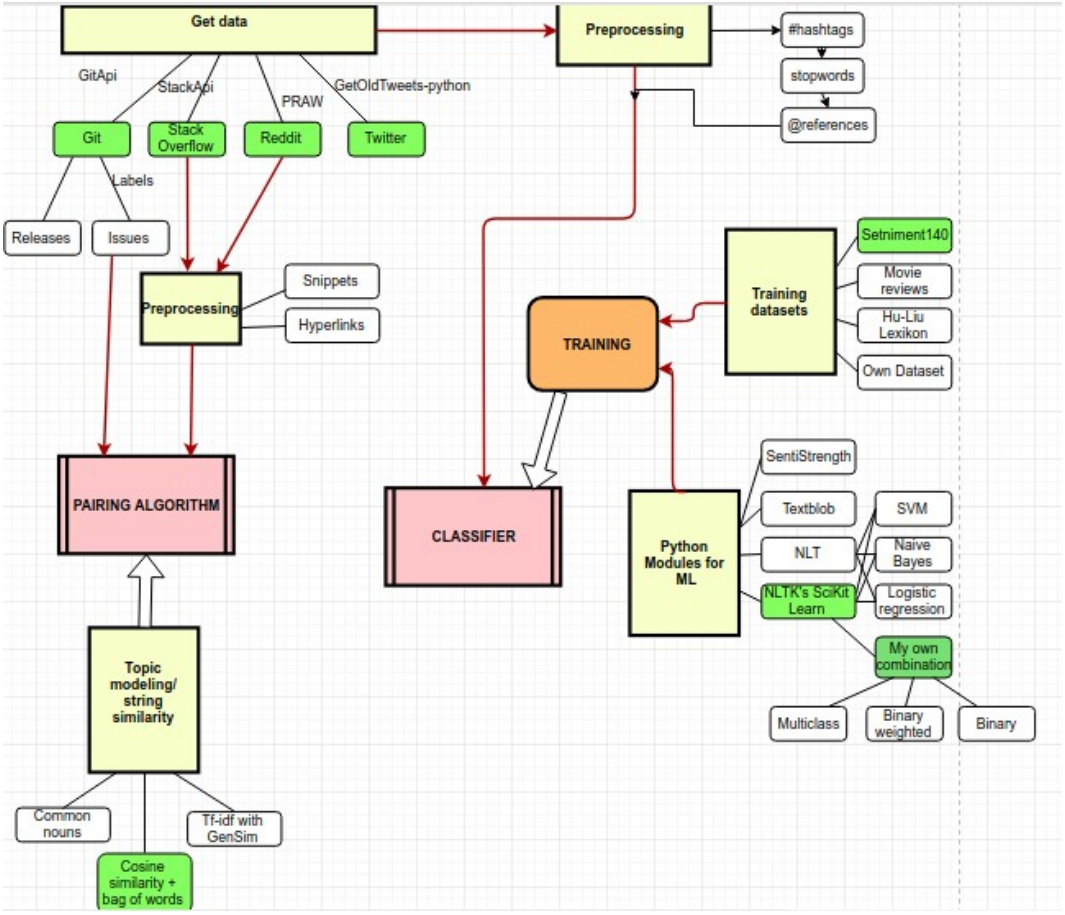


Figure 0.1: Brief sketch of dataflow in the proposed framework

As can be seen in figure 0.1, two final products of the framework are sentiment classifier and pairing algorithm. To get these, several steps need to be executed:

1. **Getting the data** - there is no data science, machine learning or natural language processing without the data. That's 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 subsectoins 6.1 and 6.2 while Twitter, Reddit and SO were mined to provide data which are the target of analysis 3.1.
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's described in the section 8.
4. **Choosing a tool/module/library as well as particular 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 don't 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 9 and 11
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 13.4.

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'm interpreting the results in subsections13.3 and 13.4.

Development of such 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 successfully pair social media entries to their respective bugs which they talk about?

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 if really wanted, most of the today's websites could be labeled as social media. 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 too to reach and interact with other users.
- **Profile pages** - pages which represent an individual, might it be a real person, group of people or company. It should contain several personal information about the user like bio, profile picture or 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[1].

2 Potential of social media data mining

Social media are changing the way that information is passed across societies and around the world.[2] 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 analyzing 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 desinformation, there is indication that social networks tend to favour valid information over rumours.[3]

3 Data

3.1 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 [4] wordcloud of SE papers before and after 2013.0.2.

Figure 0.2: Wordcloud comparison of pre and post 2013 SE papers 0.2

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 don't 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 project (https://github.com/Jefferson-Henrique/GetOldTweets-python) 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 1)
- in the interval of 2 days before and after release date (Listing 2)
- weekly

Listing 1: Creating command to get Tweets about a project version on release dates

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

Listing 2: Creating command to get Tweets about a project version in particular time interval around release date

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

Facebook: I originally planned to use Facebook statuses as a big part of analyzed data. Sadly, this was not possible since Facebook Graph API doesn't 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's 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's 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 doesn't 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 your program misbehaves for some reason, it can be more easily identified, rather than look like a browser. All mandatory arguments are shown in Listing 3.

Listing 3: Instantiating Reddit class object

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

After connection to Api is successful, senting 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 is enough.

Listing 4: Getting posts from subreddit

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

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 5

Listing 5: Getting Stackoverflow questions with StackApi

```
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
        )
```

Stack Exchange is limited on 30 requests per second what caused the process of getting the data to take much more time since program execution needed to be regularly stopped to avoid the SO throttle violation and from it resulting penalization. At first I intended to use the type posts which returns both, questions and answers, but later I've 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 X 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.

Open-source projects

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 don't necessarily know each other. These communities are usually web-based.The 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. [5] There are many other 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 for a web server implementation and in 2002 it was used on 56% of web servers worldwide [6].

4 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. [5] 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 Choosing projects of interest

There are loads of OSS projects nowadays and it turned out to be pretty interesting process to choose the correct ones for my project. 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 pairing the social media data with their corresponding bug items.

- 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.

Project of my choice are NodeJS, AngularJS, EmberJS, VueJS for frontend technologies and Laravel, Symfony and CakePhp for backend PHP technologies. Some frameworks like Django, Meteor, React 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".

Other very interesting group of OSS project to examine are cryptocurrencies. Being a very hot topic these days, I've decided to work with some of the most popular cryptocurrency repositories as well. These were Bitcoin, Ethereum, Litecoin, Dash and Ripple.

6 Github mining

Github as a most-popular version control system is often a go-to choice for many OSS projects. That happened to be the case also with projects I've chosen for my testing and case study in the thesis.

Thanks to Github Rest API v3[2] 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've decided to register an OAuth2 App under my GitHub account and that way I got non-changing token. Another way is to request a token programmatically, but I though it would be an unnecessary overhead. OAuth2 apps can be registered under *Account settings >Developer settings >Personal access tokens >Generate new token.*

6.1 Release dates

To get the project release dates, I've sent one request to the endpoint *git/refs/tags* (Listing 6)

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

Listing 6: Requesting all project tags git api tags endpoint

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

After obtaining all tags, one more request for every one of those tags was needed to get the details (Listing 7). Path to release date field withing response body as json is tag >object >url >Send new request >author >date

Listing 7: Requesting tag details and accessing release date

```
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']
```

I saved the release dates in simple text file, one per line. This code was later on changed (in subsection 13.4) to include also number of commits per release.

After executing this step, I had data needed to divide the 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 0.1

Group	Project	Release count	Duration	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 Framework	41	59	0.69
	Bootstrap	42	72	0.58

Table 0.1: OSS projects grouped according to their releasing frequency

6.2 Issues

To get the project issues, I've used the endpoint *issues*. It offers various parameters like state, labels, sort, direction or since date. I've decided to work just with closed issues and snippet where I'm sending the request can be seen in listing 8).

Listing 8: Requesting 100 closed issues

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

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.

During my later work, I've realized the ammount 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've 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 0.3 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 0.3: 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 0.2

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

Table 0.2: Reddit submissions counts

Sentiment analysis

SE and opinion mining is the field of study that analyzes people's opinions, sentiments, evaluations, attitudes, and emotions from written language.[7] It's 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. for example 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 can often have a significant impact on generalization performance of a supervised ML algorithms [8] 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, emphasized and retained while objective information like facts are discarded and ignored.

4. **Sentiment classification** - subjective expressions are classified.

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

7 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 [9]. 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 a big boom. According to data collected by Mantyla, Graziotin and Kuutila [4], 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 as published by Mantyla, Graziotin and Kuutila [4] can be seen in the graph 0.4.

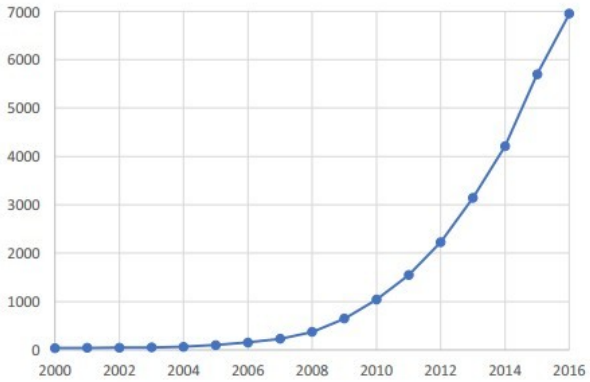


Figure 0.4: Cumulative count of papers about sentiment analysis

8 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's 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's why choosing a sufficient and fitting dataset to train my classifiers was a very important task. The datasets I've considered were:

- **Dataset140**[3] - 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 [10]

- **Movie review data**[4] - Thousand positive and thousand negative labeled movie reviews. This dataset was introduced in Pang/Lee ACL 2004 [11]

- **Hu-Liu lexicon**[5] - plain list of 6800 common English words labeled as positive and negative

- **Warriner et al lexicon**[6] - This list of words was collected with Amazon Mechanical Turk. Three components of emotions are traditionally 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) [12]. Warriner and Kuperman extended ANEW norms collected by Bradley and Lang from 1034 words to 13,915 words (lemmas).

- **Stack overflow dataset**[7] - Later into the thesis I've 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"

- **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's why I considered and even started to create my own dataset targeting specifically only cryptocurrency tweets, which I've intended to analyze as well.

Big surprise here was the lack of any bigger Reddit dataset.

9 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've always given a slight edge to the Python options. I've considered (and tested) these:

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

- **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 offeres 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 (NLP) 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.

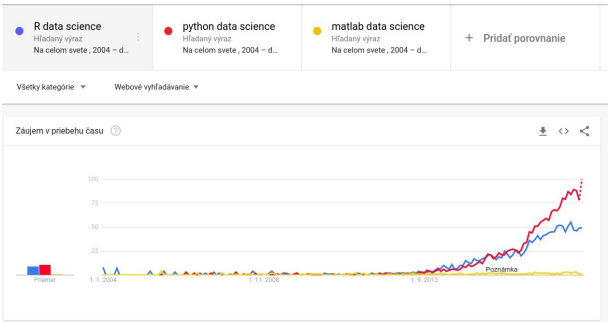


Figure 0.5: Google trend of searches regarding data science with various programming languages

10 Performance metrics

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

- **Tue 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's also useful to choose the metric before doing the analysis, so you won't 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 misused 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 9% 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 0.3. 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.[13].

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

Table 0.3: Confusion matrix

- **Precision** - Precision can be seen as a representation of a classifiers exactness. A low precision can also indicate a large number of False Positives. If the precision is high, it says that there's 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 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 for such scenarios F1 score is used.

- **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's what F1 measure (F measure) is for. It is the harmonic average of the precision and recall, where its best value is at 1.

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

11 Classifier evaluation

All the testing has been done on the testing data supplied within Sentiment140.

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 0.4 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 0.4: Textblob accuracy with various handling of neutral tweets

Honestly, none of the tree considered approaches

As mentioned earlier, accuracy is not the best metric to evaluate classifiers on, but with such low values, it is apparent that Textblob isn't performing very well. There might be several reasons for this:

- Tweets are in general difficult to analyze 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). Therefore is expected classification performance not as high as it would be with other longer text which provide more data and sentiment
- 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've decided to implement my own classifier from the ground up using Natural Language Toolkit Python module. After reading several forum discussions, I've decided to use Naive Bayes classifier as people claimed it to have the best performance for sentiment recognition of short texts like tweets. I've 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 the listing 9 and 10

Listing 9: Feature extraction and NLT classifier training

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

Listing 10: Helper methods for text features extraction

```
def get_words_in_tweets(tweets):
    all_words = []
    for (words, sentiment) in tweets:
       all_words.extend(words)
    return all_words
```

```
def get_word_features(wordlist):
    wordlist = nltk.FreqDist(wordlist)
    word_features = wordlist.keys()
    return word_features

def extract_features(document):
    document_words = set(document)
    features = {}
    for word in word_features:
       features['contains(%s)' % word] = (word in document_words)
    return features
```

The results of the 4-fold crossvalidation can be seen in table 0.5. On the listed values,it's visible that there s something not right. As written earlier, recall 100% very likely means the classifier is labeling everything as one class. That's also exactly happening here.

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

Table 0.5: Textblob accuracy with various handling of neutral tweets

conf matrix was 0,0;0,0

Sci-kit Learn

Choosing classifiers Third and final module I've 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've followed a workflow showed in the Figure 0.6

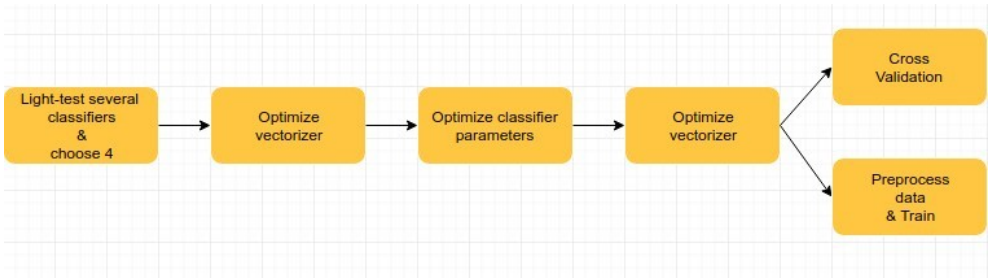


Figure 0.6: Implementation workflow of sentiment analysis using Scikit module

After manually testing and playing around with all sci-kit classifiers which are mostly used for analysis of textual data, I've 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've measured their accuracy on Sentiment140 as well as movie reviews dataset. Just out of pure curiosity I've 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 can be seen in table 0.6.

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. Not only can a vector have more than one dimension, but with text data vectors are usually high-dimensional. This is because each dimension of your feature data will correspond to a word, and the language in the documents you are examining will have thousands of words.

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 words with no duplicates would translate to an 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 nave 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 are 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 cant tell you whether or not words are unique, as the 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 (TFIDF) Matrix** as a vectorization approach. That's also the vectorizer I have used for my classifier. TFIDF 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 words frequency in a document, and how rare the word is in the entire corpus.

Once I've 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 11.

Listing 11: Tuning of vectorizer using GridSearchCV class

```
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
```

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 for 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 12.

Listing 12: Tuning of classifiers using GridSearchCV class

```
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)

    clf = GridSearchCV(LinearSVC(), SVC_parameters, cv=5,scoring=
        '%s' % score)
    clf.fit(train_corpus_tf_idf, y_train)

    print("Best␣parameters␣set␣found:")
    print(clf.best_params_)
    print("Performance␣for␣all␣combinations:")
    means = clf.cv_results_['mean_test_score']
    stds = clf.cv_results_['std_test_score']
    for mean, std, params in zip(means, stds, clf.cv_results_['
        params']):
        print("%0.3f␣(+/-%0.03f)␣for␣%r"
              % (mean, std * 2, params))

    print("Detailed␣classification␣report␣of␣model␣trained␣and␣
        evaluated␣on␣full␣dev/eval␣sets:")
    y_true, y_pred = y_test, clf.predict(test_corpus_tf_idf)
    print(classification_report(y_true, y_pred))
```

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 0.6: 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's important to realize that Tweets are very short text pieces which often don't offer much sentiment input to work with. Therefore is 78% quite promising performance for future work. For example, other paid services like MonkeyLearn [8] offer 81% accuracy on Tweets sentiment classification.

[8] https://monkeylearn.com/

Another thing I tried to increase the accuracy of the classifiers was 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 hastags. Accuracy metrics can be seen in table 0.7. Training and testing data were from Sentiment140. Performance boost was smaller that expected but still, any improvement is better than none.

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 0.7: Scikit classifiers accuracy on Sentiment140 dataset

Neutral Tweets : At this point, after all the tuning, optimizing and cross-validation, I could finally run my analyzer on the testdata 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's 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 [14] 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. [**?**] trained their classifier just on hashtags and emoticons and also had to build their own neutral training dataset. Agarwal at al [15] 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. [16] as well as Go, Huang and Bhayani [10] 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've 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 measuments are recorded in table 0.8. I've 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 0.8: Textblob 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 ways - 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 to 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 0.7

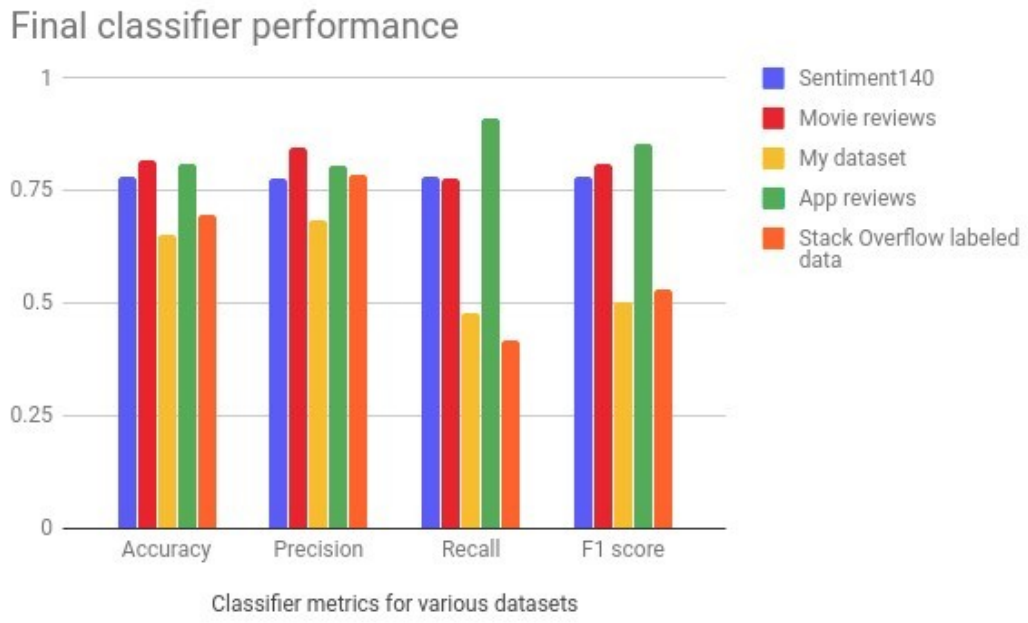


Figure 0.7: Final classifier performance for several datasets

It's clear that the best results are achieved with movie or app 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 sligthly worse compared to reviews but it's still authentic Twitter data which I'm going to analyze the most.

12 Data to analyze

I found Twitter data very suitable for sentiment analysis and therefore decided to analyze only tweets and use SO and Reddit later in chapter 13.4 about recognizing bugs in discussions.

Using the steps described in section 3.1, I've downloaded tweets with following time conditions/patterns:

- between two immediate releases
- days prior and after release
- weekly

13 Results

13.1 Immediate sentiment change with releases

First and the most simple analysis was to compare sentiment of tweets collected 3 days prior and post-release. Results are shown in table 0.9.

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 0.9: 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 noticable pattern (nicely displayed in figure 0.8) is 4% decrease in sentiment of seldom releasing projects. In groups which release often or normally, sentiment change was very small. One more thing to notice here is, that more often projects release, better the sentiment gets.

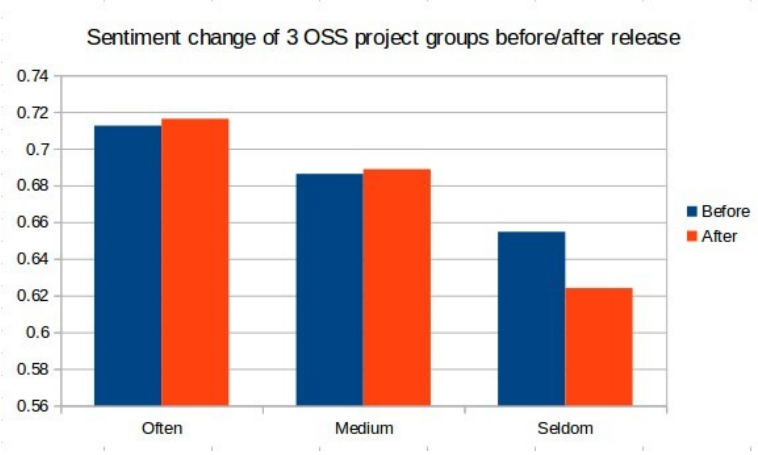


Figure 0.8: Change in sentiment for all 3 OSS groups before/after release

13.2 Results for weekly collected data

Next step was to do the analysis on the bigger scale and see if sentiment changes with the amount of releases/commits per month. For this, I'm going to use tweets collected with period one week. I've 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).

The classifier used is completely the same so there should not be big differences in the sentiment of the same projects. Results can be seen in table 0.10 and figure 0.9

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

Table 0.10: Average sentiment of tweets collected weekly

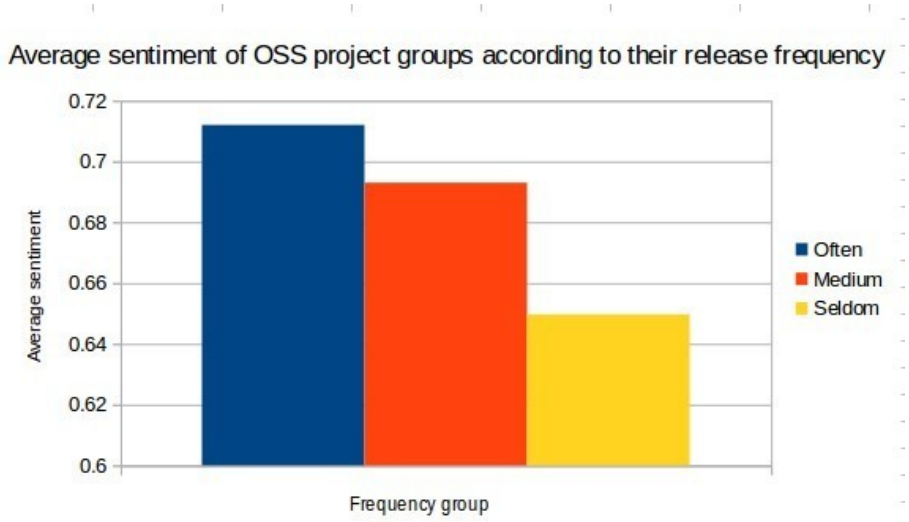


Figure 0.9: Average sentiment of OSS project groups according to their release frequency

In both of previous analysis, I've 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.

13.3 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've used the data collected weekly and analysed their relationship with number of releases per month. In the figure 0.10 are plotted sentiment difference from the average and the release count per month.

(a) EmberJS

(b) Laravel

(c) NodeJS

(d) VueJS

(e) CakePhp

(f) AngularJS

Figure 0.10: Sentiment difference from average and release counts per month

Looking at linecharts can give us some feeling and intuition whether and what type of relation there is. When working with time series, we often want to determine 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 when change in one data series transfers to the other several periods later.

To ensure a cross-correlation calculation makes sense, first I have to determine, whether are the data stationary. A stationary time series is one whose properties do not depend on the time at which the series is observed[17]. 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've used the Dickey-Fuller test method of tseries package in R. Results can be seen in the table 0.11 and 0.12

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 0.11: 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 0.12: Stationarity test of release counts

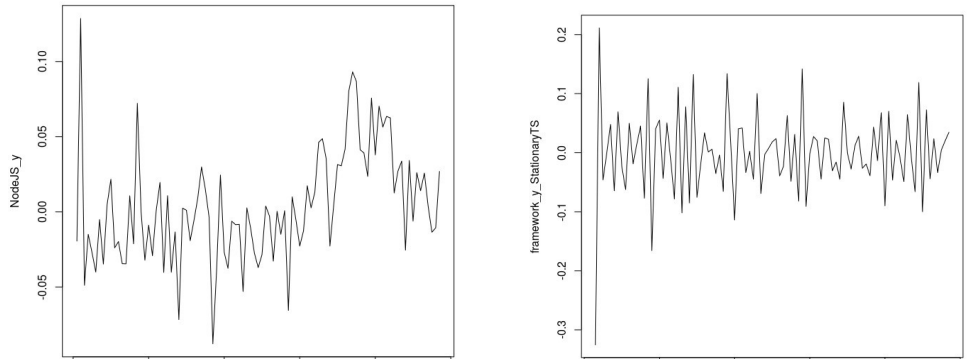
As we can see, p-values are always higher than 0.05 what indicates non-stationarity of the data, therefore I can't calculate the cross-correlation on them in this state. To transform non-stationary data into stationary, 2 approaches can be used. These are differencing and transforming. I've taken data series and differenced the values in listing 13. I've executed

both, seasonal differencing and stationary differencing although seasonal probably was not needed because the data should not be dependant on the season.

Listing 13: Used differencing method in R

```
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))
}
```

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 0.11



(a) Before differencing (b) After differencing

Figure 0.11: 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've used ccf method in R which implements Pearson's correlation calculation method. Results for all 7 OSS projects can be seen in Figure 0.12

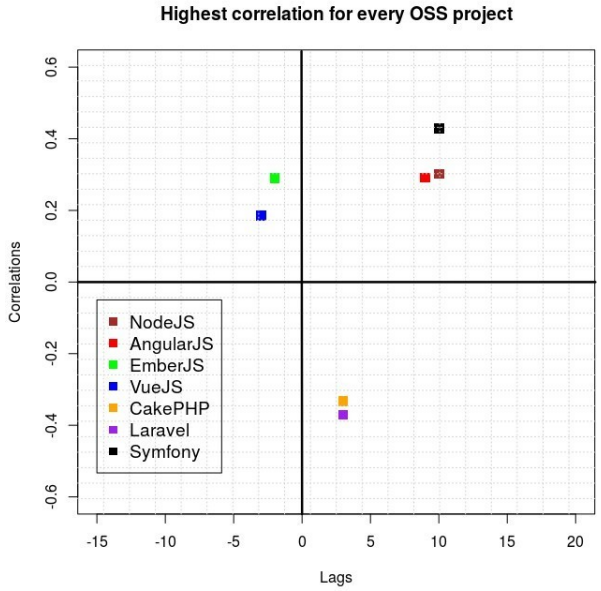


Figure 0.12: Highest correlations for every OSS project

I think there could be made a counter-point about whether the transformation to stationary data is really needed here, I've also included Figure 0.13 where are the results with the raw data.

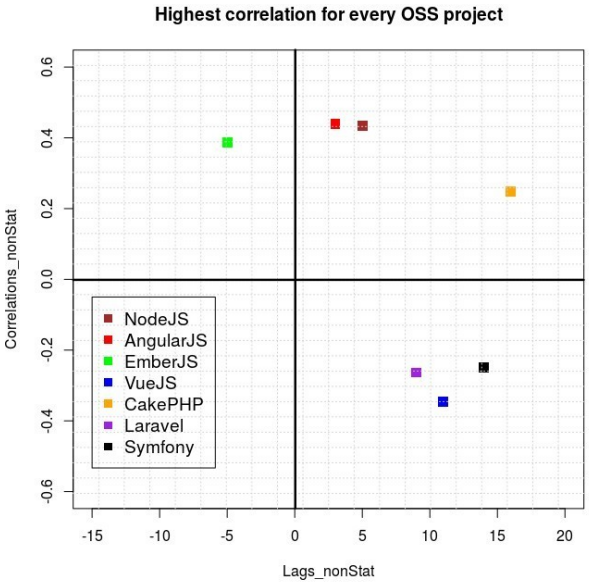


Figure 0.13: Highest correlations for every OSS project (non-stationary)

Results interpretation: As we can there is no general pattern. Maximal project correlations happen to occupy 3 of 4 possible quadrants. Each quadrant represents a different relationship between number of releases and sentiment change.

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

13.4 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 pretty straightforward task. It actually was straightforward, but as always I've encountered several unexpected problems on the way.

I intended to extend my previously used method from section 6.1 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've done some extra searching but didn't want to spend extra time so I've decided to go the way I knew will work. Instead of using Api to get the commit counts, I've 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 commit 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've 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 relaeses later on.

After getting the data ready I performed a stationarity test for commit counts. Sentiment values ar the same as before with count of releases. Results can be seen in table 0.13

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 0.13: 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 0.14

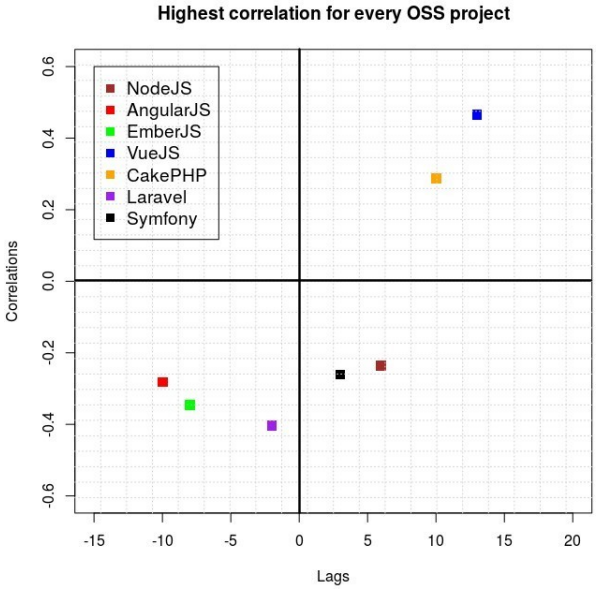


Figure 0.14: Highest correlations for every OSS project

If I would skip the step of making the data stationary, results would again look completely different 0.15.

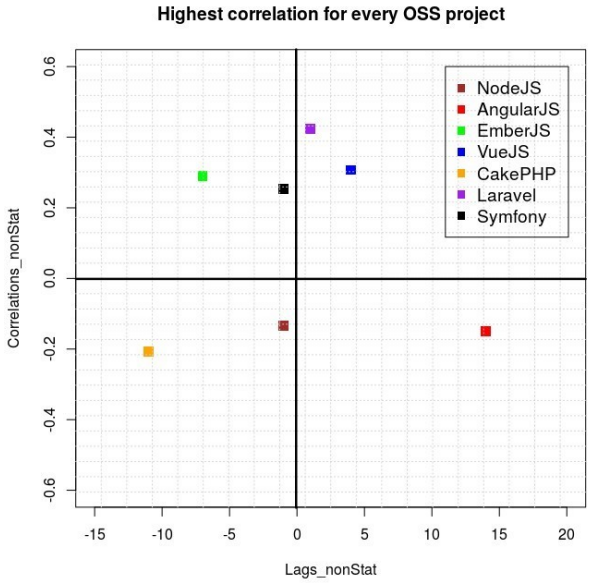


Figure 0.15: Highest correlations for every OSS project (non-stationary)

Linking bug repositories and social media

My goal of pairing 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 modeling and general text similarity algorithms.

To link any two texts based on their content, there's 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 modeling" algorithm and try to connect documents based on their matching topics.

Topic modeling is a method used to organize and summarize large textual information. It's 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 modeling for my case is that neither SO nor Reddit questions are not long enough. This could potentially be avoided by concatenating the whole discussions into one long text but these are still too topic-specific to get reasonable output. In this case, topic modeling output is not granular enough to differentiate among similar texts which are all from the very same domain.

Because the output provided by topic modeling wasn't enough to pair particular items, I searched for, found and considered several alternative approaches.

14 Aproaches

There are many ways and approaches how 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 (tf-idf) and then compute cosine similarity between them. Text transformations like Tf-idf and others are implemented in several Python packages.

I've tried following:

1. String similarity using NLTK
2. String similarity using Gensim

3. String similarity using Scikit-learn
4. The textual properties, title and description are concatenated together and then used to numeric features using the simple TakeLab system. This approach was shown by Lazar, Ritchey and Sharif. [18].

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'm comparing the whole documents and don't want to consider sentences as standalone objects, I've 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 15 illustrates the implementation of the described similarity calculation procedure.

Listing 14: Text similarity implementation with NLTK

```
tokens = word_tokenize(text)
words = [w.lower() for w in tokens]

porter = nltk.PorterStemmer()
stemmed_tokens = [porter.stem(t) for t in words]

# removing stop words
stop_words = set(stopwords.words('english'))
filtered_tokens = [w for w in stemmed_tokens if not w in
    stop_words]

# count words
count = nltk.defaultdict(int)
for word in filtered_tokens:
        count[word] += 1
return count;
```

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 doesn't analyse and consider role and position of word in a sentence (POS tagger would be required here).

Scikit-learn and TF-IDF: My next similarity checker I've implemented was using Scikit-learn module and Tf-Idf vectorizer. Term FrequencyInverse Document Frequency (TFIDF) is similar to bag-of words approach but better. While bag of words 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. Other options include term frequency adjusted for document length, logarithmically scaled frequency or augmented frequency to handle bias towards longer documents.

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

Listing 15: Text similarity implementation with Scikit using Tf-Idf model

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

15 Available data

Git issue reports: Using the approach described in the section 6.2, I've downloaded 96,651 issue reports from Git while 25,978 of those labeled as bug or similar. Because this part of the thesis was more just a PoC than the part with sentiment analysis, I've 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 0.14.

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

Table 0.14: Bug count per project

Stack overflow questions: SO mining has been described in section 3.1 and I've 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 doesn't 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 0.15). 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've decided to from opposite end and crawled through all closed git issues of projects and checked if there's a SO question linked to them. If that was the case I've saved both, git issue and SO question and that way increased the size of my dataset (can be seen in table 0.16)

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

Table 0.15: Bug count per project

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

Table 0.16: 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 0.17.

Framework	Submissions count
NodeJS	108
AngularJS	43
VueJS	20
EmberJS	13

Table 0.17: Reddit submissions counts

16 Similarity results

Linking items and in general finding similarity among these already area-specific texts proved to be a problematic task.

16.1 NLTK

Stack Overflow: The average similarity 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 0.16

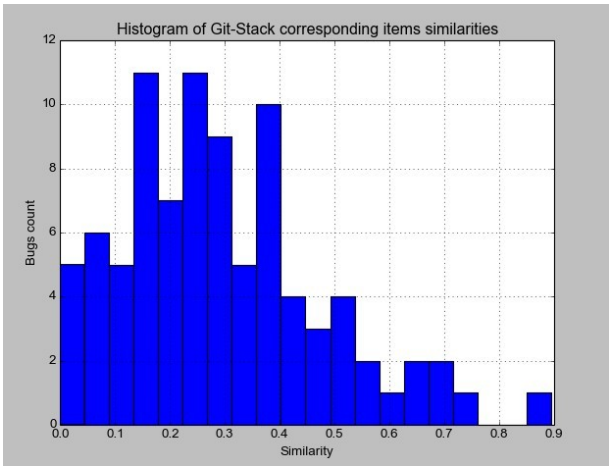


Figure 0.16: Histogram of similarities distribution among git issues and their matching SO questions

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 3 random git issues and resulting average similarity scores are in following figures.

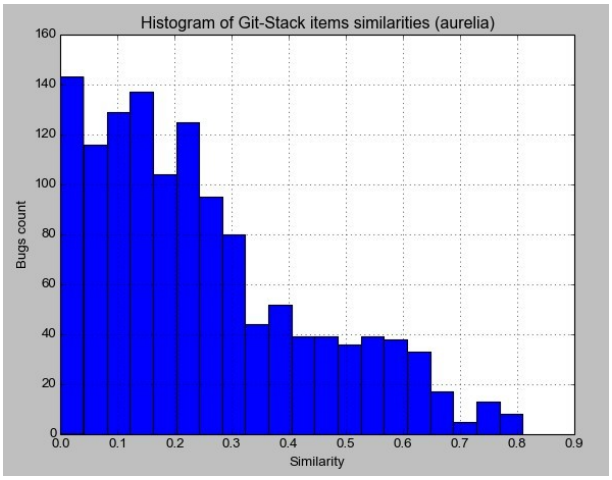
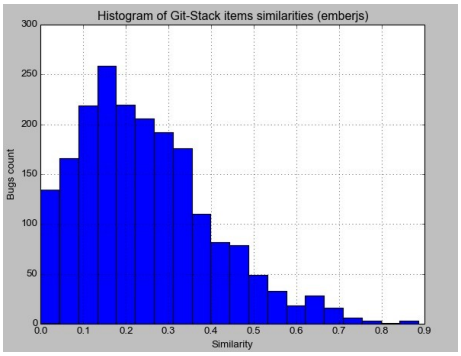
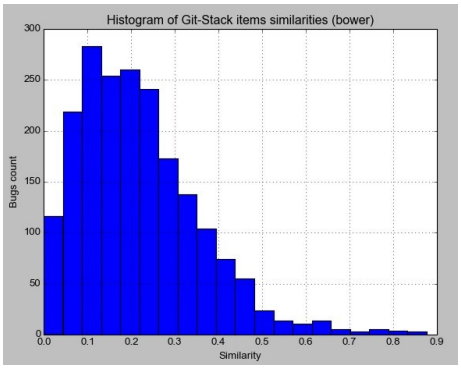


Figure 0.17: Histogram of Aurelia SO questions and random git issues. Average similarity was 0.244

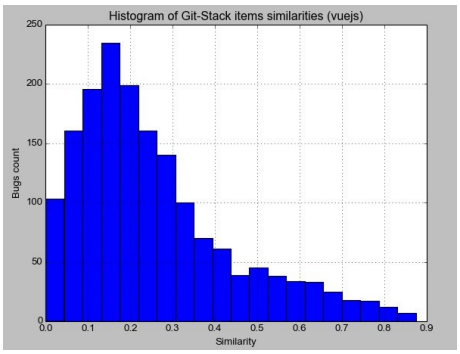


(a) EmberJS average similarity - 0.247

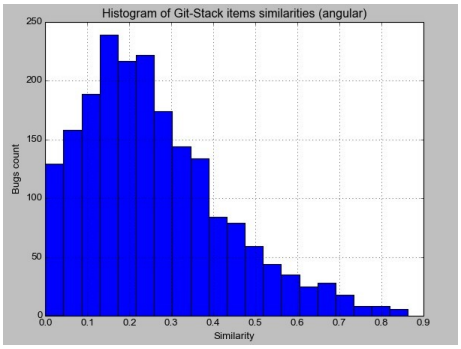


(b) Bower average similarity - 0.217

Figure 0.18: EmberJS and Bower similarity histogram



(a) VueJS average similarity - 0.255



(b) AngularJS average similarity - 0.258

Figure 0.19: VueJS and AngularJS similarity histogram

Comparing Git bugs descriptions with SO questions talking about own project issues and general issues, similarity values are following 0.18. Same table but for Reddit discussions can be seen in 0.19

Framework	**Own issues similarity**	**All issues similarity**
NodeJS	0.265	X
Angular	0.241	X
EmberJS	0.282	X
VueJS	0.261	X

Table 0.18: NLTK similarity values for SO questions

Reddit: Here I've calculated the similarity between the bug description and either particular comment in the reddit discussion which mentioned the bug or the whole discussion itself. Average similarity score for all considered projects (NodeJS, AngularJS, VueJS and EmberJS) was 0.481 for the whole discussion and 0.396 for the comment itself. Detailed scores for each project can be found in table 0.19. Subreddit for EmberJS didn't 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 0.19: Reddit NLTK similarity values

This indicates 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've plotted a relationship between similarity score and length in Figures 0.21 and 0.20.

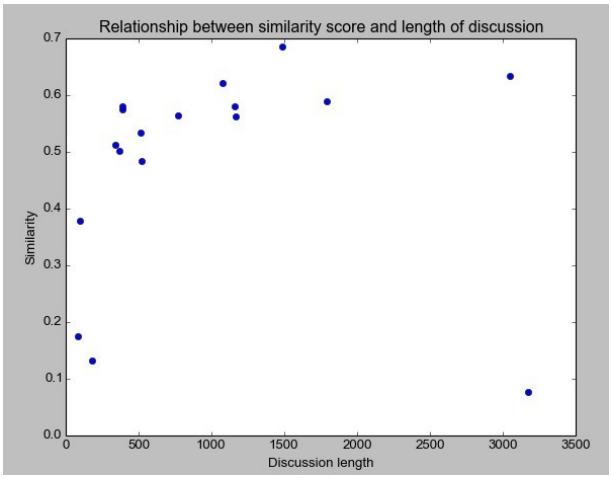


Figure 0.20: Discussion lengths and similarity scores with the issue

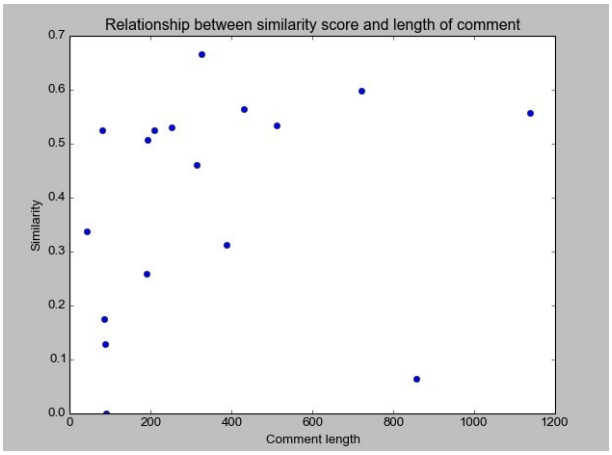


Figure 0.21: Comment lengths and similarity scores with the issue

17 Buckets

ReBucket measures the similarities of call stacks in crash reports and then assigns the reports to appropriate buckets based on the similarity values.[19]

18 Using GIT labels

One more considered approach how 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 don't 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 don't give any semantic information about the issue itself. That's the reason why this approach was rejected.

Discussion

discussion

Bibliography

[1] 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.

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

[3] 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.

[4] 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.

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

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

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

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

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

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

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

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