Github Repository Analysis Tool

System Documentation

Title of the Software: Github Repository Analysis Tool

Version Number: 1.0

Date: 17 June 2024

Authors/Contributors: Lasse Kuipers, Daniel Danilin, Eduardo Calvo, Teun de Waal, Thijs Berende, Eloy Wijlhuizen, Marie Oldeman and Maurits van ‘t Hag

Group: 7

Table of contents

System documentation

General overview

Our system is a code reviewing tool which can be used by the management or team leaders of a company to see the engagement with repositories on GitHub. It can help with management of personnel to see who is engaging with certain parts of coding projects and give some indication of the quality of work within a team.

This system is made in the form of a web application that consists of two sides of code, the frontend and the backend. The frontend is used to display data of a repository such as comments, commits, pull requests etc. as well as some scores that indicate the quality of data. It also makes sure the user can interact with the web app to display new visuals of different repositories that the user wants to analyze. The backend processes data and sends the processed data to the frontend through an API construction after saving it to a database. The backend requests data via the GitHub REST API. This data is then put in a local database which sends the correct data to the frontend whilst also sending calculated values based on the requested data e.g. engagement and a general semantic score.

Since this is an interactive app we went with a frontend-backend design. A frontend gives an easy user experience while the backend will take care of the underlying processes. For this structure, we chose to go with the combination of Django for the backend and Vue for the frontend. The main reason for this is because both are quite intuitive and easy to learn web application frameworks and are both designed with easy API integration in mind. This would make it a lot easier for us to create a working product.

Design considerations

Django

Our backend is built in Django as this already lays a good framework for dealing with API calls in an easy way. On top of this, the programming language of Django is Python, which is an easy to understand and easy to learn language with many capabilities. Still, Django also utilizes its own libraries to deal with an API framework in mind. We chose to integrate a local database that can store the data on our local system so we wouldn't need to ask for old data over and over again. This database is a SQLite file that is automatically created when starting a new Django project and will be explained more thoroughly in section Database. By saving the data to a database we improve the efficiency of our program since this leads to fewer API calls and therefore faster requests. The API calls to acquire the data are made to the GitHub REST API endpoints. With these API calls we try to get only the most necessary information needed for the analysis of a repository. For optimization, we make use of parallel processing explained in section on Asynchronous programming. When we receive the data we have multiple functions that create the engagement and semantic scores of different data types. Once all data has been stored and processed, it is sent to the frontend where a user can interact with it. From here onwards the backend awaits whether it should retrieve more data, update different scores or only send information within a specific date range.

Vue

The frontend of our web app is built in Vue.js or simply called Vue. Vue is an HTML web application framework that is specifically designed for making easy frontend design while still keeping some API functionality. It is often used together with Django. That is why we chose the combination. The languages used are CSS, HTML and Javascript, which are essential when designing a web application. None of us had any extensive previous experience with these languages but we knew there was enough documentation on these languages to help us along the way. There were two features specific to Vue that we ended up using quite a bit as they were surprisingly useful. The first was `Vue components`, this allows a programmer to create custom components that can be reused easily throughout the program. It is also possible to import certain components that were created by others. This allowed us to import components that were quite complex like a datepicker or components that we would use often like buttons. The second useful Vue feature was `Vue routing`. This let us change what was displayed on the page dependent on the URL even though we made use of a single-page application. Vue router also creates reactivity on the pages. Some examples of this are: when a button is clicked, a new page is loaded and a back button automatically leads to the previous page. These advantages of Vue and its compatibility with Django meant we were satisfied with our choice for this framework.

System architecture backend

Database

Our project utilizes SQLite3 as the backend database for data persistence. SQLite3 is a lightweight, file-based database management system (DBMS) that offers several advantages for development environments, especially for Django projects. First off all, it is built-in with Django, meaning that getting started is straightforward– no extra configuration or need for the installation of separate database software to run. Since it is natively integrated into Django, documentation about Django and SQLite3 is easily available.

Another big advantage is that SQLite3 databases are compact. All the data is stored in a single file, which makes it easy to manage and share. This is especially handy for smaller projects or during the early stages of development when storage space is not a major concern. Plus, sharing the database with your fellow developers or colleagues allows for collaboration on the same data locally.

While SQLite3 provides a lightweight and efficient way to store our project's data, we need a way to structure and organize that data for easier management and retrieval. This is where data models come into play.

These models, defined within the `models.py` file, act as blueprints for the various types of information we want to store in the database. Each model represents a specific entity or concept within our project, and defines the fields (data points) associated with it.

The objects defined in our project are: `Repo`, `PullRequest`, `User`, `Commit` and `Comment`. These encompass all types of data we need to store in our database in order to properly save and display information.

Types of models with their most important attributes needed for visualization in the frontend:

- Repo: the github repository. This is what the users will enter into our frontend.
  Important fields:
    - Personal access token: This is a token generated by GitHub to be able to use the GitHub REST API
    - Url: The URL of the repository
    - ID: Automatically generated number within database.
    - List of PullRequests: A list of all pull requests connected to the repository
- PullRequest: pull request of given repository, which is a proposal to merge a set of changes from one branch into another
  Important fields:
    - Title: The title of the pull request
    - Body: The message of the pull request
    - Date: The date when the pull request was created
    - All comments and commits: A list of all comments and commits associated with it
- User: Information of a specific user across all repositories in the database
  Important fields:
    - Name: The username of the user
    - Login: The login name of the user
    - List of all contributed PullRequests, Commits, and Comments: Lists of all pull requests, commits and comments a user made
- Comment: comments left under PullRequests, Commits or other Comments
  Important fields:
    - Date: The date when the comment was created
    - Body: The actual text of the comment
    - User: The user that made the comment
    - Related Commit: If a comment is related to a commit, the commit id is put here
    - Semantic score: The general semantic score of the comment
- Commit: commit on a PullRequest
  Important fields:
    - Date: The date when the commit was created
    - User: The user that created the commit

- Body: The associated message of the commit
- Semantic score: The general semantic score of the commit

Database Management

Django employs migrations to manage changes made to the database schema. These migrations ensure data integrity and facilitate controlled evolution of the database structure. Developers can create and apply migrations using the "`manage.py`" command. To ensure changes made to the models and data in the database, make sure to run those migrations, otherwise the implemented changes won't go into effect.

While SQLite3 is a good choice for development, Django's true strength lies in its flexibility. The project is built to work with various database backends. This means that although we currently utilize SQLite3, the "`DATABASES`" dictionary within your Django project's `settings.py` file allows for configuration with other database systems like PostgreSQL or MySQL if the needs of the user change in the future.

Endpoints

To retrieve data from GitHub, we utilize the GitHub REST API. This API supports many different endpoints that can be used to retrieve, post or delete data of certain GitHub repositories. For our project, only GET requests were used. Below is a list of the endpoints that the backend uses to retrieve data from the respective Github repository:

- **`GET/repos/{owner}/{repo}/pulls?state=all`**
  This endpoint is used to get all pull requests of a GitHub repository. By including "state=all" to the API link, both closed and open pull requests are fetched.
- **`GET/repos/{owner}/{repo}/pulls/{pull_number}/commits`**
  This endpoint is used to get all commits on a specific pull request.
- **`GET/repos/{owner}/{repo}/pulls/{pull_number}/comments`**
  This endpoint ensures that all comments, specifically review comments, are fetched from GitHub. Review comments are all linked to a certain commit of the pull request.
- **`GET/repos/{owner}/{repo}/issues/{pull_number}/comments`**
  This endpoint ensures that all comments, specifically issue comments, are fetched from GitHub. Issue comments are not linked to a certain commit, but either to another comment, or on its own.

Asynchronous programming

Early in the process of creating the software, the run times of the program became notably high. The reason for these high run times were because of a bottleneck, namely the amount of API calls that could be made per second. A good solution for tackling this problem proved to be the introduction of asynchronous API calls, as opposed to sequential API calls. Therefore, the decision has been made to integrate asynchronous programming in our software.

When looking into asynchronous processing, some important factors for what implementations should be used were found. When the program is more CPU-bound than I/O-bound, the use of threads of some sort is advised. However, the code in this software is more I/O-bound since it involves sending a HTTP request to the GitHub server and waiting for a response. As a result, asynchronous programming techniques are more valuable for this program and were used in this project for the purpose of concurrent processing.

The code for this program includes multiple levels of API calls that all need to be processed asynchronously to make the programming as fast as it could be. The levels include:

- **The pages**. When making an API request to the GitHub API, it only gives back a certain amount of information to avoid overflowing your program or database. To get more information, the same API request link should be used with the edition of a subsequent page number. This can be repeated until no pages remain
- **The pull requests**. On each page, there are pull requests of which the information can be called at the same time. Then, all information of the pull requests is handled asynchronously making the program much faster.
- **The commits/comments.** The information that is handled from the pull requests are the commits and comments. A pull request can include many commits/comments of which the information is also handled concurrently with asynchronous programming. For both commits and comments, pagination should also be taken into account which would add another layer of API calls that is handled asynchronously.

With these implementations of asynchronous programming, the program had an average speed increase of 80% in comparison to no use of asynchronous programming techniques. This speed up makes it possible to handle big GitHub repositories at almost the same speed of small repositories, making it essential for this project.

Semantic score

NLP metrics

In this section, the NLP metrics which are integrated into the web app, are outlined. The NLP metrics give a concise and descriptive overview of the contents of commit messages and comments by considering some important associated linguistic properties.

Lexical density: Theory

Lexical density is a measure which can represent the relative ‘meaningfulness’ of a text fragment. The metric is dependent on the number of words in a commit message or comment, and the number of nouns, adverbs, verbs and adjectives (part of the set of linguistic lexical categories). A high lexical density signifies that the commit message/comment is informative and context-rich, whereas a low lexical density might signify that the message lacks content or meaning. With this metric, informative and meaningful commit messages/comments can be encouraged.

For a given text fragment, the lexical density can be calculated by dividing the number of nouns, adverbs, verbs and adjectives by the total number of words in a commit message. This can be summarized in the following formula:

$$\frac{Number\ of\ nouns, adverbs, verbs, adjectives}{Total\ number\ of\ words}$$

This metric is part of the general semantic score for commit and comment messages.

Lexical density: Implementation and dependencies

The relevant functions for lexical density are carried out in the backend of the web app. The functions are implemented with the use of Python. The code contains documentation which explains the functionality of each of the functions. For calculating lexical density, a few additional dependencies are required to be installed. These specifics can be found in the User manual for this project.

Flesch reading ease: Theory

The Flesch reading ease is a measure which measures the readability of a text fragment. The Flesch reading ease is dependent on the number of words, sentences and syllables in a message. The Flesch reading ease can be calculated with the following formula:

$$206.835 - 1.015 * \frac{number\ of\ words}{number\ of\ sentences} - 84.6 * \frac{number\ of\ syllables}{number\ of\ words}$$

The Flesch reading ease maps a text fragment to a single numerical value. This statistic is part of the Flesch-Kincaid readability tests, a prominent method for the quantification of the readability of an English text fragment. A high Flesch reading ease translates to a text fragment which is more readable and easier to understand, whereas a low Flesch reading ease is found for difficult text fragments. Integrating this statistic within the web app may ensure that the quality of Git comments is maintained by not overcomplicating comments.

The total number of sentences is generally relevant for text fragments which are extracted from books, news articles and other sources which have coherent semantics across multiple sentences. For commit messages, unless they consist of multiple sentences by design, this is not always applicable. The semantics across multiple commit messages are not considered and therefore every commit message is treated as a single text fragment in this implementation.

Below is an example of a small table which represents average Flesch reading ease scores for different types of text fragments is provided below[1].

Text fragment type	**Average Flesch reading ease score**

[1] Courtesy of (https://web.archive.org/web/20160712094308/http://www.mang.canterbury.ac.nz/writing_guide/writing/flesch.shtml)

Comics	92
Sports illustrated *(Sports magazine)*	63
Wall Street Journal *(Newspaper)*	43
Standard auto insurance policy	10
Internal Revenue Code *(US federal statutory tax law)*	-6

As evident from the table above, the average Flesch reading ease score can also be below 0 for particularly difficult text fragments. In the context of our project, reading ease scores which fall below 0 are automatically set to 0 to ensure that our general semantic score is allocated within a fixed range.

Flesch reading ease: implementation and dependencies

Just like the lexical density score, the relevant functions for lexical density are carried out in the backend of the web app. The functions are implemented with the use of Python. The code contains documentation which explains the functionality of each of the functions. This NLP metric is also used in the calculation of the general semantic score.

Non-NLP metrics

In this section, the non-NLP metrics which are integrated into the web app are outlined. The non-NLP metrics encompass all metrics where linguistic properties of messages are not directly regarded, such as numerical counts, ratios and more.

Commit message length/code length ratio: Theory

The commit message length/code length ratio is a metric which contributes to the quantification of message informativeness by comparing the length of a commit message to the length of the changed code associated with a commit. With this metric, developers are encouraged to commit their code regularly to ensure commit messages are descriptive for the implemented changes. For large amounts of code adjustments associated with a short commit message, the metric returns a lower value, thus lowering the relative informativeness of a message. This metric is part of the general semantic score for commit and comment messages.

Commit message length/code length ratio: Implementation

The commit message length/code length ratio is computed as follows:

$$sigmoid(\frac{number\ of\ characters\ in\ commit\ message}{number\ of\ characters\ in\ adjusted\ code})\ *\ 100$$

where *sigmoid* represents the sigmoid function

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$

The sigmoid function, in combination with the multiplication factor of 100, ensures that for any x (representing the commit message length/code length ratio), the metric value is ranged between 0 and 100. A possible enhancement of the metric is the addition of a hyperparameter which represents the steepness of the slope in the sigmoid function, thus allowing for additional control over the influence the metric has on the general semantic score.

General semantic score

In accordance with the project requirements, a semantic score has been implemented for the web app with the aim of quantifying the informative value of commit messages and comments. An average general semantic score is available per repository, per pull request and per author. A general semantic score is also available for each commit and comment individually.

The semantic score is composed of three different metrics which have been previously introduced in this report:

NLP metrics:

- Lexical density (ranging from 0% to 100%)
- Flesch reading ease (ranging from 0% to 100%)

Non-NLP metrics:

- Commit message length/code length ratio (ranging from 0% to 100%)

The general semantic score for a commit message is computed in accordance with the following formula:

$$semantic\ score(message) = \frac{w1{*}LD(message) + w2{*}FRE(message) + w3{*}CM/CL\ ratio(message)}{w1 + w2 + w3}$$

LD denotes the lexical density metric, FRE denotes the Flesch reading ease metric and CM/CL ratio denotes the commit message length/code length ratio metric. Furthermore, $w1$, $w2$ and $w3$ represent the weights of each respective metric. By default, all metrics are weighted equally (*thus w1 = w2 = w3 = 1*), but the weights can be adjusted according to preference.

The formula is the same for the comments, without the addition of the *CM/CL* ratio and the third weight.

The semantic score per repository, pull request and author are computed as an average semantic score over all messages in the respective domains.

System architecture frontend-backend interaction

The system is designed so that the user can command the backend through the frontend. The frontend sends POST requests to the backend to retrieve the data from the database and order the backend to request data at the GitHub REST API. The backend processes and requests all the data before saving it to the database.

There are different kinds of packages sent back and forth between the backend and the frontend. The two main packages are the homepage and repository info package. All of these packages have their own path url within the backend.

Homepage package

The homepage package is collected through the `home` url in `urls.py` which leads to the `homepage_datapackage()` function. This will send all the different repositories available in the database so that the user knows what repositories they analysed previously and when they accessed it last.

Repository package

Once the user has requested a repository by either filling in the url of a new repository or clicking on an existing repository. When accessing a repository, a url containing the path `all/` is sent to the backend which requests the data from GitHub REST API. This is then stored in the database as mentioned before. The `package` url in `urls.py` links to `repo_frontend_info()`. This is used to send the data from the database to the frontend. The repository package can be specified with a date range. The repo url and date range are given as attributes of the POST request that is sent to the backend. This would make it easier to send the data if the user wants to only see a small part of the data. This package contains all information of that specific repository and this information is only updated when visiting the pull request and user page.

Delete

The delete post request is sent by frontend from the homepage when the user decides to delete a specific repository. The `delete` url takes a post request through `delete_entry_db` where the url of the repository is extracted. After this, the data related to this repository is deleted from the database.

Login / Logout

For the user to make use of the product they have to log in with account credentials. This process is done through a POST request, since the user of the product is stored in the database and therefore must be checked by accessing the database. This verification is done by accessing the `login_view` which is accessed through the `login` in `urls.py`. Logging out is done through a similar way but then `logout` in `urls.py` which automatically sends a redirect to frontend to the login page.

System architecture frontend

Router

During the planning phase it was clear that the web application would require multiple pages. This was necessary to be able to clearly display the necessary information and create a divide between all this information. The router for Vue.js allows for single page applications to run smoothly. By using a router, users can dynamically navigate between different pages.

The `index.js` file stores all the paths of all the pages, and is used to add new pages. To add a new page, one must add the page path, name and component. Other pages then refer to this file and can add new router links.

The router menu appears after a repository has been selected. This is a design choice that makes sure that the user does not open pages that do not yet have any information. The `RepositoryInfo`, `Pull Request` and `User` page are only filled after a valid already visited or new repository is selected.

Reading from JSON Files

JSON files are simple text based files that are easy to read from. It is used commonly for communicating data in web applications and is why it has been used for this project. Moreover, JSON is flexible for data exchange and is compatible with many different programming languages. These add to JSON's appeal for this project.

The frontend receives data to be displayed from the backend through JSON packages. They are extracted and can then be read in the frontend. The necessary variables are extracted in the script and returned so that they can be used later. Variables, if existent in the Json package, can be extracted and displayed as necessary. One important requirement when creating the JSON data packages was a unique value/ token for each repository, pull request and commit. This ensures that the frontend can accordingly display information and route between pages. The backend creates this information automatically when saving objects to the database the frontend has been set up to know which variables are unique for their respective repository, pull request or commit.

Security Measures

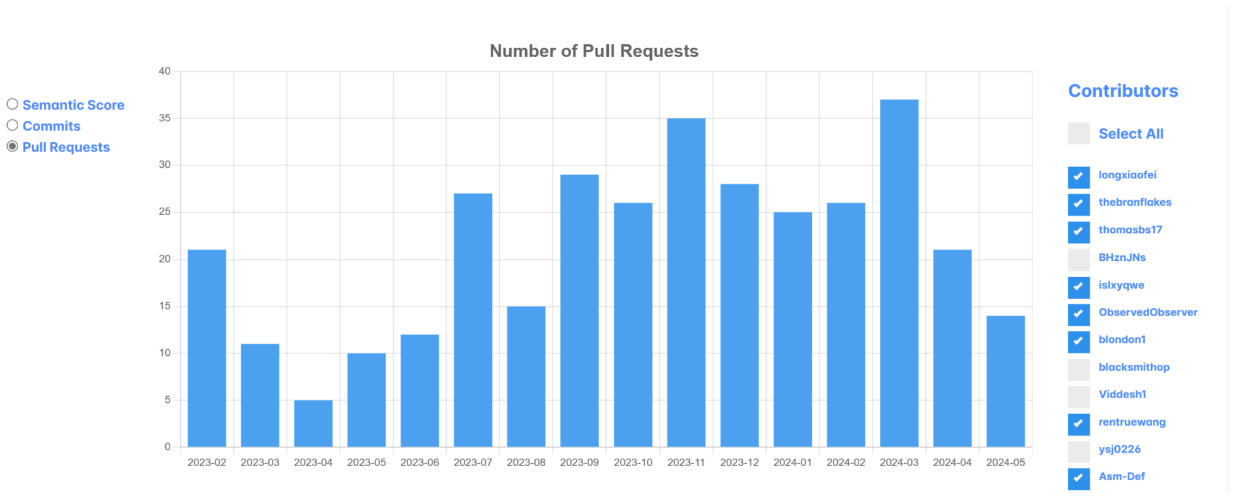
The first thing that is asked of the user on the homepage of the web application, is to enter the URL of a GitHub repository they would like to track. There is an input field which allows the user to enter anything they want. This could inevitably lead to security issues such as the entering of malicious input that could lead to an attack on the software. For example, a user could use an SQL injection to alter or delete data from our database which in turn would destroy the functionality of the web application. To prevent this from happening we deemed it necessary to check the content of the user input. We therefore created a function `checkInput(str)` which takes the user string as an input and tests whether the string matches a custom regular expression. We designed this regular expression in such a way that it only accepts valid URLs of GitHub repositories. If the string passes this test it gets sent to the backend where it will be used to fill the database with data. However, if the string does not pass this test, the user will see an error message on the home page, telling them that their input was invalid.

Sorting

Both the home page and the repository page contain lists. The home page has a list of tracked repositories, which entails all the repositories currently in the database. The repository page has a list of all contributors to that repository and a list of all pull requests made on that repository. To enhance the analytical insights, the program allows for sorting on these lists. The first sorting criteria we thought of was sorting by date, which would make it easier to find certain repositories or pull requests. Sorting on date is only possible for the list of repositories and the list of pull requests, as we did not think it would be useful or even possible to sort the contributors by date. By default, the lists are sorted by date and go from newest to oldest. However, the user can also select oldest to newest if they would rather see this. The other criteria was the semantic score of a contributor, pull request or repository. This sorting option does make sense for all the lists and would allow for a user of the web application to get certain insights: which pull request or repository is good and which is bad when it comes to semantic score. The sorting is done using a computed property which calls one of the sorting functions with the list of contributors, pull requests or repos. The sorting function for dates then sorts the lists according to dates. The sorting function for semantic score sorts the lists according to score.

Chart

The repository page contains a large central chart, which is our source of visualisation for our repository data. This chart serves as the essential tool for understanding how a team is performing overall, and even how subsets of the team are performing, across any selected time range. We used `chart.js` for its ease of use.



- ○ Semantic Score
- ○ Commits
- ◉ Pull Requests

Contributors

- ☐ Select All
- ✓ longxiaofei
- ✓ thebranflakes
- ✓ thomasbs17
- ☐ BHznJNs
- ✓ islxyqwe
- ✓ ObservedObserver
- ✓ blondon1
- ☐ blacksmithop
- ☐ Viddesh1
- ✓ rentruewang
- ☐ ysj0226
- ✓ Asm-Def

It is able to display three different sources of data: the average semantic score for commit messages over time, the number of commits over time, and the number of pull requests over time. Changing this display mode is done through the radio buttons on the left sidebar. All data is, by default, displayed on a per-month basis. This means the semantic score averages and the total counts of commits and pull requests are calculated per month. However, when a bar is clicked, the chart “zooms” into that month, showing these same calculations but on a daily basis, and limiting the range to the specific month that was clicked. On the right sidebar, there is a “Contributors” checkbox list. This displays all contributors to the repository, each with a checkbox, which allows for easy filtering of all displayed data, per user or subset of users. The contributor selection also affects the list of pull requests shown below the chart, at the bottom of the repository page. Above the chart is our date range selector, which gives one the ability to reload the repository data that is used in the frontend, as limited by the selected time range.

In addition, this entire system is functionally robust, meaning that the data mode (semantic score, commits, pull requests), zoom level (full range vs single month), contributor filter, and the date range selector, can all be modified at any point in time, regardless of the state of the chart, and work as expected.

We chose to make a single large chart - instead of multiple charts with different data on each - because that would allow for more fluid comparisons between different sources of data across a chosen timespan.

Future improvements and possible feature additions

Chart

For our chart, we could add more data sources or metrics to display, apart from the current three. Some examples could be:

- Total or active contributors over time

- An “Engagement Score”, holistically measuring active participation in the repository over time

NLP metrics

Flesch Reading Ease

- URLs (*e.g. https://github.com/google-research/google-research/tree/master/goemotions*) currently get low Flesch reading ease scores due to being treated as single words containing many syllables. Possible solutions to this may include assigning a fixed Flesch reading ease value to all URLs, or parsing the contents of URLs and treating the parsed URL as being a sentence consisting of separate words.
- The metric value output for a string consisting of 1 word, which is unknown to the NLTK ‘cmudict’ library, may vary significantly (where the bounded value ranges from 0 to 100).
- Optimize the heuristic for manual syllable count in syllableCount(word) function.
- Research possibilities of speeding up loading time by introducing asynchronicity in the functions in this file
- As in the case of lexical density, a good addition might be the calculation of an average Flesch reading ease score across multiple commit messages.

Sentiment analysis

Sentiment analysis is a technique in NLP which is used for approximating the sentiment of a text fragment. Generally speaking, the sentiment of text fragments ranges from entirely negative to entirely positive, often represented by a single numerical value which falls in the range between -1 and 1 (or sometimes 0 and 1). In this project, the two Python libraries which are responsible for sentiment analysis are TextBlob (Loria, 2018) and Vader (Hutto & Gilbert, 2014). Both libraries feature pretrained models which, with significant accuracy, can predict the sentiment of a text fragment.

In the web app, sentiment analysis should be used to monitor the dynamics of the mood in the team, based on the textual content of commit messages and comments. This metric can possibly give an insight into the development of team cohesion over a specified time range. Eventually it was decided that this metric would not be used on the creation of the semantic score. However, we still keep the file (`SentimentAnalysis.py`) in our project and this short explanation in order for the user to decide whether they want to include this metric or not.

Lexical Density

- Words unknown to NLTK get tagged as a noun by default. Given the abundant use of industry jargon etc. in messages which are exchanged within repositories, a more rigorous approach towards unknown word tagging may be a good possible future improvement.

- Research possibilities of speeding up loading time by introducing asynchronicity in the functions in this file.
- As in the case of the Flesch Reading ease, a good addition might be the calculation of an average Lexical Density score across multiple commit messages.

Non-NLP metrics

Commit message length/code length ratio

- This function does not function as expected for larger repositories due to

Sources

- Hutto, C.J. & Gilbert, E.E. (2014). VADER: A Parsimonious Rule-based Model for Sentiment Analysis of Social Media Text. Eighth International Conference on Weblogs and Social Media (ICWSM-14). Ann Arbor, MI, June 2014.
- Loria, S. (2018). textblob Documentation. Release 0.15, 2.