Software Application for the Visualization of Software Change

Student Name: **Ciaran Carroll**

Student ID: **16181492**

Supervisor: **Michael English**

Course: **BSc. Computer Systems**

Academic Year: **2019/20**

Submitted to University of Limerick 13th May 2020 in partial fulfilment of the requirements for my Final Year Project, 2019-2020

Table of Contents

Table of Figures

Acknowledgements

I would like to thank:

My supervisor, Michael English for his advice and patience throughout the year and for guiding me through the development and documentation process.

My family, especially my parents for supporting me financially and emotionally throughout my education.

My friends and girlfriend for supporting me and cheering me up in stressful times.

Abstract

Many successful software projects endure continuing change as they adapt to new requirements or fix bugs in the system. It is useful for software developers to be aware of the parts of the system that are most change prone. This can help with cost and resource estimations and with guiding code refactoring.

There are numerous existing tools which focus on the visualization of software change. However, these tools do not provide views to explore how the location of these changes, fluctuates over time. The aim of this project is to address this gap in software change visualization by mining change data from version control systems (VCS) and creating graphics to display the data.

1. Introduction

1.1 Context

Analytics is the discovery, interpretation, and communication of meaningful patterns in data while, “Software analytics is analytics on software data for managers and software engineers with the aim of empowering software development individuals and teams to gain and share insight from their data to make better decisions” (Buse & Zimmermann, 2012). It focuses on source code and software metrics and aims at describing, monitoring, predicting, and improving the efficiency and effectiveness of software engineering throughout the software lifecycle. The data collection is typically done by mining software repositories but can also be achieved by collecting user actions or production data.

Mining software repositories is an important activity during software evolution, as the extracted data is used to support a variety of software maintenance tasks. The key information extracted from these repositories gives a picture of the changes on the software system. To have a complete picture, tailored to the needs of the developer, the extracted data needs to be filtered, aggregated, and presented to the users. This manipulation of data is vital for the discovery of patterns.

This is a software analytics project which is focused on the change in source code over time and more specifically, how the location of change varies over time. The project includes the development of a software application which is able to visualize data mined from VCS repositories in order to make it easier to understand. The application is evaluated by its usefulness for identifying patterns in software development projects.

1.2 Motivation

When setting out to choose a focus area for this project, the aim was to work in an area that was both interesting and that integrated aspects of computer science. Previous experiences in software development projects was a major factor in the decision to pursue a project focused on

software change. From these experiences, it was noticed that projects change immensely over time and that Version Control Systems (VCS) are extremely helpful for recognizing where and when change is happening.

The application of the Pareto Principle in software development is also an interesting concept. More commonly known as the 80/20 rule, the Pareto Principle states that for many events, roughly 80% of the effects come from 20% of the causes. An interesting hypothesis for software development is: '80% of changes are made in 20% of the code'.
The possibility of creating an application which could investigate these was another motivation for the project selection.

1.3 Objective

The objectives of this work are as follows:

- Examine previous research in the area and technologies which could aid development.
- Design a system which can mine and visualize software change data.
- Implement the system in line with design requirements.
- Evaluate the system's ability to visualize change patterns.
- Investigate pareto principle in software change.

1.4 Report Roadmap

Chapter 2 presents a summary of findings on software change, data visualization, existing VCS visualization tools and Version Control Systems along with the APIs, Languages and Frameworks that could aid implementation. Chapter 3 presents the details concerning the architecture and design of the proposed system. Chapter 4 outlines the implementation of the proposed system while Chapter 5 documents how the system was evaluated. Finally, Chapter 6 presents conclusions and makes recommendations for future work in this area.

2. Research

This chapter outlines findings on software change and change data along with the different types of version control systems (VCS) and existing tools which visualize the data which is found within these systems. There is also an analysis of the different kinds of APIs, languages and frameworks that could be used to create the application.

2.1 Software Change

It is almost impossible to produce software systems of any size which do not need to be changed either during development or after release. Once software is put into use, new requirements emerge, and existing requirements change as the business running that software changes. Due to this continuous change, it is important to understand the change process, the different types of software change and to be able to identify key components within the change data.

In general, there are three intentions for software maintenance as described by Chapin et al. (2001). The first is to perfect the system in terms of its performance, efficiency, or maintainability (Perfective), the second is to add new functionality to the system (Adaptive) and the third is to fix faults in the system (Corrective).

Eick et al. (2002) provided some interesting insights on the change process and the key components of change data. The change process is broken down into four components. Features (functionality of the software to be added or improved), Initial Modification Requests (high level design information), Modification Requests (low level design information) and Deltas (changes to individual files).

They also describe the fundamental components of change data as can be seen in figure 1. Time is the date of change. Software space describes which files were changed and which lines were added and deleted. Developer is the engineer who made the change. Size describes how many modules, files and lines are affected. Effort is the developer hours required and interval is how long the change took in calendar time. Due to the project's focus on the location of change, the

main component of interest is software space along with the size and time components to investigate how the location and quantity of change differs over time.

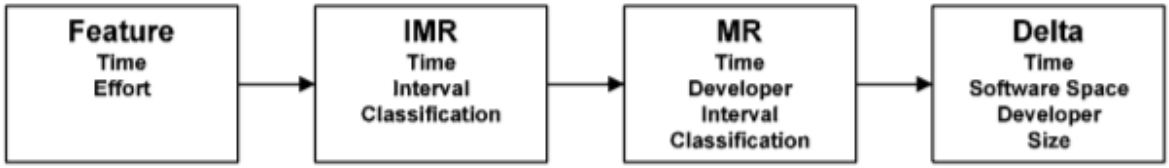

Figure 2.1: *The software change process and associated data (Eick et al., 2002)*

2.2 Data Visualization

Data visualization refers to the techniques used to communicate data or information by encoding it as visual objects (e.g., points, lines or bars) contained in graphics. Software visualization is a form of data visualization which focuses on information of and related to software systems. "Software visualization is a discipline that makes use of various forms of imagery to provide insight and understanding and to reduce complexity of the existing software system under consideration" (Knight & Munro, 1999).

When it comes to software visualization and data visualization in general, the major issue that arises is selecting the correct graphical representation or metaphor which can present the data clearly. This decision is made so difficult by the abundance of different graph types which can be displayed in both two- and three-dimensional formats. Also, progresses in modern technology have provided the ability to make graphs interactive or animated rather than static and this adds another factor to the choice. These factors highlight two major comparisons when it comes to visualization which are 2D vs. 3D graphs along with Static vs. Interactive.

Many visualization tools allow for turning graphical elements into three dimensional objects as a means of adding an extra design feature to the graph. However, the question is whether the use of the third dimension in these cases actually helps or hurts the effectiveness and appeal of the graph. Tullis and Case (2017) presents a study which conducted an online survey to determine if there is a measurable difference in performance and perception of reading 2D and 3D

representations of charts. The 1,423 participants first used either 2D or 3D versions of bar, column, donut, and pie charts to answer a series of questions and the speed and accuracy of their answers were recorded. They were then asked to rate the charts in terms of overall ease/difficulty of reading, ease of learning, effectiveness, confidence, visual appeal, familiarity with the chart type, and the overall appearance.

The findings, some of which can be seen in figure 2.2, show strong performance differences in favour of 2D charts with participants answering the questions faster and being more accurate. The 2D charts also hold a slight edge over 3D charts when it comes to the ratings. These findings help support the use of 2D representations over 3D representations however it must be remembered that in these instances the added dimension was only being used for visual effect and not for the purpose of portraying another element of data. The conclusion is that the use of a third dimension purely for decorative depth is more likely to distort and confuse the eye of viewer than to aid their understanding. The real value of a graph lies in its ability to reduce the complexity of data and the third dimension in this case, harms this value.

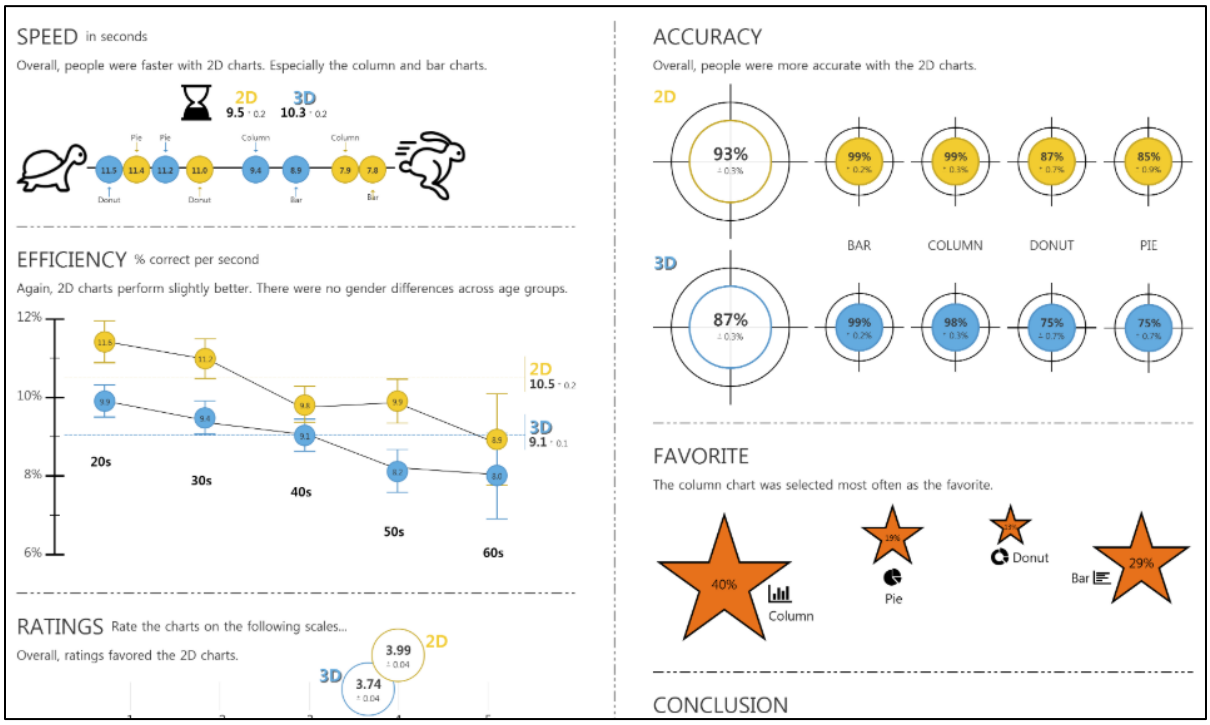

Figure 2.2: *Results from Tullis & Case (2017)*

However, there are cases including in the visualization of code structures where the use of a third dimension can be very beneficial for including another element of data. Fronk et al. (2006) presents their aim of visualizing code structures of java software systems in three dimensions. They state that 3D space allows for display of large amounts of data with optimal visual ingress. They address the issue of scalability when using UML to visualize code structures. The issue in this case is that large software systems are hard to depict with a single UML package or class diagram and integration leads to a picture which is unusable due to how confusing it is. To combat this, they create a tool which analyses java source code and creates a 3D scene which represents the structural model. This example shows where a third dimension can be helpful, and it also leads to the next comparison of static and interactive graphics.

The tool developed by Fronk et al. is only effective when combined with the ability to rotate zoom and fade the image in order to fully examine the large amount of data. Weissgerber et al. (2016) mention how compared to static visualizations, interactive graphics give freedom for additional exploration of the presented data. Interactivity is particularly powerful when you allow users to select points or series in a chart and for a summary of the relevant data or a more specific graph to then appear. This is extremely effective when looking for variation or trends over time which is big part of this project.

2.3 VCS Visualization Tools

“Version control systems (VCSs) are used to store and reconstruct past versions of program source code. As a product they also capture a great deal of contextual information about each change” (Ball et al., 1997). Due to the plethora of data that is found within these systems, mining software repositories is an important activity during software evolution and the next step is often the visualization of this data. Reuter et al. (1990) recognize the importance of data visualization. They mention its inherent power for conveying complex data due to way in which the human eye is able to percept visual representations of data. For this reason, many tools have been developed to provide visualizations of data.

During the course of my research I came across the following VCS Visualization tools:

- Seesoft (Eick et al., 1992)
- Source Viewer 3D (Maletic et al., 2003)
- Xia (Wu, 2003)
- softChange (German, 2004)
- CVSscan (Voinea et al., 2005)

For the remainder of this section, the aims and functionality of two of these tools are outlined in the form of Seesoft and softChange. These two are detailed further because they provide two differing types of software visualization tool while also both holding features which create representations of LOC (lines of code) in order to discover patterns and this correlates with the focus of this project.

2.3.1 Seesoft

The Seesoft visualization tool displays line-oriented source code statistics by reducing each file and line into a compact representation. These statistics include information such as MR (modification request to a VCS), who wrote each line of code, when it was last changed, whether it fixed a bug or added new functionality, how it is reached, how often its executed and so on. By means of direct manipulation and high interaction graphics, the user can manipulate this reduced representation of code in order to find patterns. Seesoft derives data from a variety of sources such as version control systems, static analyses (e.g. locations where a function is called) and dynamic analyses (e.g. profiling).

The reduced representation is achieved by displaying files as columns and lines of code as thin rows within these columns. The colour of each row is determined by a statistic associated with the line of code that it represents. Users are able to open code reading windows that display the actual code corresponding to the rows. This can be achieved by clicking the reading window button which creates a small magnifying box. Moving this over the rows shows the associated code. This allows the user to connect the visual aspect of the tool with the code that it represents.

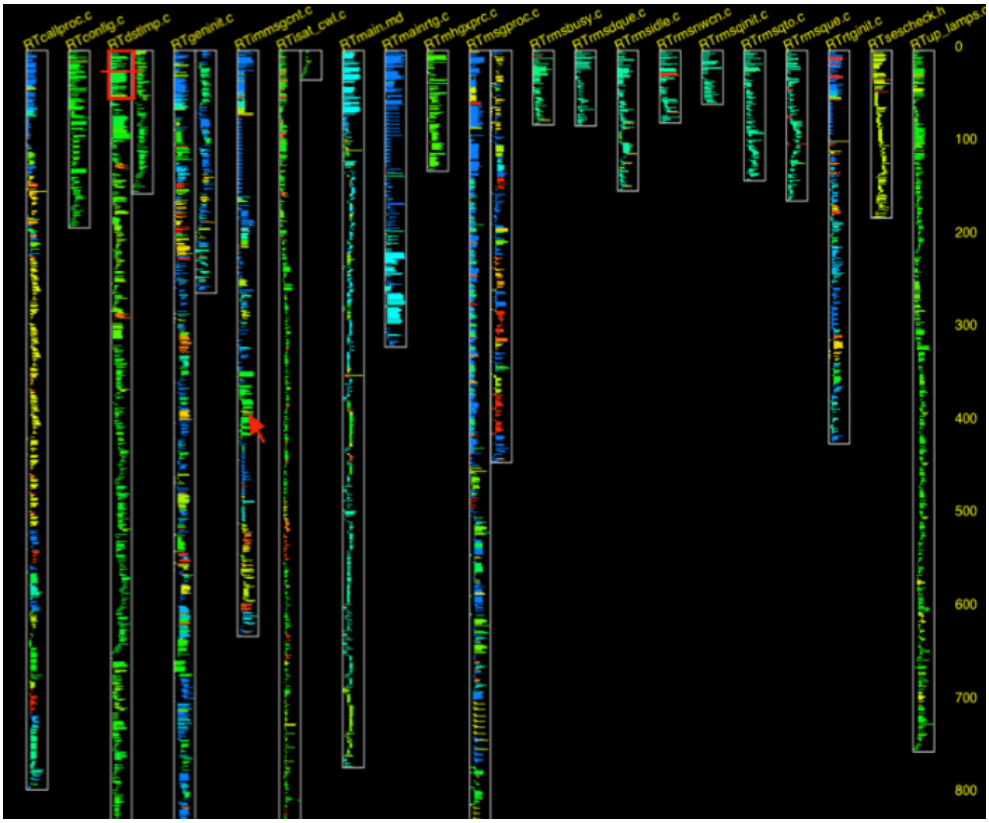

***Figure 2.3**: Seesoft GUI (Wang et al., 2015)*

2.3.2 softChange

softChange is a tool that retrieves information such as mail messages, software releases, version control logs, defect reports and then analyses and enhances it by finding new relationships amongst it. The aim being to increase understanding of how a software product evolves over time. The tool is intended for researchers who are interested in understanding how software evolves.

The softChange architecture (figure 2.4) is composed of four main components. A database to store all the historical information, a trails extractor to retrieve software trails such as ChangeLogs and bug reports, a trails analyser to generate new facts using the recovered information and a visualizer to allow the user to explore the trails. The visualizer for softChange is divided into a browser, a chart generator and an interactive graphical visualizer.

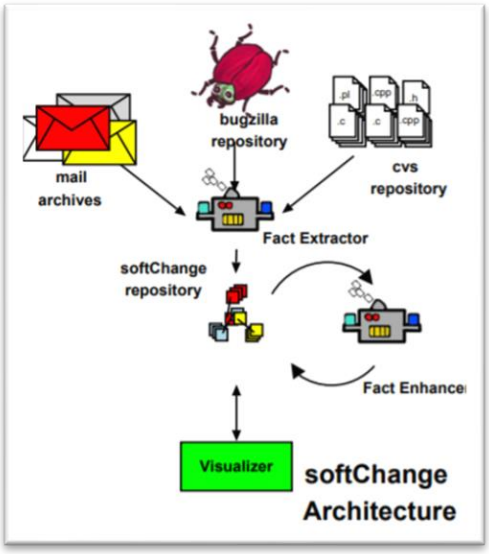

Figure 2.4: *The softChange architecture (German & Hindle, 2006)*

The softChange browser is a hypertext application that permits the exploration of MRs, the files they contain, and any changes made to these files. These changes include: a list of the functions added, modified or deleted, a pretty-printed version of the differences between source code, both in its original form, and after the comments and empty lines have been removed. The chart generator that plots different types of information as two-dimensional graphs (e.g. LOCS vs. time, Number of MRs vs. time). The graphical visualizer is an interactive application that allows the user to explore the history of the project as a collection of graphs highlighting things such as file authorship and coupling. Each visualization can provide interesting insights into the evolution of the project.

2.4 Version Control Systems

As mentioned in section 2.3, Version Control Systems record changes to a file or set of files over time so that you can recall specific versions later. They have become essential to coordinating work on a software project. In most organizations developers use either a Centralized Version Control System (CVCS) like Subversion (SVN) or Concurrent Version System (CVS) or a

Distributed Version Control System (DVCS) like Git or Mercurial. The differences between the two can be seen in figure 2.5.

CVCS came first and they are based on the idea that there is a single central copy of your project somewhere and programmers will commit their changes to this central copy. DVCS were introduced next where every developer clones a copy of a repository and has the full history of the project on their own hard drive. Since their introduction DVCS have grown in popularity and many software projects now use DVCS over CVCS. Some reasons for this are given by De Alwis and Sillito (2009). These benefits of DVCS include providing first-class access to all developers, simple automatic merging, improved support for experimental changes and support for disconnected operations. Git has become the most popular version control system as a result.

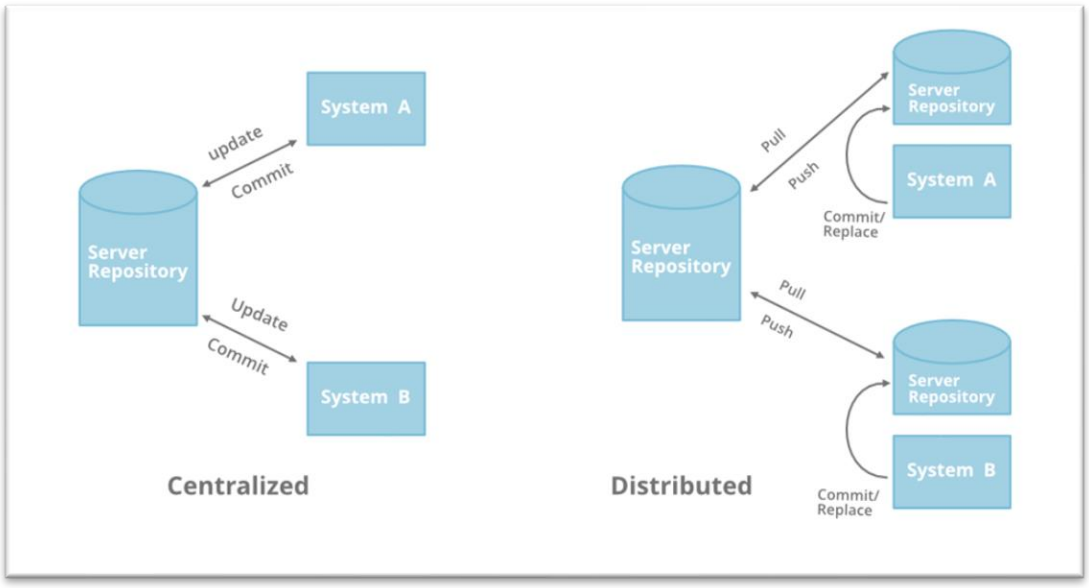

Figure 2.5: CVCS v DVCS

During recent years GitHub has become the most popular repository hosting service for Git projects. As of January 2020, GitHub (2020) reports having over 40 million users and more than 100 million repositories. For this reason, GitHub provides the greatest range of repositories from which data can be mined and then visualized. Kalliamvakou et al. (2014) discuss some of the promises and perils of mining data from GitHub. Some of the perils are that many repositories have few commits, are inactive or they are not actually for software development. They also

outline the issue that GitHub repositories do not necessarily represent entire projects due to the different branches being treated as if they are completely different repositories. This could mean that work outside of the master branch is missed when studying the repositories data. Apart from these issues, it is noted that GitHub is still a remarkable and thriving resource and it will continue to be an attractive source to mine for research. To make this research effective they suggest considering what type of repository a study needs and to then sample suitable repositories accordingly.

To study GitHub data it is first necessary to gain access to it. Access is provided by the GitHub API (2020) which is an interface provided by GitHub for developers that want to develop applications targeting GitHub. It is a RESTful (Representational State Transfer) API. A RESTful API is an API that uses HTTP requests to GET, PUT, POST and DELETE data. All API access is over HTTPS and is accessed from https://api.github.com. Data is sent and received as JSON. All information regarding the use of the APIs services can be found at https://developer.github.com/. Here explanations can be found on composing URLs to access the different resources which are available.

2.5 Languages & Frameworks

The contents of this section include my investigation into languages and frameworks which could be useful for the implementation of my application. I outline my findings on both Python and JavaScript.

2.5.1 Python

“Python has emerged over the last couple of decades as a first-class tool for scientific computing tasks, including the analysis and visualization of large datasets” (VanderPlas, 2016). This is due to the vast array of libraries dedicated to the exploration, analysis and visualization of data. Libraries such as Pandas, NumPy and SciPy provide developers with the tools needed to examine data efficiently and extract data relevant to their interests. Meanwhile, libraries such as

Matplotlib, Seaborn and Datashader offer endless opportunities when it comes to the visualization of data.

PyGitHub (2020) is a Python (2 and 3) library to use the Github API v3. With it, you can manage Github resources (repositories, user profiles, organizations, etc.) from Python scripts. It provides methods and objects which make accessing the data found within the GitHub API much simpler. To gain access to these, a developer only needs to install the PyGitHub library, import the GitHub module and provide an access token which can be created in GitHub user settings.

Plotly Dash (2020) is a Python framework for building web applications. It is built on top of Flask, Plotly.js, React and React Js. It enables developers to build dashboards using pure Python. Dash apps are rendered in the web browser. You can deploy your apps to servers and then share them through URLs. Since Dash apps are viewed in the web browser, Dash is inherently cross-platform and mobile ready. A Dash application is usually composed of two parts. The first part is the layout and describes how the app will look like and the second part describes the interactivity of the application. Dash provides HTML classes that enable us to generate HTML content with Python. The dash_html_components library provides classes for all of the HTML tags, and the keyword arguments describe the HTML attributes like style, className, and id. The dash_core_components library generates higher-level components like controls and graphs. Dash apps can be made interactive using the the app.callback decorators which wait for Inputs and update Outputs as a result.

2.5.2 JavaScript

I also investigated the capabilities of JavaScript when it came to integration with the GitHub API. JavaScript is very convenient when parsing JSON as the data format translates directly to an object literal in the language. JavaScript also holds the benefit of having many modern frameworks such as React which can be very useful for building user interfaces. React allows developers to create large web applications which can change data, without reloading the page and it is fast, scalable, and simple.

The Fetch API provides a JavaScript interface for accessing and manipulating parts of the HTTP pipeline, such as requests and responses. It also provides a global fetch() method that provides an easy, logical way to fetch resources asynchronously across the network. A basic fetch request takes one argument, the path to the resource you want to fetch and returns a promise containing a Response object. This is just an HTTP response, not the actual JSON. To extract the JSON body content from the response, the json() method is used.

In general, this method for accessing data is more complex than the simplified option which is provided by the PyGitHub library. One major factor in this is that the developer has to deal with pagination manually. For queries where there are multiple pages of results, the developer has to create a separate call to access the data of each page. On the other hand, PyGitHub calls return their own PaginatedList object of results which allows the entire result set to be iterated over with a simple for loop as can be seen in figure 2.6. This removes a lot of the complexity of accessing data through the API.

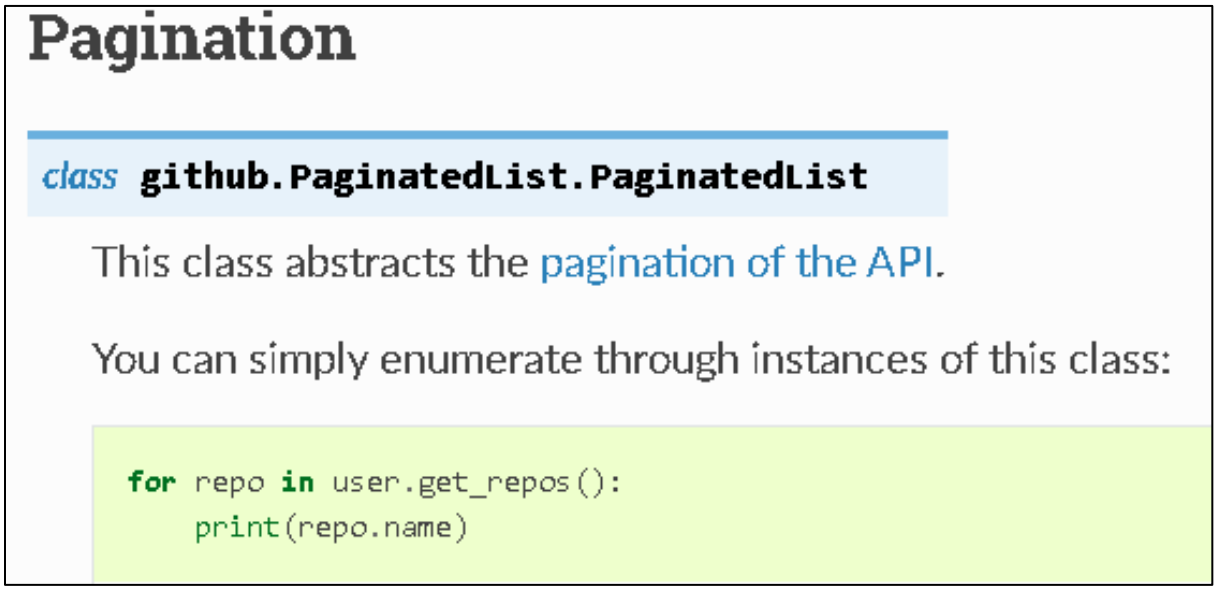

Figure 2.6: *PyGitHub (2020) Pagination*

2.6 Focus of Proposed Work

The proposed application is a web app which is capable of mining and saving data from the GitHub API including information about repositories and commits. In addition, the proposed tool provides features to visualize the data using interactive two-dimensional graphics due to them being attractive and easily understood. This is achieved by utilising the Python programming language with primary focus on the Dash Framework along with the PyGitHub and Pandas (2020) python libraries. Design considerations and implementation specifics of the proposed system are presented in chapters 3 and 4, respectively.

3. Architecture & Design

This chapter begins by presenting a brief description of the requirements of the proposed system including a system overview and its main use cases. It then presents a brief description of the architecture of the proposed system. Finally, there are sections outlining layouts, callbacks and data extraction and storage.

3.1 Requirements

The user should be able to navigate through the different parts of the web app. It should be possible to search for GitHub repositories based on certain keywords. The user should be able to enter the details of a repository and the relevant change data should be extracted and stored. Multiple visualizations should be displayed for a selected repository on another page with the option to explore at either file or folder level of abstraction. The user should be able to interact with the graphs by clicking points as well as hovering over them.

The system comprises of three main components:

- GUI / Layouts: the screens which the user sees.
- Data: extracting and storing data from GitHub.
- Callbacks: responses to different user inputs such as text, selections, clicks and hovering.

The main use cases of the system as seen in figure 3.1 are:

- User searches for a repository using keywords.
- User enters repository details to extract its change data.
- User explores a repositories data through graphics.

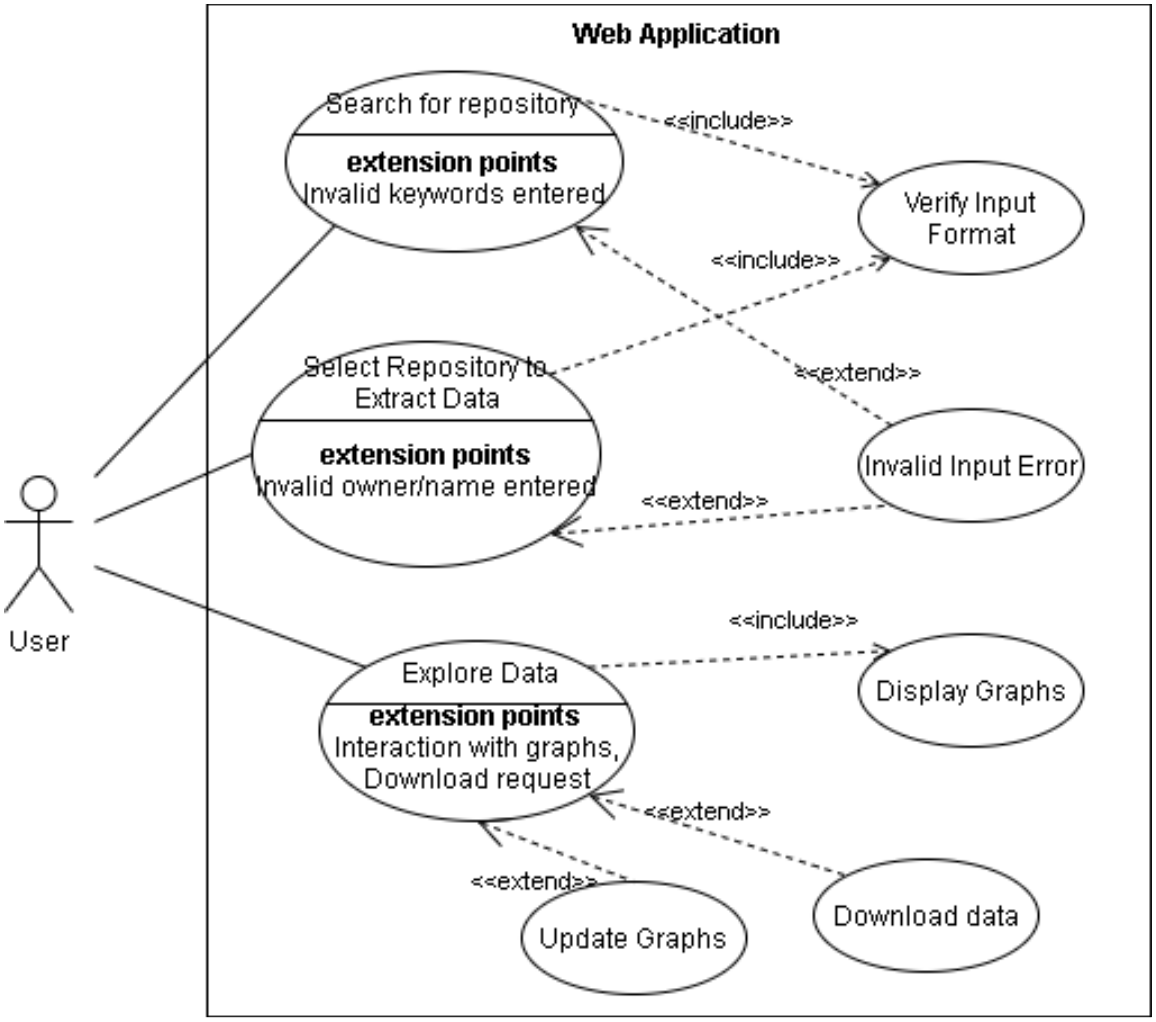

Figure 3.1: *Use case diagram for System*

3.2 Architecture

As the application is developed using the Dash python framework, it has a similar architecture to all Dash projects. Dash apps are composed of two parts. The first part is the "layout" of the app and it describes what the application looks like. The second part describes the interactivity of the application called “callbacks”. Dash renders web applications as a "single-page app" and applications can be created in a single file. However, it is possible to make the project into a multi-page app with URL Support and you can structure the app where each page/layout is contained in a separate file. This type of structure allows for the reduction of the complexity of each file making implementation and understanding of the code much simpler.

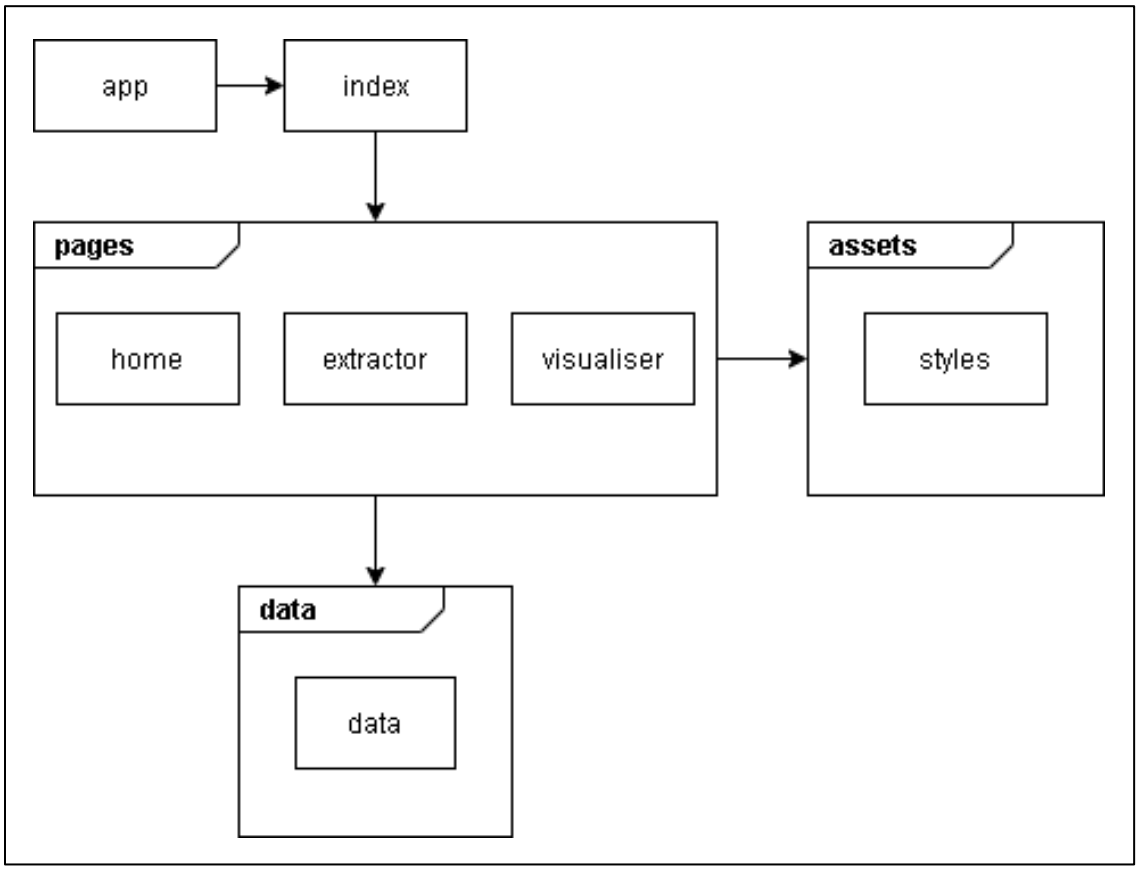

***Figure 3.2**: Structure of System*

As in figure 3.2, the structure of the project is broken down into five main parts, some of which are directories which are broken down further:

- App: Dash instance defined.
- Index: entry point for running app and provides URL routing.
- Pages: package holding files for each page.
    - Home: provide an overview of the project.
    - Extractor: extract data from GitHub.
    - Visualizer: display interactive graphics for user to explore data.
- Data: change data stored for repositories which have been mined.
- Assets: cascading style sheet for adding style to the different views.

3.3 Layouts & GUI prototypes

Dash layouts describe what the user sees. They are made up of different dash components such as the dash html, core and bootstrap component libraries. The html components library provides python structures for different html elements such as headers, divs and links. A core set of supercharged components for interactive user interfaces, written and maintained by the Dash team, is available in the core components library. This includes dropdowns, graphs, inputs and many more. Finally, dash bootstrap components are a library of Bootstrap components for Dash that make it easier to build consistently styled apps with complex, responsive layouts. Dash bootstrap provides components such as Rows, Columns and Navbars.

As there are three different pages, there is a different layout for each. The three layouts have a navigation bar to allow for navigation between the pages which is provided by the bootstrap component library. More specifically the Nav and NavItem structures. Apart from this, the home page is the least complex with it just providing plain text about the project which mainly comprises of simple html components. The other two pages are composed using the dash bootstrap Row and Col structures which allow for the creation of grids. The extractor also uses the Input and Button structures for taking user input. Finally, the visualizer page makes use of the Dropdown and Graph structures found in the dash core components library to create dropdowns and graphs which the user can interact with. The graphs used are two dimensional rather than three dimensional. This is due to the likelihood of the third dimension reducing user understanding when being used purely for visual decoration as was discovered in section 2.2.

Figures 3.3, 3.4 and 3.5 show GUI prototypes for the three pages previously described. In the figures, the black lines represent locations for text and the rectangles with an X running through them represent areas of the GUI where graphs are presented.

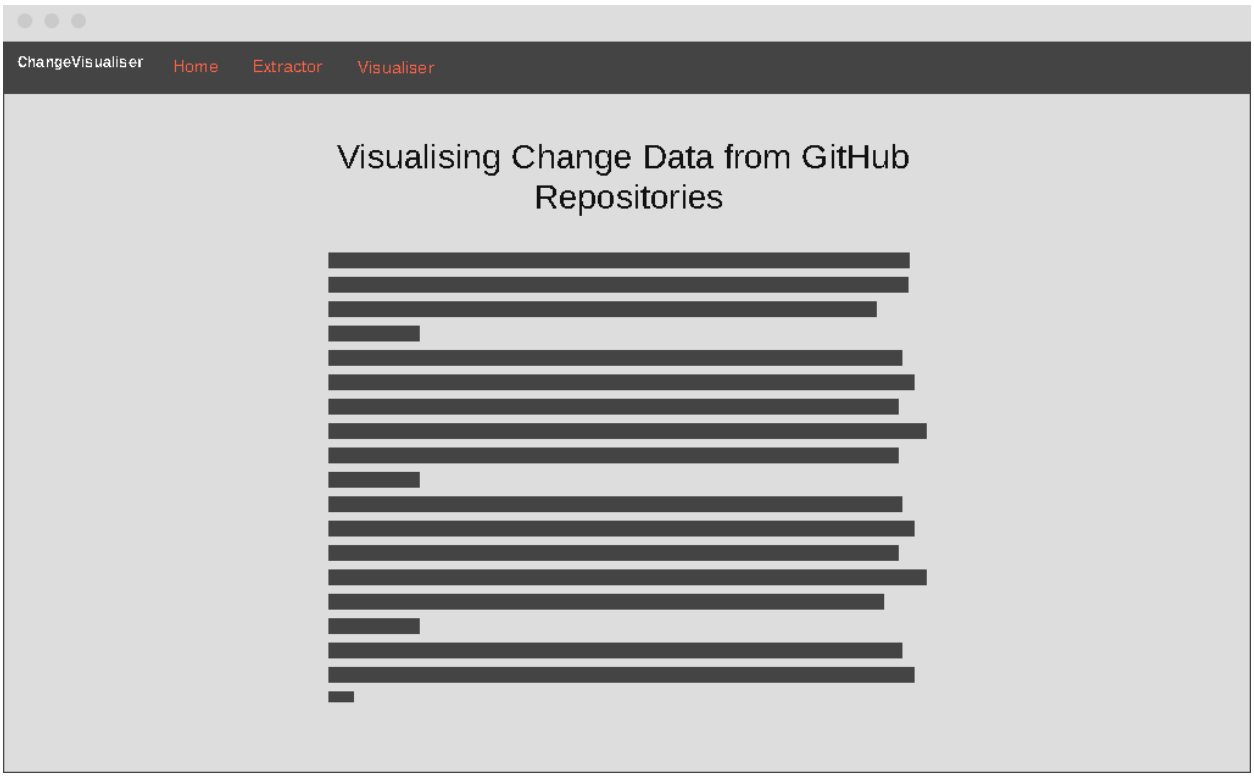

***Figure 3.3**: Home page GUI prototype*

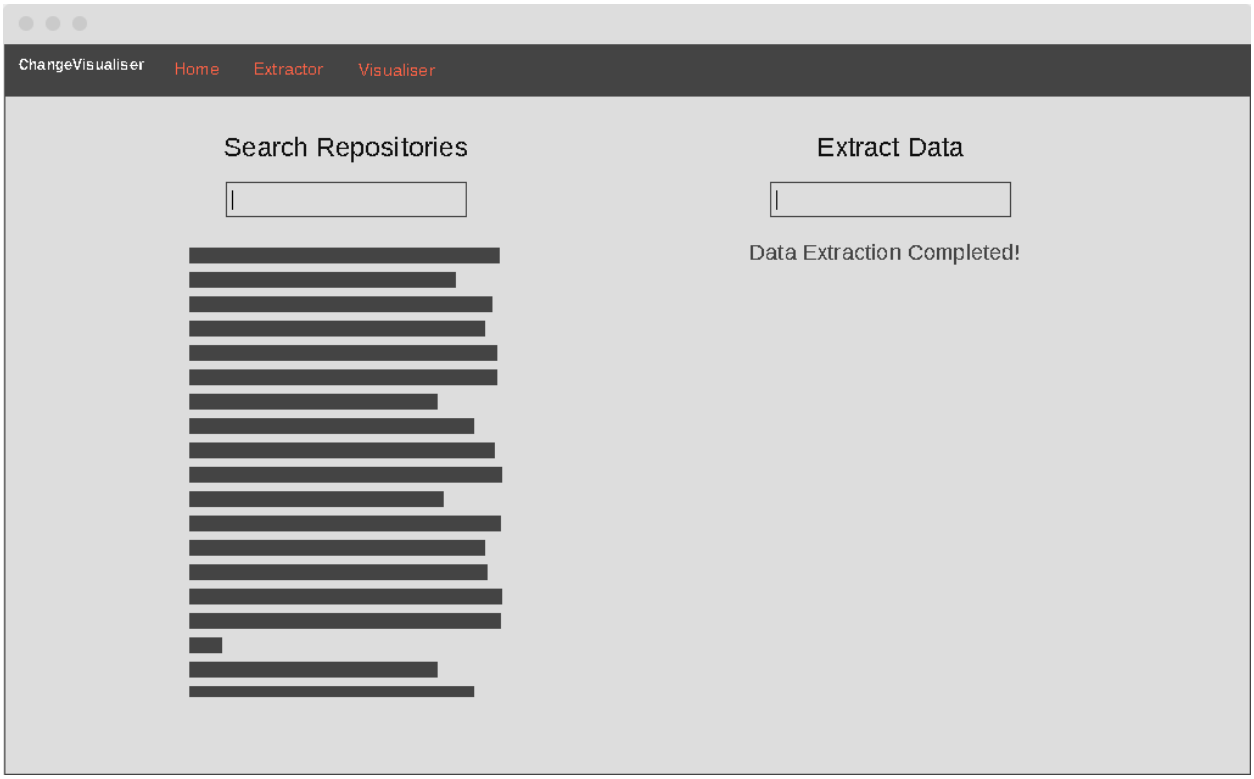

***Figure 3.4:** Extractor page GUI prototype*

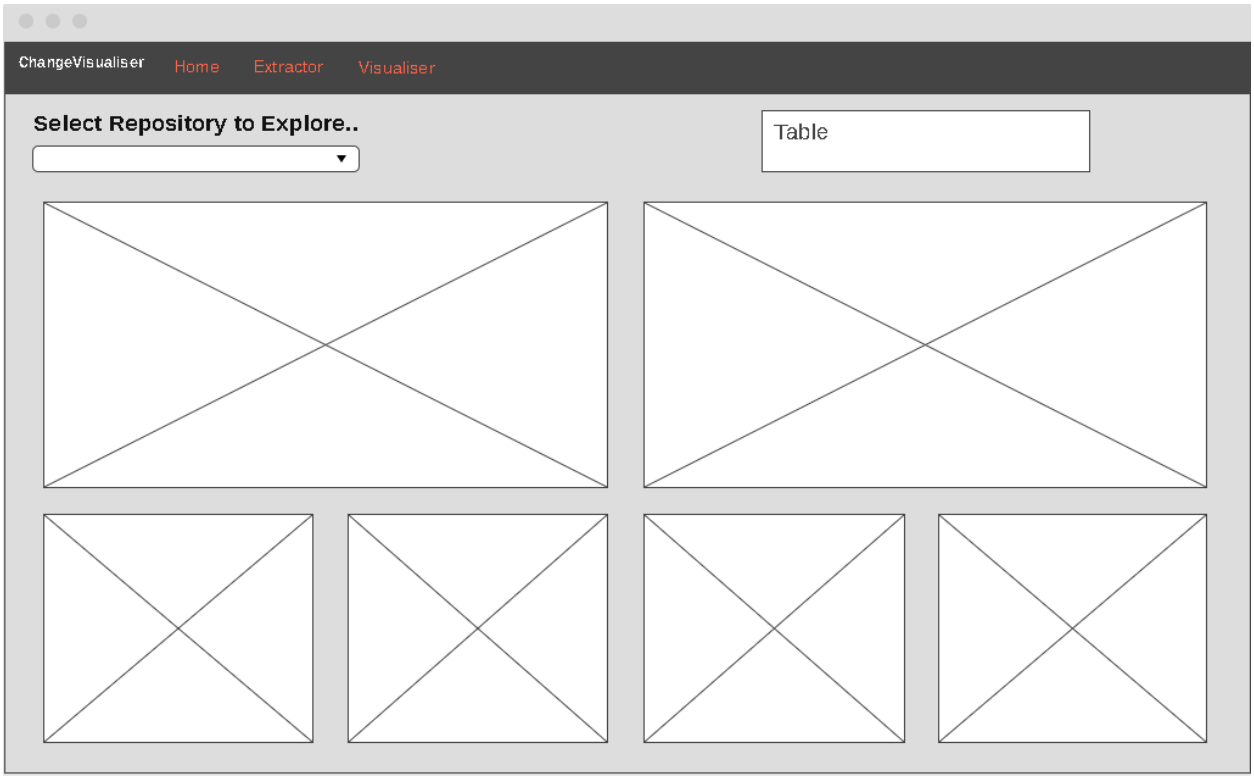

***Figure 3.5**: Visualizer page GUI prototype*

3.4 Data

In order to create visualizations, data is mined from GitHub due to it being the most popular version control system today as detailed in section 2.4. For this application, communication with the GitHub API is done using the PyGitHub python library. Regardless of the resource, which is required, the first step in this process is importing the library followed by creating a GitHub instance. The instance can be created using a personal access token or a username and password. The instance for this application is created with a read only access token because it has no need for writing data to GitHub. The first resource required is a list of repositories most relevant to the search parameters entered by the user. The application takes the name and commit count of the repositories and outputs them in table format. The next resource required is a list of all commits for a specific repository. The data is first placed in a Pandas dataframe. The repository name is stored in a repositories dataframe are there is also a dataframe which holds the relevant change data identified in section 2.1 for each commit. This being filename, total change, additions,

deletions and date. An example of the structure of one of these stored change data dataframes can be seen in figure 3.6.

Date	Filename	Total	Additions	Deletions
2020-05-01	README.md	4	4	0
2020-05-01	app/src/main/AndroidManifest.xml	2	2	0
2020-04-01	app/src/main/java/io/bluetrace/opentrace/bluet...	1	1	0
2020-04-01	README.md	5	5	0
2020-04-01	.gitignore	19	19	0

***Figure 3.6**: Example Change Data Dataframe*

The next step is storing the data in a file. The Pandas library makes storing dataframes in numerous different file format very simple. Three popular storage formats are CSV, Pickle and HDF5. If you save data as a csv you are just storing it as a comma separated list. The main advantage of saving in CSV is having a standardized format that can be opened with a wide range of software/languages but reading and writing from these files is relatively slow. The pickle module implements binary protocols for serializing and de-serializing a Python object structure. “Pickling” is the process whereby a Python object hierarchy is converted into a byte stream, and “unpickling” is the inverse operation, whereby a byte stream (from a binary file or bytes-like object) is converted back into an object hierarchy. The benefits of pickling include that it supports almost all python data types, (de)serialization is very fast and file sizes are relatively small while its negative is that it only works with python. The Hierarchical Data Format version 5 (HDF5), is an open source file format that supports large, complex, heterogeneous data. HDF5 uses a "file directory" like structure that allows you to organize data within the file in many different structured ways. This means it is possible to store multiple dataframes in a single file, each of which is referenced by a unique key while achieving similar speeds to that of pickling. Mined data is stored in a HDF5 format in the application for this reason. An example of the way the data is stored by the application is seen in figure 3.7 which shows the directory like format of HDF5. Here the Repositories “directory” or key holds a dataframe which has the name of all repositories which have been mined. There is also a key associated which each repository name which holds a change data dataframe as seen in figure 3.6.

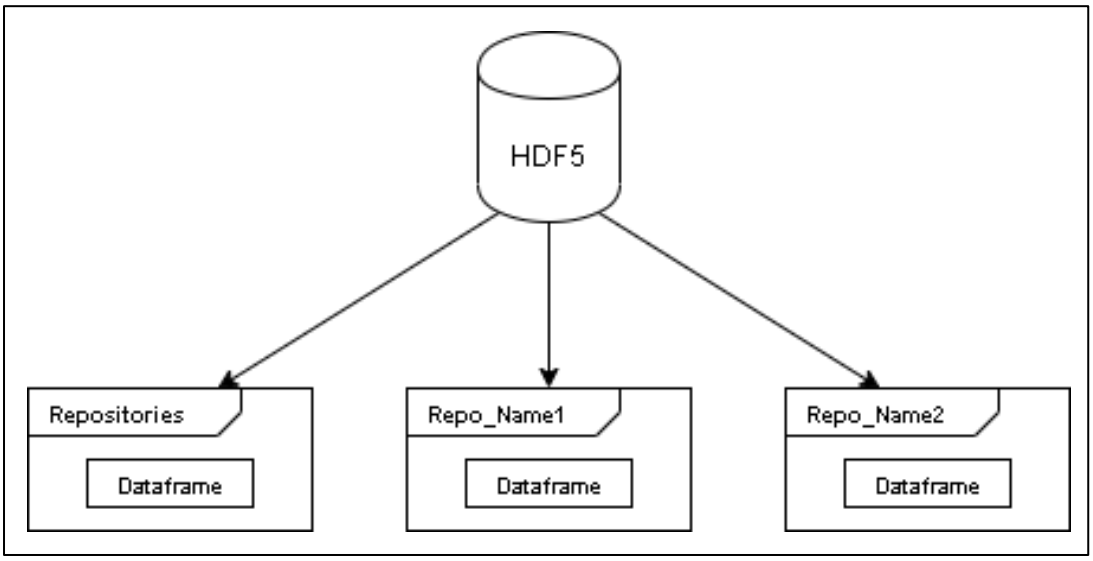

Figure 3.7: *Example Data Storage Format*

3.5 Callbacks

Interactive graphics are very effective when it comes to exploring data as was established in section 2.2. Callbacks play a major role in allowing the user to interact with visualizations created by this application. The "inputs" and "outputs" of the application interface are described declaratively through the app.callback decorator. In Dash, the inputs and outputs of the application are simply the properties of a particular component. Whenever an input property changes, the function that the callback decorator wraps will get called automatically. Dash provides the function with the new value of the input property as an input argument and Dash updates the property of the output component with whatever was returned by the function. Input and Output must be imported from dash.dependencies. This process is an example of reactive programming. Frequently the callbacks update the children of a component to display new text or the figure of a dcc.Graph component to display new data, but they can also update the style of a component or even the available options of a dcc.Dropdown component.

The first callback needed for my application is part of my index script. It makes navigation through different pages possible by mapping each specific page layout to their own URL. The

callback updates the layout based on changes to the value contained in the dash_core_components.Location component.

The home page does not require any callbacks due to it being a static page. The extractor page has two dynamic DIVs which provide results after each submit button is clicked. This functionality is provided by two separate callbacks. The first returns a table of relevant repositories and their commit count based on the keywords entered by the user. The second extracts and saves data from the repository which was selected and provides confirmation to the user.

Finally, the visualizer page requires the greatest quantity of callbacks due to it being the most dynamic page of the three. There are callbacks to update the dropdown values and to update the different graphs based on the dropdown values selected. There are also several callbacks to deal with click, select and hover events.

4. Implementation

This chapter gives details on the implementation of the application including navigation along with the extractor and visualizer pages. It also provides screenshots of the GUI and an explanation on how the application was deployed. It contains sample code to aid the explanation of the functionality of the different parts of the system. The system was written in python due to its powerful data science libraries mentioned in section 2.5.1 using PyCharm as the development environment.

4.1 Navigation

As was described in section 3.2 the application is broken into three pages. For this reason, it was necessary to implement a navigation function to allow the user to switch between these pages. Due to Dash apps being rendered as single page applications, it was necessary to create an index.py file which is the entry point for the system and sets the current layout shown to the user depending on the URL in the address bar. The index.py file first imports the dash instance which is created in the app.py file as shown in figure 4.1.

```
import dash
import dash_bootstrap_components as dbc

external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = dash.Dash(__name__, external_stylesheets=[dbc.themes.BOOTSTRAP])
app.title = 'ChangeVisualizer'
server = app.server
server.secret_key = os.environ.get('secret_key', 'secret')
app.config.suppress_callback_exceptions = True
```

Figure 4.1: *Dash instance created in app.py*

The dash instance is created in a separate file to make sure there are no circular imports. The instance being created in the index.py file would cause this because the index.py file needs to import the layout from each of the different pages while the three pages need to import the app instance to provide to the decorator of their callback functions. This implementation allows the

four python files, one for each page and the index.py file to import the app instance from the app.py file instead.

The other aspects of this file are the declaration of the app layout and a callback which decides which page to display as seen in figure 4.2. The layout is composed of two components, a Location core component which holds the value of the URL in the address bar and a Div html component which represents the entire screen space of the browser window. The callback function holds the @app.callback decorator and takes input in the form of the pathname which is stored in the Location component. The function then uses an if-else statement to decide which layout to return to the Div which represents the page content. A 404 error is returned if the pathname doesn't match with any of the available pages.

```
app.layout = html.Div([
    dcc.Location(id='url', refresh=False),
    html.Div(id='page-content')
])


@app.callback(Output('page-content', 'children'),
              [Input('url', 'pathname')])
def display_page(pathname):
    if pathname == '/extractor':
        return extractor.layout
    elif pathname == '/visualizer':
        return visualizer.layout
    elif pathname == '/':
        return home.layout
    else:
        return '404'
```

Figure 4.2: *Layout and Navigation callback - index.py*

Finally, the index.py file runs the server as seen in figure 4.3. The code checks if the current file is the main module and runs the server if true. By default, Dash apps run on localhost and can only be accessed on the machine which they are run on. To share a Dash app, it must be deployed to a public server. This will be discussed further in section 4.5.

```
if __name__ == '__main__':
    app.run_server(debug=True, dev_tools_ui=False)
```

***Figure 4.3**: Running server - index.py*

The different page file layouts must also hold a Nav bootstrap component to allow for navigation between them. The Nav component shown in figure 4.4 holds NavItems which themselves hold NavLinks. The NavLinks take a string to display to the screen and a href property which will update the address bar when one of the links is clicked. There is also a property in one of the NavLinks which makes it clear to the user which page is currently active.

```
dbc.Nav(
    [
        html.H5('ChangeVisualizer', className='header'),
        dbc.NavItem(dbc.NavLink('Home', active=True, href='/')),
        dbc.NavItem(dbc.NavLink('Extractor', href='/extractor')),
        dbc.NavItem(dbc.NavLink('Visualizer', href='/visualizer')),
    ],
    pills=True,
    style={
        'background-color': '#333',
        'margin-bottom': '20px',
        'padding': '10px'
    }
),
```

***Figure 4.4:** Nav component and descendants - home.py*

4.2 Extractor

The purpose of the extractor page is to take input from the user in order to retrieve and display desired GitHub data. In order to gain access to GitHub data, a GitHub object from the PyGitHub library needs to be created. The instance of the object can be created using a user's username and password or else a personal access token. The former is not recommended due to the security issues which come with it. Tokens can be created on the GitHub website once logged into an account and they can be created with different permissions depending on what they are for. The

token for this application only allows for reading data from public repositories because there is no need for the system to write any data to GitHub. The token is stored in a pickle which is loaded into the extractor.py file in order to create the GitHub instance as seen in figure 4.6.

```
pickle_in = open("./api_key.pkl", "rb")
api = pickle.load(pickle_in)
g = Github(api['key'])
```

Figure 4.5: *GitHub instance creation - extractor.py*

The extractor page is split into two sides. On the left is a feature for searching for repositories and on the right is a feature for selecting a repository to mine change data from it. As seen in figure 4.7, the page is split in two by using the dash bootstrap Row and Col components which allow for the creation of a grid layout in dash. Each of these main columns is made up of a FormGroup which describes how the input should be formatted and takes user text, a Row featuring a submit and info button, and a Div which is used to output the different parts of the result. The latter is placed in a dash core Loading component which displays a spinning wheel while waiting for results without needing to create a callback to do so. This ensures that the user knows the application is still working while they are waiting for a response.

```
dbc.Row(
    [
        dbc.Col(
            [
                dbc.FormGroup(
                    [
                        html.H4(dbc.Label("Search GitHub Repositories using Keywords")),
                        dbc.Input(placeholder="Enter keywords...", id='left-input-box', type="text"),
                        dbc.FormText("(format : 'keyword, keyword etc.')"),
                    ]
                ),
                dbc.Row(
                    [
                        dbc.Col(dbc.Button("Submit", color="primary", className="mr-1", id='left-button')),
                        dbc.Col(dbc.Button("Info", color="secondary", className="mr-1", id='left-open')),
                    ]
                ),
                dcc.Loading(id="loading-icon", children=[
                    html.Div(id='left-output-container', children=[
                        html.Div(id='search_api'),
                        html.Div(id='core_api'),
                        html.Br(),
                        html.Div(id='repos-found'),
                        html.Div(id='datatable')
                    ])
                ], type="default")
            ]
        ),
```

***Figure 4.6**: Layout for left Col – extractor.py*

The info buttons are implemented using Modal dash bootstrap components as seen in figure 4.8. The modal box is split into a header, body and footer. A dash callback function is required to make the modal visible when the user clicks on the info button and to make it disappear again when they click the close button. This callback can be seen in figure 4.9. The function checks if either of the buttons have been clicked and changes the state of the modal as a result.

```
dbc.Modal(
    [
        dbc.ModalHeader("Search Info"),
        dbc.ModalBody("""
            Keywords are searched for in the Name, Description and README file of public repositories.
            Results are sorted by best match, as indicated by a score field which is returned with each result. This
            is a computed value representing the relevance of an item relative to the other items in the result set.
            Multiple factors are combined to boost the most relevant item to the top of the result list.
            """),
        dbc.ModalFooter(
            dbc.Button("Close", id="left-close", className="ml-auto")
        ),
    ],
    id="left-modal",
),
```

***Figure 4.7:** Modal dash bootstrap component - extractor.py*

```
@app.callback(
    Output("left-modal", "is_open"),
    [Input("left-open", "n_clicks"), Input("left-close", "n_clicks")],
    [State("left-modal", "is_open")])
def toggle_modal(n1, n2, is_open):
    if n1 or n2:
        return not is_open
    return is_open
```

Figure 4.8: *Callback to make modal in/visible*

Each side has a callback which takes an Input box and submit Button as input, one of which can be seen in figure 4.7. For the search feature on the left, the input taken is multiple search keywords separated by a comma. As seen in figure 4.9, the callback first checks to see if a value has been entered and the submit button clicked. If so, the input string is broken up into a list of words using the split() method with a comma as the separator. These words are then joined with a '+' in between to form part of the query which will be given to the PyGitHub search_repositories() function. The rest of the query specifies where to search for the keywords and for this application that is in the name, description and README file of the repository. The result of the function is a list of Repository PyGitHub objects. Two lists are created to hold repository names and commit counts along with a count for the current index of these lists.

```
@app.callback([
    Output('search_api', 'children'),
    Output('core_api', 'children'),
    Output('repos-found', 'children'),
    Output('datatable', 'children')],
    [Input('left-button', 'n_clicks')],
    [State('left-input-box', 'value')])
def update_output(n_clicks, value):
    if n_clicks is None or value is None:
        raise dash.exceptions.PreventUpdate
    else:
        keywords = [keyword.strip() for keyword in value.split(',')]
        query = '+'.join(keywords) + '+in:name+in:readme+in:description'
        result = g.search_repositories(query)

        repos, commits = ([] for i in range(2))
        count = 0
```

Figure 4.9: *Search callback - extractor.py*

Figure 4.10 shows the applications iteration through the different repositories in the result list using a for loop. With each run through a repository name and commit count is added to each of the lists. To make sure the process is not overly time consuming, the for loop is stopped after the fifty best matches have been taken. Results are sorted by a score field which is a computed value representing the relevance of an item relative to the other items in the result set. Outputting more results would be slow because a separate core GitHub API call is made within the repo.get_commits() method. Finally, the two lists are placed in a dataframe to make outputting as a dash bootstrap Table component easier. The other parts of the result output to the screen include a total count of repositories found along with the number of Search and Core API calls remaining for the GitHub access token being used by the application. The Search and Core rates start at 30 and 5000 respectively and refresh after an hour.

```
for repo in result:
    url = repo.clone_url
    url = url.replace('https://github.com/', '')
    url = url.replace('.git', '')
    repos.append(url)

    try:
        commits.append(repo.get_commits().totalCount)
    except GithubException:
        commits.append(0)

    count += 1
    if count == 50:
        break

data = {'Owner/Name': repos, 'Commits': commits}
df = pd.DataFrame(data=data)

try:
    total_count = result.totalCount
except GithubException:
    total_count = 0

rate_limit = g.get_rate_limit()
rl_search = rate_limit.search
rl_core = rate_limit.core
```

Figure 4.10: *Search results - extractor.py*

On the right of the extractor page, there is a feature for extracting change data from a repository. The repository to be mined is selected by inputting the unique id of a repository which is the owner name and repository name combined with a backslash in between. Similarly, to the search feature, the callback for this feature takes Input and Button components as input. Once, the submit button is clicked the callback first makes sure that there is a value in the input box before checking if the file used for storing data currently exists. If so, a list of repositories previously mined is retrieved from the data file as can be seen in figure 4.11. Otherwise an empty list is created.

```
if os.path.exists('./data/data.h5'):
    reposdf = pd.read_hdf('./data/data.h5', 'repos')
    repositories = reposdf['Name'].tolist()
else:
    repositories = []
```

***Figure 4.11**: Check that data file exists - extractor.py*

```
try:
    repo = g.get_repo(value)
    languages = repo.get_languages()
    lang_exts = get_extensions(languages)
    value = value.replace("/", "_")
    value = value.replace("-", "_")
    dates, filenames, totals, adds, dels = ([] for i in range(5))
    commits = repo.get_commits()
    for commit in commits:
        date = commit.commit.author.date.date()
        date = date.replace(day=1)
        files = commit.files
        for file in files:
            fname = file.filename
            if fname.endswith(tuple(lang_exts)):
                dates.append(date)
                filenames.append(fname)
                totals.append(file.changes)
                adds.append(file.additions)
                dels.append(file.deletions)

    data = {'Date': dates, 'Filename': filenames, 'Total': totals, 'Additions': adds, 'Deletions': dels}
    df = pd.DataFrame(data=data)
    df.to_hdf('./data/data.h5', key=value)
```

***Figure 4.12**: Mining change data - extractor.py*

Figure 4.12 displays how the application extracts change data if the repository id entered is associated with a GitHub repository. The Repository object is retrieved using the PyGitHub get_repo() function. The application then gets a list of the programming languages which are used in the repository using the get_languages() function and this list is used to create a list of file extensions using the get_extensions() function part of which is shown in figure 4.13. The system then gets all the commits for the repository using the get_commits() function and it iterates through the list that is received as a result using a for loop. For each iteration, the application also looks at every file which is part of the commit and for files that match one of the collected extensions, the date of the commit, filename, total change, additions and deletions of LOC are all stored in separate lists. Once each commit has been iterated through, the five lists are placed in a dataframe before being written to a HDF5 file. As mentioned in section 3.4 multiple dataframes can be store in the same HDF5 file and this is made possible by using a unique key when reading and writing to the file. Due to the combination of owner and repository-name being unique for GitHub repositories, this string is used as the key for an individual repository. This same string is also placed in another dataframe which is part of the file with ‘repos’ as the key and this holds a list of all repositories previously mined.

```
def get_extensions(languages):
    extensions = []
    for x in languages:
        if x == 'Python':
            extensions.append('.py')
        elif x == 'Java':
            extensions.append('.java')
        elif x == 'JavaScript':
            extensions.append('.js')
        elif x == 'C':
            extensions.append('.c')
            extensions.append('.h')
```

Figure 4.13: *get_extensions() function - extractor.py*

4.3 Visualizer

The visualizer page allows the user to explore the change data through visual means such as tables and graphs. The main selections by the user are what repository and abstraction to explore. The options are provided by two dropdowns, the first holds all the names of the repositories ready to explore and the second gives a choice of file or folder level of abstraction. The layout for these dropdowns can be seen in figure 4.14.

```
dcc.Dropdown(
    id='repository_title',
    options=[{'label': i, 'value': i} for i in available_repositories],
    placeholder='Select Repository...'
),
dcc.Dropdown(
    id='abstraction',
    options=[{'label': i, 'value': i} for i in abstractions],
    placeholder='Select Abstraction...'
),
```

Figure 4.14: *Repository and Abstraction dropdowns – visualizer.py*

There are several callbacks which rely on these two values in order to create relevant graphics. The first provides a link to download the selected data as a CSV file through the browser as shown in figure 4.15. The callback takes the two selections as input, obtains the relevant data and returns a csv string to a dash hyperlink component which downloads the data to the user's device when clicked. There is another callback which returns two tables, one to display the number of files, folders and LOC change for the repository and another which presents the percentage of LOC change for the entire project that happens in the top twenty percent most change prone files. The latter is used to investigate the pareto principle '80% change happens in 20% code" which was proposed in section 1.2 and is discussed further in the evaluation of the application. The implementation of the main parts of this callback can be seen in figure 4.16. It first reads in the dataframe stored for the selected repository and this is then manipulated in order to get the number of files, folders and total LOC change. The value for the pareto principle table is calculated by dividing the total change by the change in the top twenty percent most change prone files, multiplying by one hundred and rounding the result.

```
@app.callback(
    dash.dependencies.Output('download-link', 'href'),
    [Input('repository_title', 'value'),
     Input('abstraction', 'value')])
def update_download_link(repotitle, abstraction):
    if repotitle is None or abstraction is None:
        return None
    else:
        df = pd.read_hdf('./data/data.h5', repotitle)
        if abstraction == "Folder Level":
            df['Filename'] = df['Filename'].str.rsplit("/", 1).str[0]
            df.loc[df['Filename'].str.contains('.', regex=False), 'Filename'] = ''
            df['Filename'] = df['Filename'].astype(str) + '/'
        csv_string = df.to_csv(index=True, encoding='utf-8')
        csv_string = "data:text/csv;charset=utf-8," + quote(csv_string)
        return csv_string
```

Figure 4.15: *Callback to download data as CSV – visualizer.py*

```
df = pd.read_hdf('./data/data.h5', repotitle)

df_file = df.groupby("Filename").sum()
df_file = df_file.sort_values(by=['Total'], ascending=False)
file_count = df_file['Total'].count()
change = df_file['Total'].sum()

df['Filename'] = df['Filename'].str.rsplit("/", 1).str[0]
df.loc[df['Filename'].str.contains('.', regex=False), 'Filename'] = ''
df['Filename'] = df['Filename'].astype(str) + '/'
df_folder = df.groupby("Filename").sum()
df_folder = df_folder.sort_values(by=['Total'], ascending=False)
folder_count = df_folder['Total'].count()

stats_1 = ["Folders", "Files", "Change (LOC)"]
counts_1 = [folder_count, file_count, change]
data_1 = {'': stats_1, 'Count': counts_1}
df_1 = pd.DataFrame(data=data_1)

_20p_files = round(file_count * 0.2)
file_totals = df_file['Total'].tolist()
del file_totals[int(_20p_files):]
change_20p_files = sum(file_totals)
file_percent = round((change_20p_files / change) * 100)

stats_2 = ["80% Change (LOC) in 20% Files?"]
percents_2 = [file_percent]
data_2 = {'Pareto Principle': stats_2, 'Percent': percents_2}
df_2 = pd.DataFrame(data=data_2)
```

Figure 4.16: *Collecting values for visualizer tables - visualizer.py*

The other callbacks for this page are most important for exploring how the location of change varies over time because they create the graphs which can be interacted with in order to compare different parts of the data. The initial graphs displayed are a line chart which displays values for additions, deletions and total LOC change for the different monthly periods and a bar chart which shows the total change for the most change prone files/folders. The inputs for both of these callbacks are the two dropdown values and a hover data which is associated with the opposite graph to which the callback is outputting to. Figure 4.17 shows the inputs and output for the line chart callback and the bar chart is similar but the ids for the output and hover input are switched. These callbacks output a figure to dash core Graph components which are placed in the page layout as seen in figure 4.18.

```
@app.callback(
    Output('line_chart', 'figure'),
    [Input('repository_title', 'value'),
     Input('abstraction', 'value'),
     Input('file_chart_hover', 'hoverData')])
def update_linechart(repotitle, abstraction, hoverData):
```

Figure 4.17: *Line chart output and inputs - visualizer.py*

```
dbc.Row([
    dbc.Col(dcc.Graph(id='line_chart', clear_on_unhover=True, figure=invisible_figure)),
    dbc.Col(dcc.Graph(id='file_chart_hover', clear_on_unhover=True, figure=invisible_figure))
]),
```

Figure 4.18: *Dash core Graph components for initial line and bar chart - visualizer.py*

The line chart callback starts by reading in the repository dataframe from the data file. If the hover data from the bar chart is not none, only rows where the filename column is equal to the hover value are selected using the .loc[] pandas function and the hover value is added to the end of the title of the line chart as shown in figure 4.19. The dataframe rows are then grouped by the month in the date column and the numerical values for each month are summed. Finally, the lists of names, dates, totals, additions and deletions are given as parameters to the createlinechart() function which is shown in figure 4.20 and this returns the data and layout to be output to the Graph component.

```
df = df.loc[df['Filename'] == hoverData['points'][0]['x']]
title = "LOC Change p/month: " + hoverData['points'][0]['x']
```

Figure 4.19: *Taking data for specific file/folder - visualizer.py*

```
def createlinechart(dates, totals, adds, dels, title):
    return {
        'data': [
            {
                'x': dates,
                'y': totals,
                'mode': 'lines+markers',
                'name': 'Total'
            },
            {
                'x': dates,
                'y': adds,
                'mode': 'lines+markers',
                'name': 'Additions'
            },
            {
                'x': dates,
                'y': dels,
                'mode': 'lines+markers',
                'name': 'Deletions'
            },
        ],
        'layout': {
            'title': title,
            'clickmode': 'event+select',
            'hovermode': 'closest',
            'hovermode': 'x',
            'plot_bgcolor': 'rgba(0,0,0,0)',
            'paper_bgcolor': 'rgba(0,0,0,0)',
            'font': {'color': 'black'},
            'margin': dict(b=100),
        }
    }
```

Figure 4.20: *createlinechart() function - visualizer.py*

The bar chart callback works similarly however it checks for hover data from the line chart and selects rows in the dataframe where the date column is equal to the hover value if the hover data is not none. It then groups the rows by the Filename column and sorts them in ascending order. The list of names and totals are given to a function which returns the data and layout of the chart which is then output to the Graph component.

The final callback for the visualizer page allows for the comparison of different bar charts displaying the most change prone files for different months. The callback takes the selection data from the line chart as an input which relates to points which have been 'shift + clicked' on the line chart. The same number is then placed at the start of each of the rows with the same name using a for loop as seen in figure 4.21. This is used to make comparison between the charts easier. The for loop goes through the files/folders and places the same number at the start of all names which match.

```
count = 1
for i, file_i in enumerate(fnames):
    if not fnames[i][0].isdigit():
        for j, file_j in enumerate(fnames):
            if file_i == file_j and not i == j:
                fnames[j] = str(count) + " " + fnames[j]
        fnames[i] = str(count) + " " + fnames[i]
        count += 1
```

Figure 4.21: *Numbering files/folders for comparison – visualizer.py*

The get_month_data() function in figure 4.22 is then called for each of the points selected in the mine chart. It groups and sorts the files/folder, splits the number from the start of the name and places it in its own list. The function returns the list of names, totals and numbers and the string holding which month the data relates to. The callback then calls another function to create the Graphs which are output to different Div components.

```
def get_month_data(df, month):
    df['Date'] = df.Date.astype(str)
    df = df.loc[df['Date'] == month]
    month = month[:-3]
    df = df.groupby(['Filename']).sum()
    df = df.sort_values(by=['Total'], ascending=False)
    df = df.head(10)
    filenames, filenumbers = ([] for i in range(2))
    for file in df.index.tolist():
        split = file.split()
        filenames.append(split[1])
        filenumbers.append("(" + split[0] + ")")
    filetotals = df['Total'].tolist()

    return filenames, filetotals, filenumbers, month
```

Figure 4.22: *get_month_data() function - visualizer.py*

4.4 GUI Screenshots

This section provides multiple screenshots which display the different pages and features of the application. Figure 4.23 shows the home page and the navigation bar of the application. Figures 4.24 – 4.28 show the different features of the extractor page and finally, figures 4.29 – 4.31 show the visualizer page and how it updates due to user interaction.

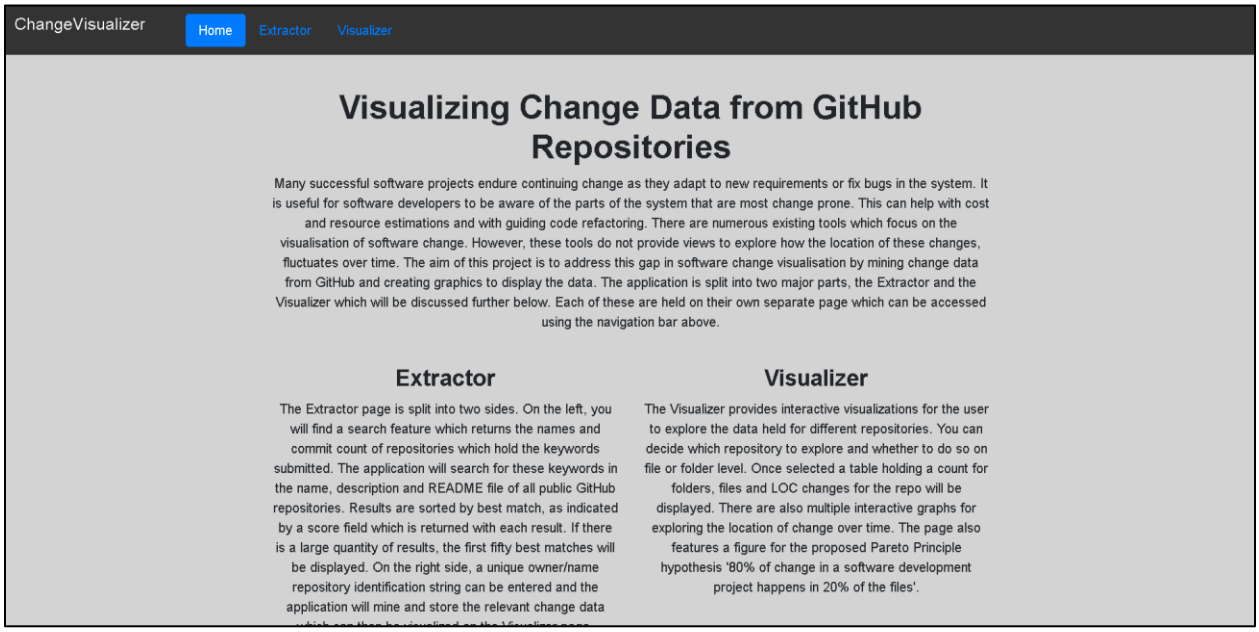

Figure 4.23: *Home page and navigation bar*

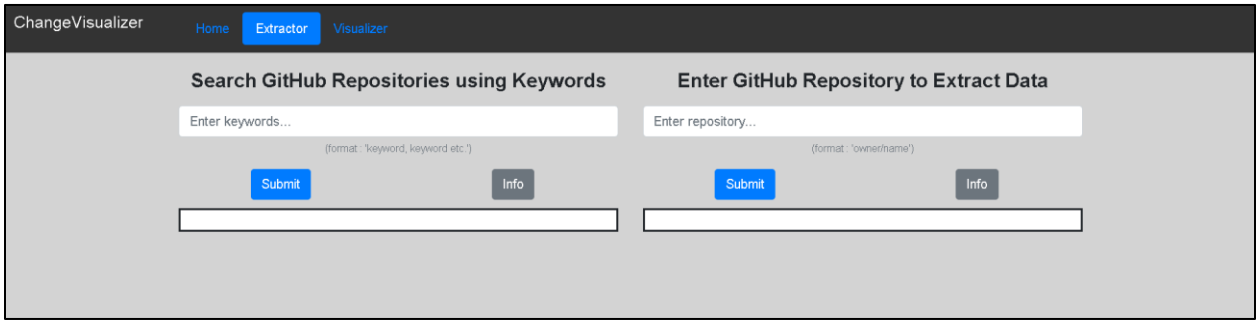

Figure 4.24*: Extractor page before input*

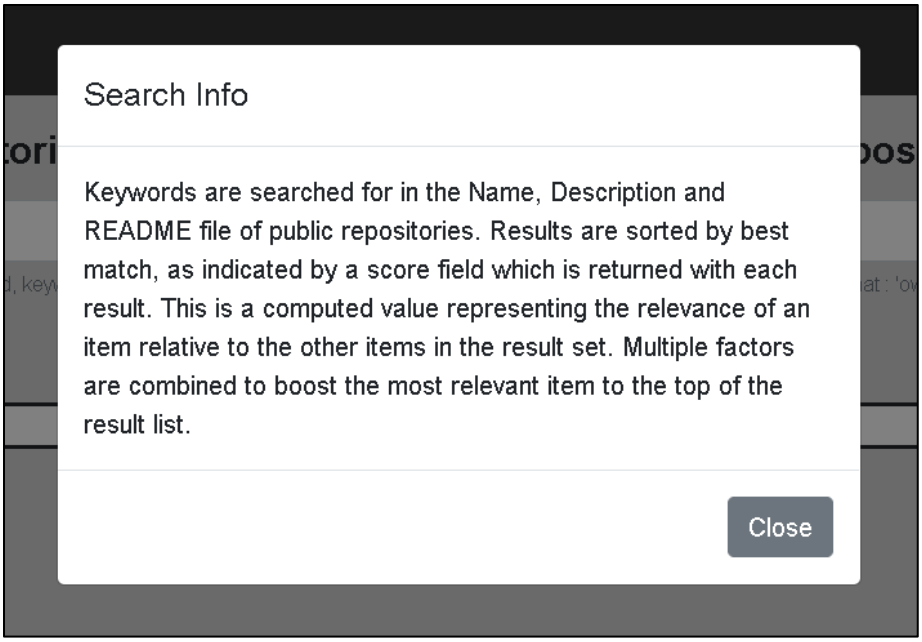

Figure 4.25: *One of Info modals on extractor page*

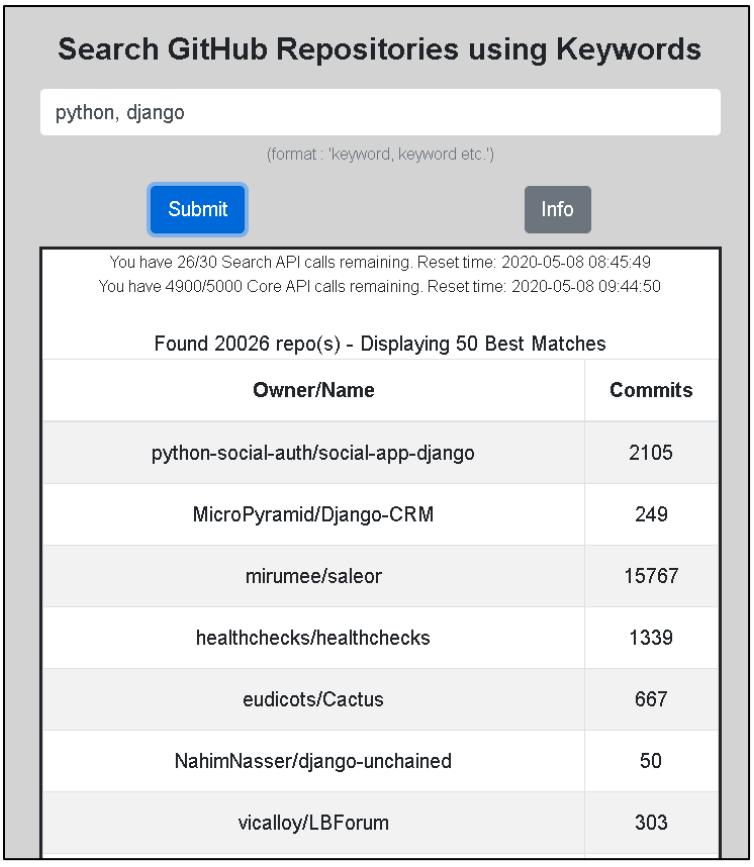

Figure 4.26: *Example results for search feature on extractor page*

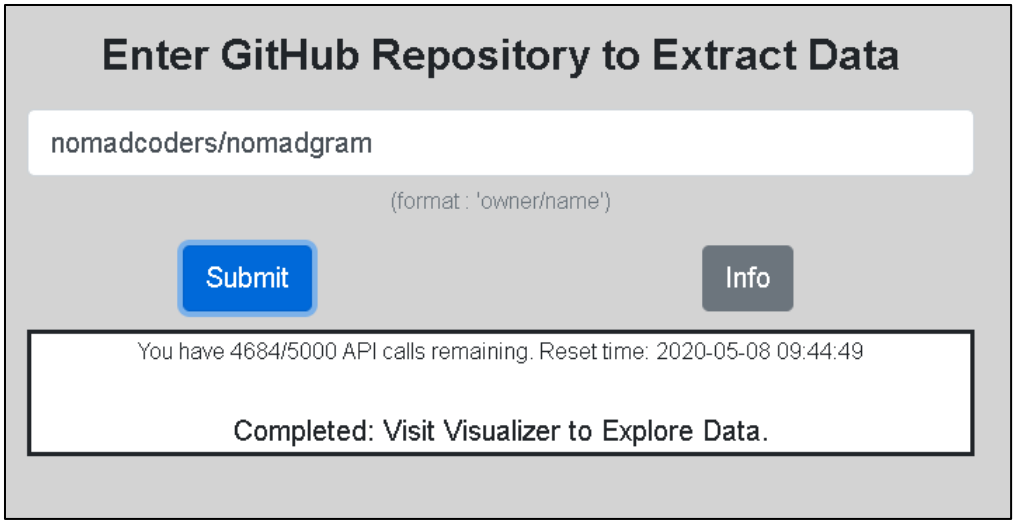

Figure 4.27: *Successful mining response*

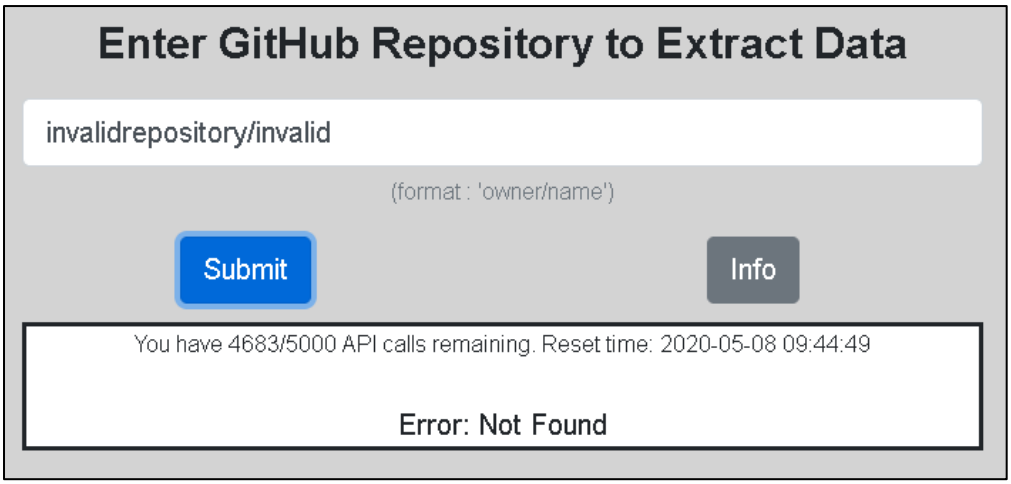

Figure 4.28: *Invalid repository details entered*

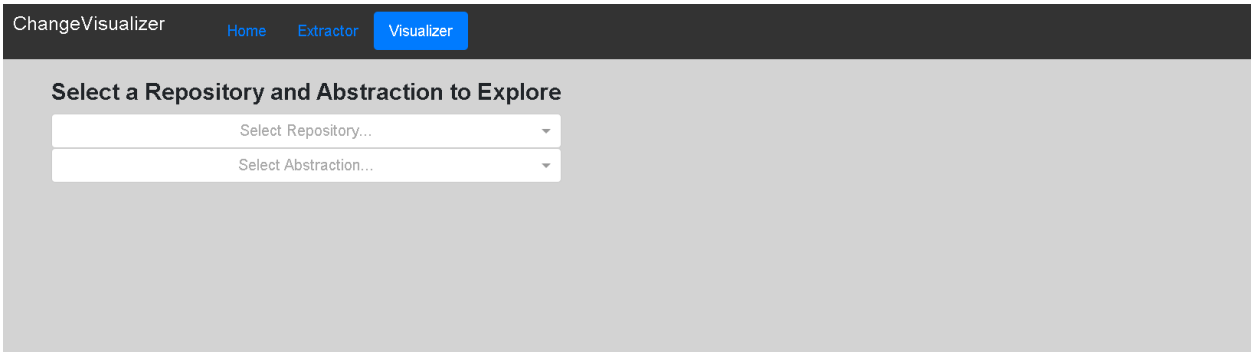

Figure 4.29: *Visualizer page*

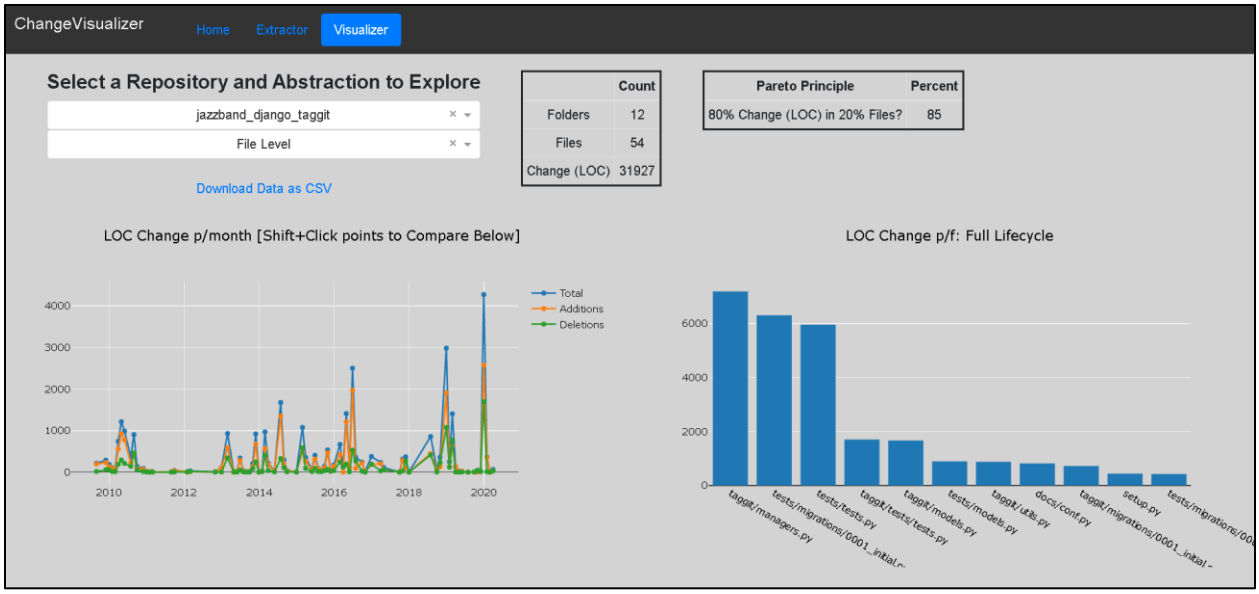

Figure 4.30: *Visualizer page after dropdown selections made*

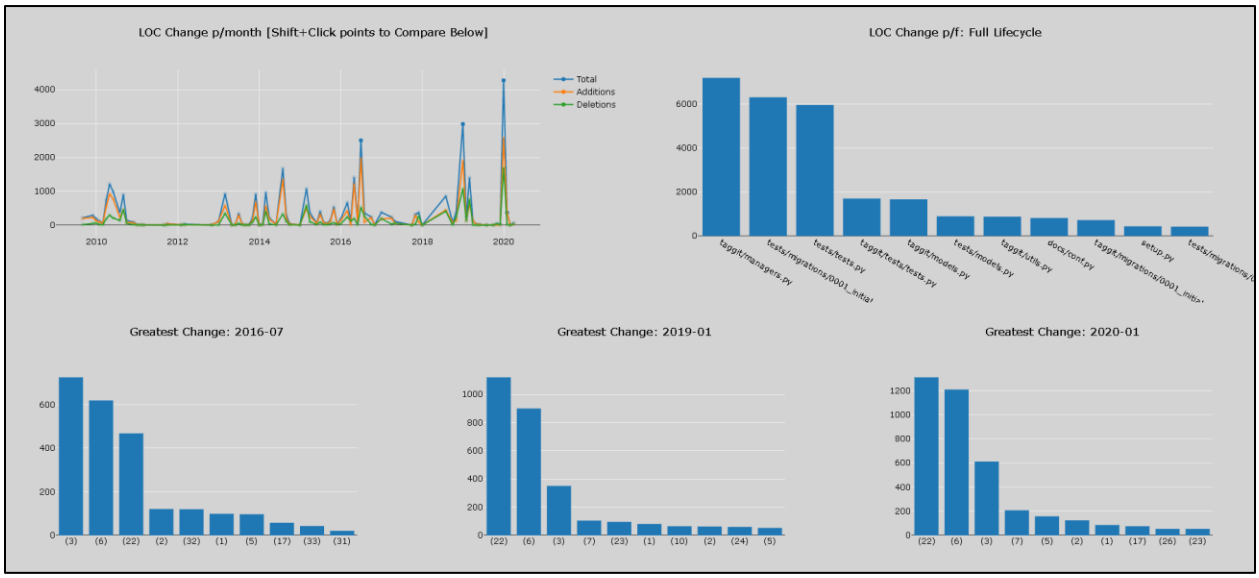

Figure 4.31: *Visualizer page after points on line chart selected*

4.5 Deployment

The application is deployed using Heroku (2020) which offers free plans for deploying applications to a public server. Heroku is a platform as a service based on a managed container system, with integrated data services and a powerful ecosystem, for deploying and running modern apps. Developers can deploy code written in Node, Ruby, Java, PHP, Python, Go, Scala, or Clojure publicly. The system and language stacks are monitored, patched, and upgraded, so

it's ready and up to date. The runtime keeps apps running without any manual intervention. The URL which the app is deployed on is “visualizing-software-change.herokuapp.com” as seen in figure 4.32.

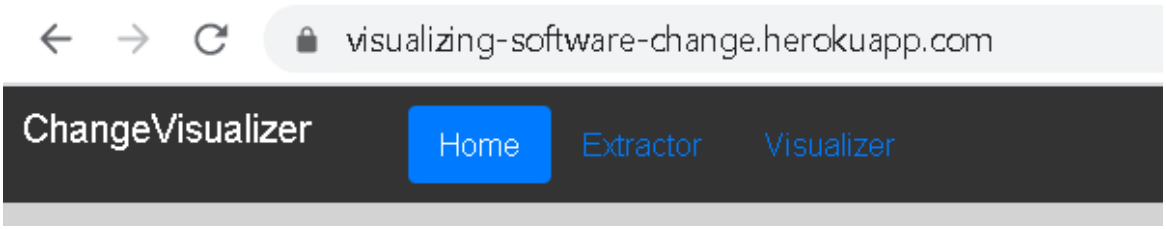

Figure 4.32: *Heroku app URL*

The application is deployed by linking the Heroku app to the GitHub repository storing the code. There is also the need for the creation of several files which are used by Heroku to set up the virtual environment and for running the application. The runtime.txt file is used to specify the version of python the app is running, the requirements.txt file specifies the different libraries and frameworks which the application needs to run and the Procfile states that the application uses a Gunicorn server and where the entry point for the application is. The Procfile for the application is shown in figure 4.33 and it points to the index.py file for running the application. The Heroku command line allows for checking the status of the application and getting recent log output. An example of log outputs is seen in figure 4.35 which shows different state changes.

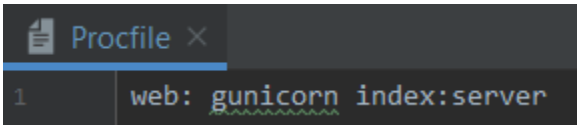

Figure 4.33: *Procfile for application*

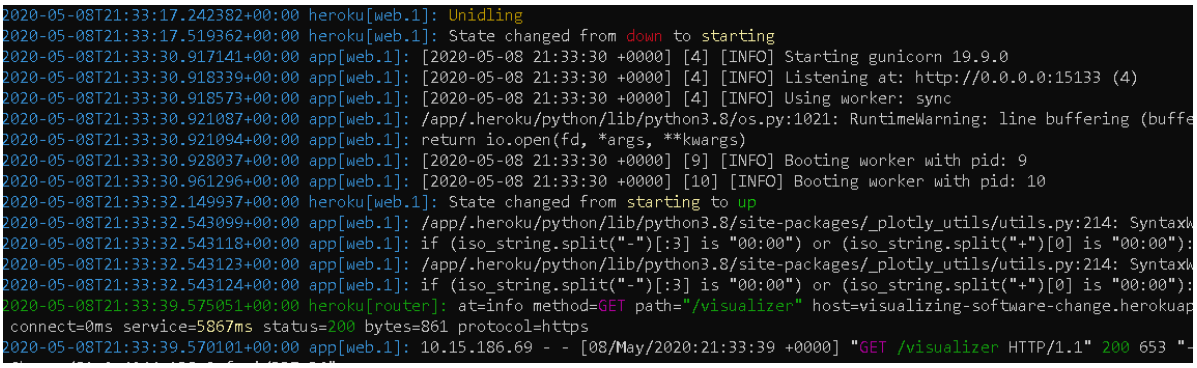

Figure 4.34: *Heroku CLI logs*

5. Testing & Evaluation

This chapter gives an outline on the testing of the system which is broken up into functional and performance testing. The rest of the chapter gives an evaluation of the usefulness of the application including its ability to visualize change patterns as well as its uses for investigating the pareto principle which was proposed at the beginning of this report.

5.1 Testing

5.1.1 Functional Testing

Functional testing is a form of software testing that validates the software system against its functional requirements. The purpose of Functional tests is to test each function of the software application, by providing appropriate input, verifying the output against the Functional requirements. This section will provide an example of some of the different functional tests executed rather than an exhaustive set of all possible functional tests.

Test 1: Navigate from home page to extractor page
Input: Click ‘Extractor’ on Navbar
Expected Result: Extractor page shown
Output: Extractor page shown
Verdict: Success

Test 2: Search for GitHub repositories
Input: ‘java, swing’ entered in input box and submit button clicked
Expected Result: No. of repositories found and table showing name and no. of commits for top fifty best matches
Output: No. of repositories and tables displayed as partially shown in figure 5.1.
Verdict: Success

Found 12334 repo(s) - Displaying 50 Best Matches

Owner/Name	Commits
ZhuangM/student	4
aterai/java-swing-tips	4030
JackJiang2011/beautyeye	153
JDatePicker/JDatePicker	135
adrian/upm-swing	397

Figure 5.1: *Partial output from functional test 2*

Test 3: Extract change data from GitHub repository

Input: 'gaborbata/jpass' entered in input box and submit button clicked

Expected Result: Extraction and storage of change data and confirmation message output to screen.

Output: Change data is mined, and confirmation message shown in figure 5.2 is displayed.

Verdict: Success

Figure 5.2: *Confirmation message for functional test 3*

Test 4: Select repository and abstraction to explore

Input: 'jazzband/Django_taggit' selected in repository dropdown and 'File level' selected in abstraction dropdown.

Expected Result: Tables displayed showing no. of files, folders, LOC change and percentage of change in top twenty percent most change prone files. Download link displayed for downloading selected data as CSV. Line chart showing additions, deletions and total LOC change over time and bar chart showing total change for the most change prone files displayed.

Output: Tables, link and charts displayed as seen in Figure 5.3.

Verdict: Success

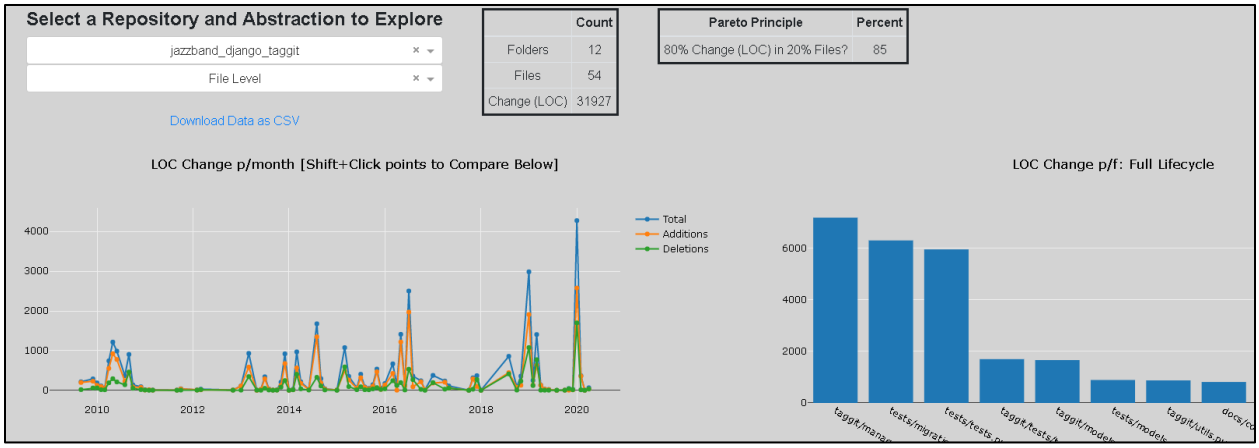

***Figure 5.3**: Output from functional test 4*

Test 5: Download change data for selected repository

Input: Click 'Download data as CSV'

Expected Result: CSV file downloaded from browser

Output: File downloaded containing relevant data as seen in figure 5.4

Verdict: Success

	A	B	C	D	E	F
1		Date	Filename	Total	Additions	Deletions
2	0	01-04-20	taggit/migı	14	7	7
3	1	01-04-20	taggit/moc	18	9	9
4	2	01-04-20	taggit/migı	14	7	7
5	3	01-04-20	taggit/moc	18	9	9
6	4	01-03-20	taggit/moc	2	1	1
7	5	01-02-20	taggit/mar	25	22	3
8	6	01-02-20	tests/migr	109	109	0

***Figure 5.4:** Partial data downloaded to CSV file from functional test 5*

Test 6: View bar chart showing file contribution to change for specific month

Input: Hover over point on the line chart

Expected Result: Bar chart updates to file contribution for specific month

Output: Bar chart displays file contribution to change for month associated with the point

Verdict: Success

Test 7: Select months to compare file contribution bar charts

Input: Shift + click three points on the line chart

Expected Result: Three bar charts displayed showing file contribution with the same number shown under matching files

Output: Bar charts displayed with numbering system as shown in figure 5.5

Verdict: Success

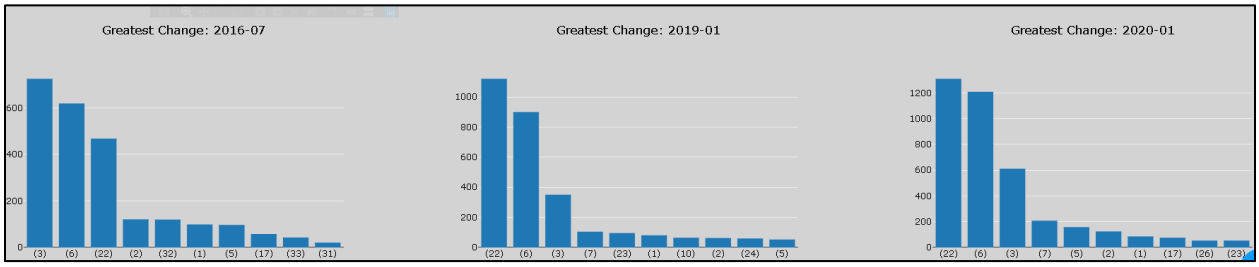

Figure 5.5: *Bar charts outputted for function test 7*

5.1.2 Performance Testing

Performance testing is the process of determining the speed, responsiveness and stability of a software system or program under a workload. Some common performance metrics include throughput (how many units of information a system processes over a specified time), memory (the working storage space available to a processor or workload) and response time (the length of time taken for the system to react to an event). The following performance tests will be focused on the response time of the system to certain user inputs.

Test 1: Observing response time of searching for GitHub repositories

Test Setup: Five different keyword combinations entered, and the runtime in seconds is recorded. Each search is run five time and the average is computed. The test is then repeated but this time the search result does not include the number of commits for each repository in the table.

Results: The results are shown in figure 5.6 and there is a clear difference between the response times depending on if the number of commits is returned or not. The average runtime with the commit count is 24 seconds while without is only 3 seconds. The reason for this is that a separate API call is made to get the number of commits for each of the repositories featured in the table.

	With No. of Commits					
Keywords	**Run 1**	**Run 2**	**Run 3**	**Run 4**	**Run 5**	**Average**
java	32.382	27.032	20.924	25.918	19.986	25.2484
java, swing	24.613	26.785	22.379	22.83	27.033	24.728
java, swing, sql	29.935	22.181	24.197	21.178	24.386	24.3754
android, game, puzzle	26.849	24.043	23.706	20.784	21.025	23.2814
algorithm, data, search	22.263	24.799	24.084	20.821	22.498	22.893
						24.10524

	Without No. of Commits					
Keywords	**Run 1**	**Run 2**	**Run 3**	**Run 4**	**Run 5**	**Average**
java	2.681	4.013	3.412	3.188	2.481	3.155
java, swing	2.973	3.044	2.465	2.754	2.864	2.82
java, swing, sql	3.416	2.744	2.231	3.254	2.41	2.811
android, game, puzzle	3.902	2.898	2.896	3.673	2.919	3.2576
algorithm, data, search	3.311	3.075	2.741	3.343	2.286	2.9512
						2.99896

Figure 5.6: *Results of performance test 1*

Test 2: Observing response time for change data extraction

Test Setup: Repositories within 50 commits of 100, 250,500,750 and 1000 commits are mined, and the runtime recorded. Three different repositories are selected at each of the 5 levels which are used to calculate an average runtime.

Results: As seen in figure 5.7, the results show the issue that as the number of commits gets larger, the runtime also gets significantly large. The average for 100 commits is 0.92 minutes whereas for 1000 commits, it has reached 8.23 minutes. These are not exact averages due to the small sample size, but they give an idea of how the application scales for larger repositories.

No. of Commits	**Repo 1**	**Repo 2**	**Repo 3**	**Average**
~100	1.03	0.78	0.96	0.92
~250	1.62	1.93	2.01	1.85
~500	3.98	4.13	3.74	3.95
~750	6.27	6.12	5.83	6.07
~1000	7.68	8.72	8.29	8.23

Figure 5.7: *Results of performance test 2*

5.2 Evaluation

The usefulness of the application is measured by its ability to display patterns in software change as well as its performance capabilities. As a result, the evaluation of the application is based on the performance testing in section 5.1.2 along with the opinions of ten Computer Systems students from the University of Limerick who tested the application on their own GitHub repositories and gave feedback on their experiences. The feedback featured what they found helpful or positive about the application as well as what they thought would be beneficial to change or add.

As the main focus of the application, feedback regarding the ability of the user to recognize change patterns in the visualizations was the most important for getting a good evaluation of the capabilities of the system. All of the participants were successful in identifying areas of their code both files and folders where the majority of the change was happening. The most used feature on the visualizer page was the comparison feature whereby they could shift + click different points on the line chart and this would result in the bar charts showing total LOC change for the most change prone files in that month. This feature was greatly used by nine out of ten of the students and this was due to having repositories which suited the visualization approach. These were repositories which had commits stretched over a number of months and even years. They were able to observe that extreme points on the line chart often related to the same group of files and folders and they could also see how their focus switched at certain stages during development. One student did not get much from this functionality due to not having repositories with commits stretched out over a decent number of months. As a result, they did give the suggestion that giving an option to observe the data in weekly intervals instead could be useful for newer or smaller repositories.

As for performance, the performance testing in section 5.1.2 show that there are issues as regard to runtime when it comes to interactions with the GitHub API. This is due to the fact that API calls are relatively slow compared to other aspects of the application and they are a major factor in the lengthening of the runtime of the two different features on the extractor page. The comparison of the search results with or without the inclusion of commit counts clearly show

how extra API calls results in a much slower application. This was particularly harmful for the application deployed on Heroku because the Heroku dynos automatically time out processes after quite a small time period. This performance issue was also noted in the feedback from the students as they found the extraction feature especially slow when they were trying to mine the change data from their larger repositories. One suggestion that was received from this was to include an input which specifies an interval of either time or commits to extract at one time.

On the other hand, the students were all happy with the response time they experienced during navigation and on the visualizer page. They found that all user interaction on the visualizer page such as dropdown selection as well as hovering and clicking resulted in an almost instant reaction from the application. This was recognized as one of the more appealing aspects of the system and they found it gave them more time to focus in on the different visualizations and gain a greater understand of the data being displayed.

The final aspect of the evaluation of the application is on its ability to investigate the proposed pareto principle that '80% of the change happens in 20% of the code'. The investigation was achieved by calculating the percentage of change which occurred in the top twenty percent most change prone files and displaying this in a table on the visualizer page. An example of this table is seen in figure 5.8. To investigate whether or not it is a valid hypothesis, the change data of 50 repositories was mined and the percentage recorded. The histogram in figure 5.9 shows the results which are grouped by ten percent intervals. The data is slightly left skewed with the largest bins being 60-70% with 14 followed by 70-80% and 80-90% with 12. This gives the perception that the proposed pareto principle is quite accurate which is backed up further by the mean percentage which is 72%. This figure is close enough to 80% to be considered a valid pareto principle for the given sample size.

Pareto Principle	Percent
80% Change (LOC) in 20% Files?	85

Figure 5.8: *Pareto principle on visualizer page*

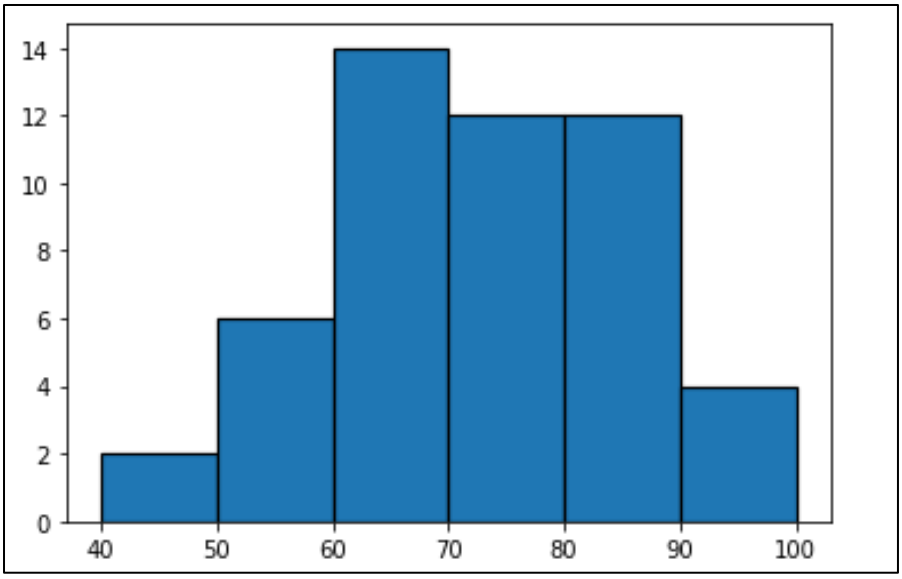

***Figure 5.9**: Histogram showing pareto percentages*

6. Conclusions & Recommendations

This chapter presents conclusions along with recommendations for future work.

6.1 Conclusions

In response to the lack of existing tools which focus on the visualization of the location of software change over time, an application has been designed and developed for this purpose. The work presents a tool which can be used to visualize change patterns in software over time. The focus of the work included software change, data visualization and version control systems which were examined through the literature and also through existing tools which have already been developed with a similar concept. The application was designed based on this research and developed using an incremental approach in the python programming language using the different data manipulation and visualization libraries and frameworks which it has to offer.

The resultant application is a web application which holds multiple interactive features including page navigation, text input and hover and click events. This allows for an in-depth exploration of change data which can be mined from all public repositories found on the GitHub software development platform. Finally, the deployment of the application using a free Heroku server has created easy access which increases the usability of the application. Overall, there are many positive features to the project as a whole however some recommendations which could improve the work in the future are included in the next section.

6.2 Recommendations

In terms of recommendations for future work, the main focus would be reducing the runtime of the features of the extractor page. One approach which could aid this goal was collected during the evaluation of the system as outlined in section 5.2. This is to add the ability to specify an interval when extracting change data from a repository which could be either based on time or

commits. This would make observing larger repositories more viable due to the fact the user would not have to wait for over ten minutes for the process to finish.

Another suggestion taken was more focused towards the other extreme which is repositories with too few commits or where commits are not stretched out over a long enough time period. It is hard to observe any type of patterns when the project only stretches out over two or three months as there aren't enough points to compare. The suggestion here is to allow for visualizing the data based on weekly intervals rather than monthly which would improve the visualizations created for these repositories.

Finally, as was discovered in the section 2.4, when mining data from GitHub it is very important to find repositories which are suited to the observations you are wishing to make. There was a clear difference in the potential of this application when using suited repositories to ones which could not be explored effectively by the visualizations created.

References

- R. Buse and T. Zimmermann, 2012, “Information Needs for Software Development Analytics,” Proc. Int’l Conf. Software Eng. (ICSE), IEEE CS; http://thomas-zimmermann.com/publications/details/buse-icse-2012.
- Chapin, N., Hale, J.E., Khan, K.M., Ramil, J.F. and Tan, W.G., 2001. Types of software evolution and software maintenance. *Journal of software maintenance and evolution: Research and Practice*, *13*(1), pp.3-30.
- Eick, S.G., Graves, T.L., Karr, A.F., Mockus, A. and Schuster, P., 2002. Visualizing software changes. *IEEE Transactions on Software Engineering*, *28*(4), pp.396-412.
- Knight, C. and Munro, M., 1999. Comprehension with [in] virtual environment visualisations. In *Proceedings Seventh International Workshop on Program Comprehension* (pp. 4-11). IEEE.
- Tullis, T.S. & Case, M., 2017. 2D vs. 3D Charts: Does 3D Representation Help or Hurt?. User Experience Professionals Association.
- Fronk, A., Bruckhoff, A. and Kern, M., 2006, September. 3D visualisation of code structures in Java software systems. In *Proceedings of the 2006 ACM symposium on Software visualization* (pp. 145-146).
- Weissgerber, T.L., Garovic, V.D., Savic, M., Winham, S.J. and Milic, N.M., 2016. From static to interactive: transforming data visualization to improve transparency. *PLoS biology*, *14*(6).
- Reuter, L.H., Tukey, P., Maloney, L.T., Pani, J.R. and Smith, S., 1990, October. Human perception and visualization. In *Proceedings of the 1st conference on Visualization'90* (pp. 401-406). IEEE Computer Society Press.
- Ball, T., Kim, J.M., Porter, A.A. and Siy, H.P., 1997. If your version control system could talk. In *ICSE Workshop on Process Modelling and Empirical Studies of Software Engineering* (Vol. 11).
- Eick, S.C., Steffen, J.L. and Sumner Jr, E.E., 1992. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, (11), pp.957-968.
- Maletic, J.I., Marcus, A. and Feng, L., 2003. Source Viewer 3D (sv3D)-a framework for software visualization. In *25th International Conference on Software Engineering, 2003. Proceedings.* (pp. 812-813). IEEE.
- Wu, X., 2003. *Visualization of version control information* (Doctoral dissertation, University of Victoria).
- Wang, Dakuo & Olson, Judith & Zhang, Jingwen & Nguyen, Trung & Olson, Gary. 2015. DocuViz. 1865-1874. 10.1145/2702123.2702517.
- German, D., 2004. Mining CVS repositories, the softChange experience. In *MSR* (Vol. 4, No. 15, pp. 17-21).

- German, D.M. and Hindle, A., 2006. Visualizing the evolution of software using softChange. *International Journal of Software Engineering and Knowledge Engineering*, *16*(01), pp.5-21.
- Voinea, L., Telea, A. and Van Wijk, J.J., 2005. CVSscan: visualization of code evolution. In *Proceedings of the 2005 ACM symposium on Software visualization* (pp. 47-56).
- Xie, X., Poshyvanyk, D. and Marcus, A., 2006. Visualization of CVS repository information. In *2006 13th Working Conference on Reverse Engineering* (pp. 231-242). IEEE.
- De Alwis, B. and Sillito, J., 2009. Why are software projects moving from centralized to decentralized version control systems?. *ICSE Workshop on Cooperative and Human Aspects on Software Engineering* (pp. 36-39). IEEE.
- VanderPlas, J., 2016. *Python data science handbook: Essential tools for working with data*. " O'Reilly Media, Inc.".
- GitHub. (2020). available: https://github.com/. [accessed 5 Jan 2020]
- Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D.M. and Damian, D., 2014, May. The promises and perils of mining GitHub. In *Proceedings of the 11th working conference on mining software repositories* (pp. 92-101).
- GitHub Dev. (2020). available: https://developer.github.com/v3/. [accessed 13 Feb 2020]
- PyGitHub. (2020). available: https://pygithub.readthedocs.io/. [accessed 22 Mar 2020]
- Mozilla. Fetch API. (2020). available: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API.
- Plotly. Dash. (2020). available: https://dash.plotly.com/. [accessed 10 Jan 2020]
- Pandas. (2020). available: https://pandas.pydata.org/. [accessed 5 Feb 2020]
- Heroku. (2020). available: https://www.heroku.com/. [accessed 16 Apr 2020]