Automatically Characterising the Purpose of GitHub Repositories

Wilson Blaikie, Christoph Treude
wilson.blaikie@student.adelaide.edu.au, christoph.treude@adelaide.edu.au

***Abstract*—GitHub hosts many invaluable libraries and repositories for programmers to utilise in their own software projects, however, communicating exactly what the code is for in short and succinct terminology is often difficult. The contents of README files can fall behind the rest of the code or not properly cover the technical details and act more like a pitch. An investigation into whether a tool can be created which can take a repositories' README and summarise it in a way that communicates the purpose of the repository is proposed using Natural Language Processing to develop a model entirely centric to README files. The tool will take the URL of a GitHub repository as a sole parameter and return a field of text with the description. Then the performance of this method will be compared to other methods of evaluating repositories alongside the different techniques utilised to construct each of these methods. Alternative methods involve defining similar repositories and cross referencing details that are shared between them or taking keywords out of the README. A benchmarking platform is developed to produce comparable results in which a blind test is conducted to evaluate the mothods' performance. While summaries written by a human outperformed the other methods, Summaries constructed from the projects README were still able to effectively communicate sufficient detail on the purpose of a repo in summarised format.**

I. Introduction & Motivation

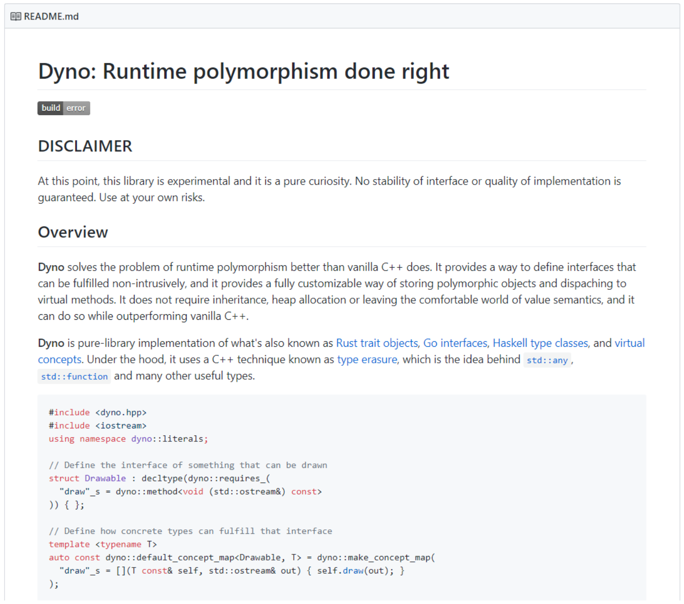
For many years now, the champion service of the open source software community has been GitHub, for over 10 years this service has provided a platform for programmers and engineers to develop their own projects or repositories and share them publicly for free. These projects can then be downloaded, installed, improved, or even re-engineered for an individual's own usage. The premise behind this philosophy on open source software is that by sharing ideas and code, communities can develop their own ideas further beyond their individual skill. In order to communicate these ideas among many programmers of varying backgrounds and experience, documentation specifying various aspects of the project are supplied alongside the code. This can be done with either comments throughout the code to define what each section of code does, or a single document that gives an overview of the project as a whole. This is called a README, READMEs have been so pervasive in open source software, that it has become an expectation that any engineered product available for others to compile is paired with an associated README. This can be partially attributed to the GNU Coding Standards put forward as a subsidiary of the Free Software Foundation. As a formal body that decrees how programmers should design their projects in order to produce high quality, understandable code that other programmers can understand and utilise in their own projects or integrate into their GNU operation system environment.

The GNU Coding Standards [1] have a set of guidelines on what an ideal README file should contain in section 7.3;

> The distribution should contain a file named README with a general overview of the package:
>
> - the name of the package;
> - the version number of the package, or refer to where in the package the version can be found;
> - a general description of what the package does;
> - a reference to the file INSTALL, which should in turn contain an explanation of the installation procedure;
> - a brief explanation of any unusual top-level directories or files, or other hints for readers to find their way around the source;
> - a reference to the file which contains the copying conditions. The GNU GPL, if used, should be in a file called COPYING. If the GNU LGPL is used, it should be in a file called COPYING.LESSER.

While this has been a foundational guideline on what good software engineering looks like, it is not law. This list is also not descriptive of everything that should be in a README, this leads to READMEs in varying quality across GitHub. One problem that arises with this is that some well engineered projects have README files that do not share this quality of construction. Some symptoms of this are non-descriptive entries that do not provide any worth to the person reading the file. This means that some programmers can find suitable libraries or API's for their needs but are unaware as the documentation on the repository they have found does not inform them of the projects functionality or usability in an effective and informative manner. This can occur when the author of the README makes this document too technical. There can be large sections of the README that are segments of code with is not necessarily constructive in defining a purpose. Another instance of text that serves no direct purpose are links, while they do provide other pages that may contain useful information. Regardless, its inclusion in the summary is not desirable. Other pitfalls for README files lay on the opposite end of the spectrum, where non-technical information is presented like Code of Conduct and contribution information.

A system that could find the essential data of a repository and effective communicate this in a ”summary” would serve as a handy guide in which libraries are right for a user. To summarise a repository, a user can get an idea on whether they want to use the repository before further investigation. This can can be completed using Natural Language Processing (NLP). NLP is the field of Artificial Intelligence revolving around how models interpret human language. This means that intelligent models can be constructed to determine what sentences and phrases bet match training examples. This requires summaries to be written by a human in order to train which in turn, requires consideration on how to effectively communicate the purpose of the repositories.


README.md

Dyno: Runtime polymorphism done right

build error

DISCLAIMER

At this point, this library is experimental and it is a pure curiosity. No stability of interface or quality of implementation is guaranteed. Use at your own risks.

Overview

Dyno solves the problem of runtime polymorphism better than vanilla C++ does. It provides a way to define interfaces that can be fulfilled non-intrusively, and it provides a fully customizable way of storing polymorphic objects and dispaching to virtual methods. It does not require inheritance, heap allocation or leaving the comfortable world of value semantics, and it can do so while outperforming vanilla C++.

Dyno is pure-library implementation of what's also known as Rust trait objects, Go interfaces, Haskell type classes, and virtual concepts. Under the hood, it uses a C++ technique known as type erasure, which is the idea behind `std::any`, `std::function` and many other useful types.

```
#include <dyno.hpp>
#include <iostream>
using namespace dyno::literals;

// Define the interface of something that can be drawn
struct Drawable : decltype(dyno::requires_(
  "draw"_s = dyno::method<void (std::ostream&) const>
)) { };

// Define how concrete types can fulfill that interface
template <typename T>
auto const dyno::default_concept_map<Drawable, T> = dyno::make_concept_map(
  "draw"_s = [](T const& self, std::ostream& out) { self.draw(out); }
);
```

Fig. 1: Dyno Framework README [2]

In this excerpt from a GitHub repository, a lot of technical language is used. While it is contextually appropriate, someone who may benefit from this library may not understand what the purpose of this library is and could end up picking something less suitable. While it does make quick reference to the fact it is a library about utilising runtime polymorphism. It’s not until much later does it describe any further detail about how it makes improvements on how the compiler and even then its mainly paragraphs of highly technical details that many may struggle to comprehend. Speculation on by the programmer that people who will utilise their library will match or even exceed them in terms of technical understanding.

To simply provide the repo to an analytical tool & have a concise summarisation would provide developers the basis to evaluate many more libraries and tools on the platform which in turn, can reduce the research & unnecessary development time on new software projects building of existing ideas. This also can expose developer’s libraries to a wider audience and build a community around the maintenance of existing projects.

II. Background & Review

Natural Language Processing (NLP) is a field of artificial intelligence in which a technical model is designed to interpret different forms of our natural language. This can be done in many forms such as speech recognition, understanding passages of text, or even generating text understandable to human beings. With the rise in popularity of artificial intelligence and deep learning modelling, the interest in NLP has risen with it. The result of this is many studies and tools for taking in text such as articles or reports and using an appropriate model to find relevant or ”high ranking” text at a high accuracy. The issue with this is that these experiments are primarily designed for very natural bodies of text. Prior research shows technical documentation like README files are not constructed with the same syntax as other literature and thus the rules of traditional NLP either struggle or completely fail to extrapolate to these texts. Even with [3]. This report seeks to analyse what differentiates READMEs from natural language and whether they reliably contain information on what the purpose of a GitHub repository is. The purpose in this study is defined as any description on what the repository is used for. Existing research is used to define what the content is in different parts of a README [4]. This means we can determine high importance areas that are more likely to contain information to be summarised. For example, sections labelled ”Contribution” will not contain information relevant to why you may want to use a library, only improve it. This can drastically reduce the amount of data that needs to be assessed to help constrain the sample space.

While many NLP models can summarise different bodies of text to a couple words or sentences, this has traditionally been completed by Extractive-based Summarisation. Extractive-based Summarisation select words and phrases from a document in order to find keywords or summarise a document. This can be done with machine learning in a supervised learning environment or with graph-based algorithms in an unsupervised learning environment. Unsupervised learning can be completed with the TextRank algorithm. TextRank collects sentences that are similar within a text and links them in a graph, once the graph is constructed they are then ranked based on which sentences are best representative of the text as a whole [5]. This method is even developed upon in the creation of the LexRank in which top ranking summaries are compared for similarity so that top results are unique [6].

Similar work to this project have been completed by GitHub [7] in the forms of their repo topics. They use some NLP techniques in order to discover what topics a repo would suit so that users may tag their repos to make them more discoverable for prospective developers. This feature helps identify similar repositories depending on their topics, unfortunately these repos are not labelled automatically & GitHub report that over a quarter of their repos they were using to develop their system, could not have their topics automatically generated by the system. This means any developed system could not rely on the GitHub API outside of collecting README content.

We define similar repos in this project as a repo that is based on an identical concept or field the inspires the creation of the repos that ultimately serve purposes that share similarities.

Since GitHub's efforts in this area are only briefly described in a blog & their developments are entirely closed source. Alternative systems for similar steps will have to be considered. Work in identifying engineered software projects resulted in a project called RepoReaper [8]. RepoReaper finds engineered repos, an engineered repo is defined in the page as such "An engineered software project is a software project that leverages sound software engineering practices in each of its dimensions such as documentation, testing, and project management". Since documentation is one of the key criteria in this study, this resultant data set that has been made public may serve useful to the project.

III. System & Experimental Design

A. Project Scope

The design of this tool was initially proposed as an application in which the URL of a GitHub repo could be provided to the program & a summarised text that explained what the repo was and its purpose would be returned. Multiple methods of extracting text from GitHub were tested for their effectiveness alongside a human made summarisation. Each of the texts that are summarised are done so using the TextRank summarisation algorithm. When summarising a GitHub repo, any prevalent sentence that includes formatted words will not rank as highly due to the words tokenising differently. The issue with markdown documents is that the formatting can abstract the contents of a sentence using the syntax of the visual formatting. By analysing the document before summarisation and removing contents like the formatting or links will provide readable sentences in the summary regardless of whether they make sense or not. Code, due to its repetitive nature for class items or methods in a library will also cause these segments to rank highest in the ranking procedure. These segments are encapsulated by triple backticks '` so they also can be filtered quite easily with regular expressions due to the syntactical consistency.

While the extractive-based summarisation could successfully take high ranking sentences from READMEs and display them together. There are READMEs out there that contain limited information and the irrelevant sentences would also be displayed in a summary. Due to the fact that to reduce a body of text, there has to be enough text present to have some removed. This means that a system would not be able to summarise repos that are not less than a paragraph. This would pose an issue in the test data where there are plenty of repos that do not have README's of sufficient size. Another issue posed with the concept of the tool is the prevalence of repos that do not have README's. Since the concept of summarising text requires that body of text to exist, these repos would be incompatible with the tool. An initial check needs to be made for the README's existence which ended up being even more prevalent of an issue.

B. Project Test Dataset

The dataset for READMEs was collected from the RepoReaper [8] project in which many engineered and non-engineered projects were analysed to be able to differentiate the two. The issue with non-engineered projects was that due to the lack of professional practises utilised in their creation they often lacked a README among other issues which can result in suboptimal results. All repos that could be summarised were used as test data to look at the degree of success in summarisation across a wide variety of content available on the GitHub platform to best represent a standard user case. READMEs unable to be summarised due to aforementioned constraints of the project were ignored.

With this dataset, the summarisation methods had to be applied on each repo link present. This meant that the tool had to be extrapolated from a single use application for a typical user, to a test benchmark that could ingest many repos over a sufficient amount of time. This lead to the RepoReaper SummarisationSpider, the spider iterates over each page of the RepoReapers database & collects all of the Repo README API links. Each time it encounters a new link it will strip the unnessecary text from the README & evaluate the length of the repo. If the repo was long enough to warrant summarisation then the different methods were applied and written to a text file. The order of each method is randomised for each repo so when assessing the effectiveness of the summarisations, the participant would not be able to identify the methods by the order & compromise the test.

C. Summarisation Methods

In order to compare performance of different summarisation techniques against eachother in a benchmark setting, there needs to be an evaluation method and even more importantly, different methods & results to compare. There are four proposed methods in this experiment ranging in complexity & specificity. There is the naive intro text summary, the standard single README summary, the more complex but broad multi-README summary, & the human written summary to act as a "best case" scenario. To create the body of text that the application would summarise. A series of regular expressions are applied on the raw markup documents analysed, this strips away all code blocks, text formatting symbols, & hyperlinks.

1) Naive Intro-Text Summary: The simplest of the methods involves taking the stripped README and tokenising the text into its sentences. From these sentences, the first four are taken and assumed that any introduction to the repo would contain enough information to describe the purpose of the repo.

2) Single README Summary: The standard methods to summarising a repo was to first take the stripped text and only applying the TextRank algorithm in order to generate the summary of the repo. While this is only applying a single algorithm on the README, the text would be entirely relevant to the repo in which the summary would be closer related to the source.

3) Multi-README Summary: A more complex method is to find similar repos to the currently assessed repo. When enough similar repos are found, they are then stripped of irrelevant format text like the current repo & concatenated together to create a single body of text, greater in length than the original repo. This text is then summarise like the Single README Summary is. This is assuming that the prevalence of the similar concepts will be enough to cause these sentences to rank higher than those specific to the repo and would not be seen as helpful. This runs the risk of providing results that while may be related to the repo, they may be more generalised and not include the specificity of the original repo. The complexity in this method arises in the ability to find similar repos without performing an extensive & time consuming search of GitHub.

4) Human Written Summary: These summaries represent an ideal system in which a human could summarise the text for another human's understanding. The inclusion of these summaries will assist in gauging the performance of these methods not only against each other, but against a theoretical upper limit. The manual summarisation of these repos will be completed blind to the other summaries created. This is to keep its generation a blind activity and prevent biases from what the application can generate like making the summaries similar in structure to trick the elavuation or writing a less effective summary to make the methods look more effective.

D. Repository Matching

Finding similar repos is also a task that the system needs to be capable of doing. GitHub natively supports assigning topics [7] to repos to help identify not only repos, but what repos could be identified similarly.

This system works by suggesting recommended topics to the user which they can select, or choose to provide their own topics for the repo. The only issue is that like the inclusion of README files, this is entirely optionally and the creators of any repo would have to elect to use this functionality. There is also no way to use this API in a way that would identify potential topics without being the repo owner. While a user could fork their own copy of the repo and then as an owner prompt GitHub for topics, this is time consuming & out of scope for this project.

While we can not rely on the presence of embedded topics, if they do exist the application can utilise this. The topics are collected as a list and each one is ranked against the summary for similarity. This is done with the Ratcliff/Obershelp pattern recognition [9] through comparing the number of matching characters in the longest matching substring sequence as a ratio of the strings length. Characters are also recursively matched between the two bodies of text outside of the substring. These topics once scored in similarity are ranked against eachother.

The highest ranking topic is used as a search method in GitHub to identify high rated repos in that topic area, the high rated repos involve high levels of community engagement which is one of the driving factors in having professional standard documentation and READMEs that sufficiently describe the purpose of the repo.

In the event that the repo does not have topics assigned to it, the application needs to be able to perform on its own. Using the Rapid Automatic Keyword Extraction algorithm (RAKE) [10], keywords from the README are collected into a list and ranked in the same method as the topics are. RAKE works by identifying small series of words that are prevalent to the text but contain no more than one stopword. The ranked keywords are then presented to the user where the user may elect to use a high ranking keyword or if desired, the user can manually override this process and enter their own keyword. This presents the user a potentially slower process with a higher degree of accuracy, this is to counteract the possibility in which repos do not contain enough identifiable information about the topic in which identifying the repo falls out of scope. Once a topic is selected it is used as a search term in GitHub just like the GitHub Topic extraction.

Once the list of similar repos are returned to the application, the top five (or all of them if five or less are found) are then selected as the similar repos to be used in summarisation.

E. Performance Evaluation Method

Once the four summaries are collected, they have to be evaluated by an independent counsel. This is to prevent bias in the results of the comparison. The proposed blind test is to provide a link to the repo with the four summaries in randomised order as to not make them identifiable outside of their performance. A master record of which repo is which will be kept, this record will have the summaries labelled to which method was utilised in generating the summary.

The independent counsel will then grade the summaries on their effectiveness in describing the purpose of the repo. This is in line with the initial problem statement. The grading system on how effective a summarisation has been defined as a 1-5 scale on how effectively the purpose of the repo was communicated to the user. The following key was used;

Scoring scale on how effectively a summarisation method performs:

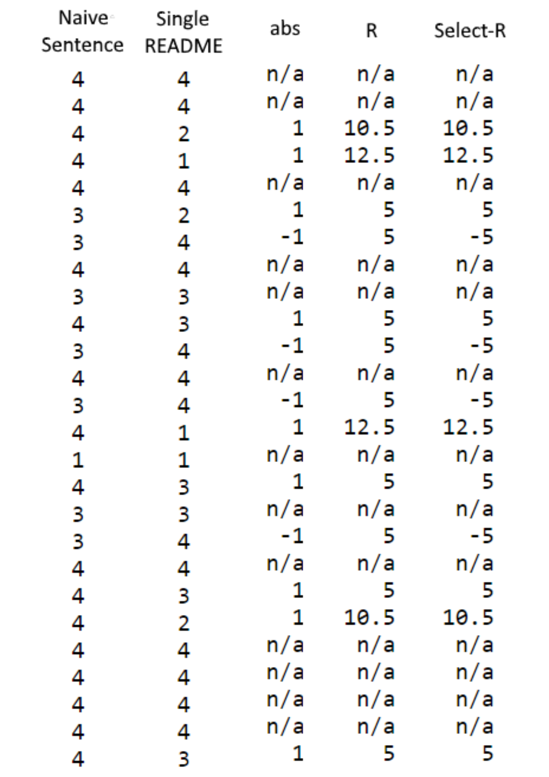
- 1: The summary was illegible or did not describe the repo & its purpose at all, no information of relevance was provided at all
- 2: The summary barely described the repo & its purpose, something regarding the repo was provided to the user but it purpose is not sufficiently communicated
- 3: The summary partially described the repo & its purpose, some information pertaining to the repo was provided but the purpose was not entirely communicated
- 4: The summary described the repo & its purpose, information relevant to the repo was provided to the user
- 5: The summary effectively described the repo & its purpose, highly insightful information was provided to the user

IV. Experimental Analysis

With the results from the blind test identified and attributed to their respecitve method, we can directly compare the results against eachother.

from this data in figure I, we can see that any algorithm is dominated by the human summarisation, receiving a perfect score in every repo assessed. The next highest performing method was the sentence summary, this method had an average score of 3.62 across the test set in only barely managed to rank in second place. Following closely behind in performance is the Single README summarisation, while this did not outperform the sentence summary it did achieve an average score of 3.192.

In order to evaluate whether the sentence summary truely outperformed the Single README Summary, a statistical test was performed. To best evaluate the series of scores, the Wilcoxon Signed-Rank Test [11] was utitilised to see if these results showed any statistical significant difference at a 95% confidence interval.



Naive Sentence	Single README	abs	R	Select-R
4	4	n/a	n/a	n/a
4	4	n/a	n/a	n/a
4	2	1	10.5	10.5
4	1	1	12.5	12.5
4	4	n/a	n/a	n/a
3	2	1	5	5
3	4	-1	5	-5
4	4	n/a	n/a	n/a
3	3	n/a	n/a	n/a
4	3	1	5	5
3	4	-1	5	-5
4	4	n/a	n/a	n/a
3	4	-1	5	-5
4	1	1	12.5	12.5
1	1	n/a	n/a	n/a
4	3	1	5	5
3	3	n/a	n/a	n/a
3	4	-1	5	-5
4	4	n/a	n/a	n/a
4	3	1	5	5
4	2	1	10.5	10.5
4	4	n/a	n/a	n/a
4	4	n/a	n/a	n/a
4	4	n/a	n/a	n/a
4	4	n/a	n/a	n/a
4	3	1	5	5

The value of z is-1.7821. The p-value is .07508.

Fig. 2: Wilcoxon Signed-Rank Test [11] for significant difference between Naive Sentence & Single README

The resultant p-score from Table 2 was 0.075 which suggests that these results could not determine a statistically significant difference between the two methods. In 50% of the results, these methods scored identically, most often in this case being when they both would score 4. This shows that they can both summarise repos to a near identical effectiveness, but the TextRank algorithm can sometimes introduce elements to a summary that make it less insightful. This can be attributed to the high ranking of sentence in a summary that may be less contstructive to someone trying to understand the repo's purpose. If there is a detailed section on package installation in the README, this is useful for developers who have already to decided to try out or use this library. Unfortunately for developers who have not yet identified the repos purpose, this does not present any current use. Nevertheless, these sections if verbose enough can cause sentences to rank highly in the summarisation. This can occur from a number of sections not directly related to the purpose like the Contribution Guide, Code of Conduct, or even the Licensing. The inclusion of these sections could be observed across many of the repos that where the README was able to be summarised. Another factor that played into the effectiveness of the sentence summarisation was that most information pertaining to the purpose of a repo are contained within the first few sentences of the repo, this aligns with the assumption made in the method's definition. With more verbose filtering of the README's to remove sections that do not directly contribute to defining a README, the performance of the methods that utilise TextRank would increase. Despite this performance, there are summaries where the Single README also outperformed Naive Sentence which shows that there is a use case for this algorithm over the naive method.

The least effective method was the Multi-README Summary, this method only achieved an average score of 1.85. This was due to the abundance of illegible README's that were produced by the spider. While only present in very few cases, there were some repos that were highly rated and identified as a top pick as a similar repo but ended up obfuscating the purpose due to the fact the README was in another language. Another cause for this is highly ranking sentences in certain repos may be those that are not relevant to the definition to the library as mentioned prior. Another notable cause for the lack of performance was in the event that one of the similar repos had a README which was significantly longer than the others. This lead to the larger & more verbose repos dominating the others. This phenomena creates summaries that are less generalised to the topic, but instead more-so a summarisation of this one repo. This creates a less effective result for the initial repo.

Another important observation was the mode of each of the method's scores. This also generalises the expected performance on a random valid repo. Due to the namture of the human made summary achieving a perfect score, the mode for this method is also 5. Both the sentence summarisaion & the single README summary have the next highest mode score at 4. This only further shows how their performance in this case are near identical. It also shows that the performance is close to that of the human summary. With the Multi-README summary, the modal value was 1. This has been explained by the methodology in which the system creates summaries. Despite the lack of performance with the modal value, there are more scores that are higher than 1 then there are of 1

TABLE I: Blind scoring of each summarisation method on a set of repos using the defined scale of performance

Summarisation Method	Summary Score																										Average Score	Modal Value
Human Written	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
Naive Sentence	4	4	4	4	4	3	3	4	3	4	3	4	3	4	1	4	3	3	4	4	4	4	4	4	4	4	3.62	4
Single README	4	4	2	1	4	2	4	4	3	3	4	4	4	1	1	3	3	4	4	3	2	4	4	4	4	3	3.19	4
Multi-README	2	1	2	1	1	1	2	2	1	1	1	4	1	1	1	2	3	3	2	2	1	3	2	3	2	3	1.85	1

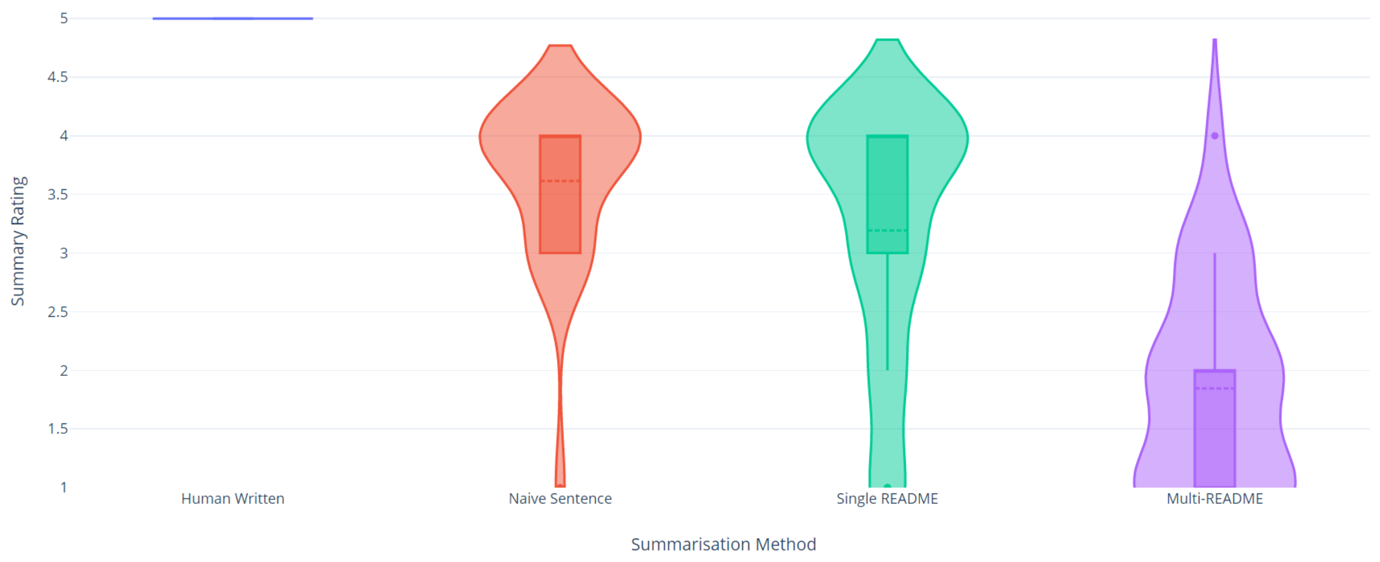


Fig. 3: Distributions of scores for each summarisation method

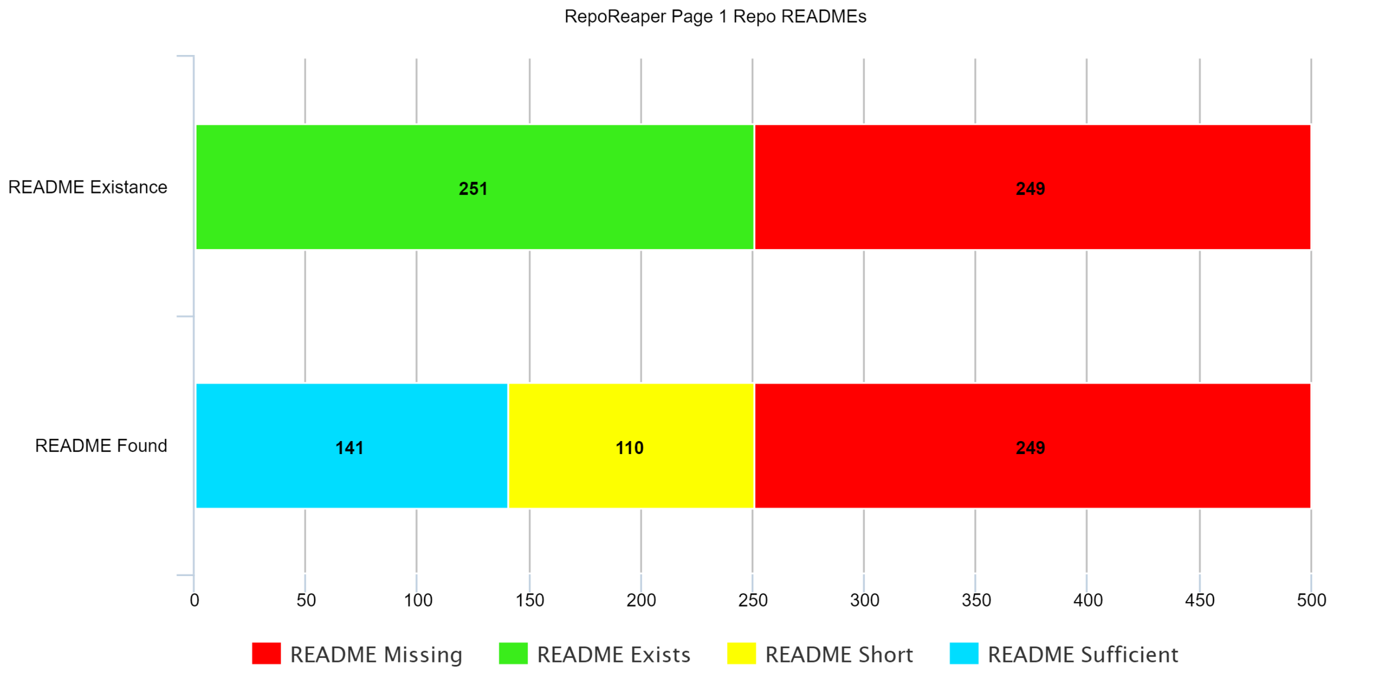


Fig. 4: Proportion of repos on Page 1 of RepoReaper database that include READMEs alongside proportion of READMEs that contain a sufficient amount of content to summarise

so this is a common outcome, more results will outperform it. Some noticeable results from Table I are also observed. One of the results from the Multi-README summary in which it scored not only a 4, but also identically to the other two non-manual methods of summarisation, this shows potential for effective summarisation utilising this technique after further investigation. This also occurred on another repo where all automated processes achieved an identical score of

1, comparing this to the 5 that the human written summary received there shows that more work needs to be completed to create a truly efficient method of summarising technical documents.

One statistic that came up within testing outside of summarisation was the number of repos that either did not include READMEs or did, but they contained next to no content. Figure 4 shows that for the first 500 repos in the database, presented on the first page, 50% of them did not include a README file at all. Another 22% did not include repos that contained enough information to be able to reduce the text with summarisation. Only 28% on this page were able to be summarised by the application. This trend persisted along the first 20 pages or 10,000 records with similar amounts of repos not containing enough valid data.

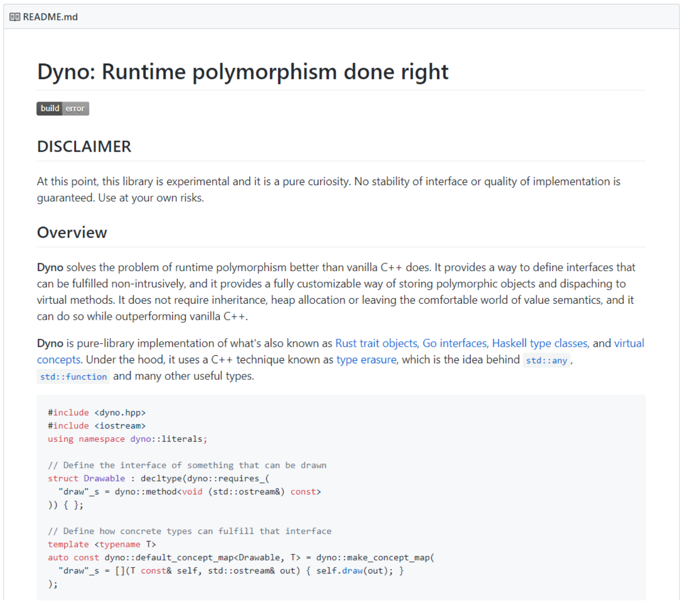
V. Conclusion & Future Work

The resulting system while it did not perform to the same degree as a human, performed similarly well with some summarisation methods. Utilising TextRank on a deformatted README or pulling the first four sentences created summaries that performed near identically as well as each other. This has provided an alternative for repos where the GitHub topics is not utilised by the owner of the repo. Utilising multiple READMEs has shown that there is potential in creating a solution that performs well. Howevern, there needs to be further considerations on filtering repos & there READMEs to classify what a similar repo is. These repos also have to be normalised in content length to best rank sentences across the texts. Further development in identifying keyword topics for technical documentation may also increase the system's abilities in identifying similar repos. A dictionary of GitHub topics could also be used with keyword extraction to identify other repos of a similar nature.

Further development of the algorithm used to summarise the text could also propel the performance of the system to that of a human if supervised learning techniques were utilised. Teaching an intelligent agent to identify important sentences in the context of software documentation would tremendously increase the performance. This would require further labelling of the RepoReaper dataset & significant training to produce competitive results. This could also be key in making a search of similar repos a viable option as identifying sentences relevant to the topic would assist in removing repo specific content from affecting the resultant summary.

Further work in increasing the usability of this application can also involve integrating this into a browser for ease of use. Running the script on each URL requires an additional window whereas running it as an extension & embedding the result into the pageview would make its use a seamless experience. Broadening the scope of repos to be summarised would also be an effective improvement. Despite GitHubs popularity there are numerous other code sharing & version control services like BitBucket that could also be integrated into the application.

By far the most effective way to increase the performance of the application is awareness, with the inclusion of more README files & README files that go in depth to not only explain the repo, but other facts for people who may not quite understand by the title or code. This will not only assist the application to identify & summarise the repos, but also expose them to other developers who may be able to utilise and or contribute to them.


README.md

Dyno: Runtime polymorphism done right

build error

DISCLAIMER

At this point, this library is experimental and it is a pure curiosity. No stability of interface or quality of implementation is guaranteed. Use at your own risks.

Overview

Dyno solves the problem of runtime polymorphism better than vanilla C++ does. It provides a way to define interfaces that can be fulfilled non-intrusively, and it provides a fully customizable way of storing polymorphic objects and dispaching to virtual methods. It does not require inheritance, heap allocation or leaving the comfortable world of value semantics, and it can do so while outperforming vanilla C++.

Dyno is pure-library implementation of what's also known as Rust trait objects, Go interfaces, Haskell type classes, and virtual concepts. Under the hood, it uses a C++ technique known as type erasure, which is the idea behind `std::any`, `std::function` and many other useful types.

```
#include <dyno.hpp>
#include <iostream>
using namespace dyno::literals;

// Define the interface of something that can be drawn
struct Drawable : decltype(dyno::requires_(
  "draw"_s = dyno::method<void (std::ostream&) const>
)) { };

// Define how concrete types can fulfill that interface
template <typename T>
auto const dyno::default_concept_map<Drawable, T> = dyno::make_concept_map(
  "draw"_s = [](T const& self, std::ostream& out) { self.draw(out); }
);
```

- Naive Sentence: dyno runtime polymorphism done right disclaimer at this point, this library is experimental and it is a pure curiosity. no stability of interface or quality of implementation is guaranteed. use at your own risks. overview dyno solves the problem of runtime polymorphism better than vanilla c++ does.
- Single README: dyno solves the problem of runtime polymorphism better than vanilla c++ dyno is pure-library implementation of what's also known as rust trait objects, go interfaces, haskell type classes, and virtual concepts. under the hood, it uses a c++ technique known as type erasure, which is the idea behind and many other useful types.
- Multi-README: for this, it includes the most popular data types, type classes and abstractions such as to empower users to write pure fp apps and libraries built atop higher order abstractions.felix dynamix was initially developed as boost.mixin(but is now a separate library, that doesn't depend on the boost libraries collection.badge.release

Fig. 5: Dyno Framework README [2] with the three automated summarisation methods run on this document

The repo containing this project alongside this paper is open-source & available at https://github.com/01010111-01000010/README-Summariser

Non-Standard Code Libraries Used

nltk, ”Natural Language Toolkit”, https://github.com/nltk/nltk
orsinium, ”TextDistance”, https://github.com/orsinium/textdistance
summanlp, ”summa - textrank”, https://github.com/summanlp/textrank
kennetrheitz, ”Requests: HTTP for Humans”, https://github.com/kennethreitz/requests
csurfer, ”rake-nltk”, https://github.com/csurfer/rake-nltk

References

[1] GNU-Project, “Gnu coding standards,” 1995.

[2] ldionne, “Dyno,” *https://github.com/ldionne/dyno*.

[3] S. Ikeda, A. Ihara, R. Kula, and K. Matsumoto, “An empirical study on readme contents for javascript packages,” *IEICE Transactions on Information and Systems*, vol. E102.D, 02 2018.

[4] G. Artha Azriadi Prana, C. Treude, F. Thung, T. Atapattu, and D. Lo, “Categorizing the content of github readme files,” *Empirical Software Engineering*, 02 2018.

[5] R. Mihalcea and P. Tarau, “Textrank: Bringing order into texts,” 07 2004.

[6] G. Erkan and D. Radev, “Lexrank: Graph-based lexical centrality as salience in text summarization,” *Journal of Artificial Intelligence Research - JAIR*, vol. 22, 09 2011.

[7] K. Ganesan, “Topic suggestions for millions of repositories,” *GitHub Blog*, 07 2017.

[8] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, “Curating github for engineered software projects,” *Empirical Software Engineering*, vol. 22, 01 2016.

[9] J. Ratcliff and D. Metzener, “Pattern matching: The gestalt approach,” *Dr. Dobb’s Journal*, vol. 141, p. 46, 07 1988.

[10] S. Rose, D. Engel, N. Cramer, and W. Cowley, *Automatic Keyword Extraction from Individual Documents*, 03 2010, pp. 1 – 20.

[11] F. Wilcoxon, “Individual comparisons by ranking methods,” *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945. [Online]. Available: http://www.jstor.org/stable/3001968