Read between the lines - Functionality Extraction From READMEs

Anonymous ACL submission

Abstract

While *text summarization* is a well-known NLP task, in this paper, we introduce a novel and useful variant of it called *functionality extraction from Git* README *files*. Though this task is a *text2text* generation at an abstract level, it involves its own peculiarities and challenges making existing *text2text* generation systems not very useful. The motivation behind this task stems from a recent surge in research and development activities around the use of large language models for code-related tasks, such as code refactoring, code summarization, etc. We also release a human-annotated dataset called FuncRead, and develop a battery of models for the task. Our exhaustive experimentation shows that small size fine-tuned models beat any baseline models that can be designed using popular black-box or white-box large language models (LLMs) such as ChatGPT (OpenAI, 2023) and Bard (Chowdhery et al., 2022). Our best fine-tuned 7 Billion CodeLlama model exhibit 70% and 20% gain on the F_1 score against ChatGPT and Bard respectively.

1 Introduction

Large Language Models (LLMs) are known to perform really well on many *text2text* (Yang and Flek, 2021) generation tasks such as *summarization* (Liu and Lapata, 2019; El-Kassas et al., 2021)), *translation* (Wang et al., 2019; Maruf et al., 2021), etc. Because of this success, there is a growing research interest in applying LLMs in novel task settings such as *explaining complex codes, generating new recipes, simplifying contents,* etc[1]. In this paper, we introduce another novel task called *functionality extraction from Git* README *files* – a variant of *text summarization* task (Prana et al., 2019) that detects all the functionalities supported by the corresponding application software. This task can also be seen as a variation of a Question-Answering (QA) (Fan et al., 2019; Soares and Parreiras, 2020) task where the question like *List all functionalities* is fixed.

The motivation to introduce *automatic functionality extraction from Git* README *files* stems from the requirement of application code refactoring to decompose a monolith application into functional microservices. Here each microservice is a collection of closely connected application artifacts (programs, tables etc.) supporting a common functionality (Lewis and Fowler, 2014; Richardson, 2018; Newman, 2021). Current microservice recommendation systems rely a lot on subject matter experts (SMEs) and falls short to correctly group artefacts since they do not have reference list of functionalities. But many application Git README files tend to contain capture *different functionalities* [2] *of the underlying software code base*[3] along with other implementation details like *what it does, how others can use it, licensing, etc.,*(Prana et al., 2019; Chen et al., 2021). As an example, the README file of the Daytrader application[4] discusses *the application overview, the technology used, licensing terms,* etc., and in between discusses *four functionalities* as highlighted in Figure 1(a).

Recently, (Doan et al., 2023) focused on leveraging LLM to generate sections of README.md like "About" section (brief 1-2 line summary of repo) but they do not aim to list all the functionalities. Extraction of the application functionalities from such README files is not straightforward. The functionalities may not be always structured and might spread across multiple paragraphs and lines. Therefore, there is a need for an intelligent system that can parse the text, understand functionality expressions, de-duplicate, and list them. To tackle this first-of-its-kind task, we also introduce and re-

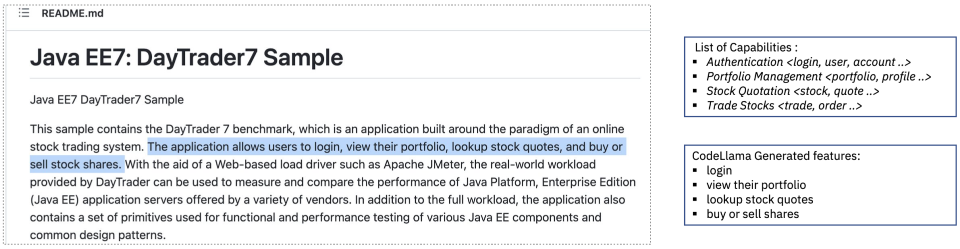
[1] https://platform.openai.com/examples

[2] Occasionally, we call *functionality* as *feature*

[3] https://docs.GitHub.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-readmes

[4] https://GitHub.com/WASdev/sample.daytrader7/

Findings of the Association for Computational Linguistics: NAACL 2024, pages 3977–3990
June 16-21, 2024 ©2024 Association for Computational Linguistics



(a) Snapshot of Description (b) Functionalities Identified

Figure 1: Snapshot of Github README content of Daytrader, an online trading application is captured in (a). The human annotated four functionalities based on the description are listed as golden truth along with the functionalities generated by fine-tuned 7 billion CodeLlama model.

lease a new dataset called `FuncRead` that will help the community to benchmark their functionality understanding module and refactor monolith applications into discovered functional microservices. The key contributions of this paper are as follows.

1. We introduce a novel *functionality extraction from Git* `README` *files* task and human-annotated dataset called `FuncRead`. This dataset captures the human-annotated lists of the functionalities in both extractive and abstractive forms for each of 2101 different GitHub `README` files following permissible licenses.
2. We perform a comparative analysis of generative models to reason out the gap in performance between different baselines on the `FuncRead` dataset. To enable comparison, we perform bipartite matching (one-to-one, many-to-one, and weighted many-to-one) to align generated functionalities with the gold functionalities.
3. We present smaller fine-tuned generative models 1&7 billion StarCoderbase, 2.7 billion phi-2, 7 billion Llama-2 & CodeLlama which give superior results compared to ChatGPT and Bard.

2 FuncRead Dataset

The `FuncRead` dataset is a first-of-its-kind dataset that consists of functionalities described in the `README` files. These functionalities were hand-curated by human annotators after carefully reading the file. For each `README` file, the functionalities are annotated in two formats - *extractive* and *abstractive*. Extractive functionalities are segments of the text or span from the `README` file; whereas abstractive functionalities are the self-explained versions of the corresponding extractive functionalities, written in the annotator's own words. Each of these format outputs are presented in the form of a list. The dataset consists of unique 2101 human annotated GitHub `README` files.

2.1 Dataset Collection

We used GitHub provided APIs to randomly select a subset of public repositories that comes with a permissible licenses. Further, we manually inspected the `README` files of these repositories and retained only the ones that comprised of at least two functionalities. Note, we do not store the `README` files for the crawled repositories, we only extracted the `README` content and other metadata like license information. We also removed markdown tags and any Personal Identifiable Information (PII) like names, email addresses etc. before further processing. The license distribution for the 2101 `README` files are as follows MIT (*1436), Apache (*334) , BSD (*334), and EPL (*6) licenses. We found that the majority of the repositories consist of 10 or lesser functionalities with an average being 5 functionality per repository. Some repository has as many as 34 different functionalities.

2.2 Dataset Annotation

We had a total of seven annotators involved in the initial data annotation process. Each annotator was asked to read the whole `README` file and perform both the annotations – *extractive* and *abstractive*. For extractive annotation, annotators were asked to select text spans from the `README` file which they felt were describing functionalities, and note them in the form of a numbered list. For abstractive annotation, each annotator was asked to describe the functionalities in their own words. All the annotators were given a disjoint set of `README` files.

2.3 Annotation Validation

We employed two new independent annotators for the purpose of human validation of the dataset obtained from the previous step. We randomly sampled 200 README files from each of these two annotators out of which 50 README files were common for both the annotators. Both of these annotators were instructed to read extractive as well as abstractive functionalities and check whether all the functionalities were included. Based on their observation, they were tasked to give a rating from 1 to 4 based on the degree of strictly necessary functionalities annotated. These ratings were used to calculate the inter-annotator agreement. We observed a Kappa score of 0.873. Figure 2 describes the ratings and the rating score distribution for both.

More details on the dataset characteristics and annotation procedure can be found in appendix.

Rating	Description	Annotator 1(%)	Annotator 2(%)
1	All functionalities are included and no unwanted functionality is included.	77.5	73.0
2	One ore more functionalities are missed but no unwanted functionality is included.	2.5	4.0
3	One ore more functionalities are included but no functionality is missed.	14.0	14.5
4	One ore more functionalities are missed and one or more unwanted functionalities are also included.	6.0	8.5

Figure 2: Ratings distribution of the two annotators during the verification step of the FuncRead dataset.

3 Task Modelling

For modeling purposes, one can view the *functionality extraction* as a generation task. In the generation mode, the goal is to generate a list of functionalities from a given README file. As ours is the first-of-its-kind dataset, we used ChatGPT and Bard models known to perform really well on most NLP and code tasks even in zero-shot setting as a baseline for our task. Among many prompts, the following prompt "*List all the features from above text. Each features should be in individual line without headings. Each features should be in individual line without headings. Do not include features related to license*" provided the best results. The actual list of prompts tried on ChatGPT and Bard can be found in section 6.5.

We wanted to study if task specific small sized models can provide competitive results. For this we considered mix of NL and code model variants like 1b and 7b StarCoderbase, 2.7b phi-2 and 7b llama-2 and CodeLlama. For fine-tuning, we pre-processed the README data through the steps listed in section 2.1. Next, we append it with "*\n##FEATURES##\n*" as the task designator prompt followed by the human annotated list of functionalities corresponding to that README file. For inference, we simply appended the task designator prompt to the README text and then allowed the model to complete sequence to generate list of functionalities.

4 Experiments and Results

For our experiments, we divided the FuncRead dataset into train, validation, and test sets comprising 1801, 100, and 200 samples respectively.

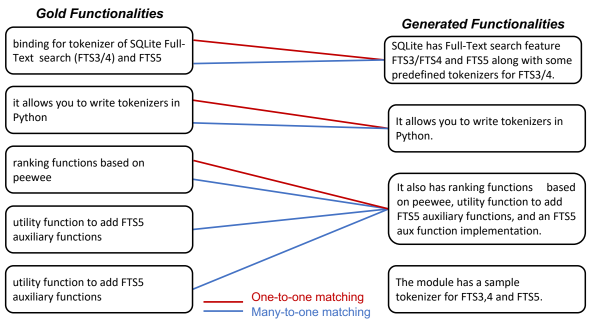


Figure 3: One-to-One bipartite matching (red color) and Many-to-one bipartite matching (blue color). Edges are established based on cosine similarity

4.1 Evaluation Metrics

To evaluate the quality of the generated functionalities, we align them to the gold annotated functionalities via bipartite matching. We perform three kinds of bipartite matching: i) one-to-one, ii) one-to-many, and iii) weighted one-to-many.

In any of these bipartite graphs, we have model-generated functionalities as nodes on one side and gold (ground truth) functionalities as nodes on the other side. The presence or absence of an edge in this bipartite graph is decided by the similarity scores between the corresponding sentences. In our experiments, we found threshold 0.3 similarity matches the most with the human judgment. We did maximum bipartite matching to compute Precision (P), Recall (R), and F_1 scores based on matched pairs to measure the generation capability.

For fine-tuning the models, we used extractive functionalities as gold, and because of it, we employed ROUGE-1, ROUGE-2, ROUGE-L scores to check the lexical matching quality of generated functionalities at an individual level. Since all the considered models are generative models, there is a high chance that it would introduce new tokens while generating functionalities. Hence, we also

Model	$F_1^{\#}$	$P^{\#}$	$R^{\#}$	F_1^{*}	P^{*}	R^{*}	F_1^{+}	P^{+}
ChatGPT	0.459	0.336	**0.900**	0.431	0.303	**0.922**	0.406	0.282
Bard	0.653	0.611	0.806	0.649	0.573	0.858	0.612	0.528
StarCoderbase-1b	0.772	0.816	0.786	0.808	0.788	0.876	0.754	0.711
StarCoderbase-7b	0.743	0.797	0.754	0.787	0.777	0.844	0.734	0.698
Phi- 2	0.231	0.172	0.656	0.226	0.159	0.733	0.207	0.144
Llama2-7b	0.698	0.748	0.715	0.715	0.700	0.795	0.658	0.622
CodeLlama-7b	**0.784**	**0.827**	0.794	**0.816**	**0.801**	0.877	**0.770**	**0.738**

Table 1: Result comparison for various fine-tuned models against out-of-the box large models for threshold = 0.3. # represents one-to-one bipartite matching, * represents many-to-one bipartite matching, + represents weighted many-to-one bipartite matching.

Model	ROUGE-1			ROUGE-2			ROUGE-L		
	F_1	P	R	F_1	P	R	F_1	P	R
ChatGPT	0.423	0.404	0.564	0.301	0.291	0.391	0.410	0.390	0.549
Bard	0.616	0.648	0.673	0.511	0.542	0.549	0.609	0.640	0.666
StarCoderbase-1b	0.759	0.750	**0.845**	0.676	0.667	**0.755**	0.757	0.747	**0.842**
StarCoderbase-7b	0.754	0.790	0.802	0.640	0.663	0.688	0.752	0.788	0.800
Phi-2	0.665	0.677	0.765	0.567	0.571	0.658	0.663	0.674	0.762
Llama2-7b	0.755	0.787	0.810	0.659	0.688	0.706	0.752	0.783	0.806
CodeLlama-7b	**0.778**	**0.815**	0.820	**0.684**	**0.710**	0.725	**0.777**	**0.813**	0.818

Table 2: Results for one-to-one matched pairs of different models generation and ground truth for threshold = 0.3.

Model	BERTScore		
	F_1	P	R
ChatGPT	0.895	0.889	0.902
Bard	0.912	0.910	0.916
StarCoderbase-1b	0.945	0.940	**0.951**
StarCoderbase-7b	0.938	0.938	0.940
Phi-2	0.928	0.925	0.933
Llama2-7b	0.936	0.935	0.939
CodeLlama-7b	**0.946**	**0.946**	0.947

Table 3: Results for one-to-one matched pairs for threshold = 0.3.

used BERTScore (Zhang et al., 2019) to capture the semantic similarity between the matched pairs.

4.2 Results

Overall, we find fine-tuned models specifically code models are reliable for this novel task. From table 1, we can observe fine-tuned models have a tendency to combine multiple functionalities into a single sentence but F_1, P, and R scores of many-to-one bipartite matching indicates that it still does less frequently. But all the fine-tuned models significantly outperform ChatGPT, Bard on P and F_1 measures. Due to inherent verbosity, R is higher for the latter models. Table 2 ROUGE scores demonstrates that the functionalities generated by the fine-tuned models have a relatively higher token similarity when matched one-to-one (it is consistent for the other two schemes as can be seen in appendix). Table 3 BERTScores are also consistent with the claims showing better semantic similarity for the fine-tuned models. We suspect code models tendency to outperform NL models can be due to their stronger exposure to Git data. In few instances the models did not list any functionalities which can be attributed to complexity and lack in standardization of GitHub `README` files. Please refer to appendix for in-depth comparisons and discussions.

5 Conclusion

We introduced a novel task *functionality extraction from Git* `README` *files* and studied on a new dataset curated from public repositories to demonstrate reliability of small sized fine-tuned LLMs.

References

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. Palm: Scaling language modeling with pathways.

Thu TH Doan, Phuong T Nguyen, Juri Di Rocco, and Davide Di Ruscio. 2023. Too long; didn't read: Automatic summarization of github readme. md with transformers. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, pages 267–272.

Wafaa S El-Kassas, Cherif R Salama, Ahmed A Rafea, and Hoda K Mohamed. 2021. Automatic text summarization: A comprehensive survey. *Expert Systems with Applications*, 165:113679.

Angela Fan, Yacine Jernite, Ethan Perez, David Grangier, Jason Weston, and Michael Auli. 2019. Eli5: Long form question answering. *arXiv preprint arXiv:1907.09190*.

J. Lewis and M. Fowler. 2014. `www.martinfowler.com/articles/microservices.html`. `www.martinfowler.com/articles/microservices.html`.

Yang Liu and Mirella Lapata. 2019. Text summarization with pretrained encoders. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3730–3740.

Sameen Maruf, Fahimeh Saleh, and Gholamreza Haffari. 2021. A survey on document-level neural machine translation: Methods and evaluation. *ACM Computing Surveys (CSUR)*, 54(2):1–36.

Sam Newman. 2021. *Building microservices*. " O'Reilly Media, Inc.".

OpenAI. 2023. Chatgpt (sep 25 version) [large language model].

Gede Artha Azriadi Prana, Christoph Treude, Ferdian Thung, Thushari Atapattu, and David Lo. 2019. Categorizing the content of github readme files. *Empirical Software Engineering*, 24(3):1296–1327.

Chris Richardson. 2018. *Microservices patterns: with examples in Java*. Simon and Schuster.

Marco Antonio Calijorne Soares and Fernando Silva Parreiras. 2020. A literature review on question answering techniques, paradigms and systems. *Journal of King Saud University-Computer and Information Sciences*, 32(6):635–646.

Qiang Wang, Bei Li, Tong Xiao, Jingbo Zhu, Changliang Li, Derek F Wong, and Lidia S Chao. 2019. Learning deep transformer models for machine translation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 1810–1822.

Diyi Yang and Lucie Flek. 2021. Towards user-centric text-to-text generation: A survey. In *International Conference on Text, Speech, and Dialogue*, pages 3–22. Springer.

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. 2019. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675*.

6 Appendix

We organize the appendix to cover the following :

- Limitations - Discuss four key limitations with this work that we plan to address in our future studies.
- Dataset - Discuss the crawled github data characteristics in detail
- Annotator Profile - Discuss the demography and key details of annotators who helped prepare the study dataset
- Annotator Instruction - Discuss in detail the instructions and guidance provided to annotators
- Annotation Validation - Discuss in detail the steps taken to review annotations
- Task Modelling using Baseline Models - List all the prompts tried to get the most accurate functionalities
- Model Hyperparameters - Key hyper-parameters used to reproduce results
- Quantitative Results - Discuss results in detail for the different settings and thresholds

6.1 Limitations

There are four major limitations in this work that could be addressed in future research. First, the study focused on 2101 samples, there could be more unknown ways of describing functionalities that the current models may not be able to handle. This can be addressed by increasing the dataset size. Second, as shown in Figure 2, we found human errors during the annotation process where, for a few samples, unwanted functionalities were added and some wanted functionalities were missed. But this can be handled by expanding the validation efforts to the rest of the samples. Third, handling very long `README` files is a challenge as we have a maximum of 2048 token limit for models. There is promising research in this direction to support longer token limit. Fourth, defining the reference set of functionalities is sometimes an ill-posed problem because different humans may perceive the `README` differently and they may conceive the set of functionalities differently. But we hope to educate annotators by discussing more number of ground truth samples.

6.2 Dataset

Table 4 shows the license distribution for the 2101. Figure 4 represents the functionalities count distribution for the repositories. `README` files. We plan to release this dataset post review period.

License	Count	Count Percentage(%)
MIT	1436	68.34
Apache	334	15.90
BSD	325	15.47
EPL	6	0.29

Table 4: License-wise split of `FuncRead` dataset.

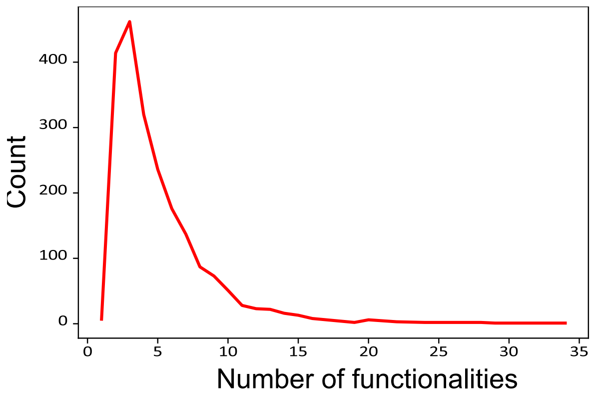


Figure 4: Functionalities count distribution of the `FuncRead` dataset.

6.3 Annotators Profile

To prepare the dataset, we requested participation from nine software engineers based out of Asia. The participants were identified based on their prior experience working on application modernization projects listed on their profile page. On an average, the participants had industrial experience of 13 years in different software engineering roles. We requested seven participants to annotate the 2101 different GitHub `README` files. Once extractive and abstractive functionalities were annotated, we employed 2 new participants to perform the verification step. We individually discussed the task details, expectations, the tentative average time that might be needed (5 minutes per annotation), and the research goal and got their consensus before providing them with the annotation instruction.

6.4 Annotation Instructions

Following were the instructions given to the seven annotators :

- *We thank you for agreeing to annotate. An excel sheet will be given with the following information*
  - *Repository id*
  - *Readme URL*
  - *Extractive functionalities*

- *Abstractive functionalities*

- *First row will be filled for convenience.*

- *For each repository id two types of annotations are requested to be done*
  - *Extractive: Copy and paste the functionalities as numbered lists.*
  - *Abstractive: Write the functionality in your own words.*
    - ***NOTE:** Please do not copy-paste for this. Please try to be as descriptive as possible i.e., introduce new words to describe instead of reusing the same set of words.*

- *Please write/copy-paste each functionality in the new line as a numbered list.*

- *Please make sure that number of abstractive and extractive functionalities are the same.*

- *Few things to take care*
  - *Do not include future/expected functionalities/roadmap/TODO/planned*
  - *Please do not click on any link to find more functionalities. Whatever functionalities are present in the* `README`*, please include those only.*
  - *Do not include application – meaning what is possible with that functionality or repository.*
  - *In Progress/partial functionalities can be included.*

All the annotators were given the same set of instructions so as to maintain consistency. Annotators' doubts were clarified on regular basis. The generated dataset was reviewed by the authors internal review board and was deemed suitable to be published for research.

6.4.1 Annotator Validation Example

Let us understand above ratings via an example. For the `README` given in Figure 1, suppose following extractive functionalities were annotated by an annotator:

- *allow users to login*
- *lookup stock quotes*
- *buy or sell stock shares*
- *provides a real-world java EE workload*

It is now clear that the annotator in this specific case has missed one of the functionality, namely "*view their portfolio*" and added an extra functionality namely "*provides a real-world java EE workload*". Therefore, a rating of 4 would be assigned during the human validation step.

6.5 Task Modelling using ChatGPT, Bard

To understand what prompts helps best to list the functionalities, we tried various prompt on ChatGPT and Bard baseline models. Some of them are as follows:

- *List all the features for the above text.*
- *List all the functionalities for the above text.*
- *List all the features from above text. Each features should be in individual line without headings.*
- *List all the features from above text. Each features should be in individual line without headings. Each features should be in individual line without headings.*
- *List all the features from above text. Each features should be in individual line without headings. Each features should be in individual line without headings. Do not include features related to license*

6.6 Evaluation Metrics

To evaluate the quality of the generated functionalities, we align them to the gold annotated functionalities via bipartite matching. We perform three kinds of bipartite matching: i) one-to-one, ii) one-to-many, and iii) weighted one-to-many.

In any of these bipartite graphs, we have model-generated functionalities as nodes on one side and gold (ground truth) functionalities as nodes on the other side. The presence or absence of an edge in this bipartite graph is decided by the similarity scores between the corresponding sentences. Figure 3 captures an illustration. For computing the similarity score, we used SentenceTransformer[5] and generated the sentence embeddings for both model-generated and gold functionalities sentences. Next, we computed a cosine similarity between these two vectors, and experimented with multiple thresholds to decide whether the edge should be present in the bipartite graph. In our experiments we found threshold 0.3 matches the most with the human judgment. A lower threshold was giving poor-quality mapping with excessively matched

[5] `https://www.sbert.net/`

pairs. A higher value was giving high-quality mapping but the number of matched pairs was very less. We used the maximum_bipartite_matching[6] function from SciPy library to perform the maximum (weighted or unweighted) bipartite matching. Based on the matched pairs, we compute Precision (P), Recall (R), and F_1 scores to measure the generation capability.

For fine-tuning the models, we used extractive functionalities as gold, and because of it, we employed ROUGE-1, ROUGE-2, ROUGE-L scores to check the lexical matching quality of generated functionalities at an individual level. Since all the considered models are generative models, there is a high chance that it would introduce new tokens while generating functionalities. Hence, we also used BERTScore (Zhang et al., 2019) to capture the semantic similarity between the matched pairs.

After analyzing the generated functionalities, we realized that the model sometimes combines multiple functionalities into a single generated sentence (see Figure 3). Therefore, there is a need for many-to-one bipartite matching where multiple gold functionalities are allowed to map into a single generated functionality. There are two kinds of results we show in many-to-one bipartite matching. The first one is *many-to-one* P, R, and F_1 scores, where all the edges in the bipartite matching are given a score of 1. The second is *weighted many-to-one* P, R, and F_1 scores, where for each of the model-generated functionality that is matched with multiple gold functionalities, each matched edge is assigned a weight that is inversely proportional to the number of functionalities matched. We take the reciprocal of the number of matched edges and assign that as a weight to all the incoming edges for that particular model-generated functionality. For example, consider the third functionality sentence generated by the model in Figure 3, which reads *"It also has ranking functions based on peewee, utility function to add FTS5 auxiliary functions and an FTS5 aux function implementation."* Now, each matched edge incident on this node gets a weight of $1/3$ for weighted many-to-one bipartite matching.

6.7 Model Hyperparameters

Table 17 shows the important hyperparamters that can be used to reproduce results. Rest of the hyperparamters are the default ones present in Huggingface Trainer API.

6.8 Quantitative Results

All experiments were performed on an $A100$ $80GB$ GPU machine.

We report results on the discussed metrics for all the fine-tuned models and compare them against the ChatGPT and Bard. Table 1 shows the P, R, and F_1 scores for the three bipartite matching schemes. We do not report R for weighted many-to-one bipartite matching as it is the same as R for many-to-one bipartite matching. Results in tables 1, 2, and 3, are restricted over that subset of test samples for which each of these models outputs a nonempty string and also yields at least one matched pair during the bipartite matching procedure. The total comparable test samples thus came down to 69.

From table 1, we can observe that all the fine-tuned models significantly outperform ChatGPT and Bard across P, R, and F_1 measures. We can see that the F_1 score of one-to-one bipartite matching for ChatGPT is 0.459 and for Bard is 0.653 which are much smaller as compared to code models. Table 2 further shows the ROUGE scores for one-to-one matched pairs. Again we see that the functionalities generated by the fine-tuned models have a relatively higher lexical similarity. Table 3 shows BERTScore which is again higher than ChatGPT and Bard. Tables 5 and 6 shows many-to-one results for threshold = 0.3. The rest of the tables show results for other threshold values 0.4 and 0.5 and matching schemes. Count of common test samples across various models which have non-empty generations and have at least one matched pair are 85 and 98 for threshold values 0.4 and 0.5 respectively. An increase in ROUGE and BERTScore gives the illusion that a higher threshold value should be preferred but as mentioned earlier the number of functionalities generated/classified decreases too which is not much helpful as we lose out on many functionalities. We recorded the responses from ChatGPT and Bard on November 25, 2023 for our experiments.

For the different task types and for threshold 0.4, please refer tables 7-11. For threshold 0.5, please refer tables 12-16.

[6] https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csgraph.maximum_bipartite_matching.html

Model	ROUGE-1			ROUGE-2			ROUGE-L		
	F_1	P	R	F_1	P	R	F_1	P	R
ChatGPT	0.607	0.576	0.792	0.467	0.448	0.604	0.589	0.558	0.772
Bard	0.687	0.719	0.764	0.583	0.617	0.636	0.681	0.711	0.758
StarCoderbase-1b	0.765	0.752	0.868	0.677	0.664	0.772	0.763	0.750	0.864
StarCoderbase-7b	0.742	0.766	0.813	0.626	0.639	0.688	0.739	0.762	0.809
Phi-2	0.664	0.667	0.775	0.567	0.567	0.662	0.661	0.663	0.769
Llama2-7b	0.734	0.762	0.806	0.637	0.655	0.699	0.732	0.758	0.802
CodeLlama-7b	0.772	0.797	0.833	0.681	0.699	0.735	0.770	0.795	0.830

Table 5: Results for many-to-one matched pairs with threshold = 0.3.

Model	BERTScore		
	F_1	P	R
ChatGPT	0.918	0.909	0.929
Bard	0.920	0.917	0.924
StarCoderbase-1b	0.950	0.944	**0.958**
StarCoderbase-7b	0.941	0.940	0.944
Phi-2	0.935	0.931	0.941
Llama2-7b	0.941	0.938	0.945
CodeLlama-7b	**0.951**	**0.950**	0.953

Table 6: Results for many-to-one matched pairs with threshold = 0.3.

Model	$F_1^{\#}$	$P^{\#}$	$R^{\#}$	F_1^*	P^*	R^*	F_1^+	P^+
ChatGPT	0.431	0.314	0.849	0.415	0.293	0.878	0.395	0.276
Bard	0.614	0.575	0.753	0.619	0.556	0.795	0.594	0.522
StarCoderbase-1b	0.738	0.778	0.752	0.771	0.767	0.819	0.735	0.712
StarCoderbase-7b	0.713	0.764	0.723	0.745	0.754	0.783	0.713	0.701
Phi- 2	0.213	0.158	0.604	0.211	0.152	0.661	0.200	0.143
Llama2-7b	0.653	0.697	0.669	0.669	0.671	0.726	0.633	0.623
CodeLlama-7b	0.752	0.792	0.761	0.777	0.780	0.816	0.750	0.737

Table 7: Result comparison for various fine-tuned models against out-of-the box large models for threshold = 0.4. # represents one-to-one bipartite matching, * represents many-to-one bipartite matching, + represents weighted many-to-one bipartite matching.

Model	ROUGE-1			ROUGE-2			ROUGE-L		
	F_1	P	R	F_1	P	R	F_1	P	R
ChatGPT	0.527	0.509	0.670	0.391	0.381	0.489	0.512	0.493	0.652
Bard	0.701	0.734	0.764	0.590	0.621	0.628	0.694	0.725	0.756
StarCoderbase-1b	0.813	0.804	0.903	0.721	0.713	0.805	0.811	0.801	0.899
StarCoderbase-7b	0.820	0.848	0.869	0.696	0.715	0.744	0.818	0.845	0.867
Phi-2	0.733	0.741	0.831	0.631	0.635	0.720	0.730	0.736	0.826
Llama2-7b	0.812	0.842	0.863	0.714	0.739	0.757	0.809	0.838	0.858
CodeLlama-7b	0.834	0.858	0.880	0.737	0.758	0.778	0.832	0.855	0.878

Table 8: Results for one-to-one matched pairs with threshold = 0.4.

Model	BERTScore		
	F_1	P	R
ChatGPT	0.906	0.901	0.913
Bard	0.923	0.919	0.927
StarCoderbase-1b	0.951	0.945	0.959
StarCoderbase-7b	0.946	0.944	0.949
Phi-2	0.940	0.937	0.944
Llama2-7b	0.946	0.943	0.950
CodeLlama-7b	0.948	0.947	0.950

Table 9: Results for one-to-one matched pairs with threshold = 0.4.

Model	ROUGE-1			ROUGE-2			ROUGE-L		
	F_1	P	R	F_1	P	R	F_1	P	R
ChatGPT	0.632	0.605	0.799	0.493	0.476	0.625	0.616	0.588	0.781
Bard	0.740	0.768	0.813	0.639	0.667	0.692	0.735	0.760	0.807
StarCoderbase-1b	0.810	0.796	0.909	0.724	0.710	0.823	0.808	0.794	0.906
StarCoderbase-7b	0.805	0.824	0.868	0.688	0.699	0.750	0.802	0.820	0.865
Phi-2	0.739	0.738	0.845	0.644	0.642	0.744	0.735	0.734	0.839
Llama2-7b	0.793	0.818	0.855	0.697	0.716	0.755	0.791	0.815	0.851
CodeLlama-7b	0.828	0.847	0.883	0.738	0.754	0.790	0.826	0.845	0.881

Table 10: Results for many-to-one matched pairs with threshold = 0.4.

Model	BERTScore		
	F_1	P	R
ChatGPT	0.921	0.912	0.930
Bard	0.925	0.922	0.930
StarCoderbase-1b	0.955	0.948	0.963
StarCoderbase-7b	0.947	0.946	0.950
Phi-2	0.947	0.943	0.952
Llama2-7b	0.946	0.943	0.951
CodeLlama-7b	0.953	0.952	0.955

Table 11: Results for many-to-one matched pairs with threshold = 0.4.

Model	$F_1^{\#}$	$P^{\#}$	$R^{\#}$	F_1^{*}	P^{*}	R^{*}	F_1^{+}	P^{+}
ChatGPT	0.398	0.290	0.783	0.392	0.280	0.806	0.380	0.269
Bard	0.553	0.520	0.672	0.562	0.514	0.702	0.547	0.492
StarCoderbase-1b	0.710	0.747	0.724	0.730	0.743	0.763	0.711	0.712
StarCoderbase-7b	0.682	0.731	0.689	0.702	0.726	0.724	0.685	0.697
Phi- 2	0.198	0.148	0.558	0.199	0.145	0.593	0.192	0.139
Llama2-7b	0.611	0.647	0.624	0.621	0.634	0.656	0.602	0.608
CodeLlama-7b	0.726	0.756	0.735	0.742	0.7506	0.769	0.726	0.723

Table 12: Result comparison for various fine-tuned models against out-of-the box large models for threshold = 0.5. # represents one-to-one bipartite matching, * represents many-to-one bipartite matching, + represents weighted many-to-one bipartite matching.

Model	ROUGE-1			ROUGE-2			ROUGE-L		
	F_1	P	R	F_1	P	R	F_1	P	R
ChatGPT	0.632	0.617	0.752	0.499	0.488	0.611	0.617	0.602	0.736
Bard	0.796	0.822	0.843	0.696	0.721	0.739	0.788	0.812	0.835
StarCoderbase-1b	0.866	0.858	0.943	0.796	0.790	0.876	0.864	0.855	0.941
StarCoderbase-7b	0.850	0.875	0.896	0.743	0.759	0.795	0.849	0.872	0.895
Phi-2	0.800	0.806	0.882	0.718	0.725	0.797	0.799	0.805	0.878
Llama2-7b	0.858	0.889	0.905	0.784	0.813	0.834	0.855	0.886	0.902
CodeLlama-7b	0.881	0.901	0.920	0.791	0.813	0.834	0.880	0.899	0.919

Table 13: Results for one-to-one matched pairs with threshold = 0.5.

Model	BERTScore		
	F_1	P	R
ChatGPT	0.920	0.914	0.928
Bard	0.937	0.934	0.941
StarCoderbase-1b	0.962	0.956	0.969
StarCoderbase-7b	0.954	0.953	0.956
Phi-2	0.956	0.953	0.959
Llama2-7b	0.954	0.953	0.956
CodeLlama-7b	0.959	0.959	0.961

Table 14: Results for one-to-one matched pairs with threshold = 0.5.

Model	ROUGE-1			ROUGE-2			ROUGE-L		
	F_1	P	R	F_1	P	R	F_1	P	R
ChatGPT	0.676	0.653	0.811	0.545	0.527	0.671	0.662	0.638	0.794
Bard	0.809	0.827	0.869	0.718	0.736	0.777	0.804	0.820	0.863
StarCoderbase-1b	0.841	0.829	0.929	0.770	0.758	0.859	0.840	0.826	0.925
StarCoderbase-7b	0.837	0.855	0.895	0.731	0.742	0.793	0.835	0.852	0.892
Phi-2	0.791	0.792	0.882	0.709	0.710	0.801	0.787	0.788	0.877
Llama2-7b	0.831	0.857	0.887	0.754	0.778	0.811	0.828	0.854	0.883
CodeLlama-7b	0.870	0.886	0.917	0.781	0.800	0.833	0.868	0.885	0.915

Table 15: Results for many-to-one matched pairs with threshold = 0.5.

Model	BERTScore		
	F_1	P	R
ChatGPT	0.928	0.919	0.937
Bard	0.938	0.936	0.942
StarCoderbase-1b	0.962	0.956	0.969
StarCoderbase-7b	0.954	0.953	0.955
Phi-2	0.958	0.956	0.962
Llama2-7b	0.953	0.951	0.956
CodeLlama-7b	0.953	0.951	0.956

Table 16: Results for many-to-one matched pairs with threshold = 0.5.

Model	Learning Rate	Learning Rate Scheduler	Batch Size	Step Size	Epochs
StarCoderbase-1b	5e-7	cosine	2	100	10
StarCoderbase-7b	5e-6	cosine	1	100	5
Phi-2	5e-7	cosine	1	100	10
Llama2-7b	5e-6	cosine	1	100	5
CodeLlama-7b	5e-5	cosine	1	100	5

Table 17: Hyperparamaters for the different fine-tuned models