University of Zurich[UZH]

Evaluating Crowdsourced Innovations Using Large Language Models

Konstantin Moser
Student ID: 17-706-169

Master Thesis
In fulfillment of the Master's Program
Master of Science UZH in Informatics
Artificial Intelligence

Responsible Professor
Prof. Dr. Martin Volk

Supervisor
Cyrille Grumbach

Submission of the thesis: March 20, 2025

Universität
Zürich[UZH]

Institut für Informatik

Declaration of independence for written work

I hereby declare that I have composed this work independently and without the use of any aids other than those declared (including generative AI such as ChatGPT). I am aware that I take full responsibility for the scientific character of the submitted text myself, even if AI aids were used and declared (after written confirmation by the supervising professor). All passages taken verbatim or in sense from published or unpublished writings are identified as such. The work has not yet been submitted in the same or similar form or in excerpts as part of another examination.

Zürich, K. Moser

Signature of student

Acknowledgements

First and foremost, I would like to express my special thanks to my supervisor, Cyrille Grumbach, PhD Candidate at ETH Zürich in the Chair of Strategic Management and Innovation. His guidance and insightful feedback made this thesis possible. Working with him was a truly pleasant experience, and I am deeply grateful for his patience and dedication throughout the past six months. His guidance allowed me to learn a great deal, as he provided valuable insights while giving me the freedom to explore and develop my own ideas.

I also want to express my gratitude to Prof. Dr. Martin Volk, Professor of Computational Linguistics at the University of Zurich for providing me with the opportunity to contribute to this research. Additionally, I would like to thank Pui Ying Wong, whose previous work laid the foundation for this thesis. Taking over her work provided a strong foundation and significantly supported my research process. Finally, I am deeply grateful to my supporting parents and my loving girlfriend for their encouragement throughout this journey. Their support has been invaluable throughout this process.

Abstract

Crowdsourcing contests generate large volumes of submissions, making efficient and fair evaluation challenging. This thesis explores how a combination of discriminative and generative artificial intelligence models can enhance solution filtering and prioritization. The discriminative model ranks submissions based on quantitative features while the generative model assesses novelty and usefulness through structured textual reasoning.

Building on prior research in AI-assisted evaluation for crowdsourcing contests, this work refines data preprocessing and refines the model's evaluation methodology. Results demonstrate that filtering out 50% of submissions retains approximately 95% of top-winning entries, confirming the discriminative model's effectiveness in removing weaker submissions while maintaining recall. At the same time, the generative model conducts a qualitative assessment of submissions, evaluating them from a different perspective than the discriminative model. A pattern emerges where the model tends to assign higher evaluations to winning submissions, indicating its ability to recognize innovation

Analyzing 112 contests, this thesis demonstrates that combining quantitative filtering with qualitative evaluation ensures that the great majority of well-documented and high-quality solutions remain in consideration. The findings also highlight the impact of domain-specific prompt adaptations in improving generative model accuracy. Future research could explore interactive evaluator chatbots and multi-modal analyses to further refine AI-driven assessments.

Contents

1. Introduction

Organizations frequently turn to crowdsourcing contests for innovation, delegating the problem-solving process to external contributors (Boudreau and Lakhani, 2013; Piezunka and Dahlander, 2019). Such contests offer a mechanism to engage contributors from diverse backgrounds who can provide innovative solutions that an organization might not have otherwise considered (Afuah and Tucci, 2012; Lifshitz-Assaf, 2018). Indeed, past research has shown that the larger and more diverse the contributing crowd, the more likely these contests are to yield highly innovative solutions (Terwiesch and Xu, 2008; Yang et al., 2009; Jeppesen and Lakhani, 2010). By integrating new knowledge incorporated into such innovative solutions, organizations stand to enhance their innovation capabilities.

Yet, extracting value from crowdsourcing contests can be challenging (Boudreau et al., 2011; Liu et al., 2014; Boudreau et al., 2016; Nagaraj and Piezunka, 2024). While involving more contributors increases the chance of surfacing innovative solutions, it also expands the pool of solutions of limited value. Even more importantly, as organizations confront larger volumes of solutions, they often resort to simplified filtering mechanisms (Piezunka and Dahlander, 2015). Under these conditions of information overload, there is a heightened risk of discarding innovative solutions due to evaluator fatigue or superficial screening processes (O'Reilly, 1980). Consequently, many innovative solutions may be filtered out before they ever receive meaningful consideration.

Ensuring that each innovative solution receives meticulous attention is vital for harnessing the full potential of crowdsourcing. However, the core challenge remains: ***How can organizations ensure sufficient attention to each innovative solution, preventing unwanted filtering?***

Recent advancements in artificial intelligence suggest that automated methods could help address this challenge (Wong, 2024). On one hand, **discriminative AI models** (e.g., classical machine learning classifiers) can filter out a substantial fraction of solutions based primarily on measurable features such as word count, code length, and number of media elements. On the other hand, **generative AI models**—particularly large language models —can provide text-focused assessments by evaluating a solution's *novelty and feasibility*. This thesis investigates how combining the strengths of both types of AI might enable organizations to effectively reduce the size of the solution pool while preserving highly innovative solutions.

The foundation for the present research builds on the work of a previous thesis (Wong, 2024), which established a baseline machine learning pipeline to classify winning vs. non-winning solution drawn from the crowdsourcing platform Hackster.io focused on hardware innovation. Hackster.io supports an array of crowdsourcing contests and attracts contributors with widely varying expertise. While that earlier thesis gathered and preprocessed extensive data from numerous Hackster.io contests, it left several evaluation challenges unresolved, including how to account for inconsistencies across contests of vastly different sizes and win percentages.

This prior thesis laid important groundwork by gathering and preprocessing data from Hackster.io,[1] and developing a classification model to predict winning solutions. However, the considerable variability among contests, ranging from small, specialized contests to large, open-ended contests, makes it difficult to apply a single contest-specific filtering threshold effectively. To address this, the present research implements a methodology that applies uniform filtering percentages across all contests, ensuring that each contest is weighted equally rather than being influenced by its inherent win rate. While this approach may lead to variations in accuracy across contests, it reduces biases introduced by imbalanced contest structures. As a result, **recall**—the proportion of winning solutions that survive the automated filtering—becomes the primary performance metric. If organizations seek to preserve innovative solutions above all else, maintaining high recall is essential.

Rather than debating whether to use discriminative or generative AI models in evaluations process, as past research did (Just et al., 2024; Bell et al., 2023; Doshi et al., 2025), I propose that aggregating both types of AI models is the most effective approach. I hypothesize that doing so enables organizations to filter a larger volumes of solutions while still preserving those deemed innovative, than using either one type of AI model in isolation. To ensure fair comparisons between these two types of AI in subsequent evaluations, a consistent textual format is employed, preventing any unintentional bias in format, language valence, or length.

Importantly, the findings of this thesis support this hypothesis. Analyzing 112 Hackster.io contests, the results show that filtering out 50% of submissions retains 95% of top-winning entries, demonstrating the discriminative model's effectiveness in removing lower-quality submissions while preserving high-quality ones. Classification distributions confirm that both models align with contest outcomes while capturing distinct aspects of submission quality. The discriminative model prioritizes documentation quantity, while the generative model evaluates novelty and usefulness. Their independent classifications highlight their complementary strengths, helping to ensure that promising innovations are not overlooked. Additionally, few-shot prompting and contest-specific adaptations enhanced the generative model's reasoning, making its assessments more precise and contextually aware.

[1]Hackster.io hosts numerous crowdsourcing contests, each with distinct requirements, prize structures, and contributor bases, leading to significant variability in both the quantity and quality of solutions.

The following chapters outline the methodological framework underlying the two AI models. First, the filtering process is explained and evaluated, demonstrating that the discriminative model can effectively prefilter submissions based on quantitative features while retaining most top entries. Then, using the discriminative model output and the LLM's assessment of novelty and usefulness, each submission is classified into one of three categories. These classification distributions are analyzed and discussed. Based on the discriminative model's ranking and a comparison of each submission with others in the same contest, a textual evaluation of documentation quantity is created. In a similar format, the LLM produces structured reasoning to justify its classification based on novelty and usefulness. Sample outputs from both models are presented and compared, illustrating how their complementary approaches provide a more comprehensive evaluation process. Finally, broader implications for innovation management and AI-assisted evaluation in diverse organizational contexts are discussed.

2. Literature Review

2.1. Crowdsourcing

Since the term was introduced by Jeff Howe in 2006, ”crowdsourcing” or delegating the problem-solving process to contributors residing outside organizational boundaries has cemented itself as an effective strategy to source innovative solutions (Jeppesen and Lakhani, 2010; Terwiesch and Xu, 2008; Howe, 2006). Indeed, crowdsourcing allows the organizations to tap in new knowledge—knowledge that is familiar to the external contributors but distant from the organizations' own knowledge stock (Afuah and Tucci, 2012). Those contributors can leverage their knowledge to develop innovative solutions that would have remained unimagined by the organizations (Park et al., 2024; Piezunka and Dahlander, 2015, 2019). Furthermore, organizations often organize crowdsourcing contests, whereby they engage with a wide array of external contributors, while rewarding only those whose solutions are deemed innovative enough (Boudreau et al., 2011).

Interestingly, crowdsourcing is not a novel phenomenon. Napoleon, for instance, leveraged crowdsourcing to find innovative solutions to the problems of food preservation—crucial when the battles were geographically distant. In 1795, Napoleon launched a crowdsourcing contest, open to all French citizens, and offered large cash prize to anyone who could develop a solution for preserving food over long periods. Many contributors, coming from all around France and from all layers of society, participated. At the end, it was Nicolas Appert, a confectioner and brewer, who came up with an innovative solution: the process of airtight food preservation, which later evolved into modern canning. Thus, the everyday food canning found in supermarkets today originated from a crowdsourcing contest. Other examples of innovative solutions that originated from past crowdsourcing contests include the development of the marine chronometer, the design of the Brooklyn Bridge or the Statue of Liberty in New York, the invention of margarine, or the development of Lindbergh's transatlantic flight route. Thus, over history, crowdsourcing has repeatedly proven to be a powerful driver of innovation, generating innovative solutions that might not have emerged within the boundaries of organizations (Piezunka and Dahlander, 2015; Howe, 2006).

Nowadays, crowdsourcing primarily involves delegating problem-solving through online platforms, targeting contributors worldwide. It has become a ubiquitous practice and has rapidly grew into a billion-dollar industry. Crowdsourcing is now leveraged by organizations for video editing (Chen and Liu, 2012), jewerley design (Füller et al., 2014), ad design (Ye and Jensen, 2022), logo design (Chen et al., 2021), naming new products (Koh, 2019), web development and design (Boudreau et al., 2011) scientific problems (Jeppesen and Lakhani, 2010), or even hardware development (Ghaleb et al., 2022).

Yet, while crowdsourcing holds promises, it faces one major bottleneck: evaluation. Organizations need to invest significant resources to evaluate all the solutions proposed by external contributors. While involving more contributors with diverse expertise increases the likelihood of generating innovative solutions, it also tends to reduce the incentives to exert efforts, leading to an increased proportion of solutions of limited value (Boudreau et al., 2011; Liu et al., 2014; Nagaraj and Piezunka, 2024). Concurrently, as organizations become overloaded by the sheer volume of solutions, they resort to increasingly simplified filtering mechanisms. Under these conditions of information overload, the likelihood of inadvertently filtering out innovative solutions increases (Piezunka and Dahlander, 2015; He et al., 2024). Consequently, organizations find themselves evaluating a vast solution pool, of which a considerable portion is of limited value, while inadvertently filtering out some that are innovative. While it is crucial for organizations to give meticulous attention to the evaluation of each innovative solution to ensure alignment with their needs, it remains unclear how they can be enabled to do so.

2.2. Artificial Intelligence: Discriminative and Generative

AI is rapidly emerging as a transformative research domain with far-reaching societal implications. Its role in modern organizations has evolved from basic rule-based systems to sophisticated technologies capable of recognizing patterns, making predictions, and processing language (Bell et al., 2023). By simulating human cognitive abilities, AI aims to perform tasks traditionally requiring human intelligence (Enholm et al., 2022). This includes the capacity to interact, learn from experience, adapt, and manage uncertainty (Legg and Hutter, 2007).

The early development of AI can be traced back to foundational work by Alan Turing on machine intelligence and von Neumann's contributions to computing architectures (McCorduck and Cfe, 2004). In its initial phase, AI primarily relied on rule-based algorithms, where decision-making followed predefined logical rules. This approach, known as symbolic AI, leveraged expert systems to structure and utilize existing knowledge bases (Shu-Hsien Liao, 2005). Despite early optimism, Herbert Simon's 1965 prediction that "we will soon have the technological means (...) to automate all managerial decisions" (Simon, 1965, p. 47) did not immediately materialize, leading to a decline in research interest and the onset of an "AI winter" (Csaszar and Steinberger, 2022).

In the last decades, AI has shifted towards subsymbolic approaches, such as machine learning, which allows systems to learn directly from data rather than relying on explicitly programmed rules (Marra et al., 2024). This shift marks a departure from traditional feature engineering towards deep learning models, significantly altering machine learning paradigms (Brynjolfsson and McAfee, 2014). Unlike earlier methods that required domain experts to manually define features, deep neural networks can autonomously extract intricate patterns from raw data. These advancements have propelled AI's success in fields such as image recognition, language translation, and strategic gameplay in Chess and Go (Agrawal et al., 2019; Brynjolfsson and McAfee, 2014).

In the last few years, AI has attracted even more attention with the rise of **Generative AI**, constrasting it with all previous approaches which are now classified as **Discriminative AI**. The paper "Attention is all you need" by Vaswani et al. (2017) triggered the latest technological breakthrough by proposing a new deep learning architecture known as the transformer-based model, which further revolutionized the field of AI and, more specifically, natural language processing (NLP). The enhanced ability facilitates pre-training on an unparalleled volume of textual data extracted from large knowledge databases, such as Wikipedia, Reddit, or Google News. Rather than associating each term within a vocabulary with a singular word embedding, transformer-based embeddings take into account the contextual surroundings of each instance of a word (Brown et al., 2020). These models have gained much attention for their text-generation capability, enabling new research and practice opportunities (Bouschery et al., 2023). This significant evolution of machine intelligence has reignited management scholars' interest within the last decades, elevating it to the "crux of the management debate" (Raisch and Krakowski, 2021, p. 193).

Recent advancements have enabled AI to take on decision-making tasks that were once the exclusive domain of humans. As AI's capabilities expand, it fosters more integrated approaches where human judgment and AI function together. This has led to the emergence of human-AI collaboration (HAI) as a dynamic research field (Weiser and von Krogh, 2023; Raisch and Krakowski, 2021; Murray et al., 2021). The core focus is on determining when and how AI can enhance decision-making in organizations, such as in evaluation processes. A key assumption in this literature is that AI differs from traditional technologies by possessing the ability to learn and act autonomously (Murray et al., 2021). This positions AI not just as a tool but as a counterpart in collaborative work systems, shaping organizations in novel ways (Anthony et al., 2023; McCorduck and Cfe, 2004). Consequently, research has increasingly explored the various forms of collaboration between humans and AI, seeking to understand the dynamics and implications of this evolving partnership.

2.3. Artificial Intelligence and Crowdsourcing Evaluations

Given the promises of AI, and the evaluation bottleneck arising in crowdsourcing, scholars have began to ponder whether AI could be integrated in the evaluation process. Interestingly, a dichotomy emerges with some scholars advocating for leveraging discriminative AI models, and other proposing to use generative AI models.

First, some scholars use discriminative AI models in their methodology. In their seminal paper, Bell et al. (2023) examined how AI could enhance the efficiency of idea screening in crowdsourcing contests, providing a more efficient approach than traditional methods relying solely on human evaluators. The authors built a simple model using LASSO that could efficiently screen out ideas considered ”bad” by human evaluators. Yet their model was also simultaneously filtered out many innovative solutions, showcasing the limitations of their paper. Thus, while promising, the paper was unable to solve the problem of filtering out solely solutions of limited value, while retaining those deemed innovative. Another example is the paper of Just et al. (2024) which employed three text embeddings—Doc2Vec, SBERT, and GPT-3-based Ada Similarity—and calculated semantic distances using different novelty detection algorithms, comparing the results with human novelty assessments. The findings indicated that SBERT-based novelty scores most closely aligned with human evaluations. Yet, similarly to the paper from Bell et al. (2023), many innovative solutions were filtered out in the process.

Conversely, other scholars leverage generative AI models in their methodology. For instance, Csaszar et al. (2024) explore how AI can enhance the evaluation of strategies. They provide empirical evidence from an accelerator program and start-up competition, showing that current large language models can evaluate strategies at a level similar to that of entrepreneurs and investors. However, although AI-based evalutions were similar, they were not identical with those of humans, and thus, the core issue persisted: many innovative solutions continued to be filtered out by AI. To overcome such issue, Doshi et al. (2025), found that combining the evaluations of multiple generative AI models reduced—but unfortunately, not signficantly—the proportion of innovative solutions being filtered out by AI.

Thus, the problem persists: leveraging AI to pre-screen solutions leads to solutions deemed innovative by human evaluators being filtered out. In this thesis, I propose that instead of debating whether to use discriminative AI models or generative AI models for evaluation processes, both should be leveraged. Thus, I propose that combining the strengths of both types of AI may be most effective at filtering out a large portion of solutions while retaining those deemed innovative. On the one hand, discriminative AI models could evaluate solutions based on ”objective” criteria, such as word length, effectively filtering out those that fall below a certain effort threshold. This argument assumes that innovation is the result of consistent effort, implying that solutions lacking effort cannot be considered innovative (Nelson and Winter, 1982; Schumpeter, 1931). On the other hand, generative AI models could evaluate on more ”subjective” criteria such as the novelty or the feasibility of solutions. Those evaluations could be aggregated to determine which solutions should ultimately be filtered out, enabling humans to focus their attention on the subset of solutions deemed innovative. Thus, I hypothesize:

Hypothesis: ***Combining discriminative and generative AI models enables a more balanced evaluation of submissions by leveraging quantitative assessments alongside qualitative reasoning, providing a more comprehensive approach than using either model alone.***

3. Methodology

This methodology is a revised and updated version of the thesis *Evaluating Innovation with AI* by Pui Ying Wong (Wong, 2024). All changes made to the prior methodology are indicated and explained through footnotes.

3.1. Data Collection

The data collection process involved crawling the Hackster.io platform using a headless Chromium browser. The crawling process was automated using the `DrissionPage` library, which allows interaction with dynamically loaded web content. The process was conducted in two main phases: extracting contest-level information and collecting detailed data on individual solutions. Significant upgrades to the methodology improved the scope and accuracy of the data gathered[1].

3.1.1. Crawling Method

The crawling script was designed to minimize server load and avoid triggering rate limits by incorporating deliberate delays (`sleep` calls) between requests. These delays ensured that all content, including dynamically loaded elements, was fully rendered before data extraction began[2]. The entire data collection process required approximately 40 hours to complete.

Contest Data

The first phase of crawling focused on contest-level data. The script navigated to the contest overview pages on Hackster.io and extracted the following metrics:

- Total number of solutions
- Total number of contributors

[1] New features such as contest start dates, detailed prize categories, and solution-level metrics were added.
[2] The previous approach included fewer delays, potentially missing dynamically loaded content.

- Contest description
- Contest winners along with prize information[3]
- Contest start dates[4]

Solution Data

In the second phase, the script navigated to individual solution pages for each contest. Detailed information about each solution was extracted, including:

- The main story text of the solution
- The number of multimedia elements, such as images, GIFs, and videos
- The total number of lines of code provided directly in the solution[5]
- The presence of external code repository links (e.g., GitHub or GitLab)[6]
- The number of components listed
- The number of CAD files and schematics provided[7]
- The total duration of all videos[8]

To determine video durations, the script used the YouTube and Vimeo APIs. If the APIs failed or videos were private, the duration was estimated using the average video length across all videos in the dataset to maintain consistency.

3.2. Data Cleaning

The data cleaning process refined the dataset by systematically addressing contests and solutions that did not meet quality criteria or introduced biases. This process ensures that the final dataset is reliable further analysis.

[3]Before, the specific prize won by each solution was not recorded, making it impossible to differentiate significant prizes from less meaningful ones in a later step.
[4]The prior methodology did not include the contests start dates, which are crucial for the data cleaning process.
[5]The previous methodology did not collect the number of code lines.
[6]External repositories were not considered before.
[7]CAD files and schematics were not explicitly counted before.
[8]Video durations were not measured or aggregated in the prior methodology.

3.2.1. Initial Dataset

The initial crawling process identified a total of 8,044 solutions across all contests. However, 183 solutions were either deleted by the administrator or set to private and could not be crawled. Consequently, data was successfully collected for the remaining 7,861 solutions.

3.2.2. Contest Filtering

Contests were filtered into three categories based on specific exclusion criteria. This step removed contests that lacked sufficient data, exhibited irregularities, or provided ambiguous outcomes.

Small Contests

Contests with fewer than 10 solutions were removed, as they lacked sufficient data for meaningful analysis. The details are summarized in Table 3.1.

Table 3.1.: Contests with fewer than 10 solutions

Contest Name	Number of solutions
Start-Your-Hardware-Startup	0
leapmotion-3d-jam-head-start	0
imp-halloween	3
Avnet	4
electroniclifeguard	5
wunderbar-holiday	5
what-will-udoo	6
ASU	6
automatizacion-del-hogar-st-micro	8
thundersoftai	8

A total of 45 solutions from 10 contests were removed, leaving 7,816 solutions remaining.

Contests Not Focused on Innovation

Contests were excluded if they exhibited irregularities, were considered "weird," or did not align with the focus on innovation. Specifically, contests with disproportionately high win rates (above

40%), suggesting low-quality evaluation or irregular prize assignments, were removed[9]. The details are summarized in Table 3.2.

Table 3.2.: Contests Not Focused on Innovation

Contest Name	Number of solutions
littlefreestemlibrary	11
sigfoxuniversities	25
arm2018	28
aarpmenopause	30
NotImpossibleAwards	40
make-halloween-2016-contest	97

A total of 231 solutions from 6 contests were removed, leaving 7,585 solutions remaining.

Contests Without Clear Winners

Contests lacking a clear order of winning solutions were removed, as they often awarded prizes across multiple categories, some of which emphasized innovation more than others, making it difficult to determine the top winners[10]. The details are summarized in Table 3.3.

Table 3.3.: Contests Without Clear Winners

Contest Name	Number of solutions
smartedgeagile	15
theta	20
sustainablefashion	29
infineon-coolmos	29
particle-iot	35
AzureEnterpriseIoT	41
Infineon3D	44
hologram	61
rapid-iot	80
LightsforLife	133

A total of 487 solutions from 10 contests were removed, leaving 7,098 solutions remaining.

[9]The prior methodology removed all contests with win rates exceeding 50%, without distinguishing between prize categories.

[10]The prior methodology did not exclude contests with no clear winners.

3.2.3. Solution Filtering

Solutions with Missing or Invalid Data

Solutions were then manually inspected to identify issues such as incorrect formats, which could prevent the crawler from accurately extracting relevant information. Common issues included:

- Links to external websites instead of direct content on Hackster.io
- Missing or improperly uploaded code
- Documentation provided only on external platforms like GitHub
- Text content uploaded as images
- Documentation uploaded as PDFs
- Deleted content

This manual review identified and removed 27 problematic solutions[11], leaving 7,071 entries.

Duplicate solutions

Solutions appearing in multiple contests were identified and handled. It is common for users to upload a project to multiple contests after initially submitting it to a relevant one, often in an attempt to win additional prize money. To address this, only the earliest relevant entry was retained, ensuring that the solution was originally created for that contest and reducing biases caused by repeated solutions[12].

The number of duplicates removed per contest is summarized in Table 3.4.

A total of 555 duplicates were identified and removed across 80 contests, reducing the dataset to 6,516 solutions.

3.2.4. Data Cleaning Summary

The data cleaning process is summarized below:

- **Contest Filtering:** 763 solutions from 26 contests removed

[11] The earlier methodology did not manually inspect solutions for missing or invalid content.
[12] Duplicate solutions were not removed in the previous methodology, potentially introducing significant biases.

Table 3.4.: Duplicates Removed Per Contest

Contest Name	Number of Duplicates Removed
2018chinausyoungmakercompetition	51
Tinkernut	38
2019chinausyoungmakercompetition	36
2020youngmakercompetition	28
SeeedEarthDay	28
UKAmazonAlexa	28
2021chinausyoungmakercompetition	27
touchlessdomore	19
2022chinausyoungmakercompetition	16
2017chinausyoungmakercompetition	12
iotinthewild	12
particle	12
aarp	11
alexa-api-contest	11
alexa-reinvent	11
chinausyoungmakercompetition	11
ESP8266	10
alexa-raspberry-pi	10
Other Contests	62 contests with fewer than 10 duplicates, totaling 184

- **Solution Filtering:** 27 solutions removed
- **Duplicate solutions:** 555 solutions removed

Total: 1,345 solutions

Final Dataset:

- **Solutions:** Reduced from 7,861 to 6,516
- **Contests:** Reduced from 139 to 112

3.2.5. Data Preprocessing

Before the data could be used for training, several preprocessing steps were applied to ensure consistency and suitability for machine learning. These included:

- Extracting contest-level and solution-level features from JSON files and combining them into a single structured CSV file, ensuring each solution record includes both solution-specific and contest-specific features
- Converting lists (e.g., videos) into numerical representations by calculating their lengths
- Encoding binary labels for categorical features such as `link`, `code`, and `winner`
- Normalizing non-binary numerical data, such as the total number of media, word count, and video duration, to a range of 0–1 using min-max scaling
- Cleaning text fields (contest descriptions, solution names, and stories) by:
  - Removing unusual characters and symbols
  - Replacing multiple newlines with spaces
  - Trimming leading and trailing whitespace
- Updating the total number of solutions per contest and the win percentage to reflect the cleaned dataset

3.3. Features and Labels

This section outlines the target variables and features derived from the dataset, which are used to train the machine learning model. It also details the prize categorization process, ensuring that the model focuses on solutions representing high-quality innovation.

3.3.1. Detailed Feature Descriptions

Each feature captures specific aspects of solution quality and content. These features are used to train machine learning models, which predict the probability of a solution winning a top prize. These are summarized as follows:

- **solution_word_count:** Total number of words in the main story text, indicating the level of detail
- **num_image:** Number of images included, reflecting visual documentation
- **num_gif:** Number of GIFs, demonstrating project animations or demonstrations
- **num_video:** Number of videos included, providing tutorials or project explanations

- **video_duration:** Combined duration of all videos (in seconds), reflecting the depth of video content
- **num_things:** Number of components used in the project, indicating complexity
- **cad:** Number of CAD files included, representing detailed 3D models
- **schematic:** Number of schematics, detailing the electrical circuitry of the project
- **code:** Binary label indicating whether code is included (**0** for no, **1** for yes)
- **code_lines:** Number of lines of code, quantifying the software component
- **link:** Binary label indicating the presence of external code repositories (**0** for no, **1** for yes)

3.3.2. Target Variable

The dataset includes a primary target variable: **winner**

The target variable determines whether a solution is considered a top prize winner:

- **0**: The solution did not win any major prize
- **1**: The solution was a top prize winner, considered one of the "best" in the contest

This label is derived from contest results on Hackster.io and serves as the key metric for model training[13].

3.3.3. Prize Categorization

To streamline the dataset, prizes were categorized into two main groups: **Top Prizes** and **Other Prizes**. Only solutions receiving **Top Prizes** are labeled as winners. This ensures that the model focuses on high-quality, innovative solutions[14].

Top Prizes

These reflect significant achievements and include:

[13] A secondary label, winner_categories, distinguishes between top-tier and medium-tier prizes. While it is not used in the current analysis, it could be explored in future work for multi-class classification approaches.

[14] In the previous methodology, all solutions winning any prize were considered winners, including even irrelevant ones such as early solution prizes.

- Grand Prize
- First Place
- Second and Third Places (in some cases)

Other Prizes

These prizes, while notable, are excluded from the winner category:

- **Medium Quality Prizes:** Finalists, Runner Ups, Top 25 Projects, Popular Vote, Fourth Places and lower
- **Thematic and Special Focus Awards:** Most Practical, AI Social Impact Award, Impact Prize Bonus, Hackster Impact Prize, The Hackster Impact Detect Award, Editor's Choice
- **Recognition and Participation Awards:** Honorable Mention, 10 Projects of Merit, Judges Choice, Judge's Freestyle Favorite (fun, silly, unexpected, nice try, or other), Most Fun Social Media Video, Crazy Popcorn Time
- **Early and Irrelevant Prizes:** Early solution Prizes, Early Birds, Special Delegate, Bonus Prizes, Lucky Draws

3.4. Discriminative Model

The evaluation of the discriminative models was designed to address biases and limitations identified in the prior methodology. Unlike the previous approach, which relied on contest-level winning percentages and evaluated all solutions together, this evaluation calculates recall contest-wise, averages the results, and incorporates multiple models with robust cross-validation while addressing class imbalance and enabling manual inspection.

3.4.1. Evaluation Process Overview

The evaluation leverages a robust **5-Fold Cross-Validation** framework ($k = 5$) with **multiple runs** to ensure comprehensive performance estimates. Each run employs distinct training, validation, and test splits, guaranteeing that every contest serves as a test set exactly once. The setup is as follows:

- The dataset is divided into **5 folds**:

  - Each fold contains a **test set** (20% of the data), which is never seen during training.
  - The remaining **train/validation set** (80% of the data) is further split into:
    * **Train Set:** 80% of the train/validation set ($\sim 64\%$ of total data)
    * **Validation Set:** 20% of the train/validation set ($\sim 16\%$ of total data)
- The process is repeated for **multiple independent runs**, where the train, validation, and test splits are randomized in each run to minimize bias caused by data partitioning.

By averaging results over multiple runs and folds, the evaluation reduces variability and mitigates bias introduced by random splits.

Data Splitting and Class Imbalance Handling

To ensure balanced training, **SMOTE** (Synthetic Minority Over-sampling Technique) (Chawla et al., 2002) is applied to mitigate class imbalance. This step reduces bias in models that might otherwise favor majority classes, providing a fairer basis for performance assessment. After splitting into train, validation, and test sets:

- The **Train Set** is oversampled using SMOTE, preserving the validation and test sets as representative of real-world distributions.
- The **Validation Set** is used for model monitoring and tuning.
- The **Test Set** remains unseen, ensuring unbiased performance estimates.

Multiple Models Evaluated

In addition to a **Multi-Layer Perceptron (MLP) (Haykin, 1998)**, which serves as a deep learning-based baseline, several classical machine learning models are trained and evaluated under identical data splits and evaluation criteria:

- **Random Forest (RF) (Ho, 1995)**
- **XGBoost (XGB) (Chen and Guestrin, 2016)**
- **Support Vector Machine (SVM) (Cortes and Vapnik, 1995)**
- **Logistic Regression (LR) (Gauss and Davis, 1857)**

This multi-model approach enables a more comprehensive understanding of performance, benchmarking neural network-based methods against traditional ML algorithms. Each model is trained for the same number of epochs (where applicable) or until convergence, ensuring comparability of results.

3.4.2. Improved Evaluation Criteria

Instead of selecting winners based on contest-level winning percentages, the evaluation ranks all solutions by their model-predicted quality scores. By evaluating recall at various filtering thresholds, this method reduces biases introduced by contest-specific win rates.

Filtering and Recall Calculation

For each test contest:

1. Predictions are generated by the chosen model.
2. Solutions are **ranked** according to their predicted scores.
3. A **filtering percentage** (e.g., from 10% to 70%) determines the subset of top-ranked solutions retained.
4. **Recall** is computed as the fraction of actual winners present in the selected subset.

By repeating this procedure across multiple models and threshold levels, the evaluation clarifies model strengths and weaknesses in different filtering scenarios.

False Negative Analysis and Quality Scores

To identify areas for improvement:

- **False negatives** (missed actual winners) are tracked and exported for further analysis. Inspecting these solutions helps highlight patterns where models consistently fail.
- Each solution is assigned a **quality score**, facilitating finer-grained ranking and enabling the analysis of model reliability in identifying high-quality solutions.

Bias Reduction

The systematic **cross-validation** setup ensures:

- Each contest is tested exactly once per run.
- Multiple runs with different random seeds reduce dependency on a single data partition.
- SMOTE balancing addresses class imbalance, fostering models that learn from a more representative sample of winners and non-winners.

3.4.3. Discriminative Evaluator

The `DiscriminativeEvaluator` is a custom class developed to transform the raw numerical output of the discriminative model into a structured textual assessment. Instead of presenting a single numerical score, this class generates a human-readable evaluation, making it more interpretable and comparable to the generative model's qualitative outputs.

This process ensures that each submission's ranking and quantitative attributes, such as text length, media usage, and documentation completeness, are communicated clearly. By converting numerical data into textual insights, the `DiscriminativeEvaluator` enhances the explainability of the model's predictions.

Evaluation Process

The discriminative evaluator operates in the following steps:

1. **Extract Precomputed Scores:** The discriminative model retrieves quality scores, already computed by the machine learning models, ranging from 0 to 1 for each submission.
2. **Compare Within Contest:** The submission's score and quantitative attributes (e.g., word count, media usage) are compared to other submissions in the same contest.
3. **Generate Structured Output:** A textual evaluation is produced, summarizing the submission's strengths and weaknesses in two distinct categories:
   - **Description and Bills of Materials:** Evaluates the length and detail of the textual description, as well as the comprehensiveness of the listed components.
   - **Visuals, Code, and Other Documentation:** Assesses the use of media, code availability, and the overall completeness of supporting documentation.

4. **Assign Overall Classification:** Based on its relative ranking, the submission is classified as:
    - **Excellent:** Among the best in the contest based on quantitative metrics.
    - **Average:** Comparable to typical entries.
    - **Poor:** Falls below other submissions.

3.5. Generative Model

This section outlines how Large Language Models are employed to perform the *qualitative* assessment of contest submissions—specifically focusing on novelty and usefulness. While the discriminative model assesses quantitative attributes, the generative model independently evaluates each submission's quality. By structuring prompts and supplying relevant examples, the framework evaluates submissions without bias from the discriminative output.

3.5.1. Prompt Engineering and Iterative Refinements

Developing an effective prompt was an iterative process with many trials. Here are the key takeaways:

- **Pass or Fail:** Initially, the prompt was lengthy and less specific. It asked the model to decide if a submission was "pass" or "fail," often resulting in over-lenient evaluations (many "pass" classifications).
- **Preliminary Screening:** An earlier prompt included a preliminary screening step to filter submissions based on completeness, alignment with contest objectives. The model was asked to assess whether a submission provided a clear and intelligible description, aligned with the contest's goals, and was at least in the prototype stage. However, this step was removed as it introduced unnecessary complexity and occasionally confused the model, leading to inconsistent evaluations.
- **Four Categories:** An attempt to classify submissions into four groups (e.g., "Top 10%," "Top 25%," etc.) caused confusion and inconsistent distinctions. This approach was replaced by three clearer classes: *Excellent*, *Average*, and *Poor*.
- **Confidence Scores:** The prompt once instructed the model to produce a confidence score (0–100%). Observations showed these scores were often random and lacked meaningful correlation, so this feature was removed.

- **Direct Classification vs. Chain-of-Thought:** Initially, the LLM was prompted to classify submissions directly before providing any reasoning. However, it became evident that having the model first reason step-by-step and then classify (Chain-of-Thought) led to more consistent outputs. Refining the prompt to prioritize reasoning before classification improved response accuracy and reduced format inconsistencies.
- **Consistency in Wording:** The prompt was standardized using the same terms (e.g., "evaluate," "solution," "novelty," "usability") throughout, avoiding synonym clutter that sometimes confused the LLM.
- **Structured Prompt Sections:** To ensure clarity and prevent the LLM from misinterpreting different components of the prompt, section delimiters such as `--- BEGIN FEW-SHOT EXAMPLES ---` and `--- END FEW-SHOT EXAMPLES ---` were introduced. This structuring improved adherence to instructions and reduced instances where the model mistakenly evaluated few-shot examples as actual submissions.
- **Encouraging Critical Evaluations:** Various strategies were tested to make the LLM more critical in its assessments. This included explicitly instructing it to "be more critical" and using constraints like "*If a solution does not clearly surpass existing alternatives, it should not be rated as 'Excellent.*" Additionally, the few-shot examples were adjusted by selecting slightly better solutions for the **Poor** and **Average** categories, helping the model calibrate its classifications more conservatively.
- **Prompt Length Management:** Certain earlier versions of the prompt provided extensive instructions and large examples, which overwhelmed smaller LLMs. A more concise and direct style improved performance.

Below is an **example** of an earlier, more extensive prompt that eventually proved too long:

```
You are an expert evaluator for technical contests. Your task is to assess a
     submission based on the following:

**Instructions:**

1. **Preliminary Screening:**
   - Does the submission provide a complete, appropriate, and intelligible
      description of the solution? (Yes/No)
   - Does the solution align with the contest's primary objectives and
      problem statement? (Yes/No)
   - Is the solution at least in the prototype stage? (Yes/No)

   If any answer is "No," provide reasoning and stop the evaluation. If all
      answers are "Yes," proceed to step 2.
```

```
2. **Detailed Evaluation:**
   - Provide reasoning on whether the submission effectively addresses the
     contest problem, noting both strengths and weaknesses.
   - Assess the innovation in the submission with specific examples.

3. **Classify the Submission:**
   - Classify the submission into one of the following categories based on
     its overall quality and alignment with the contest objectives:
     - **Top 10% (Outstanding):** Should be a winner.
     - **Top 25% (Excellent):** Strong submissions but may not be winners.
     - **Top 50% (Good):** Decent quality but lacking in innovation or
       impact.
     - **Bottom 50% (Average to Poor):** Do not meet the criteria for
       winning.

4. **Recommendation and Confidence:**
   - Based on the classification, recommend whether the submission should be
      a winner, including a brief justification.
   - Assign a confidence score (0-100%) for your evaluation.
```

Though detailed, such prompts often resulted in inconsistent or unhelpfully verbose outputs. By contrast, a concise prompt with three clear categories led to more reliable classifications.

3.5.2. Rationale for Separate Models

The framework applies two distinct models—one *generative*, the other *discriminative*—to evaluate contest submissions. The discriminative model focuses on **quantitative** attributes (e.g., length of text, number of images, video durations), whereas the generative model focuses on **qualitative** aspects (e.g., novelty, usability). Initially, there was consideration to integrate the discriminative model's output (such as numerical quality scores or submission rank) into the generative model's prompt. An example prompt snippet was:

```
This submission ranked {rank}th out of {num_submissions} submissions
in this contest, with a quality score of {quality_score:.3f} (average:
{avg_quality_score:.3f}, standard deviation: {sd_quality_score:.3f}).
The quality score was generated by a discriminative machine learning
model that evaluates the *quantity* of content...
```

However, providing this information sometimes biased the generative model's qualitative assessment and caused it to focus on quantitative signals rather than purely qualitative criteria. Therefore, the framework ultimately opted for a *sequential* approach, running each model independently so that both outputs reflect each model's strengths without cross-influence. Despite exploring the integrated approach in early tests, having completely separate evaluations yielded more transparent and interpretable results.

3.5.3. Few-Shot Prompting Implementation

The framework's generative evaluations rely on few-shot prompting, where a small set of exemplary submissions is included in the prompt. These examples help the large language model understand what constitutes *Excellent*, *Average*, or *Poor* solutions.

Selection of Few-Shot Examples

Three few-shot examples are used to cover each classification category. Manually chosen references ensure each category is well-represented. Links to the chosen few-shot submissions and their expected outputs must be specified before the evaluation. The framework then automatically inserts these examples into the prompt. Early experiments showed that:

- Fewer than three examples tended to bias the LLM toward the categories it saw explicitly.
- More than three examples often increased the prompt length and sometimes overloaded the LLM, reducing clarity.

3.5.4. Final Prompt Structure

The final prompt instructs the model to generate exactly two concise paragraphs focusing on two criteria: *novelty* and *usefulness*. It also mandates an overall recommendation from three categories (*Excellent*, *Average*, *Poor*) and specifies that approximately 80% of all evaluations should be *Average* or *Poor*, thereby reducing overly generous ratings. Three few-shot examples, each illustrating a distinct category, give the model clear anchors for classification. The prompt concludes with `<eot_id>`, ensuring a consistent termination that simplifies result extraction. Below is the complete prompt:

```
You are an expert evaluator for technical contests. Your task is to assess a
     submission based on the following:

--- BEGIN INSTRUCTIONS ---
1. Provide a structured evaluation consisting of two concise paragraphs,
    each addressing one of the following criteria in a few sentences:
  - *Novelty of the Solution*: Evaluate how novel the solution is. Search
      for similar, existing solutions, and evaluate how different and unique
```

```
     this solution is compared to those existing solutions. Identify any
     concept, feature, technology or approach that might be novel.
  - *Usefulness of the Solution*: Evaluate how useful the solution is.
     Consider factors such as practicality, usability, and relevance.
     Identify potential challenges that might hinder its real-world value.
2. Choose one of the following overall recommendations. Be critical in your
    evaluations. If a solution does not clearly surpass existing
    alternatives, it should not be rated as 'Excellent.' Carefully consider
    any limitation before rating a solution as even 'Average.' About 80% of
    the solutions should be rated as 'Average' or 'Poor.'
- Excellent: The solution demonstrates both substantial novelty and
    usefulness, far exceeding typical expectations.
- Average: The solution demonstrates a reasonable degree of novelty and
    usefulness, meeting typical expectations without exceeding them.
- Poor: The solution is only moderately novel or useful, and is thus
    unlikely to meet typical expectations.
--- END INSTRUCTIONS ---

--- BEGIN FEW-SHOT EXAMPLES ---
Below are evaluation examples that illustrate how submissions from other
    contests are evaluated according to the provided instructions and
    criteria.

**Example 1: {fs[0]['class']} Submission**
Contest Description:
"{fs[0]['overview']}"
Submission Story:
"{fs[0]['story']}"
Expected LLM Output:
"{fs[0]['output']}"

**Example 2: {fs[1]['class']} Submission**
Contest Description:
"{fs[1]['overview']}"
Submission Story:
"{fs[1]['story']}"
Expected LLM Output:
"{fs[1]['output']}"

**Example 3: {fs[2]['class']} Submission**
Contest Description:
"{fs[2]['overview']}"
Submission Story:
"{fs[2]['story']}"
Expected LLM Output:
"{fs[2]['output']}"
--- END FEW-SHOT EXAMPLES ---
```

```
--- BEGIN SUBMISSION TO EVALUATE ---
Contest Description:
{contest_description}

Submission Story:
{submission_story}
--- END SUBMISSION TO EVALUATE ---

End your response with <|eot_id|>.
```

3.5.5. Contest-Specific Prompt Adaptation

While the generative model was initially designed for broad contest evaluation, refinements were tested which adjust its assessments for specific contests. One such refinement focused on the *BuildTogether* contests, which emphasized assistive technologies for individuals with disabilities.

Motivation for Contest-Specific Prompting

The framework enabled evaluation of any contest submission using a standardized prompt and generic few-shot examples. However, this approach risked overlooking contest-specific details, especially in specialized themes like assistive technology. To address this, a targeted adaptation was tested on the **mobility impairment** track of the BuildTogether contests. By incorporating contest-specific context and examples, the goal was to enhance the LLM's ability to assess solutions within this domain.

BuildTogether Contests

The *BuildTogether* contests challenged participants to design assistive technologies that enhance accessibility and independence for individuals with disabilities. Each contest had multiple tracks, including a dedicated category for solutions targeting mobility impairments:

- **BuildTogether 1:** Participants developed assistive solutions for gaming and traveling for people with mobility impairments, alongside technologies for swimming for individuals with visual impairments. The contest emphasized co-creation with individuals with disabilities, known as "Contest Masters," who provided feedback on solution feasibility and inclusivity.

- **BuildTogether 2:** The second iteration expanded the scope, dividing the competition into two primary focus areas: visual impairments and mobility impairments. Within the mobility impairment category, participants could develop accessible home tools or sports and hobby innovations.

Targeted Prompt Modification

To test whether a contest-specific prompt and selected few-shot examples could improve the LLM's reasoning, a focused adaptation was made for the mobility impairment track of BuildTogether 2. A specialized prompt was created, and relevant few-shot examples from BuildTogether 1 were chosen. The goal was to help the model better understand and evaluate solutions in this domain by providing clearer context and relevant examples. The following adjustments were made:

1. **Refining the Instruction Section:** The opening instruction was adapted to align with the contest's specific theme, ensuring the model evaluated solutions from the perspective of accessibility and usability for individuals with mobility impairments. The modified prompt introduction was:

```
You are an expert evaluator for an innovation contest related to
developing solutions for individuals with mobility impairments.
...
```

2. **Adjusting the Few-Shot Examples:** Instead of generic few-shot examples, three mobility impairment submissions from BuildTogether 1 were chosen as reference cases. This gave the LLM relevant examples, helping it better understand and evaluate solutions within the contest's theme.

3.6. Experimental Setup and Model Execution

The evaluations were conducted on Google Colab's cloud GPUs to efficiently process LLMs. The framework is built with modular classes that handle data processing, model execution, and evaluation. Each class has a specific function but works together as a complete system. The `DriveManager` ensures dataset access through Google Drive, while `ModelLoader` provides the necessary LLM execution capabilities. `DataCleaner` and `Summarizer` refine input data for evaluation. The `SubmissionEvaluator` performs qualitative assessments, whereas `DiscriminativeEvaluator` provides a textual output regarding quantitative attributes. Finally, `ContestEvaluator` coordinates all components, automating the evaluation pipeline and saving results for analysis.

Class Overview:

- `DriveManager`: Handles Google Drive operations for dataset loading and result storage.
- `ModelLoader`: Downloads and configures LLMs from Hugging Face, optimizing execution with half-precision settings.
- `DiscriminativeModel`: Trains and evaluates machine learning models for predicting submission quality. It manages dataset splitting, cross-validation, and performance evaluation using a recall-based metric. The model outputs a numerical score between 0 and 1 for each submission.
- `DiscriminativeEvaluator`: Converts numerical model outputs into text by comparing each submission's quantitative attributes within its contest.
- `DataCleaner`: Prepares textual data by normalizing formatting and enforcing token limits.
- `Summarizer`: Summarizes long contest descriptions while preserving essential details.
- `SubmissionEvaluator`: Constructs structured prompts and evaluates individual submissions based on novelty and usefulness.
- `ContestEvaluator`: Orchestrates the complete evaluation process, executing both generative and discriminative assessments for all submissions in a contest and storing structured results.

3.6.1. Evaluation Framework

The evaluation process is managed by the `ContestEvaluator` class, which sequentially runs both the **generative model** and the **discriminative evaluator** for each contest submission. The numerical output from the `DiscriminativeModel` has already been generated at this stage. The process follows these steps:

1. Loop through all contest submissions, ensuring each entry is processed individually.
2. Retrieve the full text and associated metadata for each submission from the dataset.
3. Construct a structured prompt, incorporating few-shot examples to guide the evaluation.
4. Utilize the chosen LLM to assess the submission, focusing on novelty and usefulness.
5. Convert the discriminative model's numerical score into a comparative textual evaluation based on quantitative metrics.

6. Save all evaluation results, including generative and discriminative assessments, in a structured CSV file for further analysis and visualization.

All evaluation results are automatically stored in a structured CSV file containing:

- Submission ID and metadata.
- Generative model classification and reasoning.
- Discriminative model classification and textual evaluation.

3.6.2. Model Selection and Execution

Multiple large language models were considered, with a focus on **Llama** and **DeepSeek** variants. The key criteria for model selection included:

- **Openness**: Preference was given to open-source models that allow custom prompt injection and local GPU inference.
- **Performance Benchmarks**: Selection of state-of-the-art models with strong performance in instruction-following tasks.
- **Compatibility**: Ensuring seamless integration with the existing Python-based pipeline and Hugging Face libraries.

To evaluate their effectiveness in classifying contest submissions, multiple LLMs were tested:

- `Llama-3.2-3B-Instruct`
- `Llama-3.1-8B-Instruct`
- `DeepSeek-R1-Distill-Llama-8B`
- `DeepSeek-R1-Distill-Qwen-14B`

3.6.3. Execution and Scalability

The evaluation framework runs in Google Colab, utilizing **L4** or **A100 GPUs** to optimize inference speed and efficiency. Models are executed *sequentially*, ensuring each submission is processed independently and consistently across different models.

On average, evaluation times range from a few seconds to over a minute per submission. Evaluation time varies depending on several factors:

- **Model Size**: Larger models with many parameters take longer per submission than smaller ones.
- **GPU**: Inference time also depends on the GPU. The A100 is significantly faster than the L4.
- **Prompt Length**: Longer submissions and more extensive few-shot examples increase token usage and processing time.

4. Results

This chapter presents the evaluation results of discriminative and generative models. Discriminative models assess submissions based on structured quantitative features, while the generative model provides reasoning-based evaluations of novelty and usefulness. The performance of these models is analyzed across multiple contests, with a focus on classification accuracy, recall, and alignment with actual contest outcomes. In addition, the impact of contest-specific prompt adaptations is examined to assess their effectiveness in refining AI-driven evaluations.

4.1. Discriminative Models

This section presents the performance of the discriminative models[1]—Multi-Layer Perceptron (MLP), Random Forest, XGBoost, SVM, and Logistic Regression, under the complex evaluation procedure described in Chapter 3. The evaluation framework leverages multiple runs and multiple folds to ensure robust performance estimates.[2]

Dataset Complexity and Evaluation Setup

After the data cleaning process, a total of 112 contests remained. These contests differ significantly in size, ranging from a minimum of 11 solutions to a maximum of 347 solutions. Additionally, the amount of solutions winning prizes vary from as low as 0.6% to as high as 27% across the contests. Due to these substantial differences, the application of a uniform filtering method to all contests is not trivial.[3] To address this complexity, each contest in the data set receives equal weighting in the final metrics, regardless of size or default win percentage.

[1]Compared to the prior methodology (Wong, 2024), the model lineup and hyperparameters have been updated or re-tuned.

[2]Prior work used only a single pass with fewer splits, potentially introducing higher variance in the results.

[3]In the prior thesis, filtering thresholds were based on each contest's existing win rate, introducing biases when comparing very large vs. very small contests.

Uniform Filter Percentages Across Contests

In contrast to earlier approaches[4], the present evaluation always removes the same percentage of solutions from each contest. For example, at a filter percentage of 0.5 (50%), exactly half of the solutions in each contest are classified as "filtered out ' (predicted to be non-winners). Then I aggregate True Positives, False Positives, True Negatives, and False Negatives over all contests in the fold and compute overall recall and accuracy. This ensures that each contest is treated equivalently, mitigating the bias introduced by highly variable win rates.

The filter percentages tested range from 0.1 (10%) to 0.7 (70%). The upper limit of 70% is set because the most "aggressive" real contest scenario in the data set shows a win rate of 27%. Filtering more than 70% would systematically eliminate at least some true winners in that particular contest.

Furthermore, for each filter percentage, the final results are averaged over five runs of 5-fold cross-validation, reflecting the multi-run, multi-fold nature of the evaluation process.

Focus on Recall Over Accuracy

Since the solutions for each contest are filtered in a fixed percentage, contests with a very small or very large default win percentage may see unusual accuracy values. For example, a contest that typically awards very few winners might end up with lower accuracy if the filter percentage is relatively high compared to its real (small) win ratio. Consequently, recall is emphasized—the fraction of actual winners that remain unfiltered—as the primary metric of interest. Our practical goal is to maximize recall while filtering out as many non-winning solutions as possible.

4.1.1. Results Across Filter Percentages

Table 4.1 shows the average recall[5] for each model at filter percentages ranging from 10% to 70%. A higher filter percentage implies more solutions are removed, leaving fewer candidates; naturally, recall decreases. Figure 4.1 visualizes these results.

A clear trend emerges: as filter percentage increases, recall consistently decreases across all models. Random Forest and XGBoost display similar curves, while MLP, SVM, and Logistic Regression perform comparably and slightly outpace Random Forest and XGBoost at filter percentages above

[4]Previously, a scaled multiplier was applied to each contest's original win rate, complicating cross-contest comparisons.

[5]"Recall" here is computed by summing all true positives and false negatives across contests for each fold, then averaging over runs.

Table 4.1.: Average recall of each model at different filter percentages. Higher recall indicates fewer winners are missed.

Filter Percentage	MLP	Random Forest	XGBoost	SVM	Logistic Regression
0.1	0.9979	1.0000	1.0000	0.9984	0.9974
0.2	0.9932	0.9906	0.9885	0.9916	0.9916
0.3	0.9843	0.9708	0.9713	0.9843	0.9875
0.4	0.9728	0.9332	0.9300	0.9671	0.9734
0.5	0.9504	0.8632	0.8731	0.9452	0.9467
0.6	0.8789	0.7864	0.7953	0.8841	0.8961
0.7	0.7629	0.6789	0.6903	0.7629	0.7702

20%. These similarities and subtle differences are more apparent in the plotted curves shown in Figure 4.1.[6]

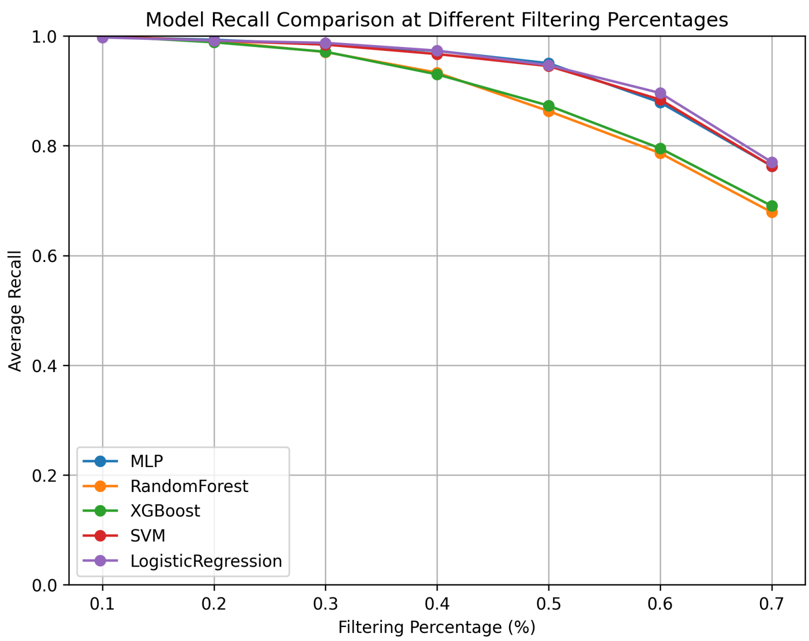

Figure 4.1.: Comparison of average recall for each model across filter percentages from 10% to 70%.

4.1.2. Per-Model Analysis

In this section, one representative run is shown for each of the five models: MLP, Random Forest, XGBoost, SVM, and Logistic Regression. Note that the overall recall values presented in

[6] In the prior thesis, Random Forest and XGBoost dominated at moderate thresholds; the new data cleaning and uniform filtering approach changed the models' relative standings.

Section 4.1.1 are the *averages* across all five runs. Therefore, there may be small discrepancies between those average values and the representative tables in this section.

MLP

Table 4.2 shows a single run of the MLP model under different filter percentages. The *Filter Percentage* column indicates the fraction of submissions being discarded, while *Remaining Size* denotes the actual percentage of the original submission pool that remains. As we increase the filter percentage from 10% to 70%, the *Recall* decreases in exchange for excluding more submissions. We observe that at 50% filtering, the MLP retains about 95.8% of the actual winners but discards nearly half of the total submissions.

Table 4.2.: MLP results for one run across different filter percentages.

Filter Percentage	Remaining Size	Accuracy	Recall	TN	FP	FN	TP
0.1	89.14	0.168	1.000	690	5307	0	383
0.2	79.24	0.266	0.992	1320	4677	3	380
0.3	69.16	0.367	0.987	1961	4036	5	378
0.4	59.22	0.465	0.977	2592	3405	9	374
0.5	49.56	0.559	0.958	3201	2796	16	367
0.6	39.27	0.653	0.880	3827	2170	46	337
0.7	29.20	0.740	0.770	4428	1569	88	295

Random Forest

Table 4.3 presents one run of the Random Forest model. Compared to the MLP results, we see that at 50% filtering, the Random Forest retains slightly fewer winners (recall of 0.872 vs. 0.958 in the MLP example). Performance trends remain consistent: recall drops as filter percentage grows.

XGBoost

In Table 4.4, XGBoost's recall also declines more rapidly compared to the MLP example. For instance, at 50% filtering, recall is 0.854, which is slightly higher than Random Forest in the same scenario but still below the MLP example.

Table 4.3.: Random Forest results for one run across different filter percentages.

Filter Percentage	Remaining Size	Accuracy	Recall	TN	FP	FN	TP
0.1	89.13	0.168	1.000	690	5307	0	383
0.2	79.21	0.266	0.984	1317	4680	6	377
0.3	69.14	0.364	0.969	1954	4043	12	371
0.4	59.18	0.460	0.935	2576	3421	25	358
0.5	49.56	0.549	0.872	3168	2829	49	334
0.6	39.26	0.642	0.789	3792	2205	81	302
0.7	29.18	0.730	0.687	4396	1601	120	263

Table 4.4.: XGBoost results for one run across different filter percentages.

Filter Percentage	Remaining Size	Accuracy	Recall	TN	FP	FN	TP
0.1	89.16	0.168	1.000	690	5307	0	383
0.2	79.23	0.266	0.990	1319	4678	4	379
0.3	69.17	0.364	0.966	1953	4044	13	370
0.4	59.19	0.458	0.922	2571	3426	30	353
0.5	49.57	0.547	0.854	3161	2836	56	327
0.6	39.28	0.641	0.783	3790	2207	83	300
0.7	29.17	0.730	0.684	4395	1602	121	262

SVM

Table 4.5 highlights the SVM performance. At 50% filtering, SVM achieves a recall of 0.948, which is substantially higher than both Random Forest and XGBoost in this single-run example. As with other models, recall declines further at higher filter percentages.

Table 4.5.: SVM results for one run across different filter percentages.

Filter Percentage	Remaining Size	Accuracy	Recall	TN	FP	FN	TP
0.1	89.15	0.168	0.997	689	5308	1	382
0.2	79.21	0.266	0.992	1320	4677	3	380
0.3	69.14	0.366	0.982	1959	4038	7	376
0.4	59.16	0.464	0.969	2589	3408	12	371
0.5	49.55	0.558	0.948	3197	2800	20	363
0.6	39.26	0.654	0.890	3831	2166	42	341
0.7	29.15	0.739	0.762	4425	1572	91	292

Logistic Regression

Finally, Table 4.6 shows one run with Logistic Regression. At 50% filtering, the recall is 0.945, which is again higher than Random Forest and XGBoost but close to MLP and SVM in this particular example.

Table 4.6.: Logistic Regression results for one run across different filter percentages.

Filter Percentage	Remaining Size	Accuracy	Recall	TN	FP	FN	TP
0.1	89.15	0.168	0.997	689	5308	1	382
0.2	79.23	0.266	0.992	1320	4677	3	380
0.3	69.15	0.367	0.990	1962	4035	4	379
0.4	59.18	0.464	0.971	2590	3407	11	372
0.5	49.57	0.558	0.945	3196	2801	21	362
0.6	39.27	0.655	0.903	3836	2161	37	346
0.7	29.17	0.742	0.781	4432	1565	84	299

4.1.3. Comparison

In comparing the tables above, Random Forest and XGBoost form one group with moderately lower recall, while MLP, SVM, and Logistic Regression show higher recall under equivalent filter thresholds. This grouping is especially noticeable at mid-range filter percentages (e.g., 40–50%), where Random Forest and XGBoost discard a higher proportion of actual winners compared to the other three models.

Overall, MLP, SVM, and Logistic Regression present similar performance levels, consistently achieving higher recall than Random Forest and XGBoost. This difference remains even as more submissions are filtered out, suggesting that these three models are more robust in retaining potential winners.

These single-run examples align with the averaged recalls presented earlier. While small numeric discrepancies appear due to each run's randomness, the overall pattern persists across all five runs. Hence, choosing MLP, SVM, or Logistic Regression may be preferable if the goal is to keep recall high when aggressively filtering out lower-ranked submissions.

4.1.4. Impact of Model Complexity

Although hyperparameter tuning and more sophisticated architectures often improve machine learning models, in this setting, simpler configurations consistently performed better. Two different MLP variants and multiple parameter grids for the other models were tested.

Simple vs. Enhanced MLP.

Two MLP architectures were tested:

- **Simple MLP:** A basic feed-forward network with two hidden layers and ReLU activations.
- **Enhanced MLP:** Added dropout layers, batch normalization, a different activation function, and a more dynamic training scheme.

Despite these additional features, the simpler MLP achieved higher recall in practice. Table 4.7 illustrates their respective recall values under five filter percentages (0.1 to 0.5). As filter percentages rise, the simpler model maintains higher recall, whereas the enhanced version's performance drops more sharply.

Table 4.7.: Recall of Simple MLP vs. Enhanced MLP at different filter percentages

Filter Percentage	Simple MLP Recall	Enhanced MLP Recall
0.1	0.9974	0.9828
0.2	0.9922	0.9593
0.3	0.9845	0.9130
0.4	0.9716	0.8729
0.5	0.9535	0.8109

Other Models. Random Forest, XGBoost, SVM, and Logistic Regression were also tested using various grid-searched parameters. For example, Random Forest explored deeper trees (larger `max_depth`), while XGBoost was tuned for `n_estimators` and `learning_rate`. Similarly, SVM and Logistic Regression used different values of `C` and kernel/penalty options. In all cases, allowing the models to grow more complex (e.g., deeper trees, higher `C` values) did not improve recall; the models overfit faster, leading to lower performance when filtering for top submissions. As a result, simpler parameter choices (`max_depth=3` in Random Forest and XGBoost, `C=0.1` for SVM and Logistic Regression) provided more reliable generalization.

4.2. Classification Distribution

The evaluation framework consists of two distinct AI models: the discriminative model, which ranks submissions based on measurable quantitative features, and the generative model, which provides structured reasoning to assess novelty and usefulness. Since both models operate independently, they classify each submission separately into one of three categories: *Excellent*, *Average*, or *Poor*. This section presents a comparative analysis of the classification distributions produced by these models to provide a structured comparison.

The x-axis of the classification plots represents the prize categories, distinguishing between submissions that won a **Top Prize**, those that received a secondary prize (**Other Prize**), and those that did not win any prize. The y-axis indicates the number of submissions classified into each category (*Excellent*, *Average*, and *Poor*). This allows for an assessment of the classification tendencies of both approaches and an analysis of whether their outputs align with contest outcomes.

The discriminative model ranks submissions based on a numerical score between 0 and 1, computed from quantitative features such as documentation length, media inclusion, and completeness. The discriminative evaluator assigns classifications based on these rankings: the top 20% of submissions in each contest are labeled as *Excellent*, the next 40% as *Average*, and the bottom 40% as *Poor*. The generative model, in contrast, follows a reasoning-based approach where classifications are determined through textual assessments of novelty and usefulness.

4.2.1. Discriminative Model Classification

The classification results from the discriminative evaluator reveal a clear trend correlating submission ranking with prize categories. Table 4.8 presents the number of submissions classified into each category based on prize outcomes.

Table 4.8.: Discriminative Model Classification by Prize Category

Prize Category	Excellent	Average	Poor
No Prize	806	2192	2543
Other Prize	216	282	90
Top Prize	244	133	10

A clear pattern emerges from these results. Submissions that won a **Top Prize** were overwhelmingly classified as *Excellent*, with 244 out of the total 387 submissions in this category receiving the highest classification. The **Other Prize** category, which represents secondary award winners, also saw a majority of submissions classified as *Excellent* or *Average*, with only 90 out of 588 entries being classified as *Poor*. In contrast, the vast majority of submissions classified as *Poor* came from the **No Prize** category, where 2543 out of 5541 submissions were ranked in the lowest tier.

Notably, only 10 **Top Prize** submissions and 90 **Other Prize** submissions were categorized as *Poor*, compared to a total of 2643 *Poor* classifications overall. This result strongly indicates that the discriminative model effectively ranks high-quality submissions above lower-quality ones, supporting its validity as a prescreening mechanism. The classification structure aligns well with contest outcomes, reinforcing that submissions with better documentation and completeness are more likely to win prizes.

The classification trends observed in Table 4.8 are further visualized in Figure 4.2. This figure illustrates the distribution of classifications across prize categories, making it evident that submissions receiving higher prizes are more frequently classified as *Excellent*, while lower-ranked submissions predominantly fall into the *Average* or *Poor* categories. The strong correlation between classification outcomes and actual contest results highlights the effectiveness of the discriminative model in distinguishing well-documented and structured submissions from weaker ones.

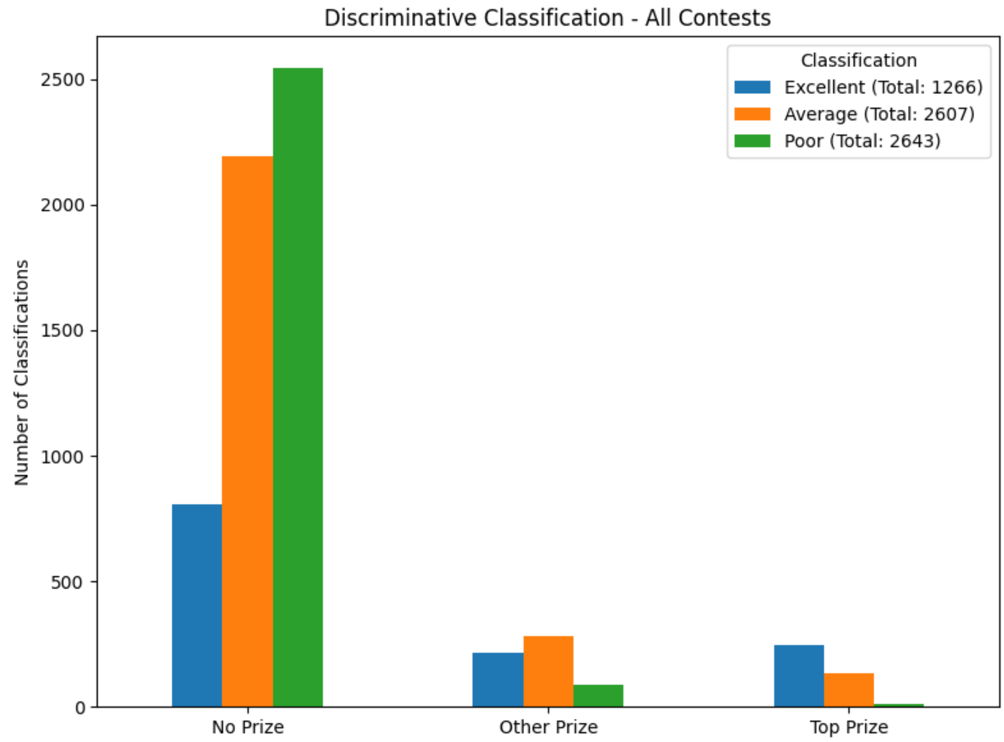

Figure 4.2.: Classification distribution of the discriminative evaluator across prize categories.

4.2.2. Generative Model Classification

The generative model evaluates submissions based on novelty and usefulness, rather than quantitative features. Unlike the discriminative model, which relies on structured numerical metrics, the generative model provides qualitative assessments through structured reasoning. Since each evaluation is based on language-based reasoning rather than predefined numerical thresholds, the primary value of the generative model lies in its ability to explain why a submission demonstrates innovation and practical applicability. The classification distribution presented here serves as an overview of how the model categorizes submissions, but the core function of the generative model remains its ability to provide detailed reasoning. Example outputs illustrating this reasoning will be presented in later sections.

The table below summarizes how the generative model categorized submissions into *Excellent*, *Average*, and *Poor*, grouped by prize category.

Table 4.9.: Generative Model Classification by Prize Category

Prize Category	Excellent	Average	Poor
No Prize	1741	3356	444
Other Prize	279	298	11
Top Prize	250	128	9

The classification pattern observed in the generative model follows a trend similar to that of the discriminative model, with higher-ranked submissions receiving more *Excellent* ratings and lower-ranked ones being classified as *Average* or *Poor*. Among the **Top Prize** winners, nearly two-thirds were classified as *Excellent*, demonstrating that the model successfully recognized the strongest submissions as highly novel and useful. Only a small fraction (9 submissions) in this category were rated as *Poor*, indicating that the generative model rarely assigns low scores to highly ranked submissions.

For the **Other Prize** category, the classification is more evenly split between *Excellent* and *Average*, with similar numbers in both categories. This suggests that while these submissions exhibit notable strengths, they do not consistently meet the highest standards of novelty and usefulness required for *Excellent* classification.

The **No Prize** category shows a more varied distribution, with twice as many submissions classified as *Average* compared to *Excellent*. Furthermore, nearly all of the *Poor* classifications (444 out of 464 total) are concentrated in this group. This indicates that the model successfully distinguishes between weaker submissions and those demonstrating stronger innovation. However, compared to the discriminative model, the generative model assigns *Poor* classifications far less frequently. This can be attributed to the qualitative nature of the evaluations, where the model often finds at least some degree of novelty or usefulness in most submissions, making outright rejection less common.

Figure 4.3 visualizes these classification trends, highlighting the relationship between prize-winning status and the model's assessment of novelty and usefulness.

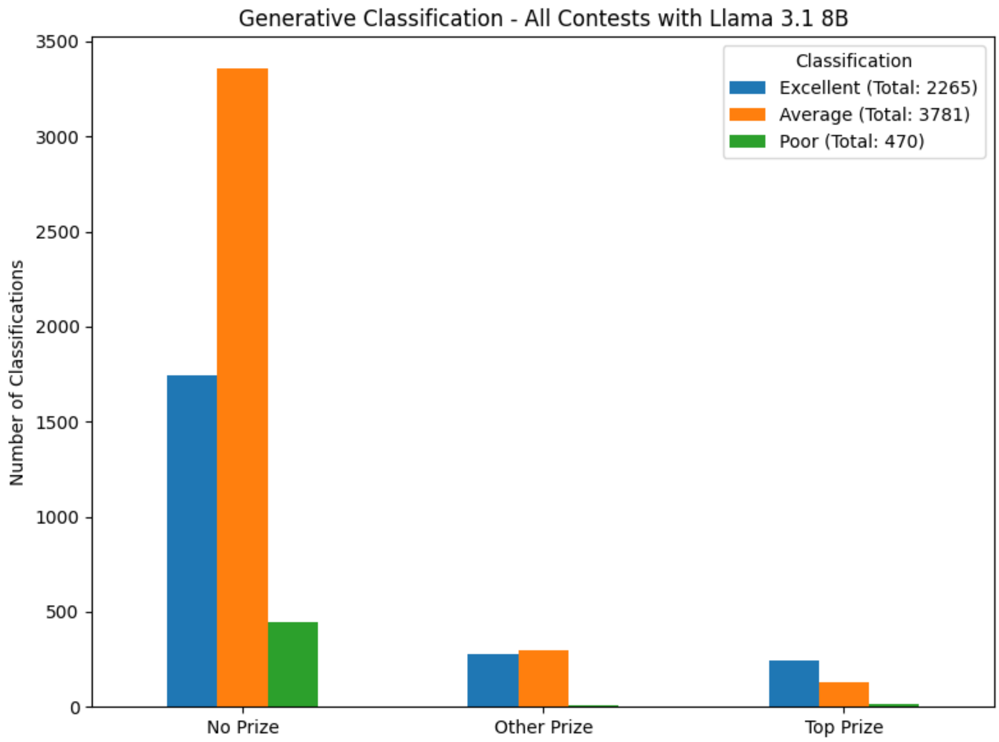

Figure 4.3.: Generative Model Classification Distribution Across Prize Categories

While the classification pattern aligns with expectations—prize-winning submissions are more likely to be classified as *Excellent*—the distinction between categories is not as sharp as in the discriminative model. This is an inherent characteristic of the generative approach, as it does not strictly follow numerical thresholds but rather reasons through each submission's merits. The relatively low number of *Poor* classifications also suggests that the model leans towards recognizing at least some value in most submissions, potentially making it more forgiving than the discriminative model. Despite this, the overall trends indicate that the generative model effectively prioritizes highly innovative solutions while identifying weaker ones with reasonable accuracy.

4.3. Model Comparison

While the primary role of the generative model is to provide structured reasoning rather than categorical classification, summarizing its output in this manner only offers a broad overview of its decision-making patterns. The classification distribution for a single contest will still be shown to illustrate how both models evaluated the submissions. However, since this alone does not fully capture the model's reasoning, examples of its textual outputs will follow, providing a clearer view of how individual submissions were assessed.

4.3.1. Output Format

The textual output of the generative model follows a structured format, ensuring consistency in evaluation. Each assessment consists of three key sections: an analysis of the submission's **novelty**, an evaluation of its **usefulness**, and a final **classification** into one of the three categories. This structured output allows for systematic extraction of different components, making it possible to analyze model outputs efficiently. An example of this output is shown below:

> **Novelty of the Solution:**
> Reasoning about the novelty of the solution.
>
> **Usefulness of the Solution:**
> Reasoning about the usefulness of the solution.
>
> **Overall Recommendation:**
> *Poor, Average, or Excellent*

Similarly, the discriminative evaluator provides a structured output based on quantitative analysis. It evaluates submissions based on factors such as text length, number of visuals, and presence of detailed documentation. The output consists of three sections, each corresponding to a key aspect of submission completeness:

> **Description and Bills of Materials:**
> Evaluation of the documentation length and the comprehensiveness of the listed materials.
>
> **Visuals, Code, and Other Documentation:**
> Evaluation of the inclusion of images, videos, schematics, and code.
>
> **Overall Recommendation:**
> *Poor, Average, or Excellent*

By maintaining these structured formats, different parts of the evaluation can be extracted using regex, enabling further analysis and experimentation.

4.3.2. Example Contest

In the following section, the contest *KinetisFlexIO* will be briefly introduced, followed by an analysis of how both models evaluated its submissions. Outputs and classification overviews from both models will be compared to assess their alignment and differences in evaluating contest entries.

KinetisFlexIO

The *KinetisFlexIO* contest focused on enabling innovative designs using NXP's Kinetis microcontrollers, particularly emphasizing energy efficiency, security, and flexibility for IoT and wearable applications. The contest awarded a total of five prizes, with four top prizes of $3,000 each and one secondary prize of $200. All remaining submissions did not receive a prize.

Generative Evaluation The generative model classified the 59 submissions into three categories based on novelty and usefulness. A total of 17 submissions were rated as *Excellent*, indicating strong innovation and practical value. The majority, 38 submissions, were classified as *Average*, meeting general expectations but not significantly exceeding them. Finally, only four submissions were categorized as *Poor*, suggesting limited novelty or practical application.

All five prize-winning submissions were classified as *Excellent*, indicating that the model successfully identified the top projects as highly novel and useful. Among the submissions that did not win any prize, the majority were classified as *Average*, with a few receiving a *Poor* rating. The classification distribution for this contest is shown in Figure 4.4.

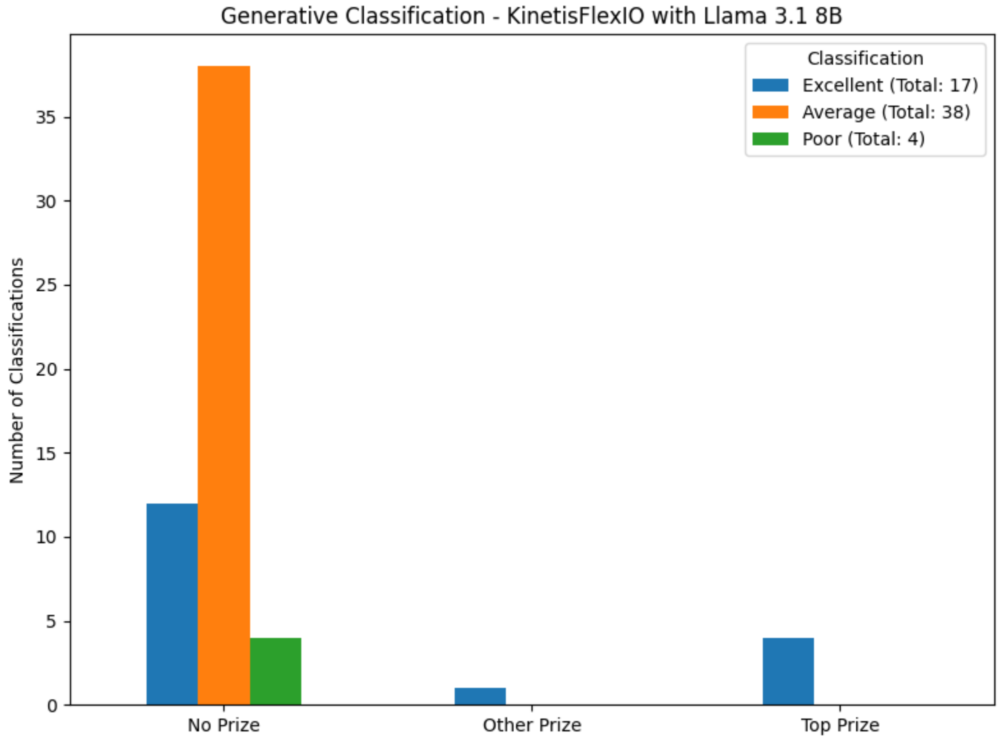

Figure 4.4.: Generative Model Classification for KinetisFlexIO Contest

Discriminative Evaluation The discriminative model evaluates submissions based on quantitative attributes. In total, it classified 11 submissions as *Excellent*, 24 as *Average*, and 24 as *Poor* as shown in Figure 4.5.

The classification of the five prize-winning submissions closely matched the results of the generative model. The four **Top prize** winners were classified as *Excellent* by both models, indicating that these submissions combined strong innovation with well-structured and detailed documentation. The **Other Prize** winner was classified as *Average* by the discriminative model. This suggests that while the submission demonstrated sufficient novelty and usefulness, it lacked the same level of extensive documentation and completeness seen in the top prize winners.

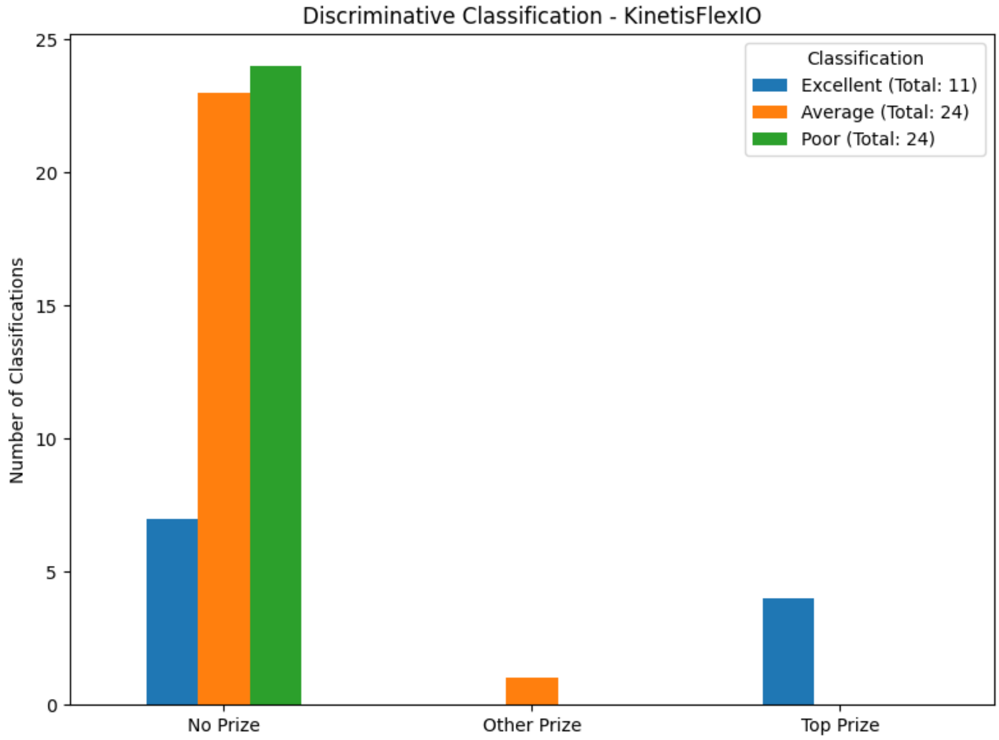

Figure 4.5.: Discriminative Model Classification for KinetisFlexIO Contest

4.3.3. Example Submissions

The following sections present and evaluate the outputs of both the generative and discriminative models. Their assessments will be compared to determine how well they align with the judged outcome and whether their evaluations provide meaningful insights into the project's quality.

Project Evaluated: FlexIO Based Multi-Copter Rotor Control

This project, one of the top prize winners from the KinetisFlexIO Contest, focuses on real-time monitoring and control of multicopter propellers using FlexIO. The main challenge addressed is the ability to detect propeller hits or slowdowns in real time to prevent damage to both the drone and the object it is inspecting. This project won one of the top prizes, meaning the judges recognized it as one of the best submissions in the contest.

Generative Evaluation:

Novelty of the Solution:
The project presents a unique approach by integrating FlexIO module to monitor the

velocity of the propellers of a multicopter in real-time, providing the needed feedback for safety, velocity control, and system performance monitoring. The use of FlexIO to track propeller speed control and make instant notifications of any failure to maintain commanded propeller speed is an innovative application of the technology. While multicopter control systems exist, this solution uniquely combines FlexIO, Processor Expert, and the Kinetis SDK to achieve real-time monitoring and control. The addition of a multicopter build and the use of optical sensors to detect propeller rotations adds to the novelty of the project.

Usefulness of the Solution:
The practicality of this solution is exceptional, addressing real-world challenges such as safety, velocity control, and system performance monitoring in multicopter applications. The ability to detect propeller hits and take necessary actions to prevent damage to the object, the propeller, the motor drive mechanisms, and motor drive electronics is a significant improvement over existing multicopter control systems. The use of a simple operational amplifier circuit to create logic-level tachometer signals for the FRDM-K82F and the implementation of a software trigger function to cause the PWM duty cycle to update are effective solutions to the challenges faced in the project. The project's ability to adapt to different environments and applications, such as smart roads, smart homes, industrial IoT, wearable devices, and security, further increases its usefulness.

Overall Recommendation:
Excellent
The project demonstrates both substantial novelty and usefulness, far exceeding typical expectations. The innovative application of FlexIO, Processor Expert, and the Kinetis SDK to achieve real-time monitoring and control of multicopter propellers is a significant contribution to the field. The project's ability to address real-world challenges and its adaptability to different environments and applications make it an excellent solution.

Generative Evaluation Analysis The generative model correctly identifies the project's strong novelty and real-world applicability. This project won one of the main prizes, making the *Excellent* classification appropriate.

The **novelty** evaluation accurately highlights the innovative use of FlexIO for real-time multicopter propeller monitoring. While similar control systems exist, this project uniquely integrates FlexIO, Processor Expert, and Kinetis SDK to detect and respond to propeller slowdowns instantly. The combination of timestamp-based tracking, optical sensors, and real-time adjustments distinguishes it from standard drone velocity control methods.

In terms of **usefulness**, the project addresses a critical issue in commercial drone applications, where precise flight control near objects is essential to prevent damage. FlexIO enables immediate feedback and corrective action, reducing collision risks beyond human reaction times. Its adaptability to various fields, including industrial IoT, security, and wearables, further increases its value.

The **overall recommendation** of *Excellent* is well justified. The project enhances existing drone control systems with precise real-time monitoring, making it highly relevant for professional applications. The generative model's assessment aligns with the project's demonstrated strengths.

Discriminative Evaluation:

Description and Bills of Materials:
The submission's description is excellent compared to other entries. It contains an extensive amount of text that thoroughly explains the project, covering all essential aspects in detail. This level of textual detail exceeds that of most other submissions. The length and level of detail in the description reflect the overall excellent efforts of the submission, mirroring the high standards seen throughout the project. The submission features excellent documentation of the components, providing a detailed and well-organized list of materials used.

Visuals, Code, and Other Documentation:
This entry provides average visual documentation, incorporating a moderate number of images and videos. While the visuals contribute to understanding the project, they are not as extensive as in the highest-ranked submissions. Even though the available visuals aid comprehension, additional supporting media could have further enhanced clarity and engagement. It fully shares its code and provides a repository link to encourage collaboration. These factors influence how easily others can reproduce or build upon the work.

Overall Recommendation:
Excellent

Discriminative Evaluation Analysis The discriminative model assigned this submission a score of 0.96 on a scale from 0 to 1, ranking it as the 7th best submission in the contest. This score placed it within the threshold for an *Excellent* classification. The evaluation highlights that the submission excels in textual description and component documentation, providing extensive details compared to other entries. However, its visual documentation is categorized as average, indicating that while the provided images and videos contribute to understanding, they are not as extensive as in the highest-ranked submissions. The mention of a shared repository link is also relevant, as it enhances accessibility and reproducibility. This evaluation provides insight into how the submission compares to others in terms of documentation quality.

These results demonstrate the complementary strengths of both models in this contest. Their strong agreement on winning submissions suggests that combining qualitative reasoning with quantitative evaluation provides a robust filtering mechanism for identifying top projects for this contest. While some discrepancies exist for non-winning entries, the overall alignment reinforces the effectiveness of this dual approach.

4.3.4. Impact of Contest-Specific Prompt Adaptation

The effectiveness of large language models in evaluating contest submissions depends significantly on the clarity and specificity of the prompt. The introduction of contest-specific adaptations, including tailored instructions and refined few-shot examples, enhances the accuracy and relevance of the model's assessments. This section compares the outputs generated by a general prompt with those produced by a contest-specific adaptation, using a case study to illustrate the improvements in evaluation quality.

Project Evaluated: IR Remote Control, Light Monitor & Fire Alarm

The evaluation focuses on a submission titled *IR Remote Control, Light Monitor & Fire Alarm Using Blues Notecarrier-A, Cardputer & XIAO ESP32S3*. The project integrates an infrared (IR) remote control, a light monitoring system, and a fire alarm, designed specifically for individuals with mobility impairments. It aims to improve accessibility by enabling users to control multiple home devices with a single universal remote while incorporating an automated fire detection system for enhanced safety.

General Model:

> **Novelty of the Solution:**
> The proposed solution demonstrates some level of innovation by integrating multiple sensors and technologies to create a comprehensive system for people with disabilities. The use of a single infrared remote control to activate multiple devices is a novel approach, as is the integration of a wrist cuff for comfortable use. However, similar concepts have been explored in accessible technology, and the solution does not significantly push the boundaries of existing solutions. The addition of a light monitor with fire alarm is an interesting feature, but it is not a novel concept in itself.

Usefulness of the Solution:
The solution has practical value for users with disabilities, particularly those with mobility impairments. The use of a single infrared remote control and a wrist cuff makes it easier for users to interact with multiple devices without the need to carry multiple remotes. The light monitor with fire alarm feature provides an additional layer of safety and convenience. However, the solution may face challenges in terms of setup and maintenance, particularly for users with limited technical expertise.

Overall Recommendation:
Average

Contest-Specific Model:

Novelty of the Solution:
The submission proposes a multi-functional assistive technology solution consisting of an IR remote control and a light monitor with a fire alarm. While the individual components may not be entirely novel, the combination and integration of these features into a single system for individuals with mobility impairments demonstrate some level of innovation. However, similar solutions that integrate multiple assistive technologies exist, such as smart home systems and wearable devices. The novelty of this solution lies in its specific application and design for users with mobility impairments, making it somewhat unique but not groundbreaking.

Usefulness of the Solution:
The solution has practical value for individuals with mobility impairments, particularly those who struggle with controlling multiple devices or navigating their environment. The IR remote control and light monitor with fire alarm can enhance accessibility and safety. However, the solution's effectiveness depends on the user's ability to operate the remote control and navigate the menu, which may be challenging for some individuals. The design also assumes a certain level of technical understanding, which might be a barrier for some users. Furthermore, the solution's scalability and maintainability are not explicitly addressed, which could be a concern for long-term use.

Overall Recommendation:
Average

Comparison of Evaluation Outputs

Table 4.10 presents the evaluations produced using the general and contest-specific prompts. Both assessments categorize the submission as *Average*, but the reasoning differs in depth, clarity, and specificity.

Table 4.10.: Comparison of Evaluations: General vs. Contest-Specific Prompts

Evaluation Criterion	General Prompt Output	Contest-Specific Prompt Output
Novelty	Recognizes integration of multiple sensors but states that similar concepts exist. Provides minimal comparison to other assistive technologies.	Acknowledges that the combination of features adds some innovation but emphasizes that similar assistive technologies, such as smart home and wearable systems, already exist. Highlights that the innovation lies in its adaptation for users with mobility impairments.
Usefulness	Emphasizes practical value for mobility-impaired users but provides only a broad assessment of challenges.	Identifies key benefits but also highlights potential challenges, such as difficulty in menu navigation and the need for technical understanding. Discusses scalability and maintainability, which were overlooked in the general assessment.
Overall Evaluation	Categorizes as *Average* but lacks clear justification beyond general usefulness.	Also categorizes as *Average* but provides a more structured rationale, explaining both the strengths and limitations in greater detail.

Advantages of Contest-Specific Adaptation

The contest-specific prompt and refined few-shot examples significantly improved the evaluation by enhancing the model's ability to assess the submission within the appropriate context. The following key improvements were observed:

The general prompt identified that the submission integrates multiple technologies but did not provide sufficient context regarding its uniqueness within the field of assistive technology. In contrast, the contest-specific prompt encouraged the model to explicitly compare the submission to existing smart home and wearable assistive devices, making the novelty assessment more precise.

The contest-specific adaptation also led to a more thorough examination of potential usability constraints. While the general prompt mentioned “setup and maintenance” challenges in a vague manner, the adapted version explicitly pointed out:

- The potential difficulty of menu navigation for mobility-impaired users.
- The assumption that users have sufficient technical understanding to operate the system.
- The lack of explicit discussion on the solution’s long-term scalability and maintainability.

These details provide a more practical and user-centered assessment.

Although both assessments categorized the submission as *Average*, the contest-specific evaluation justified this decision with a structured rationale. It acknowledged the submission’s strengths while critically addressing its limitations, making the overall evaluation more constructive and informative.

The comparison demonstrates that incorporating contest-specific adaptations in LLM-based evaluation frameworks significantly improves the precision and relevance of assessments. By refining the instruction set and using tailored few-shot examples, the model better contextualizes novelty, considers real-world usability challenges, and provides a more balanced evaluation. This approach enhances the credibility and usefulness of AI-driven assessments in innovation contests, particularly those involving specialized domains such as assistive technology.

4.4. Case Study

The following case study examines the *BuildTogether2* contest to analyze how the discriminative and generative AI models evaluated its submissions. The primary focus is on the generative model, where prompt adaptations were introduced to reduce the number of submissions classified as *Excellent* and increase the number of *Poor* classifications. This section outlines the contest’s objectives, presents classification results from both AI models, and discusses the impact of prompt modifications.

4.4.1. Contest Description and Goals

The *BuildTogether2* contest is an innovation challenge aimed at developing assistive technologies for individuals with disabilities. Following the success of its predecessor, this contest seeks to push inclusive innovation further by encouraging participants to design novel and useful solutions that enhance accessibility and independence for people with disabilities.

The contest is structured around two primary focus areas: Visual Impairments and Mobility Impairments. Each focus area is divided into two specific tracks, guiding participants toward solutions tailored to distinct challenges. For visual impairments, submissions may focus on adaptations for either outdoor or indoor activities. For mobility impairments, participants are encouraged to develop solutions for either home accessibility and tools or sports and hobbies. Contest Masters, experts in the disability space, provide tailored feedback to support participants in refining their projects.

4.4.2. Discriminative AI Results

The discriminative model was applied to submissions from the *BuildTogether2* contest to classify them into *Excellent*, *Average*, or *Poor* based on structured quantitative features. The classification results are summarized in Table 4.11.

Table 4.11.: Discriminative Model Classification for BuildTogether2

Prize Category	Excellent	Average	Poor
No Prize	10	28	51
Other Prize	15	24	4
Top Prize	2	2	0

The results indicate that the discriminative model performs effectively, with a classification pattern that aligns well with prize outcomes. Among the 55 submissions classified as *Poor*, 51 did not receive any prize, confirming that the model successfully identifies lower-ranked submissions. Additionally, the proportion of submissions classified as *Excellent* decreases as prize ranking declines. The **Top Prize** category has the smallest number of submissions, but a majority were classified as *Excellent* or *Average*, with none falling into the *Poor* category. In contrast, the **No Prize** category has the highest proportion of *Poor* classifications and the lowest proportion of *Excellent* classifications.

This classification trend suggests that the discriminative model effectively distinguishes high-quality submissions from weaker ones. However, in the **Top Prize** category, the model classified two submissions as *Average*, which, while still favoring higher-ranked submissions, does not provide a clear distinction between the very best entries. The distribution is further visualized in Figure 4.6, illustrating the decreasing trend of *Excellent* classifications and the increasing proportion of *Poor* classifications as submission quality declines.

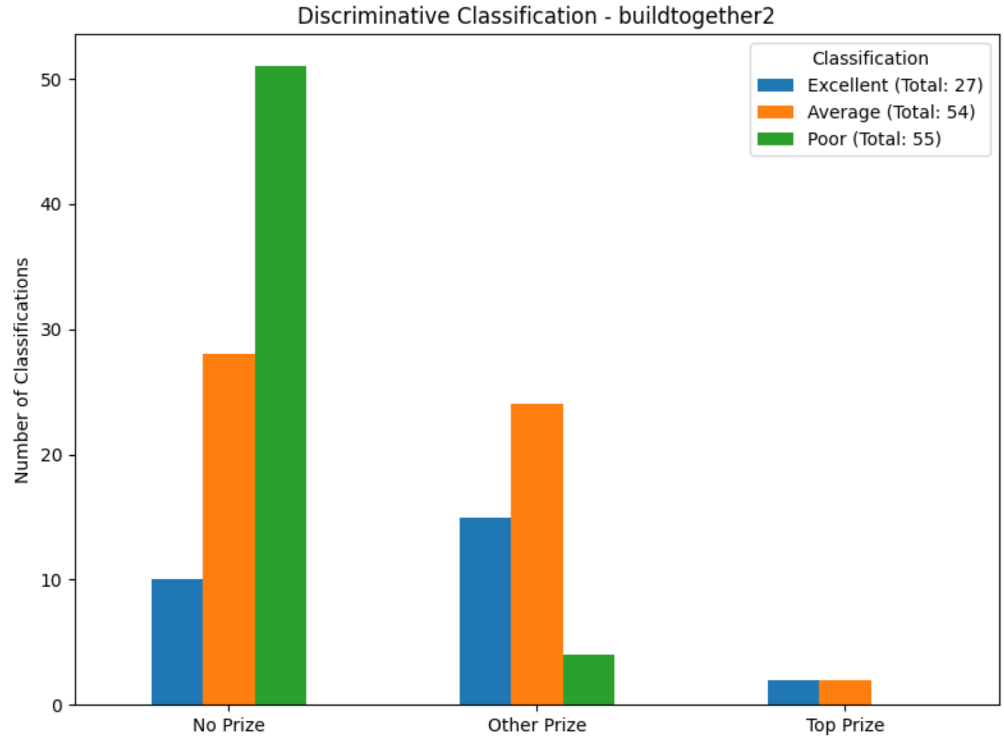

Figure 4.6.: Discriminative Model Classification Distribution for BuildTogether2

4.4.3. Generative AI Results

The generative model was initially evaluated with the general prompt from Subsection 3.5.4 that was not specifically adapted for the *BuildTogether2* contest. The classification distribution for this setup is shown in Table 4.12.

Table 4.12.: Generative Model Classification by Prize Category (General Prompt)

Prize Category	Excellent	Average	Poor
No Prize	40	46	3
Other Prize	26	16	1
Top Prize	3	1	0

The initial results indicate a strong bias toward *Excellent* classifications, particularly in the **No Prize** category, where 40 submissions were classified as *Excellent* despite not receiving any award. Additionally, the model assigned almost no submissions to the *Poor* category across all prize groups, making it difficult to differentiate lower-quality submissions.

To improve classification accuracy, the prompt was iteratively refined, and few-shot examples from the predecessor contest were incorporated, as outlined in Section 4.3.4. The tailored prompt led to a more balanced classification distribution, shown in Table 4.13.

Table 4.13.: Generative Model Classification by Prize Category (Tailored Prompt)

Prize Category	**Excellent**	**Average**	**Poor**
No Prize	9	61	19
Other Prize	6	29	8
Top Prize	2	2	0

With the tailored prompt, the number of *Excellent* classifications decreased significantly, particularly in the **No Prize** category, while the number of *Poor* classifications increased to a more reasonable level. Despite these improvements, the model still exhibited reluctance to assign *Poor* classifications.

A significant modification in the tailored prompt was the explicit emphasis on critical evaluation. Several adjustments were introduced to encourage the model to classify fewer submissions as *Excellent* and increase the proportion of *Poor* classifications. The updated prompt includes strict guidelines on novelty and usefulness, ensuring that a submission must clearly surpass existing alternatives to be considered *Excellent*. If there is any doubt regarding a solution's impact or originality, the model is instructed to default to a *Poor* rating.

Additionally, the few-shot examples were reduced to only two: one rated *Poor* and the other rated *Average*. This adjustment was made to further discourage the model from assigning *Excellent* classifications by limiting exposure to high-rated examples. The inclusion of concrete rejection criteria for *Poor* classifications—such as AI-generated content, lack of originality, vague execution plans, and excessive use of buzzwords—also played a role in refining the model's judgment.

Figure 4.7 visualizes the classification distribution resulting from this adapted prompt. The tailored instructions successfully increased the number of *Poor* classifications while reducing the number of submissions rated *Excellent*, leading to a more balanced distribution.

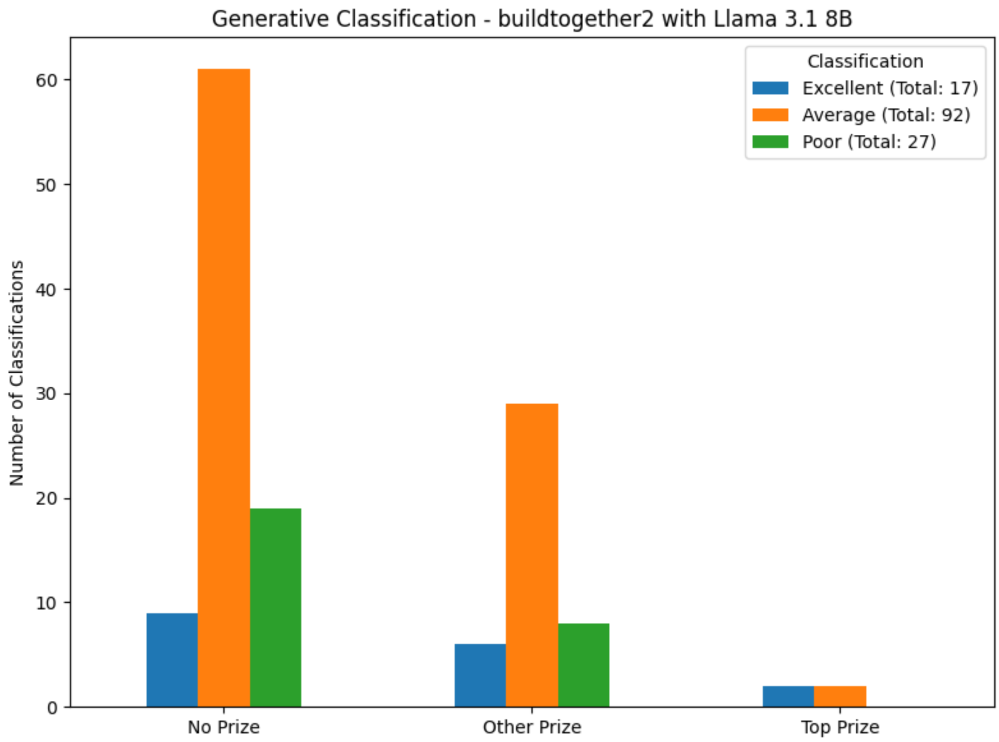

Figure 4.7.: Classification distribution of the generative evaluator for the BuildTogether2 contest after prompt refinement.

While these modifications led to a more critical evaluation process, they also introduced potential drawbacks. The prompt became more contest-specific, as the rejection criteria and assessment structure were fine-tuned to match the characteristics of *BuildTogether2*. Additionally, the stricter criteria may have made the model overly conservative, undervaluing some submissions. Despite these trade-offs, the results demonstrate that iterative prompt refinement and careful selection of few-shot examples can effectively recalibrate the model's classification tendencies.

The following is the refined prompt used for improved classification:

```
You are a critical expert evaluator for an innovation contest related to
    developing solutions for individuals with disabilities. Your task is to
    evaluate a solution based on specific criteria. The evaluation should
    focus on evaluating the solution and should not restate what the
    solution is about.

1. Provide a structured evaluation consisting of two concise paragraphs,
    each addressing one of the following criteria in a few sentences:
- *Novelty of the Solution*: Evaluate whether the solution introduces any
    new concepts, features, or technologies. Compare it to existing
    solutions and assess whether it offers a significant improvement. If it
    only repackages existing ideas or lacks substantial differences, this
    should be noted.
```

```
- *Usefulness of the Solution*: Critically assess the solutions practical
     value. Identify any major limitations that reduce its real-world
    applicability. If the usefulness is not clearly demonstrated or depends
    on vague assumptions, highlight these concerns.

2. Choose one of the following overall recommendations. Be highly critical
    in your evaluations. A solution should only be rated *Excellent* if it
    clearly surpasses existing alternatives in both novelty and usefulness.
    If there are any doubts, it should be rated *Poor*.

Most solutions should be rated Poor by default. *Average* should only be
    used in rare cases where a solution has clear, demonstrated value but
    still does not exceed typical expectations.

    **Solutions that should be rated *Poor* include:**
    - **AI-generated, generic, or filler content**: If the solution appears
        formulaic, vague, or lacks substantive detail, it is likely AI-
        generated and should be marked *Poor*.
    - **Lack of originality**: If the solution closely resembles existing
        products, known technologies, or common DIY solutions, it should be
        rated *Poor* unless it offers a unique improvement.
    - **No clear execution or technical depth**: If the description lacks
        concrete steps, technical details, or a structured plan for
        implementation, it should be rated *Poor*.
    - **Excessive repetition or buzzwords**: If a solution is wordy but
        lacks meaningful depth or repeatedly states obvious facts without
        adding substance, it should be rated *Poor*.

- *Excellent*: The solution is exceptionally novel and useful, setting a
    clear benchmark beyond typical expectations. It must demonstrate a
    significant advancement over existing solutions and have no major
    drawbacks.
- *Average*: The solution has some merit but still does not significantly
    exceed expectations. It must clearly describe a working concept and
    prove some value. Any notable limitations should push it towards *Poor*.
- *Poor*: The solution lacks originality, technical depth, or a clear
    implementation plan. If it is vague, overly generic, appears AI-
    generated, or fails to provide concrete improvements over existing
    solutions, it must be rated *Poor*.
```

4.4.4. Pros and Cons of Discriminative AI and Generative AI

Both the discriminative and generative AI models contribute unique strengths to the evaluation process, addressing different aspects of submission quality. The discriminative model efficiently processes large volumes of submissions by ranking them based on structured features. This ensures that well-documented submissions are prioritized, providing a reliable filtering mechanism. However, it only assesses surface-level completeness and may overlook highly novel submissions that lack extensive documentation.

In contrast, achieving the desired evaluation distribution with the generative model required multiple iterations and extensive trial and error. Refining tailored prompts and calibrating outputs to maintain consistency proved to be a time-intensive process. The model's inherent variability made it challenging to fine-tune, demanding careful adjustments to align with expectations. Additionally, its reliance on computationally expensive language models further complicated objective validation compared to numerical scoring methods.

Despite their limitations, the two models complement each other. The discriminative model ensures that submissions meet fundamental quality thresholds, while the generative model provides additional insights into their originality and impact. Differences in classification outcomes highlight that each model evaluates distinct dimensions of quality. By combining both approaches, a more balanced and comprehensive evaluation framework is achieved.

5. Discussion

5.1. Discriminative Model

Crowdsourcing contests generate a large volume of submissions, making manual evaluation impractical due to time constraints. The discriminative model addresses this challenge by efficiently filtering out lower-quality submissions while preserving the majority of winning entries. By eliminating up to 50% of submissions, human evaluators can focus on projects with the highest potential rather than being overwhelmed by the sheer number of entries. This targeted approach ensures that contest organizers maintain a high standard of evaluation without the excessive burden of reviewing every submission in detail.

The Discriminative Evaluator classifies submissions based on the model's numerical ranking. Each contest's top 20% of submissions are labeled as Excellent, the next 40% as Average, and the bottom 40% as Poor. Both the filtering threshold and classification cutoffs can be adjusted depending on contest characteristics. When the general standard of submissions is high, the filtering threshold can be lowered to retain more entries for further assessment. Conversely, in contests dominated by lower-quality submissions, a more aggressive filtering strategy can be applied to prioritize efficiency.

The classification results confirm that the discriminative model effectively ranks high-quality submissions above weaker ones. The clear correlation between classification and prize-winning status, as shown in Figure 4.2, indicates that the model successfully identifies well-documented and structured submissions. The relatively low number of Poor classifications among Top Prize and Other Prize submissions further reinforces its reliability as a prescreening mechanism. However, the rigid classification structure—where 40% of submissions are categorized as Poor regardless of the contest's overall quality—presents a limitation. Future improvements could explore adaptive filtering thresholds to better accommodate variations in contest difficulty and submission quality, ensuring a more tailored and efficient evaluation process.

The dataset used in this thesis encompasses contests with varying characteristics, including differences in submission volume, average documentation length, and default win rates. Each submission contains many structured quantitative features providing a measurable basis for evaluation.

While deeper models with more parameters could, in theory, capture intricate patterns across these features, the results indicate that increased complexity often leads to a decline in recall. Overly flexible models risk overfitting to anomalies rather than recognizing broader trends, reducing their generalizability across different contests.

For example, a winning submission with only 600 words, no images, but 484 lines of code could mislead a highly complex model into assuming that *low-documentation, high-code* projects are strong candidates. Such a localized pattern does not generalize well, as the majority of winning projects feature comprehensive documentation alongside technical depth. Simpler models, by contrast, focus on overarching trends—such as the correlation between extensive documentation and submission success—without being misled by outliers. This makes them more reliable in large-scale screening tasks.

A key reason why simpler models perform better lies in the relative homogeneity of features that correlate with high-quality submissions. While individual features such as image count or documentation length vary, there is an underlying pattern that well-documented submissions with additional media tend to perform better. A straightforward classifier, such as a shallow MLP, logistic regression, or linear SVM, effectively captures this relationship. More complex models attempt to account for outliers and edge cases, often at the expense of general recall. Given that contests range from small-scale challenges with fewer than 15 entries to large competitions exceeding 300 submissions, the variability in winning criteria further complicates the learning process. A deeper neural network or an ensemble of decision trees may learn rules that are only applicable to specific contest conditions, failing to generalize effectively across different datasets. In contrast, simpler models maintain a stable decision boundary, ensuring higher recall when applied to entirely new contests.

The findings indicate that minimalist model configurations deliver the best recall performance for large-scale screening. While deeper networks and more intricate ensembles were tested, their performance deteriorated due to excessive sensitivity to uncommon submission features. Simpler approaches, which emphasize high-level trends such as submission length, media inclusion, and structural completeness, consistently demonstrated higher recall. Given that filtering out the lowest-ranked 50% of submissions preserves approximately 95% of actual winners, this strategy effectively maximizes recall while ensuring that promising projects are retained.

5.2. Generative Model

The generative model evaluates contest submissions by assessing their novelty and usefulness. Unlike the discriminative model, which relies on structured numerical indicators the generative model performs a qualitative analysis. Since both models operate independently, they specialize in their respective tasks without influencing each other. The discriminative model identifies well-documented submissions, while the generative model determines their conceptual merit, ensuring that a solution is not only well-presented but also innovative and practically valuable.

The generative model's classification results, shown in Table 4.9, indicate that it generally aligns with contest outcomes but applies a more lenient evaluation framework compared to the discriminative model. Notably, among Top Prize submissions, two-thirds were classified as Excellent, suggesting that the model effectively recognizes highly novel and useful solutions. However, a key difference emerges in the frequency of Poor classifications. The generative model assigns significantly fewer submissions to this category, reflecting its qualitative assessment approach, which often finds at least some value in most entries. Unlike the discriminative model, which imposes predefined ranking thresholds, the generative model produces more interpretative evaluations, sometimes leading to less distinct classification boundaries.

Evaluating qualitative outputs presents unique challenges compared to the more structured metrics used in the discriminative model. While recall, precision, and accuracy provide clear performance indicators for classification tasks, the generative model produces textual reasoning, making direct evaluation more complex. Consistency is a major challenge, as the same submission may receive slightly different responses due to prompt variations and model randomness. Interpretability also poses difficulties, since textual assessments require human judgment to determine their validity, unlike numerical scores that offer direct comparability. Additionally, LLMs initially exhibited a tendency to overestimate novelty and usefulness, requiring explicit constraints in the prompt to enforce critical assessments.

To address these challenges, few-shot prompting was employed instead of fine-tuning. This approach enhances adaptability and computational efficiency, allowing the model to generalize to new contest submissions without the need for retraining. Research has demonstrated that few-shot prompting enables LLMs to generalize effectively to new tasks without requiring extensive retraining (Brown et al., 2020; Gao et al., 2021). In the context of contest evaluation, where submissions vary significantly, the ability to adjust evaluation criteria through example modifications rather than retraining is particularly advantageous.

Despite these advantages, few-shot prompting introduces its own set of challenges. Smaller models struggled to balance few-shot examples with the rest of the prompt, occasionally misinterpreting the provided examples as new submissions. Variability in model responses made it difficult to isolate improvements caused by prompt refinements from the natural fluctuations in LLM outputs. Some models, such as DeepSeek, demonstrated an initial tendency to assign overly generous evaluations, requiring additional constraints to ensure a more critical assessment. Iterative prompt refinements, combined with explicit instructions to consider both strengths and limitations, helped to improve classification consistency and mitigate excessive leniency.

Contest-specific adaptations further enhanced the generative model's accuracy, particularly in domains where contextual understanding is essential. Without such modifications, the model tended to produce generic evaluations that lacked insight into the domain-specific challenges of a submission. For example, in an assistive technology contest, a general prompt might recognize that a submission integrates multiple components but fail to assess its relevance within the accessibility field. A tailored prompt, in contrast, explicitly compares the submission against existing assistive technologies such as smart home systems and adaptive wearable devices. This adaptation leads to a more precise analysis of whether a solution introduces meaningful functionality and places greater emphasis on usability factors relevant to mobility-impaired users. It also improves the justification for classification decisions by ensuring that the model considers real-world application constraints rather than solely focusing on general innovation.

By dynamically adjusting prompts and few-shot examples, the generative model produces evaluations that are both more detailed and more aligned with the contest's focus. The ability to refine assessments through structured reasoning allows for a more balanced evaluation of each submission, ensuring that high-quality ideas are recognized even when their documentation is less extensive.

5.3. Combination of Models

Both models contribute distinct but complementary strengths to the evaluation process. The discriminative model excels at identifying structural indicators of strong submissions—such as longer descriptions, multimedia inclusion, and comprehensive technical documentation—while efficiently ranking and categorizing entries. The generative model complements this by providing deeper assessments of novelty and usefulness, capturing qualitative aspects that purely numerical features cannot address. While the filtering process could be applied before generative evaluation, at this stage, all submissions were assessed by both models independently.

By combining both models, the evaluation process benefits from a layered approach. The discriminative model ensures that submissions with strong documentation and technical detail are recognized, while the generative model identifies highly novel and impactful ideas, even if they are not as well-documented. This dual perspective minimizes the risk of prematurely discarding promising projects while maintaining efficiency in screening large volumes of submissions. The ability of both models to classify independently while still aligning on high-quality submissions suggests that their integration offers a balanced framework optimizing reviewer effort.

5.4. Future Work

While the proposed evaluation framework effectively combines discriminative and generative models to assess contest submissions, several possibilities exist for further refinement and extension. Future work could focus on improving the generative model's reasoning capabilities, enhancing the adaptability of the evaluation process, incorporating multi-modal assessment, and expanding the framework's applicability beyond technical innovation contests.

One key direction for improvement lies in refining the generative model. Its ability to assess novelty and usefulness remains highly dependent on prompt design and the selection of few-shot examples. Future research could explore expanding the framework to support multi-modal evaluation. Additionally, the use of larger language models with more parameters could further improve evaluation. Another promising approach could be the development of an interactive chatbot that enables human evaluators to query the model directly about specific aspects of a submission, facilitating more nuanced and responsive assessments. Also worth investigating is the trade-off between fine-tuning and few-shot prompting. While few-shot learning was chosen for its adaptability and computational efficiency, fine-tuning may provide more stable and contest-specific assessments. A systematic comparison of these approaches across different contest types could yield insights into their respective advantages.

While the generative model provides structured reasoning, its alignment with expert human evaluations has not yet been systematically analyzed. Future research could compare model-generated assessments with those of experienced judges to identify areas where LLM evaluations diverge from human reasoning. This would provide insights into potential biases in LLM-generated assessments and inform refinements to improve the model's reliability.

A further extension of this work could involve integrating real-time feedback mechanisms into the contest submission process. Currently, the models operate in a post-submission evaluation setting, but future iterations could provide real-time feedback to participants as they draft their entries. Rather than suggesting what participants should do or proposing entirely new ideas, the system should focus on highlighting the strengths of what has already been done, offering insights into which aspects are particularly well-executed. Additionally, it could guide contestants by indicating which directions are most promising for further refinement and which aspects may not contribute significantly to the overall quality of the submission. By providing structured feedback on documentation completeness, novelty, and usability without imposing prescriptive recommendations, this enhancement could help contestants make more informed decisions while maintaining creative autonomy. Such a system has the potential to improve the overall quality of contest entries while also reducing the burden on human evaluators.

These future directions present valuable opportunities to enhance AI-assisted contest evaluation. Refining model assessments, incorporating multi-modal capabilities, expanding to new domains, and enabling real-time feedback could make the evaluation process more efficient, accurate, and fair.

6. Conclusion

Automated evaluation of crowdsourcing contest submissions presents a significant challenge due to the large volume of entries and the need to fairly assess both well-documented and highly innovative solutions. This thesis demonstrated that a combination of discriminative and generative AI models can effectively address this challenge by providing complementary insights into submission quality. The discriminative model efficiently filters out lower-ranked submissions based on structured quantitative features, while the generative model evaluates the novelty and usefulness of each solution.

The classification analyses revealed consistent trends across both models, highlighting their ability to distinguish between varying quality levels of submissions. Notably, both models exhibited a pattern of assigning higher evaluations to winning submissions, reinforcing their alignment with prize outcomes. Since all submissions were evaluated by both models without pre-filtering, further optimizations could improve computational efficiency. Future implementations may apply filtering before generative evaluation to reduce the number of processed submissions while ensuring that innovative projects are not prematurely excluded. The discriminative model's ability to remove a large portion of lower-ranked submissions with minimal impact on prize-winning entries suggests that such filtering could be effective.

The classification results also highlighted a key limitation of the generative model: it rarely assigned *Poor* classifications, favoring *Average* even for weaker submissions. This suggests that the model tends to recognize some level of value in most entries, which could introduce a bias toward leniency. Future refinements could involve stricter calibration, additional evaluation criteria, or dynamic prompt adjustments to ensure that truly weak submissions are identified more reliably.

Beyond classification, contest-specific prompt adaptations significantly improved the reasoning capabilities of the generative model. When primed with domain-relevant instructions and tailored few-shot examples, the model produced more precise and meaningful evaluations compared to a general prompt. This refinement ensures that novelty and usefulness are assessed within the appropriate context, reducing the likelihood of generic or irrelevant responses.

Few-shot prompting played a crucial role in calibrating the generative model, helping standardize its output format and improving consistency. However, limitations remain. No large-scale LLMs were used, meaning that the reasoning capabilities of smaller models may still be constrained. Additionally, human evaluators were only involved in the form of binary winner classifications, preventing a direct assessment of how well AI-generated reasoning aligns with expert judgment.

Despite its advantages, AI-driven evaluation presents certain risks. Automated systems may overlook submissions that are truly innovative but lack sufficient documentation. Enhancing evaluation fairness is an important area for future development, particularly through the integration of multimodal analysis. Allowing the models to assess not only textual descriptions but also images, CAD files, and videos could provide a more comprehensive and equitable evaluation process.

Expanding the model's contextual understanding of submissions would further enhance evaluation accuracy. Providing additional metadata or developing interactive chatbots that allow human reviewers to query model insights in real time could contribute to more refined assessments. With these advancements, AI-assisted contest evaluation could become an even more effective and reliable tool, significantly reducing human workload while preserving fairness and accuracy in the selection of top submissions.

Bibliography

Allan Afuah and Christopher L. Tucci. Crowdsourcing As a Solution to Distant Search. *Academy of Management Review*, 37(3):355–375, July 2012. ISSN 0363-7425. doi: 10.5465/amr.2010.0146. URL https://journals.aom.org/doi/abs/10.5465/amr.2010.0146. Publisher: Academy of Management.

Ajay Agrawal, Joshua Gans, and Avi Goldfarb. *The Economics of Artificial Intelligence: An Agenda*. University of Chicago Press, June 2019. ISBN 978-0-226-61347-5. Google-Books-ID: 4GyVDwAAQBAJ.

Callen Anthony, Beth A. Bechky, and Anne-Laure Fayard. "Collaborating" with AI: Taking a System View to Explore the Future of Work. *Organization Science*, 34(5):1672–1694, September 2023. ISSN 1047-7039. doi: 10.1287/orsc.2022.1651. URL https://pubsonline.informs.org/doi/full/10.1287/orsc.2022.1651. Publisher: INFORMS.

J. Jason Bell, Christian Pescher, Gerard J. Tellis, and Johann Füller. Can AI Help in Ideation? A Theory-Based Model for Idea Screening in Crowdsourcing Contests. *Marketing Science*, 43 (1):54–72, April 2023. ISSN 0732-2399. doi: 10.1287/mksc.2023.1434. URL https://pubsonline.informs.org/doi/abs/10.1287/mksc.2023.1434. Publisher: INFORMS.

Kevin J Boudreau and Karim R Lakhani. Using the crowd as an innovation partner. *Harvard business review*, 91(4):60–9, 140, April 2013. ISSN 0017-8012.

Kevin J. Boudreau, Nicola Lacetera, and Karim R. Lakhani. Incentives and Problem Uncertainty in Innovation Contests: An Empirical Analysis. *Management Science*, 57(5):843–863, May 2011. ISSN 0025-1909. doi: 10.1287/mnsc.1110.1322. URL https://pubsonline.informs.org/doi/abs/10.1287/mnsc.1110.1322. Publisher: INFORMS.

Kevin J. Boudreau, Karim R. Lakhani, and Michael Menietti. Performance responses to competition across skill levels in rank-order tournaments: field evidence and implications for tournament design. *The RAND Journal of Economics*, 47(1):140–165, 2016. ISSN 1756-2171. doi: 10.1111/1756-2171.12121. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/1756-2171.12121. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/1756-2171.12121.

Sebastian G. Bouschery, Vera Blazevic, and Frank T. Piller. Augmenting human innovation teams with artificial intelligence: Exploring transformer-based language models. *Journal of Product Innovation Management*, 40(2):139–153, 2023. ISSN 1540-5885. doi: 10.1111/jpim.12656. URL `https://onlinelibrary.wiley.com/doi/abs/10.1111/jpim.12656`. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/jpim.12656.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020. URL `https://proceedings.neurips.cc/paper_files/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html`.

Erik Brynjolfsson and Andrew McAfee. *The Second Machine Age: Work, Progress, and Prosperity in a Time of Brilliant Technologies*. W. W. Norton & Company, January 2014. ISBN 978-0-393-23935-5. Google-Books-ID: WiKwAgAAQBAJ.

N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. SMOTE: Synthetic Minority Oversampling Technique. *Journal of Artificial Intelligence Research*, 16:321–357, June 2002. ISSN 1076-9757. doi: 10.1613/jair.953. URL `https://www.jair.org/index.php/jair/article/view/10302`.

Liang Chen and De Liu. Comparing strategies for winning expert-rated and crowd-rated crowdsourcing contests: First findings. *AMCIS 2012 Proceedings - All Submissions*, 1, January 2012.

Pei-Yu Chen, Paul Pavlou, Shinyi Wu, and Yang Yang. Attracting High-Quality Contestants to Contest in the Context of Crowdsourcing Contest Platform. *Production and Operations Management*, 30(6):1751–1771, June 2021. ISSN 1059-1478. doi: 10.1111/poms.13340. URL `https://doi.org/10.1111/poms.13340`. Publisher: SAGE Publications.

Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794, San Francisco California USA, August 2016. ACM. ISBN 978-1-4503-4232-2. doi: 10.1145/2939672.2939785. URL `https://dl.acm.org/doi/10.1145/2939672.2939785`.

Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, September 1995. ISSN 0885-6125, 1573-0565. doi: 10.1007/BF00994018. URL `http://link.springer.com/10.1007/BF00994018`.

Felipe A. Csaszar and Tom Steinberger. Organizations as Artificial Intelligences: The Use of Artificial Intelligence Analogies in Organization Theory. *Academy of Management Annals*, 16(1):1–37, January 2022. ISSN 1941-6520. doi: 10.5465/annals.2020.0192. URL https://journals.aom.org/doi/abs/10.5465/annals.2020.0192. Publisher: Academy of Management.

Felipe A. Csaszar, Harsh Ketkar, and Hyunjin Kim. Artificial Intelligence and Strategic Decision-Making: Evidence from Entrepreneurs and Investors. *Strategy Science*, 9(4):322–345, December 2024. ISSN 2333-2050. doi: 10.1287/stsc.2024.0190. URL https://pubsonline.informs.org/doi/full/10.1287/stsc.2024.0190. Publisher: INFORMS.

Anil R. Doshi, J. Jason Bell, Emil Mirzayev, and Bart S. Vanneste. Generative artificial intelligence and evaluating strategic decisions. *Strategic Management Journal*, 46(3):583–610, 2025. ISSN 1097-0266. doi: 10.1002/smj.3677. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/smj.3677. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smj.3677.

Ida Merete Enholm, Emmanouil Papagiannidis, Patrick Mikalef, and John Krogstie. Artificial Intelligence and Business Value: a Literature Review. *Information Systems Frontiers*, 24(5): 1709–1734, October 2022. ISSN 1572-9419. doi: 10.1007/s10796-021-10186-w. URL https://doi.org/10.1007/s10796-021-10186-w.

Johann Füller, Katja Hutter, Julia Hautz, and Kurt Matzler. User Roles and Contributions in Innovation-Contest Communities. *Journal of Management Information Systems*, 31(1):273–308, July 2014. ISSN 0742-1222, 1557-928X. doi: 10.2753/MIS0742-1222310111. URL https://www.tandfonline.com/doi/full/10.2753/MIS0742-1222310111.

Tianyu Gao, Adam Fisch, and Danqi Chen. Making Pre-trained Language Models Better Few-shot Learners, June 2021. URL http://arxiv.org/abs/2012.15723. arXiv:2012.15723 [cs].

Carl Friedrich Gauss and Charles Henry Davis. *Theory of the Motion of the Heavenly Bodies Moving about the Sun in Conic Sections: A Translation of Gauss's "Theoria Motus." With an Appendix*. Little, Brown, 1857. Google-Books-ID: I37LpyiNRloC.

Taher Ahmed Ghaleb, Daniel Alencar da Costa, and Ying Zou. On the Popularity of Internet of Things Projects in Online Communities. *Information Systems Frontiers*, 24(5):1601–1634, October 2022. ISSN 1572-9419. doi: 10.1007/s10796-021-10157-1. URL https://doi.org/10.1007/s10796-021-10157-1.

Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, USA, 2nd edition, June 1998. ISBN 978-0-13-273350-2.

Tianyu He, Marco S. Minervini, and Phanish Puranam. How Groups Differ from Individuals in Learning from Experience: Evidence from a Contest Platform. *Organization Science*, 35(4): 1512–1534, July 2024. ISSN 1047-7039. doi: 10.1287/orsc.2021.15239. URL `https://pubsonline.informs.org/doi/full/10.1287/orsc.2021.15239`. Publisher: INFORMS.

Tin Kam Ho. Random decision forests. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, volume 1, pages 278–282 vol.1, August 1995. doi: 10.1109/ICDAR.1995.598994. URL `https://ieeexplore.ieee.org/document/598994/?arnumber=598994`.

Jeff Howe. The Rise of Crowdsourcing. (14), 2006.

Lars Bo Jeppesen and Karim R. Lakhani. Marginality and Problem-Solving Effectiveness in Broadcast Search. *Organization Science*, 21(5):1016–1033, October 2010. ISSN 1047-7039, 1526-5455. doi: 10.1287/orsc.1090.0491. URL `https://pubsonline.informs.org/doi/10.1287/orsc.1090.0491`.

Julian Just, Thomas Ströhle, Johann Füller, and Katja Hutter. AI-based novelty detection in crowdsourced idea spaces. *Innovation*, 26(3):359–386, July 2024. ISSN 1447-9338, 2204-0226. doi: 10.1080/14479338.2023.2215740. URL `https://www.tandfonline.com/doi/full/10.1080/14479338.2023.2215740`.

Tat Koon Koh. Adopting Seekers' Solution Exemplars in Crowdsourcing Ideation Contests: Antecedents and Consequences. *Information Systems Research*, 30(2):486–506, June 2019. ISSN 1047-7047, 1526-5536. doi: 10.1287/isre.2018.0810. URL `https://pubsonline.informs.org/doi/10.1287/isre.2018.0810`.

Shane Legg and Marcus Hutter. A Collection of Definitions of Intelligence, June 2007. URL `http://arxiv.org/abs/0706.3639`. arXiv:0706.3639 [cs].

Hila Lifshitz-Assaf. Dismantling Knowledge Boundaries at NASA: The Critical Role of Professional Identity in Open Innovation. *Administrative Science Quarterly*, 63(4):746–782, December 2018. ISSN 0001-8392. doi: 10.1177/0001839217747876. URL `https://doi.org/10.1177/0001839217747876`. Publisher: SAGE Publications Inc.

Tracy Xiao Liu, Jiang Yang, Lada A. Adamic, and Yan Chen. Crowdsourcing with All-Pay Auctions: A Field Experiment on Taskcn. *Management Science*, 60(8):2020–2037, August 2014. ISSN 0025-1909. doi: 10.1287/mnsc.2013.1845. URL `https://pubsonline.informs.org/doi/abs/10.1287/mnsc.2013.1845`. Publisher: INFORMS.

Giuseppe Marra, Sebastijan Dumančić, Robin Manhaeve, and Luc De Raedt. From Statistical Relational to Neurosymbolic Artificial Intelligence: a Survey, January 2024. URL `http://arxiv.org/abs/2108.11451`. arXiv:2108.11451 [cs].

Pamela McCorduck and Cli Cfe. *Machines Who Think: A Personal Inquiry into the History and Prospects of Artificial Intelligence*. A K Peters/CRC Press, New York, 2 edition, March 2004. ISBN 978-0-429-25898-5. doi: 10.1201/9780429258985.

Alex Murray, Jen Rhymer, and David G. Sirmon. Humans and Technology: Forms of Conjoined Agency in Organizations. *Academy of Management Review*, 46(3):552–571, July 2021. ISSN 0363-7425, 1930-3807. doi: 10.5465/amr.2019.0186. URL `http://journals.aom.org/doi/10.5465/amr.2019.0186`.

Abhishek Nagaraj and Henning Piezunka. The Divergent Effect of Competition on Platforms: Deterring Recruits, Motivating Converts. *Strategy Science*, 9(3):277–296, September 2024. ISSN 2333-2050. doi: 10.1287/stsc.2022.0125. URL `https://pubsonline.informs.org/doi/full/10.1287/stsc.2022.0125`. Publisher: INFORMS.

Richard R. Nelson and Sidney G. Winter. The Schumpeterian Tradeoff Revisited. *The American Economic Review*, 72(1):114–132, 1982. ISSN 0002-8282. URL `https://www.jstor.org/stable/1808579`. Publisher: American Economic Association.

Charles A. O'Reilly. Individuals and Information Overload in Organizations: Is More Necessarily Better? *Academy of Management Journal*, 23(4):684–696, December 1980. ISSN 0001-4273. doi: 10.5465/255556. URL `https://journals.aom.org/doi/abs/10.5465/255556`. Publisher: Academy of Management.

Sanghyun Park, Henning Piezunka, and Linus Dahlander. Coevolutionary Lock-In in External Search. *Academy of Management Journal*, 67(1):262–288, February 2024. ISSN 0001-4273. doi: 10.5465/amj.2022.0710. URL `https://journals.aom.org/doi/abs/10.5465/amj.2022.0710`. Publisher: Academy of Management.

Henning Piezunka and Linus Dahlander. Distant Search, Narrow Attention: How Crowding Alters Organizations' Filtering of Suggestions in Crowdsourcing. *Academy of Management Journal*, 58(3):856–880, June 2015. ISSN 0001-4273. doi: 10.5465/amj.2012.0458. URL `https://journals.aom.org/doi/abs/10.5465/amj.2012.0458`. Publisher: Academy of Management.

Henning Piezunka and Linus Dahlander. Idea Rejected, Tie Formed: Organizations' Feedback on Crowdsourced Ideas. *Academy of Management Journal*, 62(2):503–530, April 2019. ISSN 0001-4273. doi: 10.5465/amj.2016.0703. URL `https://journals.aom.org/doi/abs/10.5465/amj.2016.0703`. Publisher: Academy of Management.

Sebastian Raisch and Sebastian Krakowski. Artificial Intelligence and Management: The Automation–Augmentation Paradox. *Academy of Management Review*, 46:192–210, January 2021. doi: 10.5465/amr.2018.0072.

Josef Schumpeter. The Present World Depression: A Tentative Diagnosis. *The American Economic Review*, 21(1):179–182, 1931. ISSN 0002-8282. URL `https://www.jstor.org/stable/1802985`. Publisher: American Economic Association.

Shu-Hsien Liao. Expert system methodologies and applications—a decade review from 1995 to 2004. *Expert Systems with Applications*, 28(1):93–103, January 2005. ISSN 0957-4174. doi: 10.1016/j.eswa.2004.08.003. URL `https://www.sciencedirect.com/science/article/pii/S0957417404000934`.

Herbert Alexander Simon. *The Shape of Automation for Men and Management*. Harper & Row, 1965. Google-Books-ID: P01VAAAAMAAJ.

Christian Terwiesch and Yi Xu. Innovation Contests, Open Innovation, and Multiagent Problem Solving. *Management Science*, 54(9):1529–1543, September 2008. ISSN 0025-1909. doi: 10.1287/mnsc.1080.0884. URL `https://pubsonline.informs.org/doi/abs/10.1287/mnsc.1080.0884`. Publisher: INFORMS.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL `https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html`.

Ann-Kristin Weiser and Georg von Krogh. Artificial intelligence and radical uncertainty. *European Management Review*, 20(4):711–717, 2023. ISSN 1740-4762. doi: 10.1111/emre.12630. URL `https://onlinelibrary.wiley.com/doi/abs/10.1111/emre.12630`. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/emre.12630.

Pui Ying Wong. Evaluating Innovation with AI. Master's thesis, University of Zurich, September 2024.

Yang Yang, Pei-Yu Chen, and Paul Pavlou. Open Innovation: An Empirical Study of Online Contests. 2009.

Jonathan (Hua) Ye and Matthew Jensen. Effects of introducing an online community in a crowdsourcing contest platform. *Information Systems Journal*, 32(6):1203–1230, 2022. ISSN 1365-2575. doi: 10.1111/isj.12397. URL `https://onlinelibrary.wiley.com/doi/abs/10.1111/isj.12397`. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/isj.12397.

A. Appendix

This appendix contains the complete code implementations used throughout the thesis. It includes the core model framework, data gathering processes, and preprocessing scripts essential for contest evaluation and analysis.

A.1. Model Framework

This section provides a complete listing of the classes referenced in Section 3.6. These implementations are essential components of the experimental setup, supporting data preprocessing, model execution, and evaluation processes.

- `DriveManager`: Handles Google Drive operations for dataset loading and result storage.
- `ModelLoader`: Downloads and configures LLMs from Hugging Face, optimizing execution with half-precision settings.
- `DiscriminativeModel`: Trains and evaluates machine learning models for predicting submission quality. It manages dataset splitting, cross-validation, and performance evaluation using a recall-based metric. The model outputs a numerical score between 0 and 1 for each submission.
- `DiscriminativeEvaluator`: Converts numerical model outputs into text by comparing each submission's quantitative attributes within its contest.
- Machine Learning Models:
  - `SimpleMLP`: A fully connected neural network implemented with PyTorch Lightning. It processes submission features and predicts their quality using a multi-layer perceptron.
  - `OptimizedModelWrapper`: A flexible wrapper that trains and optimizes multiple machine learning models, including Random Forest, XGBoost, SVM, and Logistic Regression, using hyperparameter tuning.

  - `EnhancedMLP`: An improved neural network model incorporating dropout, batch normalization, and a learning rate scheduler to enhance prediction stability and accuracy.

- `DataCleaner`: Prepares textual data by normalizing formatting and enforcing token limits.
- `Summarizer`: Summarizes long contest descriptions while preserving essential details.
- `SubmissionEvaluator`: Constructs structured prompts and evaluates individual submissions based on novelty and usefulness.
- `ContestEvaluator`: Orchestrates the complete evaluation process, executing both generative and discriminative assessments for all submissions in a contest and storing structured results.

A.1.1. Drive Manager

```python
class DriveManager:
    """
    Class Name: DriveManager

    Purpose:
        - Handle mounting Google Drive in a Colab environment.
        - Provide convenient path handling if needed.

    Responsibilities:
        - Mount/unmount Google Drive.
        - Potentially provide standard path resolutions for loading/saving.

    Example Usage:
        drive_mgr = DriveManager()
        drive_mgr.mount_drive()
    """

    def __init__(self, mount_path="/content/drive/"):
        """
        Constructor for DriveManager.

        Args:
            mount_path (str): The path at which to mount Google Drive.

        Attributes:
            mount_path (str): Where Google Drive is mounted in Colab.
        """
        self.mount_path = mount_path

    def mount_drive(self, force_remount=True):
        """
        Mounts Google Drive using the provided mount path.

        Args:
            force_remount (bool): Whether to force-remount if already mounted.
        """
        from google.colab import drive
```

```
        drive.mount(self.mount_path, force_remount=force_remount)
```

Listing A.1: Drive Manager Python Code

A.1.2. Model Loader

```
class ModelLoader:
    """
    Class Name: ModelLoader

    Purpose:
        - Load and configure a Hugging Face model and tokenizer.
        - Provide a convenient method for text generation.

    Responsibilities:
        - Download/load a tokenizer and model from a given model name.
        - Handle device placement (CPU/GPU) and half-precision casting if desired.
        - Provide a generate_text() method for inference.

    Example Usage:
        model_loader = ModelLoader("meta-llama/Llama-3.1-8B-Instruct")
        model_loader.load_model_and_tokenizer()
        response = model_loader.generate_text(prompt, max_length=200)
    """

    def __init__(self, model_name, device=None):
        """
        Constructor for ModelLoader.

        Args:
            model_name (str): The name or path of the model on Hugging Face.
            device (str, optional): The device to load model onto (e.g., "cuda" or "cpu").
                                    Defaults to "cuda" if available, else "cpu".

        Attributes:
            model_name (str): The Hugging Face model name.
            tokenizer (AutoTokenizer): The loaded tokenizer (populated after
                load_model_and_tokenizer()).
            model (AutoModelForCausalLM): The loaded model (populated after
                load_model_and_tokenizer()).
            device (str): The device for model inference.
        """
        self.model_name = model_name
        self.tokenizer = None
        self.model = None
        self.device = device if device else ("cuda" if torch.cuda.is_available() else "cpu")

    def load_model_and_tokenizer(self):
        """
        Loads the tokenizer and model from Hugging Face.
        Casts the model to half-precision if the device is GPU.
        """
        print(f"Loading model: {self.model_name} on device {self.device}")
        self.tokenizer = AutoTokenizer.from_pretrained(self.model_name)
        self.model = AutoModelForCausalLM.from_pretrained(
```

```
            self.model_name,
            device_map="auto", # if self.device == "cuda" else None,
            torch_dtype=torch.float16 # if self.device == "cuda" else torch.float32
        ).to(self.device)

    def generate_text(self, prompt, max_length=512, temperature=0.7, stop_token_ids=None):
        """
        Generates text using the loaded model.

        Args:
            prompt (str): The input text/prompt for generation.
            max_length (int): Maximum number of tokens for the output.
            temperature (float): Sampling temperature.

        Returns:
            str: The generated text from the model.
        """
        if self.tokenizer is None or self.model is None:
            raise ValueError("Tokenizer/Model not loaded. Call load_model_and_tokenizer() first
                .")

        inputs = self.tokenizer(prompt, return_tensors="pt").to(self.device)

        outputs = self.model.generate(
            inputs["input_ids"],
            max_length=max_length,
            temperature=temperature,
            attention_mask=inputs["attention_mask"],
        )
        return self.tokenizer.decode(outputs[0], skip_special_tokens=False)
```

Listing A.2: Model Loader Python Code

A.1.3. Discriminative Model

```
class DiscriminativeModel:
    """
    Class Name: DiscriminativeModel

    Purpose:
        - Orchestrate model training (across multiple runs/folds) and evaluate performance.
        - Provide a high-level interface to:
            1. Split data into train/val/test sets.
            2. Train either PyTorch Lightning models (e.g., SimpleMLP) or scikit-learn models
               (wrapped in OptimizedModelWrapper).
            3. Evaluate on a test set using a submission-based recall metric.
            4. (Optionally) Compute and save false negatives, all submissions with model-
                generated scores,
               and summary metrics (Accuracy, Recall, TN, FP, FN, TP).

    Responsibilities:
        - Manage K-Fold logic, train/val/test splitting, and repeated runs.
        - Contain a method to perform evaluation on a given test set (`submission_evaluation`).
        - Contain a method (`run_evaluation`) that orchestrates multiple runs/folds and then
          aggregates fold-level metrics into a final run-level table.
```

```

    Example Usage:
        discriminative_model = DiscriminativeModel(df, output_path="/content/drive/MyDrive
            /...")
        discriminative_model.run_evaluation(n_runs=5, n_folds=5, model_type="LogisticRegression
            ", dataset_name="28_11.csv")
    """

    def __init__(self, df, output_path):
        """
        Constructor for DiscriminativeModel.

        Args:
            df (pd.DataFrame): The full dataset.
            output_path (str): Where to store evaluation CSVs.
        """
        self.df = df
        self.output_path = output_path

    def submission_evaluation(
        self,
        model,
        test_df,
        run,
        fold,
        dataset_name,
        model_type,
        save_contest_results=False,
        save_false_negatives=False,
        save_all_submissions=False,
        compute_additional_metrics=True
    ):
        """
        Evaluate a trained model on the test set by filtering out various percentages of
            submissions.
        Optionally save false negatives and all submissions with model scores.
        Optionally compute additional per-filter metrics (Accuracy, Recall, TN, FP, FN, TP).

        Args:
            model: The trained model (could be PyTorch Lightning or sklearn).
            test_df (pd.DataFrame): DataFrame containing the test data.
            run (int): Current run number.
            fold (int): Current fold index.
            dataset_name (str): Name of the dataset (used for saving results).
            model_type (str): Type of the model ('MLP', 'LogisticRegression', 'RandomForest', '
                XGBoost', 'SVM').
            save_contest_results (bool): If True, save per-contest results to CSV.
            save_false_negatives (bool): If True, save the false negatives to CSV.
            save_all_submissions (bool): If True, save all submissions with scores to CSV.
            compute_additional_metrics (bool): If True, compute & summarize Accuracy, Recall,
                TN, FP, FN, TP for each filter.

        Returns:
            (df_results, accuracy, overall_recall, conf_matrix, metrics_summary_df,
                fold_metrics_data):
                - df_results (pd.DataFrame): Per-contest results with recall at each filter
                    percentage.
                - accuracy (float or None): Overall accuracy at 50% filtering (or None if no
                    data).
```

```
                - overall_recall (float or None): Overall recall at 50% filtering (or None if no data).
                - conf_matrix (np.array or None): Confusion matrix at 50% filtering (or None if no data).
                - metrics_summary_df (pd.DataFrame or None): Additional metrics at each filter (this fold).
                - fold_metrics_data (dict): Raw fold-level sums/averages that can be aggregated across folds.
        """
        print(f"----- Testset Evaluation by filtering out varying percentages of submissions for each contest -----")
        filter_percentages = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7]

        # Lists/dicts to store results
        per_contest_results = []
        false_negatives = []
        all_submissions = []


        metrics_dict = {
            p: {"all_preds": [], "all_labels": []}
            for p in filter_percentages
        }

        # Group the test set by 'contest_name'
        for contest_name, contest_data in test_df.groupby('contest_name'):

            # Extract the 'prizes_sum' information for the contest
            prizes_sum = contest_data['prizes_sum'].iloc[0] if 'prizes_sum' in contest_data.columns else None

            # Convert into dataset
            dataset = CSVDataset(contest_data, smote=False)
            x = torch.tensor(dataset.x, dtype=torch.float)
            y = torch.tensor(dataset.y, dtype=torch.float)

            total_submissions = len(y)
            total_winners = int(y.sum().item())

            if total_submissions == 0:
                continue

            contest_info = {
                'contest_name': contest_name,
                'prizes_sum': prizes_sum,
                'nr_winners': total_winners,
                'nr_submissions': total_submissions
            }

            # Generate predictions
            with torch.no_grad():
                if model_type == "MLP":
                    # MLP (PyTorch Lightning) forward
                    y_hat = model(x).squeeze()
                else:
                    # sklearn model or wrapper
                    if hasattr(model, "predict_proba"):
                        y_hat = torch.tensor(model.predict_proba(x.numpy())[:, 1])
                    else:
```

```
                        y_hat = torch.tensor(model.predict(x.numpy()))

            y_hat_flat = y_hat.flatten()
            sorted_probs, sorted_indices = torch.sort(y_hat_flat, descending=True)

            # Store quality scores for all submissions (for optional saving)
            for idx, submission in enumerate(contest_data.itertuples(index=False)):
                submission_info = submission._asdict()
                submission_info['quality_score'] = y_hat[idx].item()
                submission_info['model'] = fold + 1
                all_submissions.append(submission_info)

            # For each filter percentage, compute recall and store predictions
            for percentage in filter_percentages:
                keep_percentage = 1 - percentage
                top_n = int(keep_percentage * total_submissions)
                top_n = max(top_n, 1)

                top_indices = sorted_indices[:top_n]
                preds = torch.zeros_like(y_hat)
                preds[top_indices] = 1

                # Per-contest recall
                recall = recall_score(y.cpu().numpy(), preds.cpu().numpy())
                percentage_label = f'recall_{int(percentage*100)}%'
                contest_info[percentage_label] = recall


                if compute_additional_metrics:
                    metrics_dict[percentage]["all_preds"].append(preds)
                    metrics_dict[percentage]["all_labels"].append(y)

                if abs(percentage - 0.5) < 1e-9:
                    # Identify false negatives
                    for idx, (true_label, pred_label) in enumerate(zip(y, preds)):
                        if true_label == 1 and pred_label == 0:
                            fn_info = contest_data.iloc[idx].to_dict()
                            fn_info['quality_score'] = y_hat[idx].item()
                            fn_info['model'] = fold + 1
                            false_negatives.append(fn_info)

            per_contest_results.append(contest_info)

        # ----------------------------
        # Compute metrics at 50% filtering (fold-level)
        # ----------------------------
        p_50 = 0.5
        if len(metrics_dict[p_50]["all_preds"]) > 0:
            all_preds_50 = torch.cat(metrics_dict[p_50]["all_preds"]).cpu().numpy()
            all_labels_50 = torch.cat(metrics_dict[p_50]["all_labels"]).cpu().numpy()

            accuracy = accuracy_score(all_labels_50, all_preds_50)
            overall_recall = recall_score(all_labels_50, all_preds_50)
            conf_matrix = confusion_matrix(all_labels_50, all_preds_50)

            print("\n----- Overall Evaluation at 50% filtering (FOLD LEVEL) -----")
            print(f"Fold {fold+1} accuracy: {accuracy}")
            print(f"Fold {fold+1} recall: {overall_recall}")
            print(f"Fold {fold+1} Confusion Matrix:\n{conf_matrix}")
```

```
            print(f"Fold {fold+1} Remaining % size of Submissions: {(sum(all_preds_50) / len(all_preds_50)) * 100:.2f}%")
        else:
            accuracy = None
            overall_recall = None
            conf_matrix = None

        df_results = pd.DataFrame(per_contest_results)

        # -----------------------------
        # Save per-contest results CSV
        # -----------------------------
        if save_contest_results:
            output_csv = os.path.join(self.output_path, f'{model_type}_results_{dataset_name[:-4]}_run{run}.csv')
            file_exists = os.path.exists(output_csv)
            df_results.to_csv(output_csv, mode='a', header=not file_exists, index=False)
            print(f"\nPer-contest results (fold-level) saved to {output_csv}")

        # -----------------------------
        # Save false negatives if requested
        # -----------------------------
        if save_false_negatives:
            if false_negatives:
                false_negatives_df = pd.DataFrame(false_negatives)
                false_negatives_csv = os.path.join(self.output_path, f'{model_type}_fn_{dataset_name[:-4]}_run{run}.csv')
                file_exists = os.path.exists(false_negatives_csv)
                false_negatives_df.to_csv(false_negatives_csv, mode='a', header=not file_exists, index=False)
                print(f"False negatives saved to {false_negatives_csv}")
            else:
                print("No false negatives found at 50% filtering in this fold.")

        # -----------------------------
        # Save all submissions if requested
        # -----------------------------
        if save_all_submissions:
            if all_submissions:
                all_submissions_df = pd.DataFrame(all_submissions)
                all_submissions_csv = os.path.join(self.output_path, f'{model_type}_all_submissions_{dataset_name[:-4]}_run{run}.csv')
                file_exists = os.path.exists(all_submissions_csv)
                all_submissions_df.to_csv(all_submissions_csv, mode='a', header=not file_exists, index=False)
                print(f"All submissions with quality scores saved to {all_submissions_csv}")
            else:
                print("No submissions found to save for this fold.")

        # -----------------------------
        # Compute & return additional metrics across all filter percentages (fold-level)
        # -----------------------------
        metrics_summary_df = None
        fold_metrics_data = {}

        if compute_additional_metrics:
            all_metrics_rows = []
            for p in sorted(filter_percentages):
                preds_list = metrics_dict[p]["all_preds"]
```

```python
                labels_list = metrics_dict[p]["all_labels"]
                if len(preds_list) == 0:
                    continue

                all_preds_p = torch.cat(preds_list).cpu().numpy()
                all_labels_p = torch.cat(labels_list).cpu().numpy()

                # confusion matrix
                tn, fp, fn, tp = confusion_matrix(all_labels_p, all_preds_p).ravel()
                accuracy_p = (tp + tn) / (tp + tn + fp + fn)
                recall_p = tp / (tp + fn) if (tp + fn) > 0 else 0.0
                remain_subs_pct = (sum(all_preds_p) / len(all_preds_p)) * 100

                row = {
                    "Filter_Percentage": p,
                    "Remaining_Percentage_Size_of_Submissions": remain_subs_pct,
                    "Accuracy": accuracy_p,
                    "Recall": recall_p,
                    "True_Negatives": tn,
                    "False_Positives": fp,
                    "False_Negatives": fn,
                    "True_Positives": tp
                }
                all_metrics_rows.append(row)

            metrics_summary_df = pd.DataFrame(all_metrics_rows)
            print("\nFold-Level Additional Metrics (All Contests Combined) for each Filter %:")
            print(metrics_summary_df)

            # Prepare the fold_metrics_data to later be aggregated in run_evaluation
            fold_metrics_data = {
                p: {
                    "tn": 0, "fp": 0, "fn": 0, "tp": 0,
                    "acc_list": [], "recall_list": [], "remain_pct_list": []
                }
                for p in sorted(filter_percentages)
            }

            for _, row in metrics_summary_df.iterrows():
                p = row["Filter_Percentage"]
                fold_metrics_data[p]["tn"] += row["True_Negatives"]
                fold_metrics_data[p]["fp"] += row["False_Positives"]
                fold_metrics_data[p]["fn"] += row["False_Negatives"]
                fold_metrics_data[p]["tp"] += row["True_Positives"]
                fold_metrics_data[p]["acc_list"].append(row["Accuracy"])
                fold_metrics_data[p]["recall_list"].append(row["Recall"])
                fold_metrics_data[p]["remain_pct_list"].append(row["Remaining_Percentage_Size_of_Submissions"])

        return (
            df_results,
            accuracy,
            overall_recall,
            conf_matrix,
            metrics_summary_df,
            fold_metrics_data  # raw sums/lists for each filter
        )

    def run_evaluation(
```

```python
        self,
        n_runs,
        n_folds,
        model_type,
        dataset_name,
        mlp_hidden_size=32,
        mlp_epochs=20,
        save_contest_results=False,
        save_false_negatives=False,
        save_all_submissions=False,
        compute_additional_metrics=True
    ):
        """
        Run the training & evaluation loop multiple times (n_runs),
        each time with K-Fold splitting (n_folds).

        After each run, we aggregate fold-level metrics to produce a
        run-level summary table of filter percentages vs.
        (avg) Accuracy, (avg) Recall, (avg) Remaining%, (sum) TN/FP/FN/TP.

        Args:
            n_runs (int): Number of times to run the cross-validation process.
            n_folds (int): Number of folds to use in K-Fold.
            model_type (str): Type of model to train ('MLP', 'LogisticRegression', etc.).
            dataset_name (str): Name of the dataset (used for saving results).
            mlp_hidden_size (int): Hidden layer size for MLP (if MLP is used).
            mlp_epochs (int): Number of epochs for MLP training (if MLP is used).
            save_contest_results (bool): If True, save per-contest results to CSV.
            save_false_negatives (bool): If True, saves false negatives to CSV in `submission_evaluation`.
            save_all_submissions (bool): If True, saves all submissions to CSV in `submission_evaluation`.
            compute_additional_metrics (bool): If True, compute & print Accuracy, Recall, and confusion matrix values
                                                for each filter percentage, plus final run-level table.
        """
        from sklearn.model_selection import KFold, train_test_split
        import numpy as np

        # Unique contest names
        contest_names = self.df['contest_name'].unique()

        # For printing final average recall at 50% filter across folds
        recall_per_fold = []

        # Predefine the filter percentages
        filter_percentages = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7]

        for run in range(1, n_runs + 1):
            print(f"\n========== STARTING RUN {run}/{n_runs} for {model_type} ==========")

            run_metrics_dict = {
                p: {
                    "tn": 0, "fp": 0, "fn": 0, "tp": 0,
                    "acc_list": [], "recall_list": [], "remain_pct_list": []
                }
                for p in filter_percentages
            }
```

```

            kf = KFold(n_splits=n_folds, shuffle=True, random_state=None)
            contest_folds = list(kf.split(contest_names))

            for fold, (train_val_idx, test_idx) in enumerate(contest_folds):
                print(f"\n--- Run {run}, Fold {fold+1}/{n_folds} ---")
                test_contests = contest_names[test_idx]
                train_val_contests = contest_names[train_val_idx]

                # 80/20 split of the 80% (train_val_contests)
                train_contests, val_contests = train_test_split(train_val_contests, test_size
                    =0.2, random_state=None)

                train_df = self.df[self.df['contest_name'].isin(train_contests)]
                val_df = self.df[self.df['contest_name'].isin(val_contests)]
                test_df = self.df[self.df['contest_name'].isin(test_contests)]

                # Prepare DataLoaders for MLP
                train_dataset = CSVDataset(train_df)
                val_dataset = CSVDataset(val_df, smote=False)
                train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
                val_loader = DataLoader(val_dataset, batch_size=32)

                # --------------------------------
                # Train model
                # --------------------------------
                if model_type == "MLP":
                    model = SimpleMLP(
                        input_size=11,
                        hidden_size=mlp_hidden_size,
                        output_size=1
                    )
                    trainer = pl.Trainer(
                        max_epochs=mlp_epochs,
                        enable_checkpointing=False,
                        logger=False
                    )
                    trainer.fit(model, train_loader, val_loader)
                    model.eval()

                else:
                    # Sklearn-based model
                    model_wrapper = OptimizedModelWrapper(
                        model_name=model_type,
                        train_X=train_dataset.x,
                        train_y=train_dataset.y,
                        val_X=val_dataset.x,
                        val_y=val_dataset.y
                    )
                    model = model_wrapper.model  # The optimized sklearn model

                # --------------------------------
                # Evaluate model (fold-level)
                # --------------------------------
                df_results, accuracy, overall_recall, conf_matrix, fold_metrics_df,
                    fold_metrics_data = self.submission_evaluation(
                    model,
                    test_df,
                    run,
```

```
                    fold,
                    dataset_name,
                    model_type,
                    save_contest_results=save_contest_results,
                    save_false_negatives=save_false_negatives,
                    save_all_submissions=save_all_submissions,
                    compute_additional_metrics=compute_additional_metrics
                )

                recall_per_fold.append(overall_recall)

                # ---------------------------------
                # ACCUMULATE fold-level metrics into run_metrics_dict
                # ---------------------------------
                if compute_additional_metrics and fold_metrics_data:
                    for p in fold_metrics_data.keys():
                        run_metrics_dict[p]["tn"] += fold_metrics_data[p]["tn"]
                        run_metrics_dict[p]["fp"] += fold_metrics_data[p]["fp"]
                        run_metrics_dict[p]["fn"] += fold_metrics_data[p]["fn"]
                        run_metrics_dict[p]["tp"] += fold_metrics_data[p]["tp"]

                        run_metrics_dict[p]["acc_list"].extend(fold_metrics_data[p]["acc_list"])
                        run_metrics_dict[p]["recall_list"].extend(fold_metrics_data[p]["recall_list"])
                        run_metrics_dict[p]["remain_pct_list"].extend(fold_metrics_data[p]["remain_pct_list"])

                # Clear GPU cache if using GPU
                torch.cuda.empty_cache()

            # ----------------------------
            # After all folds of this run, build final run-level metrics
            # ----------------------------
            if compute_additional_metrics:
                final_rows = []
                for p in filter_percentages:
                    data_p = run_metrics_dict[p]

                    # Sum of confusion matrix terms across folds
                    tn = data_p["tn"]
                    fp = data_p["fp"]
                    fn = data_p["fn"]
                    tp = data_p["tp"]
                    total = tn + fp + fn + tp

                    # Compute Accuracy & Recall from the overall sums
                    accuracy_from_sums = (tn + tp) / total if total > 0 else 0.0
                    recall_from_sums = tp / (tp + fn) if (tp + fn) > 0 else 0.0

                    # For Remaining% we can still average across folds
                    avg_remain_pct = np.mean(data_p["remain_pct_list"]) if data_p["remain_pct_list"] else 0.0

                    row = {
                        "Filter_Percentage": p,
                        "Remaining_Percentage_Size_of_Submissions": avg_remain_pct,
                        "Accuracy": accuracy_from_sums,
                        "Recall": recall_from_sums,
```

```
                        "True_Negatives": tn,
                        "False_Positives": fp,
                        "False_Negatives": fn,
                        "True_Positives": tp
                    }
                    final_rows.append(row)

                run_metrics_summary_df = pd.DataFrame(final_rows)
                print(f"\n========== RUN {run} - FINAL ADDITIONAL METRICS ACROSS ALL FOLDS ==========")
                print(run_metrics_summary_df)

                # Save this run-level summary to CSV
                additional_metrics_csv = os.path.join(
                    self.output_path,
                    f"{model_type}_additional_metrics_{dataset_name[:-4]}_run{run}.csv"
                )
                run_metrics_summary_df.to_csv(additional_metrics_csv, index=False)
                print(f"Run-level Additional Metrics saved to {additional_metrics_csv}")

            # -----------------------------
            # Print average recall for this run (at 50% filter) if desired
            # -----------------------------
            valid_recalls = [r for r in recall_per_fold if r is not None]
            avg_recall_50 = np.mean(valid_recalls) if len(valid_recalls) > 0 else 0.0
            print(f"\nAverage Recall at 50% filter across all folds (RUN {run}): {avg_recall_50:.4f}")

    def aggregate_average_recall_across_runs(self, model_type, dataset_name, n_runs=5):
        """
        Reads all run-level CSVs for a given model_type (e.g., "MLP") and dataset_name,
        then computes the average Recall across all runs for each Filter_Percentage.

        Args:
            model_type (str): The model type ("MLP", "RandomForest", "XGBoost", "SVM", "LogisticRegression").
            dataset_name (str): The dataset filename (used to locate CSVs).
            n_runs (int): Number of runs that were performed (defaults to 5).

        Returns:
            pd.DataFrame or None:
                A DataFrame with two columns: ["Filter_Percentage", "Average_Recall"],
                or None if no CSV files were found.

        Example:
            avg_recall_df = discriminative_model.aggregate_average_recall_across_runs("MLP", "28_11.csv", 5)
        """
        import os
        import pandas as pd

        dfs = []
        # Gather all CSV files for the specified runs
        for run in range(1, n_runs + 1):
            csv_name = f"{model_type}_additional_metrics_{dataset_name[:-4]}_run{run}.csv"
            csv_path = os.path.join(self.output_path, csv_name)
            if os.path.exists(csv_path):
                df_run = pd.read_csv(csv_path)
                dfs.append(df_run)
```

```python
            else:
                print(f"Warning: CSV not found for run {run}: {csv_path}")

        if not dfs:
            print("No CSV files found for the specified model_type and dataset_name.")
            return None

        # Concatenate all runs
        combined_df = pd.concat(dfs, ignore_index=True)

        # Compute average recall for each Filter_Percentage
        avg_recall_df = (
            combined_df
            .groupby("Filter_Percentage", as_index=False)["Recall"]
            .mean()
            .rename(columns={"Recall": "Average_Recall"})
        )

        # Print and return the resulting DataFrame
        print(f"\n=== Average Recall Across {n_runs} Runs for {model_type} ===")
        print(avg_recall_df)
        return avg_recall_df
```

Listing A.3: Discriminative Model Python Code

A.1.4. Machine Learning Models

```python
# --------------------------------------------------
# SimpleMLP
# --------------------------------------------------
class SimpleMLP(pl.LightningModule):
    """
    Class Name: SimpleMLP

    Purpose:
        - A simple Multilayer Perceptron (MLP) model built with PyTorch Lightning.

    Responsibilities:
        - Define a forward pass through a 3-layer fully-connected neural network.
        - Handle training/validation steps and loss computation using BCE.

    Example Usage:
        model = SimpleMLP(input_size=11, hidden_size=32, output_size=1)
        trainer = pl.Trainer(max_epochs=10)
        trainer.fit(model, train_loader, val_loader)
    """

    def __init__(self, input_size, hidden_size, output_size):
        """
        Constructor for SimpleMLP.

        Args:
            input_size (int): Number of input features.
            hidden_size (int): Number of units in hidden layers.
            output_size (int): Number of output units (1 for binary classification).
```

```
        """
        super().__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        """
        Forward pass of the MLP.

        Args:
            x (torch.Tensor): Input tensor.

        Returns:
            torch.Tensor: Sigmoid output (probability of class 1).
        """
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return torch.sigmoid(x)

    def training_step(self, batch, batch_idx):
        """
        Training step: computes binary cross-entropy loss.

        Args:
            batch (tuple): (x, y) data from the DataLoader.
            batch_idx (int): Batch index (not used).

        Returns:
            torch.Tensor: Training loss.
        """
        x, y = batch
        y_hat = self(x).squeeze()
        y = y.float()
        loss = F.binary_cross_entropy(y_hat, y)
        self.log('train_loss', loss)
        return loss

    def validation_step(self, batch, batch_idx):
        """
        Validation step: computes binary cross-entropy loss.

        Args:
            batch (tuple): (x, y) data from the DataLoader.
            batch_idx (int): Batch index (not used).
        """
        x, y = batch
        y_hat = self(x).squeeze()
        y = y.float()
        loss = F.binary_cross_entropy(y_hat, y)
        self.log('val_loss', loss)

    def configure_optimizers(self):
        """
        Configures the Adam optimizer with a fixed learning rate.
        """
        return torch.optim.Adam(self.parameters(), lr=1e-4)

```

```python
# ---------------------------------------------------
# OptimizedModelWrapper
# ---------------------------------------------------
class OptimizedModelWrapper(pl.LightningModule):
    """
    Class Name: OptimizedModelWrapper

    Purpose:
        - A wrapper that trains various scikit-learn models (RandomForest, XGBoost, SVM, LogisticRegression)
          and performs hyperparameter optimization (via GridSearchCV where applicable).

    Responsibilities:
        - Select model based on provided model_name.
        - Perform grid search (where applicable).
        - For XGBoost, handle early stopping via eval set.
        - Provide a forward method to integrate with PyTorch Lightning workflow.
    """

    def __init__(self, model_name, train_X, train_y, val_X=None, val_y=None):
        """
        Constructor for OptimizedModelWrapper.

        Args:
            model_name (str): Name of the model to train (RandomForest, XGBoost, SVM, LogisticRegression).
            train_X (numpy.ndarray): Training features.
            train_y (numpy.ndarray): Training labels.
            val_X (numpy.ndarray, optional): Validation features (required for XGBoost).
            val_y (numpy.ndarray, optional): Validation labels (required for XGBoost).
        """
        super().__init__()
        self.model_name = model_name
        self.train_X = train_X
        self.train_y = train_y
        self.val_X = val_X
        self.val_y = val_y
        self.model = self._get_optimized_model()

    def _get_optimized_model(self):
        """
        Internal method: Defines and performs the required optimization for the selected model.

        Returns:
            sklearn model: Trained and optimized model.
        """
        if self.model_name == "RandomForest":
            param_grid = {
                'n_estimators': [100],
                'max_depth': [3],
                'min_samples_split': [10]
            }
            model = RandomForestClassifier()

        elif self.model_name == "XGBoost":
            if self.val_X is None or self.val_y is None:
                raise ValueError("Validation data is required for XGBoost with early stopping.")
            model = XGBClassifier(
```

```
                use_label_encoder=False,
                eval_metric='logloss',
                max_depth=3,
                n_estimators=50,
                learning_rate=0.05,
                subsample=0.8,
                colsample_bytree=0.8
            )
            model.fit(
                self.train_X,
                self.train_y,
                eval_set=[(self.val_X, self.val_y)],
                verbose=True  # Set to False if you want to suppress the training output
            )
            return model

        elif self.model_name == "SVM":
            param_grid = {
                'C': [0.1],
                'kernel': ['linear']
            }
            model = SVC(probability=True)

        elif self.model_name == "LogisticRegression":
            param_grid = {
                'C': [0.1],
                'penalty': ['l2']
            }
            model = LogisticRegression(max_iter=500)

        else:
            raise ValueError("Invalid model name. Choose from: RandomForest, XGBoost, SVM,
                LogisticRegression")

        # For non-XGBoost models, perform Grid Search
        scorer = make_scorer(recall_score)
        grid_search = GridSearchCV(estimator=model, param_grid=param_grid, cv=3, scoring=scorer
            )
        grid_search.fit(self.train_X, self.train_y)
        print(f"Best Parameters for {self.model_name}: {grid_search.best_params_}")
        print(f"Best Cross-Validation Recall: {grid_search.best_score_}")
        return grid_search.best_estimator_

    def validation_step(self, batch, batch_idx):
        """
        Placeholder method for validation in a PyTorch Lightning loop.

        Args:
            batch (tuple): (features, labels).
            batch_idx (int): Batch index (not used).
        """
        x, y = batch
        x, y = x.cpu(), y.cpu()
        preds = self.forward(x)
        loss = F.binary_cross_entropy(preds, y.float())
        self.log("val_loss", loss)

    def forward(self, x):
        """
```

```
        Forward method to integrate with PyTorch Lightning.

        Args:
            x (torch.Tensor): Input features.

        Returns:
            torch.Tensor: Predictions as probabilities for class 1.
        """
        x_np = x.cpu().numpy()
        probs = self.model.predict_proba(x_np)[:, 1]
        return torch.tensor(probs, dtype=torch.float32)

    def configure_optimizers(self):
        """
        No optimizer is used here, as training occurs via scikit-learn's methods.
        """
        return None

# ---------------------------------------------------
# EnhancedMLP
# ---------------------------------------------------
class EnhancedMLP(pl.LightningModule):
    def __init__(self, input_size, hidden_size, output_size, dropout_prob=0.3, lr=1e-3):
        super(EnhancedMLP, self).__init__()
        # Save hyperparameters
        self.save_hyperparameters()

        # Define layers
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.bn1 = nn.BatchNorm1d(hidden_size)
        self.dropout1 = nn.Dropout(dropout_prob)

        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.bn2 = nn.BatchNorm1d(hidden_size)
        self.dropout2 = nn.Dropout(dropout_prob)

        self.fc3 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = F.leaky_relu(self.bn1(self.fc1(x)))
        x = self.dropout1(x)
        x = F.leaky_relu(self.bn2(self.fc2(x)))
        x = self.dropout2(x)
        x = self.fc3(x)
        return torch.sigmoid(x).squeeze()

    def training_step(self, batch, batch_idx):
        x, y = batch
        y_hat = self(x)
        # Compute loss
        loss = F.binary_cross_entropy(y_hat, y)
        # Log training loss
        self.log('train_loss', loss, on_step=False, on_epoch=True, prog_bar=True)
        return loss

    def on_validation_epoch_start(self):
        # Initialize a list to store outputs
        self.validation_step_outputs = []

```

```python
    def validation_step(self, batch, batch_idx):
        x, y = batch
        y_hat = self(x)
        # Compute loss
        loss = F.binary_cross_entropy(y_hat, y)
        # Log validation loss
        self.log('val_loss', loss, on_step=False, on_epoch=True, prog_bar=True)
        # Store predictions and targets for metrics
        self.validation_step_outputs.append({'preds': y_hat.detach(), 'targets': y.detach()})

    def on_validation_epoch_end(self):
        # Concatenate all predictions and targets
        preds = torch.cat([x['preds'] for x in self.validation_step_outputs]).cpu()
        targets = torch.cat([x['targets'] for x in self.validation_step_outputs]).cpu()
        # Binarize predictions
        preds = torch.round(preds)
        # Compute metrics
        acc = accuracy_score(targets, preds)
        precision = precision_score(targets, preds, zero_division=0)
        recall = recall_score(targets, preds, zero_division=0)
        f1 = f1_score(targets, preds, zero_division=0)
        # Log metrics
        self.log('val_acc', acc, prog_bar=True)
        self.log('val_precision', precision)
        self.log('val_recall', recall)
        self.log('val_f1', f1)
        # Clear the outputs
        self.validation_step_outputs.clear()

    def configure_optimizers(self):
        # Define optimizer
        optimizer = torch.optim.AdamW(self.parameters(), lr=self.hparams.lr)
        # Define learning rate scheduler
        scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
            optimizer, mode='min', factor=0.1, patience=3, verbose=True
        )
        # Return optimizer and scheduler
        return {
            'optimizer': optimizer,
            'lr_scheduler': {
                'scheduler': scheduler,
                'monitor': 'val_loss',  # Metric to monitor
                'interval': 'epoch',
                'frequency': 1
            }
        }
```

Listing A.4: Machine Learning Models Python Code

A.1.5. Discriminative Evaluator

```python
class DiscriminativeEvaluator:
    """
    Class to evaluate submissions based on quantitative features and generate textual feedback.
    """
```

```python

    CATEGORY_EXCELLENT = 3
    CATEGORY_AVERAGE   = 2
    CATEGORY_POOR      = 1

    def __init__(self, df):
        """
        Args:
            df (pd.DataFrame): DataFrame containing all submissions.
            top_percentile (float): Percentile threshold for determining Excellent category.
            average_percentile (float): Percentile threshold for determining Average category.
        """
        self.df = df
        self.top_percentile = 0.8
        self.average_percentile = 0.4


        # Simple synonyms for the 3 categories to add mild variation
        self.category_synonyms = {
            self.CATEGORY_EXCELLENT: ["excellent"], # "outstanding"
            self.CATEGORY_AVERAGE:   ["average"], # "moderate"
            self.CATEGORY_POOR:      ["poor"] # "weak"
        }

    def _choose_random_synonym(self, category):
        """Randomly pick a synonym for the given category."""
        return random.choice(self.category_synonyms[category])

    def evaluate_label_category(self, submission_row, same_contest_df, label):
        """
        Evaluate the given label for a submission and categorize it as ’Excellent’, ’Average’,
            or ’Poor’.

        Args:
            submission_row (pd.Series): Row of the submission to evaluate.
            same_contest_df (pd.DataFrame): Filtered DataFrame with the same contest.
            label (str): Label to evaluate (e.g., ’submission_word_count’, ’num_image’).

        Returns:
            int: The category (3 for Excellent, 2 for Average, 1 for Poor).
        """
        label_series = same_contest_df[label].dropna()
        submission_value = submission_row[label]

        # Handle edge case for low averages
        if label_series.mean() < 1 and submission_value == 0:
            return self.CATEGORY_AVERAGE

        # Determine percentile rank
        rank_percentile = label_series.rank(pct=True)[submission_row.name]

        if rank_percentile >= self.top_percentile:
            return self.CATEGORY_EXCELLENT
        elif rank_percentile >= self.average_percentile:
            return self.CATEGORY_AVERAGE
        else:
            return self.CATEGORY_POOR

    def calculate_percentile(self, submission_row, same_contest_df, label):
```

```
        """
        Calculate the percentile of a submission's value for a given label.
        """
        label_series = same_contest_df[label].dropna()

        # Handle case with many zeroes
        if label_series.mean() <= 0.5 and submission_row[label] == 0:
            return 0.5  # Treat as average if the mean is very low

        return label_series.rank(pct=True)[submission_row.name]

    def discriminative_evaluation(self, submission_url):
        """
        Perform a discriminative evaluation and return both textual feedback and a string classification.

        Returns:
            tuple: (evaluation_text, disc_classification_str)
                  where disc_classification_str     {"Excellent", "Average", "Poor"}
        """
        try:
            submission_row = self.df[self.df["submission_url"] == submission_url].iloc[0]
        except IndexError:
            raise ValueError(f"Submission URL {submission_url} not found in the dataset.")

        # Filter for the same contest
        contest_name = submission_row["contest_name"]
        same_contest_df = self.df[self.df["contest_name"] == contest_name]

        # Determine overall category based on rank
        rank = submission_row['qs_rank']
        num_submissions = submission_row['num_of_submissions']
        # Higher percentiles = better rankings
        rank_percentile = (num_submissions - rank) / num_submissions

        if rank_percentile >= self.top_percentile:  # High percentile = Excellent
            overall_category = self.CATEGORY_EXCELLENT
        elif rank_percentile >= self.average_percentile:  # Mid percentile = Average
            overall_category = self.CATEGORY_AVERAGE
        else:
            overall_category = self.CATEGORY_POOR  # Low percentile = Poor


        # --- Paragraph 1: Description ---
        desc_value = submission_row["submission_word_count"]
        desc_cat = self.evaluate_label_category(submission_row, same_contest_df, "submission_word_count")
        desc_syn = self._choose_random_synonym(desc_cat)

        if desc_value < 50:
            description_paragraph = (
                "The submission's description contains minimal content, making a proper evaluation difficult. "
                "It falls short of providing key details that could help understand the project effectively."
            )
            alignment_text = ""
        else:
            if desc_cat == self.CATEGORY_EXCELLENT:
```

```
            description_paragraph = (
            f"The submissions description is {desc_syn} compared to other entries. "
            "It contains an extensive amount of text that thoroughly explains the project, covering all essential aspects in detail. "
            "This level of textual detail exceeds that of most other submissions."
            )
            if desc_cat == overall_category:
                alignment_text = (
                    "The length and level of detail in the description reflect the overall excellent efforts of the submission, "
                    "mirroring the high standards seen throughout the project."
                )
            elif desc_cat > overall_category:
                alignment_text = (
                    "The description stands out with superior detail compared to the overall submission quality, "
                    "indicating that the textual documentation may have been prioritized."
                )

        elif desc_cat == self.CATEGORY_AVERAGE:
            description_paragraph = (
            f"The projects description is {desc_syn} and offers a moderate amount of text that covers the key points of the project. "
            "It provides a level of detail that is comparable to the average submission."
            )
            if desc_cat == overall_category:
                alignment_text = (
                    "The description's moderate level of detail is consistent with the overall average quality of submissions, "
                    "providing an amount of text that aligns with what is typically seen in similar entries."
                )
            elif desc_cat < overall_category:
                alignment_text = (
                    "Although the overall submission is deemed excellent, the description offers only an average level of detail, "
                    "suggesting that additional elaboration could have further strengthened its documentation compared to other high-quality entries."
                )
            else:
                alignment_text = (
                    "The description stands out with more text and detail than what is generally seen in submissions with overall poor quality, "
                    "indicating that extra effort was put into documenting the project relative to its peers."
                )
        else:  # Poor
            description_paragraph = (
                f"The description is {desc_syn} compared to other entries. "
                "It contains very little text, providing only a brief overview of the project. "
                "Compared to most other submissions, it lacks the level of detail needed to fully understand the projects scope and functionality."
            )
            if desc_cat == overall_category:
                alignment_text = (
                    "The description's brevity is consistent with the overall poor quality of the submission, "
```

```
                        "indicating that minimal documentation effort was put into this project."
                    )
                elif desc_cat < overall_category:
                    alignment_text = (
                        "While the overall submission is evaluated as higher quality, the description remains relatively brief, "
                        "suggesting that additional elaboration could have further strengthened its documentation and better supported the project's overall presentation."
                    )

        # BOM part
        bom_cat = self.evaluate_label_category(submission_row, same_contest_df, "num_things")
        bom_syn = self._choose_random_synonym(bom_cat)

        if bom_cat == self.CATEGORY_EXCELLENT:
            comp_doc = f"The submission features {bom_syn} documentation of the components, providing a detailed and well-organized list of materials used. "
        elif bom_cat == self.CATEGORY_AVERAGE:
            comp_doc = f"The amount of components used in the project is {bom_syn}, aligning with the average level seen in similar submissions. "
        else:
            comp_doc = (
                f"The amount of components is {bom_syn}, indicating either a very basic implementation or potentially missing material details. "
                "This could hinder the ability to replicate or understand the projects full scope. "
            )

        description_paragraph += f" {alignment_text} {comp_doc}"


        # --- Documentation Paragraph ---

        # Visual Documentation part
        visual_labels = ["num_image", "num_video"]
        percentiles = [self.calculate_percentile(submission_row, same_contest_df, label) for label in visual_labels]
        avg_percentile = sum(percentiles) / len(percentiles)

        if avg_percentile >= self.top_percentile:
            visual_category = self.CATEGORY_EXCELLENT
        elif avg_percentile >= self.average_percentile:
            visual_category = self.CATEGORY_AVERAGE
        else:
            visual_category = self.CATEGORY_POOR

        visual_syn = self._choose_random_synonym(visual_category)
        images_present = submission_row["num_image"] > 0
        videos_present = submission_row["num_video"] > 0
        gifs_present = submission_row["num_gif"] > 0

        present_formats = [fmt for fmt, present in zip(["images", "videos"], [images_present, videos_present]) if present]
        format_text = " and ".join(present_formats) if present_formats else "visual content"

        # Main visual documentation assessment
        if visual_category == self.CATEGORY_EXCELLENT:
```

```
            visual_doc = (
                f"This entry provides {visual_syn} visual documentation, utilizing many {format_text}. "
                "The provided media offer a clear representation of the project, making it easier to understand and replicate. "
                "Compared to other submissions, it includes a notably high amount of visual material."
            )
        elif visual_category == self.CATEGORY_AVERAGE:
            visual_doc = (
                f"This entry provides {visual_syn} visual documentation, incorporating a moderate number of {format_text}. "
                "While the visuals contribute to understanding the project, they are not as extensive as in the highest-ranked submissions. "
                "Even though the available visuals aid comprehension, additional supporting media could have further enhanced clarity and engagement."
            )
        else:
            visual_doc = (
                f"This entry provides {visual_syn} visual documentation, with only limited {format_text} available. "
                "Compared to other entries, the visual content is sparse, making it more difficult to grasp the project fully. "
                "A stronger emphasis on visual documentation would have significantly improved the submissions presentation."
            )

        # Additional checks for missing content
        if gifs_present:
            visual_doc += " Additionally, animated GIFs are included, helping to illustrate certain aspects dynamically. "

        missing_resources = [res for res, present in zip(["images", "videos"], [images_present, videos_present]) if not present]

        if missing_resources:
            conjunction = "However" if visual_category in (self.CATEGORY_EXCELLENT, self.CATEGORY_AVERAGE) else "Also"
            missing_text = f" {conjunction}, it lacks key resources such as {', '.join(missing_resources)}. "

            if "videos" in missing_resources:
                missing_text += (
                    "Without video documentation, understanding how the project operates in real-time is challenging. Including a demonstration video could have showcased key interactions or features more effectively."
                )

            if "images" in missing_resources:
                missing_text += (
                    " Including images would have helped document the project's components and overall design more effectively, "
                    "ensuring that viewers can quickly understand its structure and purpose."
                )

            visual_doc += missing_text


        # Code
```

```
        code_provided = (submission_row["code"] == 1)
        link_provided = (submission_row["link"] == 1)

        if code_provided and link_provided:
            code_text = "It fully shares its code and provides a repository link to encourage collaboration."
        elif code_provided and not link_provided:
            code_text = "The submission includes some code but lacks a central repository link."
        else:
            code_text = "Unfortunately, no code is provided, which could leave technical aspects undisclosed."

        code_doc = f"{code_text} These factors influence how easily others can reproduce or build upon the work."

        documentation_paragraph = visual_doc + code_doc

        # # --- Paragraph 3: Overall Recommendation ---
        if overall_category == self.CATEGORY_EXCELLENT:
            disc_classification_str = "Excellent"

        elif overall_category == self.CATEGORY_AVERAGE:
            disc_classification_str = "Average"

        else:  # Poor Overall Category
            disc_classification_str = "Poor"

        # Final text assembly using list comprehension
        paragraphs = [
            f"**Description and bills of materials:**\n{description_paragraph}",
            f"**Visuals, code and other documentation:**\n{documentation_paragraph}",
            f"**Overall Recommendation:**\n*{disc_classification_str}*"
        ]

        final_text = "\n\n".join([p for p in paragraphs if p])


        return final_text, disc_classification_str
```

Listing A.5: Discriminative Evaluator Python Code

A.1.6. Data Cleaner

```
class DataCleaner:
    """
    Class Name: DataCleaner

    Purpose:
        - Provide static methods to clean text and calculate token sizes.

    Responsibilities:
        - Clean text by removing special characters (except dots).
        - Calculate the number of tokens of a given text using a tokenizer.
    """
```

```python

    @staticmethod
    def clean_text(text):
        """
        Clean a text field by removing unnecessary symbols (except dots),
        handling newlines, etc.

        Args:
            text (str): Input text to clean.

        Returns:
            str: Cleaned text or an empty string if invalid/empty.
        """
        if not isinstance(text, str) or not text.strip():
            return ""

        # Remove weird signs except for dots
        cleaned = re.sub(r'[^\w\s.]', '', text)
        # Replace multiple newlines with space
        cleaned = re.sub(r'\n+', ' ', cleaned)
        return cleaned.strip()

    @staticmethod
    def calculate_token_size(text, tokenizer):
        """
        Calculate the number of tokens in a given text using the tokenizer.

        Args:
            text (str): The input text.
            tokenizer: The tokenizer from Hugging Face.

        Returns:
            int: Number of tokens.
        """
        tokens = tokenizer(text, return_tensors="pt")["input_ids"]
        return tokens.shape[1]

    @staticmethod
    def strip_prompt_and_eos_token(response, prompt, tokenizer):
        """
        Strips the prompt from the response and removes the EOS token if it exists.

        Args:
            response (str): The generated text response from the model.
            prompt (str): The input prompt used for the model.
            tokenizer: The tokenizer used by the model, providing the EOS token.

        Returns:
            str: The cleaned response text.
        """
        # Strip the prompt from the response
        response_start = response.find(prompt[-100:])
        if response_start != -1:
            response = response[response_start + 100:].strip()

        # Check if the response ends with the EOS token and remove it
        eos_token = tokenizer.eos_token  # e.g., "<|eot_id|>"
        if response.endswith(eos_token):
            response = response[: -len(eos_token)].strip()
```

```

        return response
```

Listing A.6: Data Cleaner Python Code

A.1.7. Summarizer

```
class Summarizer:
    """
    Class Name: Summarizer

    Purpose:
        - Summarize a given contest description using the model.
    """

    def __init__(self, model_loader: ModelLoader, data_cleaner: DataCleaner, df=None,
                 max_allowed_tokens=100000, summary_size=400):
        """
        Args:
            model_loader (ModelLoader): The loaded model/tokenizer wrapper.
            data_cleaner (DataCleaner): Utility class for cleaning text & counting tokens.
            df (pd.DataFrame, optional): DataFrame with contest info.
            max_allowed_tokens (int): Max tokens for prompt+answer.
            summary_size (int): Approx. size for the answer portion.
        """
        self.model_loader = model_loader
        self.data_cleaner = data_cleaner
        self.df = df
        self.max_allowed_tokens = max_allowed_tokens
        self.summary_size = summary_size

    def summarize_contest_description(self, contest_name):
        """
        Summarize the contest description for the given contest_name.

        Returns:
            str: A cleaned summary of the contest description.
        """
        if self.df is None:
            raise ValueError("A DataFrame must be provided to Summarizer to look up submissions
                .")

        # Grab the relevant row
        try:
            selected_entry = self.df[self.df['contest_name'] == contest_name].iloc[0]
        except IndexError:
            raise ValueError(f"Contest name {contest_name} not found in the dataset.")

        # Clean text
        contest_description = self.data_cleaner.clean_text(selected_entry['overview'])

        prompt = f"""
          You are an expert at summarizing information for clarity and relevance. Below is a
              description of a contest.
          Your task is to summarize it by focusing on:
```

```
          - What is the contest about?
          - Main goals and challenges described?
          - Topics or themes participants address?
          - Overall purpose of the contest?

          Write the summary as a single, continuous paragraph and leave out irrelevant details (prizes, hardware giveaways, registration, etc.).

          Contest Description:
          {contest_description}

          End the Summary with '<|eot_id|>'.
          """

        prompt_tokens = self.data_cleaner.calculate_token_size(prompt, self.model_loader.tokenizer)
        total_max_length = prompt_tokens + self.summary_size
        if total_max_length > self.max_allowed_tokens:
            raise ValueError("Contest description + answer size exceed max token limit.")

        raw_text = self.model_loader.generate_text(
            prompt,
            max_length=total_max_length,
            temperature=0.7,
        )
        cleaned_response = self.data_cleaner.strip_prompt_and_eos_token(raw_text, prompt, self.model_loader.tokenizer)
        torch.cuda.empty_cache()
        return cleaned_response
```

Listing A.7: Summarizer Python Code

A.1.8. Submission Evaluator

```
class SubmissionEvaluator:
    """
    Class Name: SubmissionEvaluator

    Purpose:
        - Evaluate a single submission for classification (Excellent, Average, Poor)
          using a language model with few-shot examples.
        - If no contest_description is provided, it uses Summarizer to generate one.
    """

    def __init__(self, model_loader: ModelLoader, data_cleaner: DataCleaner, df, fewshot_df, fewshot_urls,
                 summarizer: Summarizer, output_length=800):
        """
        Args:
            model_loader (ModelLoader): Contains the loaded model and tokenizer.
            data_cleaner (DataCleaner): Used for cleaning text.
            df (pd.DataFrame): Main submissions DataFrame.
            fewshot_df (pd.DataFrame): DataFrame containing few-shot examples.
            fewshot_urls (list): List of URLs for few-shot examples.
            summarizer (Summarizer): Used to generate a contest summary if none is provided.
```

```
            output_length (int): Maximum length of the output text.
        """
        self.model_loader = model_loader
        self.data_cleaner = data_cleaner
        self.df = df
        self.fewshot_df = fewshot_df
        self.fewshot_urls = fewshot_urls
        self.summarizer = summarizer
        self.output_length = output_length



    def evaluate_submission(self, submission_url, contest_description=None):
        """
        Evaluate a single submission. Builds a prompt with few-shot examples.
        If contest_description=None, we'll use Summarizer to generate one from the submission's contest_name.

        Returns:
            str: The model-generated text (cleaned).
            str: The prompt used.
        """
        try:
            selected_entry = self.df[self.df['submission_url'] == submission_url].iloc[0]
        except IndexError:
            raise ValueError(f"Submission URL {submission_url} not found in the dataset.")

        # If no contest description, we summarize from the submission's contest_name
        if contest_description is None:
            contest_name = selected_entry['contest_name']
            contest_description = self.summarizer.summarize_contest_description(contest_name)
            print("Contest Summary:\n" + contest_description + "\n")

        # Clean text
        submission_story = self.data_cleaner.clean_text(selected_entry['story'])

        # Prepare few-shot examples
        fewshot_submissions_filtered = self.fewshot_df[self.fewshot_df['submission_url'].isin(self.fewshot_urls)]
        fewshot_submissions_filtered = fewshot_submissions_filtered.set_index('submission_url') \
                                                                    .reindex(self.fewshot_urls) \
                                                                    .reset_index()
        fs = []
        for i in range(len(fewshot_submissions_filtered)):
            row = fewshot_submissions_filtered.iloc[i]
            fs.append({
                    "class": row['class'],
                    "overview": row['overview'],
                    "story": row['story'],
                    "output": row['expected_response']
                })


        fewshot_str = f"""
          --- BEGIN FEW-SHOT EXAMPLES ---
          Below are evaluation examples that illustrate how submissions from other contests are evaluated according to the provided instructions and criteria.
```

```
      **Example 1: {fs[0]['class']} Submission**
      Contest Description:
      "{fs[0]['overview']}"
      Submission Story:
      "{fs[0]['story']}"
      Expected LLM Output:
      "{fs[0]['output']}"

      **Example 2: {fs[1]['class']} Submission**
      Contest Description:
      "{fs[1]['overview']}"
      Submission Story:
      "{fs[1]['story']}"
      Expected LLM Output:
      "{fs[1]['output']}"

      **Example 3: {fs[2]['class']} Submission**
      Contest Description:
      "{fs[2]['overview']}"
      Submission Story:
      "{fs[2]['story']}"
      Expected LLM Output:
      "{fs[2]['output']}"
      --- END FEW-SHOT EXAMPLES ---
      """


    prompt = f"""
      You are an expert evaluator for technical contests. Your task is to assess a submission based on the following:

      --- BEGIN INSTRUCTIONS ---
      1. Provide a structured evaluation consisting of two concise paragraphs, each addressing one of the following criteria in a few sentences:
        - *Novelty of the Solution*: Evaluate how novel the solution is. Search for similar, existing solutions, and evaluate how different and unique this solution is compared to those existing solutions. Identify any concept, feature, technology or approach that might be novel.
        - *Usefulness of the Solution*: Evaluate how useful the solution is. Consider factors such as practicality, usability, and relevance. Identify potential challenges that might hinder its real-world value.

      2. Choose one of the following overall recommendations. Be critical in your evaluations. If a solution does not clearly surpass existing alternatives, it should not be rated as 'Excellent.' Carefully consider any limitation before rating a solution as even 'Average.' About 80% of the solutions should be rated as 'Average' or 'Poor.'
        - Excellent: The solution demonstrates both substantial novelty and usefulness, far exceeding typical expectations.
        -  Average : The solution demonstrates a reasonable degree of novelty and usefulness, meeting typical expectations without exceeding them.
        -  Poor : The solution is only moderately novel or useful, and is thus unlikely to meet typical expectations.
      --- END INSTRUCTIONS ---

      {fewshot_str}

      --- BEGIN SUBMISSION TO EVALUATE ---
      Contest Description:
```

```
            {contest_description}

            Submission Story:
            {submission_story}
            --- END SUBMISSION TO EVALUATE ---

            End your response with '<|eot_id|>'.
            """

        prompt_tokens = self.data_cleaner.calculate_token_size(prompt, self.model_loader.
            tokenizer)
        total_max_length = prompt_tokens + self.output_length
        stop_token_ids = torch.tensor([128000, 27, 91, 9684, 91, 29])

        response = self.model_loader.generate_text(
            prompt,
            max_length=total_max_length,
            temperature=0.7,
            stop_token_ids=stop_token_ids
        )
        cleaned_response = DataCleaner.strip_prompt_and_eos_token(response, prompt, self.
            model_loader.tokenizer)
        torch.cuda.empty_cache()
        return cleaned_response, prompt
```

Listing A.8: Submission Evaluator Python Code

A.1.9. Contest Evaluator

```
class ContestEvaluator:
    """
    Orchestrates evaluation of all submissions in a given contest.
    Saves results with extra columns: "contest_description", "discriminative",
    "disc_classification", "generative", "gen_classification", ...
    """

    def __init__(self, submission_evaluator, discriminative_evaluator, summarizer):
        """
        Args:
            submission_evaluator: Evaluates single submissions (generative).
            discriminative_evaluator: Numeric-based evaluation text + classification.
            summarizer: Summarizes the contest if no description is given.
        """
        self.submission_evaluator = submission_evaluator
        self.discriminative_evaluator = discriminative_evaluator
        self.summarizer = summarizer

    def parse_llm_response(self, llm_response):
        """
        Extract classification label (Excellent, Average, or Poor) enclosed in **...**,
        returning the last occurrence in the text.
        Example matches: **Excellent**, **Average**, **Poor** (case-insensitive).
        """
        # Find all occurrences of **Excellent**, **Average**, or **Poor** (case-insensitive)
        matches = re.findall(r"(Excellent|Average|Poor)", llm_response, re.IGNORECASE)
```

```
        if not matches:
            return None
        # Take the last match and capitalize it (e.g., "Excellent", "Average", or "Poor")
        return matches[-1].capitalize()

    def evaluate_full_contest(self, contest_name, df, output_csv_path=None, contest_description=None):
        """
        Evaluate all submissions in a contest. Adds:
            "disc_classification" from DiscriminativeEvaluator
            "gen_classification" from parse_llm_response.
        """
        if contest_description is None:
            contest_description = self.summarizer.summarize_contest_description(contest_name)

        # Filter for the target contest
        contest_df = df[df['contest_name'] == contest_name].copy()
        # Sort by 'quality_score' descending
        contest_df = contest_df.sort_values(by='qs_raw_avg', ascending=False)
        # # For demonstration, selecting bottom 50% only (custom logic)
        # contest_df = contest_df.iloc[len(contest_df) // 2:]

        results_df = pd.DataFrame(columns=[
            "submission_url", "submission_name", "contest_name",
            "num_of_submissions", "rank", "quality_score",
            "contest_description",
            "prompt",
            "discriminative",
            "disc_classification",
            "generative",
            "gen_classification",
            "winner_categories"
        ])

        for idx, row in tqdm(contest_df.iterrows(), total=len(contest_df), desc="Evaluating submissions"):
            submission_url = row['submission_url']
            submission_name = row['submission_name']
            num_submissions = row['num_of_submissions']
            rank = row['qs_rank']
            quality_score = row['qs_raw_avg']
            winner_categories = row['winner_categories']

            # (1) Discriminative
            disc_text, disc_classification = self.discriminative_evaluator.discriminative_evaluation(submission_url)

            # (2) Generative
            try:
                generative_resp, prompt = self.submission_evaluator.evaluate_submission(
                    submission_url,
                    contest_description=contest_description
                )
            except Exception as e:
                print(f"Error processing {submission_url}: {e}")
                continue

            # (3) Extract generative classification
            gen_classification = self.parse_llm_response(generative_resp)
```

```

            new_row = {
                "submission_url": submission_url,
                "submission_name": submission_name,
                "contest_name": contest_name,
                "num_of_submissions": num_submissions,
                "rank": rank,
                "quality_score": quality_score,
                "contest_description": contest_description,
                "prompt": prompt,
                "discriminative": disc_text,
                "disc_classification": disc_classification,
                "generative": generative_resp,
                "gen_classification": gen_classification,
                "winner_categories": winner_categories
            }

            results_df = pd.concat([results_df, pd.DataFrame([new_row])], ignore_index=True)


            if output_csv_path:
                results_df.to_csv(output_csv_path, index=False)

        return results_df
```

Listing A.9: Contest Evaluator Python Code

A.2. Data Gathering and Preprocessing

This section provides an overview of the scripts responsible for collecting and preparing contest data for analysis. The data gathering process involves web crawling to extract contest details, submission content, and prize information. Once collected, the data undergoes preprocessing to ensure consistency, remove redundancies, and enhance its usability for machine learning models.

- `dynamic_crawl_contest_link.py`: Crawls the Hackster.io contest page to extract all contest links, including past and recent contests. It navigates the webpage dynamically, handles scrolling, and filters out redundant links before saving them.
- `dynamic_crawl_contests_content.py`: Extracts detailed information from each contest, including the contest overview, number of participants, number of submissions, prize details, and the contest start date. It organizes this data and saves it in structured JSON and CSV formats.
- `dynamic_crawl_contest_prizes.py`: Collects prize information from contest pages, identifying winning submissions and their associated rewards. The script parses the webpage, extracts prize categories, and saves the data to a CSV file for further analysis.

- `dynamic_crawl_contests_submissions.py`: Crawls through all submissions of each contest, extracting details such as project descriptions, images, CAD files, schematics, code availability, and video content. It organizes this information into structured JSON files for each submission.

- `dataset_creator.py`: Constructs the dataset by extracting contest and submission details from JSON files and merging them into a structured CSV file. It processes submission metadata, filters out incomplete entries, and ensures a clean dataset for further analysis.

- `finalize_data.py`: Prepares the dataset for analysis by standardizing contest and submission data. It verifies contest-submission matches, assigns default winner labels, counts words in submission descriptions, and saves the processed data in a finalized structure.

- `data_preprocessor.py`: Refines the dataset for machine learning models by cleaning and organizing the data. It removes duplicate submissions, standardizes date formats, maps missing video durations, and updates contest-level statistics. Finally, it saves the fully processed dataset for further modeling.

A.2.1. Crawling Contest Links

```python
class LinkCrawler:
    def __init__(
        self,
        website="https://www.hackster.io/contests",
        task="Task",
    ):
        self.options = webdriver.ChromeOptions()
        self.options.headless = True  # Run Chrome in headless mode (without a UI)
        self.driver = webdriver.Chrome(
            service=Service(ChromeDriverManager().install()), options=self.options
        )

        # Open the website
        self.driver.get(website)
        time.sleep(2)  # Wait for the page to load

        self.task = task
        self.website = website

    def get_task(self):
        return self.task

    def slow_scroll_to_bottom(
        self, scroll_increment=100, scroll_pause_time=0.1, start=0
    ):
        current_scroll_position, new_height = 0, 1
        while current_scroll_position <= new_height:
            current_scroll_position += scroll_increment
            self.driver.execute_script(
                f"window.scrollTo({start}, {current_scroll_position});"
```

```
            )
            time.sleep(scroll_pause_time)
            new_height = self.driver.execute_script("return document.body.scrollHeight")

        return new_height

    def get_elements_by_selector(self, selector, scroll=None):
        if scroll:
            # Scroll the page to load elements
            self.slow_scroll_to_bottom(**scroll)

        WebDriverWait(self.driver, 20).until(
            EC.presence_of_all_elements_located((By.CSS_SELECTOR, selector))
        )

        return self.driver.find_elements(By.CSS_SELECTOR, selector)

    def click_on_element(self, button_xpath):
        # Try to find the button
        button = WebDriverWait(self.driver, 10).until(
            EC.element_to_be_clickable((By.XPATH, button_xpath))
        )
        # Scroll the button into view
        ActionChains(self.driver).move_to_element(button).perform()
        button.click()

        time.sleep(2)  # Wait new content to load

    def clean_reduntant_links(self, competitions_links):
        # Remove None reduntant links
        competitions_links = [link for link in competitions_links if link]

        # Remove reduntant links ending with #winners
        competitions_links = [
            link for link in competitions_links if not link.endswith("#winners")
        ]

        # Remove reduntant links not include /contests/ (company links)
        competitions_links = [
            link for link in competitions_links if "/contests/" in link
        ]

        # Remove reduntant links
        competitions_links = list(set(competitions_links))
        return competitions_links

    def get_recent_competitions(self):
        try:
            logger.info(f"Crawling recent competitions section")
            # Crawl recent competitions section
            recent_competitions_selector = "div.cards__wrapper__fXLS1.
                contests_page__recentCard__Dgi3d > div > div > a.cards__title__eutG1.
                hckui__typography__bodyM.hckui__typography__linkCharcoal.
                hckui__layout__marginTop15.hckui__layout__marginBottom10"

            recent_competitions = self.get_elements_by_selector(
                recent_competitions_selector,
                scroll={"scroll_increment": 100, "scroll_pause_time": 0.1, "start": 0},
            )
```

```python

            recent_competitions_links = [
                a.get_attribute("href") for a in recent_competitions
            ]

            recent_competitions_links = self.clean_reduntant_links(
                recent_competitions_links
            )

            logger.info(f"Found {len(recent_competitions_links)} recent competitions")

            return recent_competitions_links

        except Exception as e:
            print("An error occurred while crawling:", str(e))

    def get_past_competitions(self):
        logger.info(f"Crawling past competitions section")
        try:
            while True:
                # Try to find the "Show More" button
                show_more_button_xpath = (
                    '//*[@id="main"]/div/div/div[2]/div/div/div/div/div[2]/button'
                )
                show_more_button = WebDriverWait(self.driver, 10).until(
                    EC.element_to_be_clickable((By.XPATH, show_more_button_xpath))
                )
                # Scroll the "Show More" button into view (optional)
                ActionChains(self.driver).move_to_element(show_more_button).perform()
                # Click the "Show More" button
                show_more_button.click()
                # Wait for the content to load
                time.sleep(2)
        except Exception as e:
            print("No more 'Show More' buttons to click or an error occurred:", str(e))
        self.slow_scroll_to_bottom(200, 0.1)
        past_competitions_selector = "#main > div > div > div.hckui__layout__flexJustifyCenter.hckui__layout__paddingTop30.hckui__layout__paddingBottom45.hckui__layout__paddingLeft15.hckui__layout__paddingRight15.contests_page__bannerListRoot__ubtVD"
        past_competitions_div = self.get_elements_by_selector(
            past_competitions_selector,
            scroll={"scroll_increment": 100, "scroll_pause_time": 0.1, "start": 0},
        )
        past_competitions = past_competitions_div[0].find_elements(By.TAG_NAME, "a")

        past_competitions_links = [a.get_attribute("href") for a in past_competitions]

        past_competitions_links = self.clean_reduntant_links(past_competitions_links)

        logger.info(f"Found {len(past_competitions_links)} recent competitions")

        return past_competitions_links

    def formating(self, competitions):
        competition_data = []
        for comp in competitions:
            title = clean_text(comp.text)  # Extract the title
            link = comp.get_attribute("href")  # Get the link
```

```python
            competition_data.append([title, link])

        return competition_data

    def save_to_csv(self, filename, headers, rows):
        with open(filename, "w", newline="", encoding="utf-8") as file:
            writer = csv.writer(file)
            writer.writerow(headers)
            for r in rows:
                writer.writerow(r)

    def close(self):
        self.driver.quit()
```

Listing A.10: dynamic_crawl_contest_link.py Python Code

A.2.2. Extracting Contest Content

```python
def get_prize_numbers(text):
    """
    Extract prize amount from the given text.
    :param text: The text to extract the prize from (e.g., "$1000").
    :return: The numeric prize value as an integer, or 0 if not found.
    """
    amount = re.search(r"\$\d+", text)
    if amount is None:
        return 0
    return int(amount.group(0).replace("$", ""))

def get_overview():
    """
    Collects the contest overview and metrics (e.g., number of participants, submissions).
    :return: A dictionary with contest details.
    """
    # Dictionary to store contest metrics
    competition_metric = {}

    # Get the current page URL (contest URL)
    competition_metric["url"] = page.url

    # Initialize default metrics
    apply_for_hardware = 0
    num_of_submissions = 0
    num_of_participants = 0

    # Extract submission, participant, and hardware info from page
    nags = page.eles(
        ".hckui__typography__bodyM hckui__typography__link hckui__typography__hoverUnderline"
    )
    for nag in nags:
        nag_text = nag.ele(".hckui__typography__bold").text
        # Use match-case for structured handling of different contest metrics
        match nag_text:
            case "Apply for hardware":
                apply_for_hardware = int(
```

```python
                    nag.ele(
                        ".hckui__layout__marginLeft10 hckui__typography__bodyS"
                    ).text.replace(",", "")
                )
            case "Submissions":
                num_of_submissions = int(
                    nag.ele(
                        ".hckui__layout__marginLeft10 hckui__typography__bodyS"
                    ).text.replace(",", "")
                )
            case "Participants":
                num_of_participants = int(
                    nag.ele(
                        ".hckui__layout__marginLeft10 hckui__typography__bodyS"
                    ).text.replace(",", "")
                )

    # Store extracted values in the competition metric dictionary
    competition_metric["apply_for_hardware"] = apply_for_hardware
    competition_metric["num_of_submissions"] = num_of_submissions
    competition_metric["num_of_participants"] = num_of_participants

    # Extract additional information from the overview and prize sections
    challenge_main = page.ele(".challenge-column-side-main").eles("tag:section")
    for section in challenge_main:
        # Get contest overview text
        if section.attr("id") == "overview-description":
            competition_metric["overview"] = section.text

        # Get prize details, store the list of prizes
        if section.attr("id") == "overview-prizes":
            prizes = section.eles(".challenge-prize-copy")
            competition_metric["prizes_list"] = [prize.text for prize in prizes]
            for prize in prizes:
                winner = prize.ele(".hckui__typography__bodyL hckui__typography__bold")
                amount = prize.ele(".challenge-prize-name-value")
                if amount.ele(".hckui__typography__pebble"):
                    amount = amount.ele(".hckui__typography__pebble")
    return competition_metric

def get_overview():
    """
    Collects the contest overview and metrics (e.g., number of participants, submissions).
    :return: A dictionary with contest details.
    """
    # Dictionary to store contest metrics
    competition_metric = {}

    # Get the current page URL (contest URL)
    competition_metric["url"] = page.url

    # Initialize default metrics
    apply_for_hardware = 0
    num_of_submissions = 0
    num_of_participants = 0

    # Extract submission, participant, and hardware info from page
    nags = page.eles(
        ".hckui__typography__bodyM hckui__typography__link hckui__typography__hoverUnderline"
```

```python
    )
    for nag in nags:
        nag_text = nag.ele(".hckui__typography__bold").text
        if nag_text == "Apply for hardware":
            apply_for_hardware = int(
                nag.ele(
                    ".hckui__layout__marginLeft10 hckui__typography__bodyS"
                ).text.replace(",", "")
            )
        elif nag_text == "Submissions":
            num_of_submissions = int(
                nag.ele(
                    ".hckui__layout__marginLeft10 hckui__typography__bodyS"
                ).text.replace(",", "")
            )
        elif nag_text == "Participants":
            num_of_participants = int(
                nag.ele(
                    ".hckui__layout__marginLeft10 hckui__typography__bodyS"
                ).text.replace(",", "")
            )

    # Store extracted values in the competition metric dictionary
    competition_metric["apply_for_hardware"] = apply_for_hardware
    competition_metric["num_of_submissions"] = num_of_submissions
    competition_metric["num_of_participants"] = num_of_participants

    # Extract additional information from the overview and prize sections
    challenge_main = page.ele(".challenge-column-side-main").eles("tag:section")
    for section in challenge_main:
        # Get contest overview text
        if section.attr("id") == "overview-description":
            competition_metric["overview"] = section.text

        # Get prize details, store the list of prizes
        if section.attr("id") == "overview-prizes":
            prizes = section.eles(".challenge-prize-copy")
            competition_metric["prizes_list"] = [prize.text for prize in prizes]

    # Look for the timeline container
    side_overview = page.ele(".challenge-column-side-overview hckui__util__overflowFlexHack")
    if side_overview:
        # Get all past events (sections with timeline information)
        past_events = side_overview.eles(".side_panel__pastEvent__NgPGp")
        # Iterate through each event to find the "Contest begins" text
        for event in past_events:
            # Get the title and date elements
            title_element = event.ele(".hckui__typography__bodyL hckui__typography__bold")
            date_element = event.ele(".hckui__typography__bodyS")
            # Check if the title text is "Contest begins"
            if title_element and "Contest begins" in title_element.text:
                competition_metric["contest_begins_date"] = date_element.text
                break
    else:
        print("side_overview not found")
    return competition_metric

def get_all_submissions():
    """
```

```
    Collects all submission links and titles from the contest submissions page using DrissionPage.
    :return: Two lists containing submission links and titles.
    """
    links = []
    titles = []
    page_number = 1
    max_pages = 100  # To prevent infinite loops

    while page_number <= max_pages:
        try:
            print(f"Processing page {page_number}")

            # Wait for a brief moment to ensure the page content is loaded
            time.sleep(3)
            move_screen(page)

            # Extract submission data
            cards_ = page.eles(".card-body")
            for card in cards_:
                project_link = card.ele(".project-link-with-ref")
                if project_link:
                    href = project_link.attr("href")
                    title = project_link.attr("title")
                    if href and href not in links:
                        links.append(href)
                        titles.append(title)
            # Check if there is a "Next" button and if it is enabled
            button = page.ele("text:Next    ", timeout=5)
            if not button:
                print("No 'Next' button found. Reached the last page.")
                break
            # Ensure 'Next' button is not disabled
            parent_li = button.parent()
            if parent_li and 'disabled' in (parent_li.attr('class') or ''):
                print("Next button is disabled. Reached the last page.")
                break
            # Click the 'Next' button
            button.click()
            page_number += 1

            # Wait for new content to load
            time.sleep(2)
        except Exception as e:
            print("Error while fetching submissions or navigating pages:", e)
            time.sleep(2)  # Add a delay before retrying or breaking out of the loop
            break
    return links, titles


def crawl(website, task=["contest_overview", "submissions"]):
    """
    Crawl the contest website and collect the overview and submissions.
    :param website: The contest website to crawl.
    :param task: The tasks to perform. Default is ["contest_overview", "submissions"].
    """
    # Extract contest title from the URL
    title = website.split("/")[-1]

```

```python
    # Create a directory for saving the contest data
    os.makedirs(f"./data/contest_text/{title}", exist_ok=True)

    # Collect and save the contest overview
    if "contest_overview" in task:
        overview_website = website + "#challengeNav"
        page.get(overview_website)
        move_screen(page)
        contest_overview = get_overview()

        # Add the contest name to the overview and save as JSON
        contest_overview["name"] = title
        with open(f"./data/contest_text/{title}/contest_overview.json", "w") as f:
            json.dump(contest_overview, f)

    # Collect and save the submissions data
    if "submissions" in task:
        submissions_website = website + "/submissions#challengeNav"
        page.get(submissions_website)
        submissions_links, submissions_titles = get_all_submissions()
        # Save submissions to CSV file
        with open(f"./data/contest_text/{title}/submissions.csv", "w", newline='', encoding='
            utf-8') as f:
            writer = csv.writer(f)
            writer.writerow(["Title", "Link"])
            for link, title in zip(submissions_links, submissions_titles):
                writer.writerow([title, link])
    return
```

Listing A.11: dynamic_crawl_contests_content.py Python Code

A.2.3. Crawling Contest Prizes

```python
class SubmissionPrizeCrawler:
    def __init__(self, headless=True):
        self.options = webdriver.ChromeOptions()
        if headless:
            self.options.headless = True  # Run in headless mode
        self.driver = webdriver.Chrome(
            service=Service(ChromeDriverManager().install()), options=self.options
        )

    def fetch_page(self, url):
        self.driver.get(url)
        try:
            # Wait for key elements to load
            WebDriverWait(self.driver, 10).until(
                EC.presence_of_element_located((By.CLASS_NAME, 'winner-card-grid'))
            )
        except Exception as e:
            print(f"Timeout waiting for page to load: {url}. Error: {e}")

    def get_winners(self, competition_url):
        try:
            self.fetch_page(competition_url)
```

```python
        except Exception as e:
            print(f"Failed to load page: {competition_url}. Skipping. Error: {e}")
            return []

        soup = BeautifulSoup(self.driver.page_source, 'html.parser')
        winners_data = []

        try:
            prize_containers = soup.find_all('div', class_='winner-card-grid')
            for container in prize_containers:
                submission_cards = container.find_all('div', class_='winner-card')
                for card in submission_cards:
                    prize_details = card.find('h5', class_='hckui__typography__h5').text.strip()
                    submission_name_element = card.find('a', class_='hckui__typography__bodyM hckui__typography__link hckui__typography__bold')
                    if submission_name_element:
                        winners_data.append({
                            'competition_url': competition_url,
                            'category': prize_details.split(':')[0].strip(),
                            'prize': prize_details,
                            'submission': submission_name_element.text.strip(),
                            'submission_link': f"https://www.hackster.io{submission_name_element['href']}"
                        })
        except Exception as e:
            print(f"Error parsing competition page: {competition_url}. Error: {e}")
        return winners_data

    def save_to_csv(self, filename, data):
        headers = ['competition_url', 'category', 'prize', 'submission', 'submission_link']

        with open(filename, 'w', newline='', encoding='utf-8') as file:
            writer = csv.DictWriter(file, fieldnames=headers)
            writer.writeheader()
            for row in data:
                writer.writerow(row)

    def close(self):
        self.driver.quit()
```

Listing A.12: dynamic_crawl_contest_prizes.py Python Code

A.2.4. Crawling Contest Submissions

```python
def login():
    """Login to the website"""
    lg = page.ele(
        "css=.hckui__buttons__cancel.hckui__buttons__lg,hckui__layout__hiddenMedLargeDown.nav-item.reactPortal",
        timeout=5,
    )
    if not lg:
        return
    time.sleep(0.5)
```

```
    lg.click()

    google = page.ele("text=Log in with Google")
    time.sleep(0.5)
    google.click()

    if not new_page and (e := page.ele("text=Use other account", timeout=5)):
        e.click()

    un = page.ele("@aria-label=Email or phone", timeout=5)
    if not un:
        return
    time.sleep(0.5)
    un.input(username)

    nt = page.ele("text=Next")
    time.sleep(0.5)
    nt.click()

    pwd = page.ele("@aria-label=Enter your password")
    time.sleep(0.5)
    pwd.input(password)

    nt = page.ele("text=Next")
    time.sleep(0.5)
    nt.click()

def save_img(contest_name, submission_name, number, ele):
    """Save images"""
    imgs = ele.eles("tag:img")
    for idx, img in enumerate(imgs, start=number):
        Path(f"crawler/data/image/{contest_name}/{submission_name}").mkdir(parents=True,
            exist_ok=True)
        with open(f"crawler/data/image/{contest_name}/{submission_name}/{submission_name}-{idx
            }.png", mode="wb") as f:
            rep = requests.get(img.attr("src"))
            f.write(rep.content)
            time.sleep(0.5)

    return len(imgs) + number

def move_screen(page):
    """Move the screen to the bottom of the page efficiently."""
    scroll_step = 400  # Scroll by larger increments to reach the bottom faster
    max_retries = 5    # Stop if the position doesn't change after a few tries
    retries = 0

    while retries < max_retries:
        previous_position = page.rect.page_location[1]
        page.run_js(f"window.scrollBy(0, {scroll_step});")
        time.sleep(0.2)  # Short delay to allow content to load if necessary

        new_position = page.rect.page_location[1]
        if new_position == previous_position:
            retries += 1  # Increment retries if no movement
        else:
            retries = 0  # Reset retries if there was movement

def extract_youtube_video_id(url):
```

```
    """Extract the YouTube video ID from a URL."""
    try:
        if not url.startswith(('http://', 'https://')):
            url = 'https:' + url
        parsed_url = urllib.parse.urlparse(url)
        if parsed_url.hostname == 'youtu.be':
            return parsed_url.path[1:]
        elif parsed_url.hostname in ('www.youtube.com', 'youtube.com'):
            if parsed_url.path == '/watch':
                query = urllib.parse.parse_qs(parsed_url.query)
                return query.get('v', [None])[0]
            elif parsed_url.path.startswith('/embed/'):
                return parsed_url.path.split('/')[2]
            elif parsed_url.path.startswith('/v/'):
                return parsed_url.path.split('/')[2]
        return None
    except Exception as e:
        logger.error(f"Error extracting YouTube video ID from URL {url}: {e}")
        return None

def get_youtube_video_duration(video_id):
    """Get the duration of a YouTube video in seconds."""
    try:
        youtube = build('youtube', 'v3', developerKey=YOUTUBE_API_KEY)
        response = youtube.videos().list(part='contentDetails', id=video_id).execute()
        items = response.get('items', [])
        if items:
            duration = items[0]['contentDetails']['duration']
            return isodate.parse_duration(duration).total_seconds()
        else:
            logger.warning(f"No details found for YouTube video ID {video_id}")
            return 0
    except HttpError as e:
        logger.error(f"YouTube API error: {e}")
        return 0
    except Exception as e:
        logger.error(f"Error getting YouTube video duration: {e}")
        return 0

def extract_vimeo_video_id(url):
    """Extract the Vimeo video ID from a URL."""
    try:
        if not url.startswith(('http://', 'https://')):
            url = 'https:' + url
        parsed_url = urllib.parse.urlparse(url)
        if 'player.vimeo.com' in parsed_url.hostname:
            return parsed_url.path.split('/')[-1]
        elif 'vimeo.com' in parsed_url.hostname:
            return parsed_url.path.strip('/')
        return None
    except Exception as e:
        logger.error(f"Error extracting Vimeo video ID from URL {url}: {e}")
        return None

def get_vimeo_video_duration(video_id):
    """Get the duration of a Vimeo video in seconds."""
    try:
        headers = {'Authorization': f'Bearer {VIMEO_ACCESS_TOKEN}'}
        response = requests.get(f'https://api.vimeo.com/videos/{video_id}', headers=headers)
```

```
        if response.status_code == 200:
            return response.json().get('duration', 0)
        else:
            logger.error(f"Vimeo API error {response.status_code}: {response.text}")
            return 0
    except Exception as e:
        logger.error(f"Error getting Vimeo video duration: {e}")
        return 0

def save_video_link(contest_name, submission_name, link, platform, length):
    """Save the video link data (YouTube, Vimeo, or unknown) to a CSV file."""
    csv_file_path = "your_csv_file_path.csv"
    file_exists = os.path.isfile(csv_file_path)

    # Open the CSV file in append mode
    with open(csv_file_path, mode='a', newline='', encoding='utf-8') as file:
        writer = csv.writer(file)
        if not file_exists:
            # Write the header if the file doesn't exist
            writer.writerow(["contest_name", "submission_name", "link", "platform", "length"])
        # Write the video link data
        writer.writerow([contest_name, submission_name, link, platform, length])

def log_error(contest_name, submission_name, url, error_message):
    """Log the error to a CSV file."""
    log_file_path = "your_log_file_path.csv"
    file_exists = os.path.isfile(log_file_path)

    with open(log_file_path, mode='a', newline='', encoding='utf-8') as file:
        writer = csv.writer(file)
        if not file_exists:
            writer.writerow(["contest_name", "submission_name", "url", "error_message"])
        writer.writerow([contest_name, submission_name, url, error_message])

def get_data(contest_name, submission_name, url, get_all_overview=False, save_images=False):
    """Get the data and handle errors appropriately, logging any issues."""
    try:
        if get_all_overview:
            overview = page.ele("#overview")
            content = overview.text if overview else ""

        if save_images:
            try:
                number = 1
                number = save_img(contest_name, submission_name, number, overview)
            except Exception as e:
                log_error(contest_name, submission_name, url, f"Error saving images: {e}")

        data = {
            "contest_name": contest_name,
            "submission_name": submission_name,
            "submission_url": url,
            "story": "",
            "image": 0,
            "gif": 0,
            "videos": [],
            "video_duration": 0,
            "things": [],
            "num_things": 0,
```

```
            "cad": 0,
            "schematic": 0,
            "code": 0,
            "code_lines": 0,
            "link": 0,
        }

        description = page.ele("#description").eles("tag:section")
        for section in description:
            try:
                if section.attr("id") == "things":
                    components = section.eles(".hckui__typography__bodyL")
                    data["things"] = [c.text for c in components]
                    data["num_things"] = len(components)

                if section.attr("id") == "story":
                    data["story"] = section.text

                    total_video_duration = 0
                    processed_videos = set()  # Track unique video URLs for the current submission

                    if videos := section.eles(".embed-frame", timeout=2):
                        for video in videos:
                            try:
                                # Check for iframe first, then fallback to anchor tag
                                iframe = video.ele("tag:iframe", timeout=5)
                                anchor = video.ele("tag:a", timeout=5)
                                src = iframe.attr("src") if iframe else (anchor.attr("href") if anchor else None)

                                if src and src not in processed_videos:
                                    # Add the unique src to the set for this submission only
                                    processed_videos.add(src)
                                    data["videos"].append(src)

                                    platform = "unknown"
                                    length = None

                                    # Process YouTube links
                                    if 'youtube.com' in src or 'youtu.be' in src:
                                        platform = "youtube"
                                        video_id = extract_youtube_video_id(src)
                                        length = get_youtube_video_duration(video_id) if video_id else None
                                        total_video_duration += length if length else 0

                                    # Process Vimeo links
                                    elif 'vimeo.com' in src:
                                        platform = "vimeo"
                                        video_id = extract_vimeo_video_id(src)
                                        length = get_vimeo_video_duration(video_id) if video_id else None
                                        total_video_duration += length if length else 0

                                    # Save video data to CSV (platform can be "youtube", "vimeo", or "unknown")
                                    save_video_link(contest_name, submission_name, src, platform, length)
```

```

                        except Exception as e:
                            # Log specific missing element cases for better debugging
                            missing_element = "iframe" if not iframe else "anchor" if not anchor else "unknown"
                            log_error(contest_name, submission_name, url, f"Error processing video (missing {missing_element}): {e}")

                data["video_duration"] = total_video_duration

                try:
                    if imgs := section.eles("tag:img", timeout=2):
                        data["image"] = len(imgs)
                except Exception as e:
                    log_error(contest_name, submission_name, url, f"Error processing images: {e}")

                try:
                    if gifs := section.eles("tag:video", timeout=2):
                        data["gif"] = len(gifs)
                except Exception as e:
                    log_error(contest_name, submission_name, url, f"Error processing GIFs: {e}")

            if section.attr("id") == "cad":
                try:
                    cad_elements = section.eles(".project-attachment")
                    data["cad"] = len(cad_elements)
                except Exception as e:
                    log_error(contest_name, submission_name, url, f"Error processing CAD elements: {e}")

            if section.attr("id") == "schematics":
                try:
                    schematic_elements = section.eles(".project-attachment")
                    data["schematic"] = len(schematic_elements)
                except Exception as e:
                    log_error(contest_name, submission_name, url, f"Error processing schematics: {e}")

            if section.attr("id") == "code":
                data["code"] = 1
                try:
                    section_html = section.html
                    soup = BeautifulSoup(section_html, "html.parser")

                    # Check for lines of code if applicable
                    line_elements = soup.find_all("span", id=lambda x: x and x.startswith("line-"))
                    if line_elements:
                        data["code_lines"] = len(line_elements)

                    # Check for any links indicating external code repositories or downloadable code files
                    code_links = soup.find_all("a", href=lambda href: href and (
                        "github.com" in href or
                        "gitlab.com" in href or
                        "bitbucket.org" in href or
                        "arduino.cc" in href or
```

```python
                            "attachments" in href or
                            href.endswith(".zip") or
                            href.endswith(".tar") or
                            href.endswith(".tar.gz") or
                            href.endswith(".ino")
                        ))

                        # Mark link presence if any matching links are found
                        if code_links:
                            data["link"] = 1

                        # Log error if no code lines or external links are found
                        if data["code_lines"] == 0 and data["link"] == 0:
                            log_error(contest_name, submission_name, url, "No code found in code section")

                    except Exception as e:
                        log_error(contest_name, submission_name, url, f"Error processing code section: {e}")

            except Exception as section_error:
                log_error(contest_name, submission_name, url, f"Error processing section {section.attr('id')}: {section_error}")

        output_dir = f"crawler/data/contest_submissions/{contest_name}"
        os.makedirs(output_dir, exist_ok=True)
        with open(f"{output_dir}/{submission_name}.json", "w", encoding="utf-8") as f:
            json.dump(data, f, ensure_ascii=False, indent=4)

    except Exception as e:
        log_error(contest_name, submission_name, url, f"General error in get_data: {e}")
```

Listing A.13: dynamic_crawl_contests_submissions.py Python Code

A.2.5. Dataset Creator

```python
FILE_PATH = Path(__file__).resolve().parent
DATA_PATH = FILE_PATH / "finalized_data"

CONTEST_DIR = DATA_PATH / "contest_text"
SUBMISSIONS_DIR = DATA_PATH / "contest_submissions"

def get_submission_data(submission_json):
    with open(submission_json, "r", encoding="utf-8") as f:
        data = json.load(f)

    return [
        # submission features
        data["submission_url"],
        data["word_count"],
        data["image"],
        data["gif"],
        len(data["videos"]),
        data["video_duration"],
        data["num_things"],
```

```
        int(data["cad"]),
        int(data["schematic"]),
        int(data["code"]),
        data.get("code_lines", 0),
        int(data["link"]),
        # submission text
        data["submission_name"],
        data["story"],
        data["contest_name"],
        # label
        int(data["winner"]),
    ]

def get_contest_data(contest_json):
    with open(contest_json, "r", encoding="utf-8") as f:
        data = json.load(f)

    return [
        # contest features
        data["apply_for_hardware"],
        data["num_of_submissions"],
        data["num_of_participants"],
        data["word_count"],
        data["prizes_sum"],
        data["num_of_winners"],
        data["num_of_submissions"] / data["num_of_participants"] if data["num_of_participants"]
             > 0 else 0,
        data["contest_begins_date"],
        # contest text
        data["overview"],
    ]

def convert2df():
    # Check if the dataset is already converted
    if Path("dataset.csv").exists():
        return pd.read_csv("dataset.csv")

    # Create a DataFrame with the updated columns
    df = pd.DataFrame(
        columns=[
            # contest features
            "apply_for_hardware",
            "num_of_submissions",
            "num_of_participants",
            "contest_word_count",
            "prizes_sum",
            "num_of_winners",
            "submission_rate",
            "contest_begins_date",
            "overview",
            # submission features
            "submission_url",
            "submission_word_count",
            "num_image",
            "num_gif",
            "num_video",
            "video_duration",
            "num_things",
            "cad",
```

```python
            "schematic",
            "code",
            "code_lines",
            "link",
            "submission_name",
            "story",
            "contest_name",
            "winner",
        ]
    )

    # Iterate over each contest and submission to populate the DataFrame
    for contest in tqdm(CONTEST_DIR.iterdir()):
        contest_name = contest.name
        contest_json = CONTEST_DIR / contest_name / "contest_overview.json"
        contest_submissions = SUBMISSIONS_DIR / contest_name

        if not contest_json.exists():
            logger.warning(f"Missing contest overview for {contest_name}")
            continue

        contest_data = get_contest_data(contest_json)

        for submission in contest_submissions.rglob("*.json"):
            submission_data = get_submission_data(submission)
            combined_data = contest_data + submission_data
            df = pd.concat(
                [df, pd.DataFrame([combined_data], columns=df.columns)],
                ignore_index=True,
            )

    # Filter out incomplete or irrelevant rows
    df = df.dropna(subset=["story"])
    df = df[df["submission_name"] != "Deleted_by_Admin"]
    df = df[df["submission_name"] != "Project_under_progress"]

    # Save the combined dataset as a single CSV file
    df.to_csv("dataset.csv", index=False)

    return df
```

Listing A.14: dataset_creator.py Python Code

A.2.6. Finalizing Dataset

```python
FILE_PATH = Path(__file__).resolve().parent

class DataPreprocessing:
    def __init__(self, original_data_path):
        self.original_data_path = original_data_path
        self.submissions_dir = original_data_path / "contest_submissions"
        self.contest_dir = original_data_path / "contest_text"

        # Get all contest names with submissions
        self.contest_names = [
```

```
            x.name for x in self.submissions_dir.iterdir() if x.is_dir()
        ]

    def check_contests_match(self):
        contest_names = [x for x in self.contest_dir.iterdir() if x.is_dir()]
        contest_sub_dir = [x for x in self.submissions_dir.iterdir() if x.is_dir()]

        a = set([x.name for x in contest_names])
        b = set([x.name for x in contest_sub_dir])
        print(len(a), len(b))
        print(a - b)
        if a == b:
            print("Contests match")
        else:
            print("Contests do not match")
        return

    def set_winner(self, data):
        # Set winner status to False (or 0) for all submissions
        return {**data, "winner": 0}

    def count_words(self, data, key="story"):
        text = data[key]
        count = len(re.findall(r"\w+", text))
        return {**data, "word_count": count}

    def run(self):
        # Loop through all contests
        for contest_name in self.contest_names:
            # Get contest_overview path
            contest_overview = self.contest_dir / contest_name / "contest_overview.json"

            # Read contest_overview
            with open(contest_overview, "r", encoding="utf-8") as f:
                contest_overview_data = json.load(f)

            # Set num_of_winners and num_of_submissions to 0
            contest_overview_data["num_of_winners"] = 0

            # Add overview word count
            contest_overview_data = self.count_words(contest_overview_data, key="overview")

            # Make path if it does not exist
            new_contest_overview = str(contest_overview).replace(
                "your_path",
                "your_new_path"
            )
            Path(new_contest_overview).parent.mkdir(parents=True, exist_ok=True)

            # Write updated contest overview JSON
            with open(new_contest_overview, "w", encoding="utf-8") as f:
                json.dump(contest_overview_data, f, indent=4)

            # Get all submissions paths
            submissions = [
                x for x in (self.submissions_dir / contest_name).iterdir() if x.is_file()
            ]

            # Process each submission JSON file and update with winner status
```

```
            for submission in submissions:
                # Read JSON file
                with open(submission, "r", encoding="utf-8") as f:
                    data = json.load(f)

                # Set winner status to 0 and count words
                data = self.set_winner(data)
                data = self.count_words(data)

                # Make path if it does not exist
                new_submission = str(submission).replace(
                    "your_path",
                    "your_new_path"
                )
                Path(new_submission).parent.mkdir(parents=True, exist_ok=True)

                # Write updated submission JSON
                with open(new_submission, "w", encoding="utf-8") as f:
                    json.dump(data, f, indent=4)
```

Listing A.15: finalize_data.py Python Code

A.2.7. Data Preprocessing

```
class DataPreprocessor:
    def __init__(self):
        # Initialize paths
        self.base_path = Path(__file__).resolve().parent
        self.dataset_path = self.base_path / "dataset"
        self.to_delete_path = self.base_path / "to_delete"

        # Load datasets
        self.dataset = pd.read_csv(self.dataset_path / "dataset.csv")
        self.winners_labelled = pd.read_csv(self.dataset_path / "winners_labelled.csv")

        # Initialize 'winner_categories' column in dataset
        self.dataset['winner_categories'] = 0

        # Load contests to exclude
        with open(self.to_delete_path / "contests_to_delete.txt", "r") as f:
            self.contests_to_exclude = [line.strip() for line in f if line.strip()]

        # Load submissions to delete
        with open(self.to_delete_path / "submissions_to_delete.txt", "r") as f:
            self.links_to_delete = [line.strip() for line in f if line.strip()]

        # Define a mapping for timezones
        self.tzinfos = {
            'PDT': gettz('America/Los_Angeles'),  # Pacific Daylight Time
            'PST': gettz('America/Los_Angeles'),  # Pacific Standard Time
            }

        # Initialize not_found list
        self.not_found = []

```

```python
    def locate_winners(self, include_category_1=False):
        """
        Update dataset with winner information from winners_labelled.csv.

        Args:
            include_category_1 (bool): If True, treat 'winner_category' 1 as a winner and set 'winner' to 1.
        """
        for _, row in self.winners_labelled.iterrows():
            contest_name = row['contest_name']

            # Skip if contest is in the exclusion list
            if contest_name in self.contests_to_exclude:
                continue

            submission_link = row['submission_link']
            winner_category = row['winner']  # This is 1 or 2

            # Find matching row in dataset
            match = (self.dataset['contest_name'] == contest_name) & \
                    (self.dataset['submission_url'] == submission_link)

            if match.any():
                # Update 'winner_categories' with the highest value
                self.dataset.loc[match, 'winner_categories'] = self.dataset.loc[match, 'winner_categories'].apply(
                    lambda x: max(x, winner_category)
                )

                # Update 'winner' column based on winner_category and include_category_1
                if winner_category == 2 or (include_category_1 and winner_category == 1):
                    self.dataset.loc[match, 'winner'] = 1
                else:
                    # Only set 'winner' to 0 if it's not already 1
                    self.dataset.loc[match & (self.dataset['winner'] != 1), 'winner'] = 0
            else:
                # Track submissions not found
                self.not_found.append((contest_name, submission_link))


    def clean_dataset(self):
        """Clean the dataset by removing specified contests and submissions."""
        # Remove entries with contests to delete
        initial_length = len(self.dataset)
        self.dataset = self.dataset[~self.dataset['contest_name'].isin(self.contests_to_exclude)]
        final_length = len(self.dataset)
        print(f"Deleted {initial_length - final_length} entries based on contests to delete.")

        # Remove entries with submission links to delete
        initial_length = len(self.dataset)
        self.dataset = self.dataset[~self.dataset['submission_url'].isin(self.links_to_delete)]
        final_length = len(self.dataset)
        print(f"Deleted {initial_length - final_length} entries based on submission links to delete.")


        # Convert 'contest_begins_date' to datetime format
```

```
        self.dataset['contest_begins_date'] = self.dataset['contest_begins_date'].apply(lambda x: parser.parse(x, tzinfos=self.tzinfos))
        # Sort the dataset by 'contest_begins_date' in ascending order (earliest date first)
        self.dataset = self.dataset.sort_values(by='contest_begins_date').reset_index(drop=True)
        # Track the initial number of rows per contest
        initial_counts = self.dataset.groupby('contest_name').size()
        # Remove duplicates based on 'submission_name', keeping the earliest (first) occurrence
        initial_length = len(self.dataset)
        self.dataset = self.dataset.drop_duplicates(subset='submission_name', keep='first')
        final_length = len(self.dataset)
        # Track the final number of rows per contest
        final_counts = self.dataset.groupby('contest_name').size()
        # Initialize a list to store the results
        duplicate_data = []
        # Calculate the number of duplicate submissions removed per contest
        print(f"Removed {initial_length - final_length} duplicate submissions based on 'contest_begins_date'.")
        # Collect detailed information per contest
        for contest_name in initial_counts.index:
            removed_count = initial_counts[contest_name] - final_counts.get(contest_name, 0)
            if removed_count > 0:
                # print(f"Contest '{contest_name}': Removed {removed_count} duplicate submissions.")
                # Append the contest name and removed count to the list
                duplicate_data.append({'contest_names': contest_name, 'nr_of_duplicates': removed_count})
        # Convert the list to a DataFrame
        duplicates_df = pd.DataFrame(duplicate_data)

        # Update 'num_of_submissions' and 'num_of_winners' for each contest
        submission_counts = self.dataset['contest_name'].value_counts()
        self.dataset['num_of_submissions'] = self.dataset['contest_name'].map(submission_counts)

        winner_counts = self.dataset[self.dataset['winner'] == 1]['contest_name'].value_counts()
        self.dataset['num_of_winners'] = self.dataset['contest_name'].map(winner_counts).fillna(0).astype(int)

    def estimate_missing_videos(self):
        """Estimate missing video durations based on average duration per video."""
        # Calculate average video duration per video
        valid_videos_df = self.dataset[(self.dataset['num_video'] > 0) & (self.dataset['video_duration'] > 0)]
        average_video_duration = int((valid_videos_df['video_duration'] / valid_videos_df['num_video']).mean())
        print(f"Average video duration per video: {average_video_duration} seconds")

        # Fill missing video durations
        condition = (self.dataset['num_video'] > 0) & (self.dataset['video_duration'] == 0)
        self.dataset.loc[condition, 'video_duration'] = self.dataset.loc[condition, 'num_video'] * average_video_duration

        # Convert 'video_duration' to integer
        self.dataset['video_duration'] = self.dataset['video_duration'].astype(int)

    def save_data(self, filename):
        """Save the processed dataset to a CSV file."""
```

```
        output_path = self.dataset_path / filename
        self.dataset.to_csv(output_path, index=False)
        print(f"Dataset saved to {output_path}")

    def run_all_steps(self):
        """Run all preprocessing steps in order."""
        self.locate_winners(include_category_1=False)
        if self.not_found:
            print("The following submissions were not found in the dataset:")
            for contest, link in self.not_found:
                print(f"Contest: {contest}, Submission Link: {link}")
        else:
            print("All submissions from winners_labelled.csv were found and processed.")

        self.clean_dataset()
        self.estimate_missing_videos()
        self.save_data("final_dataset.csv")
```

Listing A.16: data_preprocessor.py Python Code