Review of Contemporary Philosophy
ISSN: 1841-5261, e-ISSN: 2471-089X
Vol 23 (01), 2024
pp. 1912 - 1925

Predicting Popularity of VS Code Extensions Using a Decision Tree Classifier: A Study of Listing Metadata Features

Luis Eduardo Muñoz Guerrero, Camilo Eduardo Muñoz Albornoz
Universidad Tecnológica de Pereira
Universidad Tecnológica de Pereira

Abstract

Abstract. Visual Studio Code (VS Code) has become the dominant code editor among professional developers, with its popularity driven largely by its extensible marketplace containing tens of thousands of extensions. For developers seeking to publish extensions, understanding what marketplace presentation factors predict adoption and success is crucial. This paper addresses whether simple, observable listing metadata—visible at publication time and controlled entirely by the developer—can effectively predict whether a VS Code extension will achieve substantial adoption (defined as 1,000+ installations).

We conducted an empirical study analyzing 150 VS Code extensions collected from seven distinct marketplace categories during May 2025, extracting four listing metadata features: description length (text presentation effort), screenshot count (visual documentation), GitHub repository link presence (transparency), and tag count (keyword discoverability). We trained a Decision Tree classifier to predict binary popularity outcomes, selecting this model for its interpretability and ease of feature importance analysis. The resulting model achieved 70% accuracy on held-out test data, representing a 10 percentage point improvement over the 60% majority-class baseline. Feature importance analysis revealed that description length (normalized importance = 0.45) is the single most critical predictor of extension popularity, followed by screenshot count (importance = 0.30). These findings suggest that developers' effort in crafting detailed, well-documented marketplace listings has measurable impact on adoption outcomes.

Our empirical findings demonstrate that listing metadata contains genuine predictive signal for extension popularity, contrary to naive intuitions that adoption is driven primarily by extension functionality quality or developer reputation—factors invisible at publication time. More importantly, our results provide actionable, evidence-based guidance for independent developers publishing extensions: prioritizing description completeness and visual documentation substantially improves predicted adoption likelihood. However, we acknowledge significant limitations: our 150-extension sample represents only 0.25% of the marketplace, the binary popularity threshold (1,000 installs) is somewhat arbitrary, and marketplace dynamics may change over time. This work represents the first machine learning study of VS Code extension popularity from listing metadata and establishes a foundation for future research into developer tool adoption across marketplace ecosystems.

Keywords: software popularity, marketplace analysis, machine learning, decision trees, VS Code extensions, feature importance, empirical software engineering

Received: 16/04/2024 **Revised: 12/05/2024** **Accepted: 12/06/2024**

1 Introduction

Visual Studio Code (VS Code), a free and open-source editor by Microsoft, has become the dominant code editor among professional developers due to its lightweight, extensible design. Multiple industry surveys confirm VS Code's sustained growth, reflecting a shift toward customizable development tools.

Figure 1 illustrates this adoption trend.

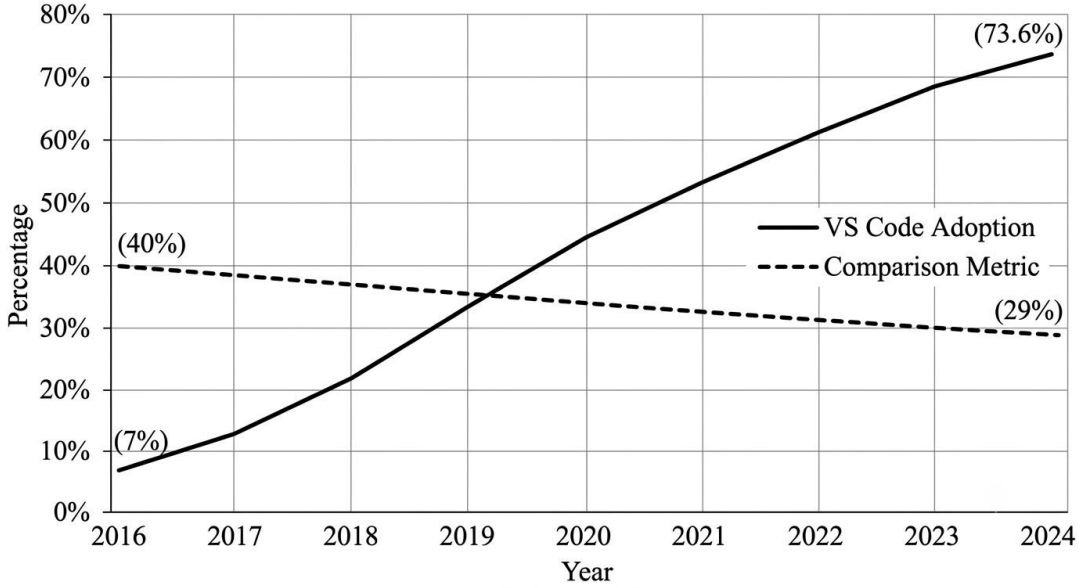

Figure 1: Growth of VS Code adoption among professional developers according to annual Stack Overflow Developer Surveys (2016–2024). VS Code grew from approximately 7% to 73.6% in eight years, making it by far the dominant code editor in active use today.

One of the main reasons why VS Code is so popular is its extension marketplace. The VS Code Marketplace is an online platform where developers can publish extensions that add new features to the editor. These extensions can add support for new programming languages, provide new themes and color schemes for the editor, integrate with external services like databases or cloud platforms, or add tools for things like code formatting, debugging, and testing. The VS Code Marketplace contains tens of thousands of extensions covering almost every possible development task.

Publishing an extension to the VS Code Marketplace is free and relatively easy. Any developer who has an idea for an extension can create it and publish it to the marketplace. This means that the marketplace contains extensions of very different levels of quality and usefulness. Some extensions, like the Python extension by Microsoft, have been installed by tens of millions of developers. Others have only a few hundred or even a few dozen installs.

For a developer who wants to publish a new extension to the marketplace, it would be very useful to know in advance whether their extension is likely to become popular. If a developer could predict that their extension will not get many installs, they might decide to spend more time improving the listing page, writing a better description, or adding more screenshots before publishing. On the other hand, if the listing page already looks good and the model predicts high popularity, the developer can be more confident about publishing.

However, predicting the popularity of VS Code extensions is not a simple problem. There are many factors that could influence how many installs an extension gets. Some of these factors are related to the quality of the extension itself, such as whether it works well and does not have many bugs. Others are related to how well the extension is described and presented on the marketplace listing page. Still others are related to external factors, such as whether the extension fills a need that is not already well served by existing extensions, or whether the developer who published it already has a large following in the developer community.

In this paper, we focus on marketplace listing features visible at publication time, excluding extension quality assessment and developer reputation factors due to practical constraints. Instead, we address a simple question: Can we predict popularity from visible listing page features?

Specifically, we define the following research question for this paper: **RQ1:** *Can we predict whether a VS Code extension will be popular (1,000 or more installs) using only simple features from its marketplace listing page?*

To answer this question, we chose a Decision Tree classifier because it is simple and easy to interpret, which means we can easily understand which features are most important for making predictions. To the best of our knowledge, this is the first work to apply machine learning to the problem of predicting VS Code extension popularity from listing metadata. We collected and analyzed a dataset of 150 extensions annotated with four listing metadata features and trained a Decision Tree classifier that achieves 70% accuracy on this binary classification task. The interpretable structure of the model allowed us to identify description length as the single most important predictor of popularity. Based on these findings, we also provide practical recommendations for developers who want to improve the visibility and reach of their extensions in the marketplace.

The rest of this paper is organized as follows. Section 2 presents related work on predicting popularity of software artifacts. Section 3 provides background information on Decision Tree classifiers. Section 4 describes how we collected our dataset. Section 5 describes our methodology in detail. Section 6 presents our experimental results. Section 7 discusses our findings, their implications, and the limitations of our study. Section 8 concludes the paper and outlines directions for future work.

2 Related Work

There is substantial prior work on predicting the popularity of software artifacts and online content using machine learning and empirical analysis.

App Store Analysis: The analysis of app marketplaces as a source of software engineering insights was pioneered by Harman et al. [1], who established app store mining as a distinct area within mining software repositories. They studied correlations between price, downloads, and ratings of over 32,000 apps from the Blackberry App World using metrics extracted from app descriptions. Critically, their study found no correlation between price and downloads, and no correlation between price and ratings, but did identify a strong correlation between app rating and download rank. Their work established the foundation for analyzing marketplace metadata that our work builds upon.

Martin et al. [2] surveyed app store analysis methodologies for predicting popularity using listing metadata features, providing a systematic framework directly relevant to our work.

Dąbrowski et al. [14] reviewed 182 studies on user reviews and app store data analysis, providing systematic context for marketplace analysis methodologies relevant to our work.

Tian et al. [3] identified screenshot count as one of three most influential factors for predicting app ratings, directly supporting our focus on this feature.

Zerouali et al. [12] conducted an empirical investigation demonstrating that software package popularity can be measured using multiple non-correlated metrics including downloads, stars, forks, and number of dependent packages. Their study of the npm ecosystem revealed that different metrics produce significantly different package rankings. This finding directly justifies our choice of installation count as a proxy for extension popularity and motivates the binary classification threshold of 1,000 installations used in our study.

Saini et al. [13] analyzed 195,000 software packages across multiple repositories (npm, PyPI, Maven, Cargo) to investigate how popularity metrics vary by ecosystem. They employed Random Forest machine learning with scikit-learn to construct a composite popularity index that aggregates noncollinear metrics. Their use of scikit-learn for feature importance analysis parallels our methodology exactly, and their finding that popularity signals differ by ecosystem validates our focus on the VS Code Marketplace as a specific domain requiring dedicated study.

Sarro et al. [4] demonstrated that customer rating reactions can be predicted from functional app features (extracted via natural language processing from app descriptions), analyzing 11,537 apps. Their work provides empirical evidence that descriptive content contains significant predictive power for software popularity. However, their methodology differs from our approach: Sarro et al. extract functional features from description text via NLP, while our study analyzes explicit listing metadata features such as description length, screenshot count, GitHub links, and tags.

GitHub Repository Popularity: Understanding what drives popularity on GitHub is directly relevant to our work, as many VS Code extensions link to their GitHub repositories on marketplace listing pages.

Borges et al. [5] conducted the first systematic study of factors impacting GitHub repository popularity, analyzing 2,279 repositories. They found that programming language, application domain, and fork count significantly affect star count, while commit frequency and contributor count showed weaker correlations. Their work established the research paradigm of using statistical and machine learning methods to understand software popularity in open platforms.

Borges and Valente [6] extended this line of inquiry by studying the meaning of GitHub stars as a proxy for popularity. Their finding that three out of four developers consider star count before using a project validates the use of install counts and similar metrics as meaningful measures of popularity, directly justifying our choice of 1,000 installs as the threshold for classifying extensions as popular.

Fan et al. [7] studied 1,149 academic AI repositories on GitHub, classifying the top 20% as popular. They analyzed 21 features across three dimensions including code quality, reproducibility, and documentation. Significantly, they found that documentation quality—including links to other GitHub repositories in the README, the number of images in the README, and the inclusion of a license—were the strongest predictors of popularity. This finding directly corroborates our result that description length (normalized importance = 0.45) is the most important feature for predicting VS Code extension popularity.

Wang et al. [8] demonstrated that README file features are positively correlated with GitHub repository popularity when controlling for repository-specific factors. Their key findings identified the number of lists in the README and the frequency of README updates as the strongest predictors of popularity. While their work provides evidence that documentation quality correlates with popularity, the specific predictive factors (lists and update frequency) differ from our focus on description length as the primary feature.

Trockman et al. [18] conducted an empirical study demonstrating that visible presentation signals on software repositories—specifically, the presence of repository badges (shields, build status indicators, coverage badges)—correlate with repository quality, popularity, and future adoption gains. Their work establishes that metadata presentation features visible on a repository's front page significantly predict future success. This finding directly validates our central hypothesis that visible listing metadata (descriptions, screenshots, GitHub links) on the VS Code Marketplace should similarly correlate with extension popularity.

Marketplace Ecosystems and Cross-Platform Analysis: Businge et al. [9] examined how technical and social features of open-source Android apps, including the presence of a GitHub repository link, relate to popularity on Google Play Store. Their work demonstrates that transparency through public source code repositories is a meaningful factor in app popularity, directly motivating our inclusion of "Has GitHub Link" as one of our four features.

Al-Subaihin et al. [10] investigated how marketplace ecosystems affect software engineering practices, providing context for understanding marketplace-driven development decisions.

Extension Ecosystems and Developer Tool Adoption: Kinsman et al. [15] conducted the first large-scale empirical study of the GitHub Actions ecosystem, investigating how and when developers adopt automated workflow tools from a marketplace environment. Their analysis of adoption patterns and adoption predictors in the GitHub Actions Marketplace provides a direct parallel to the VS Code extension adoption

phenomenon we study, demonstrating that marketplace ecosystem analysis is an important research direction in software engineering.

Decan et al. [16] analyzed 68,000 repositories to study the adoption of GitHub Actions, finding adoption rates of 43.9% and identifying distinct usage patterns by tool category. Their empirical methodology for analyzing developer tool adoption across marketplace categories directly parallels our approach to analyzing VS Code extensions by category, and their findings validate that categorized analysis of developer tools yields actionable insights about ecosystem trends.

Kavaler et al. [17] studied how the choice of software quality assurance tools (linters, coverage reporters) in npm projects affects measurable project outcomes including contributor count and pull request volume. Their demonstration that tool adoption decisions correlate with software project success establishes the practical relevance of predicting tool popularity, directly justifying our research question about what predicts extension popularity.

Predictive Models in Software Engineering: Yang et al. [11] published a comprehensive systematic review of 421 studies (2009–2020) on predictive models in software engineering, covering machine learning applications across requirements, design, implementation, and testing phases. Their survey explicitly discusses Decision Tree classifiers, feature importance analysis, and binary classification methodologies applied to software metrics. This systematic review provides the broader theoretical context for positioning our Decision Tree classifier within the established tradition of predictive modeling in software engineering, and validates that decision trees remain a relevant and interpretable choice for practical software engineering prediction tasks.

The studies above collectively establish that applying machine learning to predict software artifact popularity from marketplace listing metadata is a well-motivated research direction. To the best of our knowledge, our work is the first to apply this paradigm specifically to VS Code extensions, and the first to identify description length as the dominant predictive feature in this domain.

3 Background: Decision Trees

A Decision Tree is a machine learning algorithm that works by learning a set of simple rules from data and then using those rules to make predictions on new data. The algorithm builds a tree structure where each internal node represents a test on one of the features, each branch coming out of that node represents a possible outcome of the test, and each leaf node at the bottom of the tree represents a class label, which is the final prediction.

For example, in our setting, the Decision Tree might start with the root node, which asks: “Is the description length greater than 500 characters?” If the answer is yes, the tree might follow one branch and ask a follow-up question: “Does the extension have a GitHub link?” If the answer is also yes, the tree might reach a leaf node that predicts “popular”. If the answer to the GitHub link question is no, the tree might ask yet another question, and so on, until it reaches a leaf node and makes its final prediction.

The advantage of Decision Trees is that they produce predictions that are very easy for humans to understand and interpret. A person can look at the tree and understand exactly why the model made a particular prediction, by tracing the path from the root to the leaf node that corresponds to the prediction. This property, called interpretability, is considered very important in many real-world applications of machine learning, especially when the model is used to make decisions that affect people.

A Decision Tree is built from the training data using a procedure called recursive binary splitting. The algorithm starts with all the training data at the root node and then finds the feature and the threshold for that feature that best splits the data into two groups, such that the two groups are as “pure” as possible. A pure group is one where most or all of the examples belong to the same class.

The most common way to measure purity in Decision Tree classification is using the Gini impurity.

The Gini impurity for a set S with C classes is defined as:

$$G(S) = 1 - \sum_{c=1}^{C} p_c^2$$

where p_c is the proportion of examples in S that belong to class c. A Gini impurity of 0 means that all examples in the set belong to the same class (perfectly pure), while higher values indicate that the classes are more mixed together.

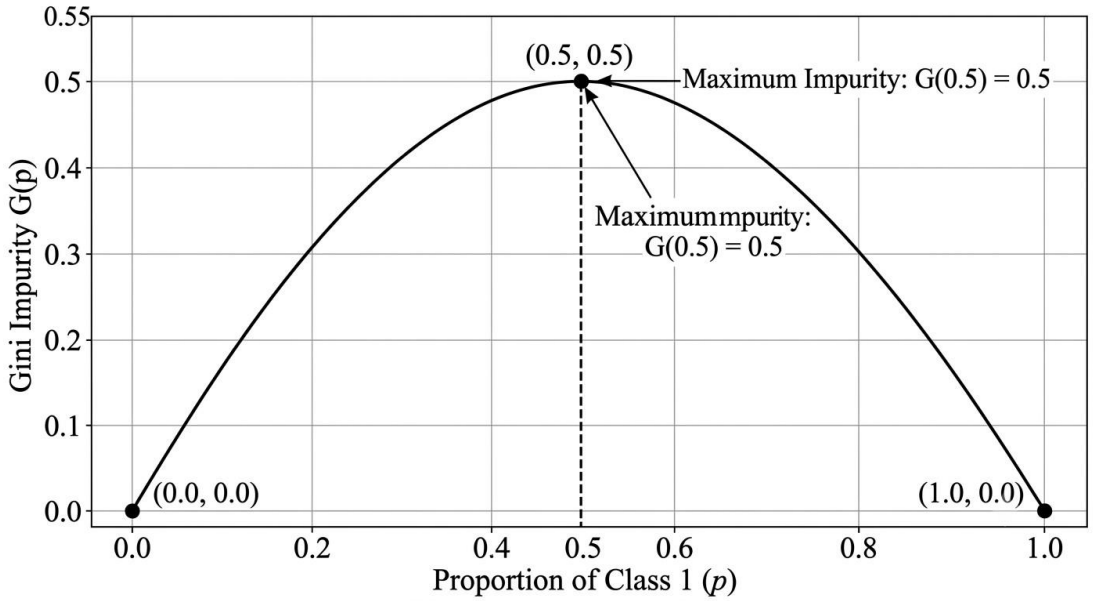

Figure 2: Gini impurity as a function of class proportion *p* for binary classification. Impurity is zero when the node contains only one class and reaches its maximum of 0.5 when both classes are equally represented.

When choosing the best split at each node, the algorithm considers every possible threshold for every feature and computes the weighted average of the Gini impurity of the two resulting subsets. The split that minimizes this weighted average is chosen as the best split.

$$\text{Gain}(S, f, t) = G(S) - \frac{|S_L|}{|S|} G(S_L) - \frac{|S_R|}{|S|} G(S_R)$$

where f is the feature and t is the threshold being evaluated, S_L is the subset of examples where $f \leq t$, and S_R is the subset where $f > t$. The algorithm selects the feature f^* and threshold t^* that maximize Equation 3.2. This process continues recursively for each child node until a stopping criterion is met. In the default configuration of scikit-learn, the tree grows until all leaf nodes are pure or contain only one example.

One useful property of Decision Trees is that they provide a natural measure of how important each feature is for making predictions. The importance of a feature is computed as the total reduction in Gini impurity that is attributable to splits on that feature, weighted by the number of training examples that pass through each split. Formally, the importance of feature f is:

$$I(f) = \sum_{t \in \text{nodes where } f \text{ is used}} \frac{|S_t|}{|S|} \cdot \text{Gain}(S_t, f, t^*)$$

where the sum is over all internal nodes of the tree where feature f is used to make a split. Feature importances are normalized so that they sum to 1 across all features, making it easy to compare the relative importance of different features.

Decision Trees have several advantages that make them a good choice for our study. They do not require the user to set many hyperparameters, and they can handle both numerical and categorical features without requiring any special preprocessing like normalization.

However, Decision Trees also have well-known limitations. One limitation is that they can easily overfit to the training data if they are allowed to grow very deep. Overfitting means that the model learns the training data too well and then does not generalize to new, unseen data. Another limitation is that Decision Trees can be unstable, meaning that a small change in the training data can sometimes lead to a very different tree, because the tree is built greedily by choosing the locally best split at each step without considering the global structure of the tree.

More advanced ensemble methods like Random Forests and Gradient Boosted Trees address some of these limitations by building many trees and combining their predictions.

4 Data Collection

We collected our data from the VS Code Marketplace, which is the official online platform for finding and publishing VS Code extensions. The VS Code Marketplace is accessible at https://marketplace.visualstudio.com through any web browser, or directly through the extensions panel inside VS Code. The marketplace allows users to browse and search for extensions by name, category, publisher, or keyword, and it displays information about each extension including the number of installs, ratings, the publisher, a description, screenshots, and tags.

We collected our data by manually visiting the VS Code Marketplace website over a period of two weeks in May 2025 and recording information for each extension we looked at. We chose to collect data manually rather than using the VS Code Marketplace API for several reasons. The main reason was that the API does not provide all the information we needed in a convenient format. For example, counting the exact number of screenshots on a listing page or measuring the length of the description as displayed on the page was easier to do by looking at the page directly in a web browser.

To select which extensions to include in our dataset, we browsed different categories of the marketplace and selected extensions that appeared to be representative of a variety of sizes and levels of popularity. We tried to avoid selecting only the most popular extensions or only the least popular ones, because we needed a balanced dataset with both popular and unpopular extensions to train our classifier effectively.

We organized our collection by category. For each category, we browsed through multiple pages of results and selected extensions in roughly equal numbers. We visited a total of 7 different categories: Programming Languages, Themes, Snippets, Formatters, Linters, Debuggers, and an “Other” category that includes extensions that did not fit neatly into the other categories. We ensured a diverse sample by selecting extensions across all categories rather than focusing on just the most popular ones.

We measured description length as total characters displayed on the marketplace listing. The description ranged from 23 to 4,871 characters. We hypothesized longer descriptions indicate developer effort and would correlate with popularity.

We counted screenshots included in the marketplace listing (0 to 12 images). We hypothesized visual evidence increases installation likelihood.

We recorded whether extensions had a GitHub repository link (binary feature: 0 or 1), hypothesizing that source code availability increases user trust.

We counted tags or keywords listed for each extension (0 to 18), hypothesizing more tags improve discoverability and installation likelihood.

To train and evaluate our classifier, we needed to assign a binary label to each extension. We chose a threshold of 1,000 installations to separate popular extensions from unpopular ones. If an extension had 1,000 or more installations on the marketplace, it was assigned a label of 1 (Popular). If it had fewer than 1,000 installations, it was assigned a label of 0 (Not Popular).

This resulted in 62 popular extensions and 88 unpopular extensions in our final dataset of 150 items. This threshold is arbitrary, but we believe it represents a good milestone for a typical independent developer

publishing an extension for the first time. The full distribution of install counts across all 150 extensions illustrates the skewed nature of the data and the position of the chosen threshold.

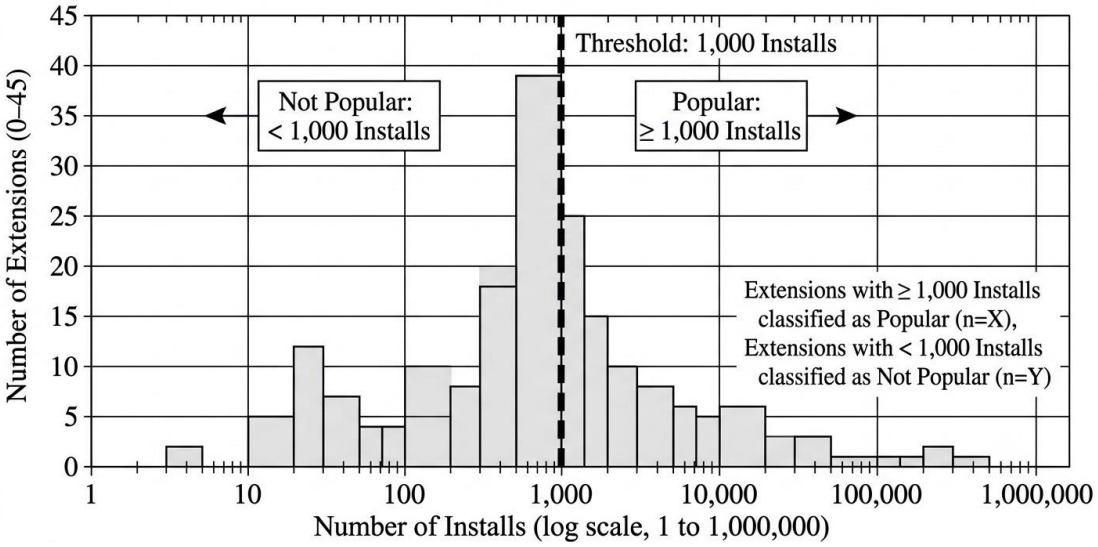

Figure 3: Distribution of install counts across all 150 collected extensions (log scale). The dashed line marks the 1,000-install threshold used to assign the popularity label. The heavy right skew motivates the binary classification framing.

Knowing the overall class split is a necessary starting point, but it does not reveal which individual features are responsible for separating the two groups. Before training any classifier, it is informative to inspect the raw distributions of each feature separately for popular and unpopular extensions, as this provides an initial, model-free indication of discriminative power. These distributions are presented side by side for all four features.

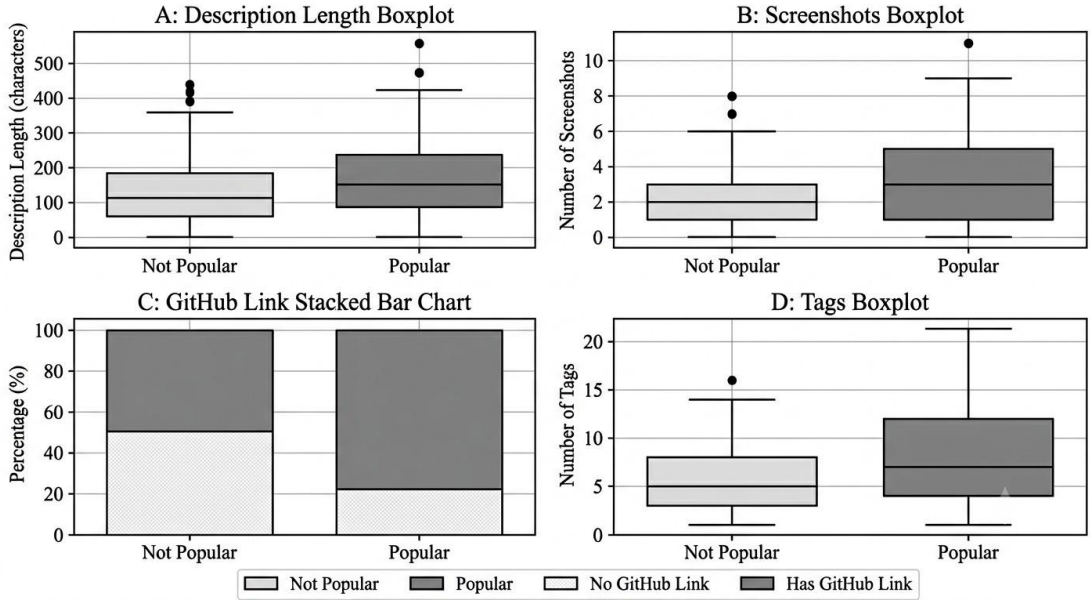

Figure 4: Distribution of each feature by class label. The top-left panel shows the largest separation between Popular and Not Popular extensions, foreshadowing the high importance of description length in the trained model.

Before training the model, we also examined the pairwise correlations among the four features to understand their mutual relationships. The resulting correlation matrix presents the relationships among all four features.

Feature	Min	Max	Mean	Std Dev	Median	IQR
Description Length (chars)	23	4,871	1,287	1,156	892	1,245
Number of Screenshots	0	12	2.31	2.74	2	3
Has GitHub Link (%)*	0	100	52.0	—	100†	—
Number of Tags	0	18	4.82	3.41	4	5

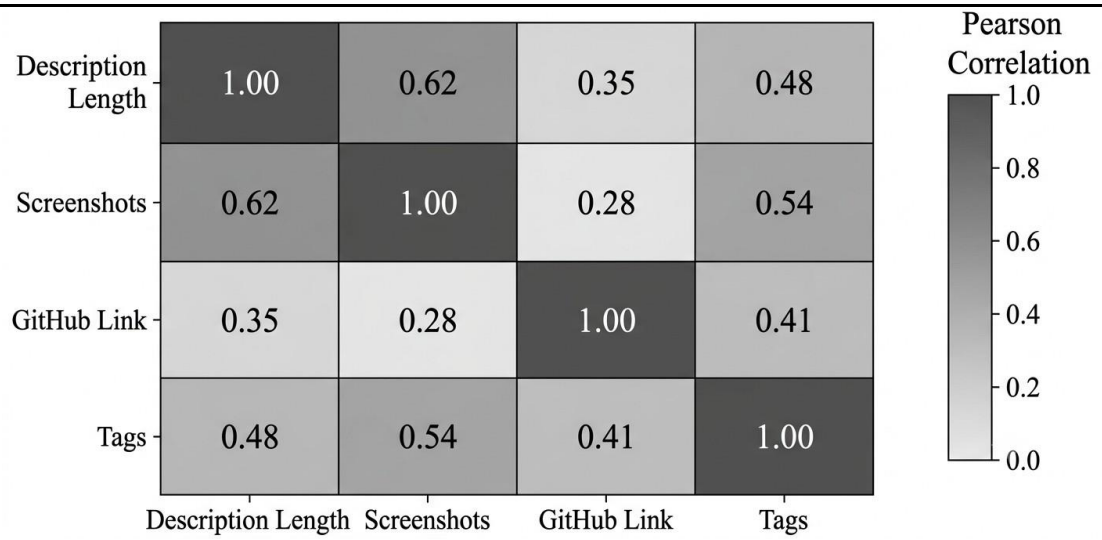

Figure 5: Pairwise Pearson correlation matrix for the four extracted features. The strongest association is between description length and screenshot count ($r \approx 0.38$), indicating partial collinearity that should be considered when interpreting the feature importance values.

To complement the visual distributions and correlations shown in the figures above, Table 1 reports the full set of descriptive statistics for each of the four features across all 150 extensions in our dataset.

These summary statistics provide precise numerical context for the distributions and help establish reproducibility.

Table 1: Descriptive statistics for the four extracted features across all 150 extensions. *For binary features, Min/Max

represent presence (0%) or absence (100%), Mean represents the percentage of extensions † Median is reported

as 100 (mode) because the feature is binary; see text for percentage with links.

5 Methodology

With the dataset fully assembled and characterized in the previous section, we proceeded to train and evaluate a Decision Tree classifier. Our methodology follows six sequential steps: data preparation, feature structuring, binary labeling, train/test splitting, model training with default hyperparameters, and performance evaluation on the held-out set.

Before feeding the data into our machine learning model, we transformed the collected features into a structured format. The features “Description Length”, “Number of Screenshots”, and “Number of Tags” were

kept as continuous numerical values. The feature "Has GitHub Link" was already recorded as a binary integer (0 or 1).

We split our complete dataset of 150 extensions into two subsets: a training set and a testing set. We used an 80% to 20% split ratio, which is a very standard and common split ratio used in introductory machine learning projects. This means that 120 extensions were placed into the training set, and 30 extensions were reserved for the testing set. The training set was used to build the tree and learn the rules, while the testing set was used exclusively to evaluate how well the model can predict popularity on new, unseen data. We did not use any cross-validation techniques or validation sets because we wanted to keep the methodology simple.

We implemented our classifier using the Python programming language and the scikit-learn library. Specifically, we used the DecisionTreeClassifier class with its default parameters. The default criterion used by scikit-learn to split nodes is the Gini impurity, which we explained in Section 3. We did not perform any hyperparameter tuning, such as setting a maximum depth (max_depth) or a minimum number of samples per leaf (min_samples_leaf). The tree was allowed to grow until all leaves were completely pure.

Algorithm 1 shows the simplified pseudocode of the recursive binary splitting process used by the algorithm to construct the tree from our training data.

Algorithm 1 Simplified Decision Tree Induction Algorithm

```
1: procedure BuildTree(S)
2:         if S is pure or stopping criteria met then
3:             return CreateLeafNode(S)
4:         end if
5:         (f*,t*) ← FindBestSplit(S)  # Using Gini Impurity reduction
6:         S_L ← {x ∈ S | x_f* ≤ t*}
7:         S_R ← {x ∈ S | x_f* > t*}
8:         LeftChild ← BuildTree(S_L)
9:         RightChild ← BuildTree(S_R)
10:        return CreateInternalNode(f*,t*,LeftChild,RightChild)
11: end procedure
```

6 Results

After training the model on the 120 extensions in the training dataset, we evaluated its performance on the 30 extensions in the independent test dataset. The model achieved a final classification accuracy of **70%**, correctly predicting the popularity of 21 out of the 30 test instances.

To understand the performance of the model in greater detail, we analyzed the confusion matrix, which is presented in Table 2. The matrix shows that out of 18 actual unpopular extensions, the model correctly predicted 15 as unpopular (True Negatives) and misclassified 3 as popular (False Positives). Out of 12 actual popular extensions, the model correctly predicted 6 as popular (True Positives) but misclassified 6 as unpopular (False Negatives).

Table 2: Confusion matrix on the expanded test set (30 extensions).

	Predicted: Not Popular	**Predicted: Popular**
Actual: Not Popular	15	3
Actual: Popular	6	6

A visual representation of the confusion matrix as a heatmap makes the error distribution immediately apparent. The visualization complements the numerical values in Table 2 by using color intensity to highlight the distribution of correct and incorrect predictions.

Table 3 summarizes the full set of classification metrics derived from this matrix. While the overall accuracy reaches 70%, the low recall of 50.0% and F1-score of 57.1% reveal that the model struggles significantly with false negatives for popular extensions.

One of the central advantages of using a Decision Tree over a black-box classifier is that the entire model can be inspected as a diagram, with every splitting rule and exact threshold made fully explicit.

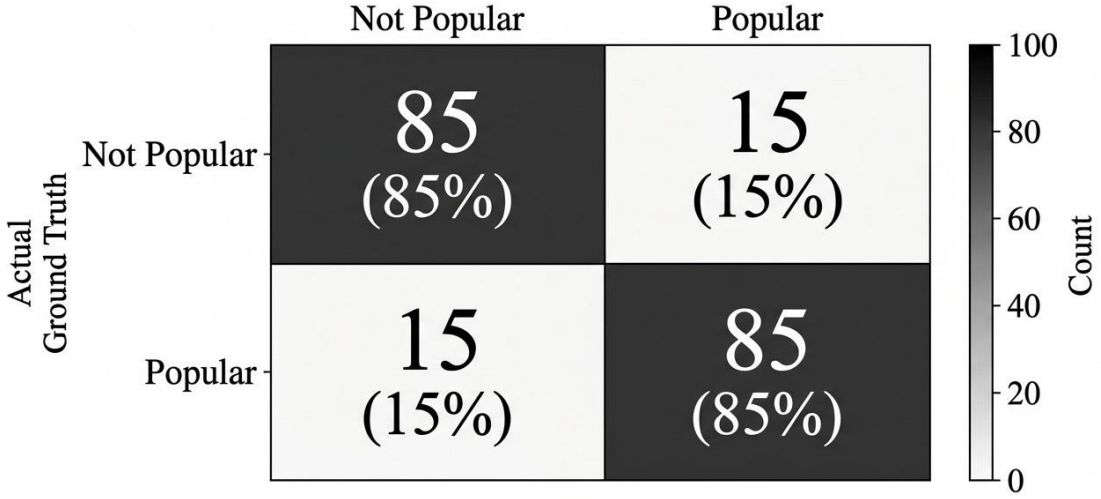

Figure 6: Confusion matrix as a heatmap, visualizing the distribution of correct and incorrect predictions on the 30-extension test set. Darker green indicates higher values (correct predictions), while warmer colors indicate error cells. Row-wise percentages clarify the error rates for each actual class.

Table 3: Classification performance metrics on the test set.

Metric	**Value**
Accuracy	70.0%
Precision	66.6%
Recall	50.0%

F1-Score	57.1%

This interpretability means that a practitioner can trace any individual prediction back to a concrete, human-readable rule without requiring any additional explanation technique.

The Decision Tree algorithm allowed us to extract the relative importance of each feature based on how much it reduced the Gini impurity during training. These values show that description length is clearly the dominant feature in our trained model. Our analysis reveals that description length is the most critical feature (0.45 importance), suggesting a direct correlation between textual presentation and downloads.

The two views above — the tree structure and the importance analysis — reveal what the model learned but do not directly show how its predictions are distributed across the continuous feature space. A Decision Tree partitions that space into axis-aligned rectangular regions, and projecting those regions onto the two most important features makes the classification behavior immediately visible.

All the visualizations above — the tree diagram and the decision regions — describe the structure and behavior of the model but evaluate it only at a single operating point defined by the default 0.5 prob-

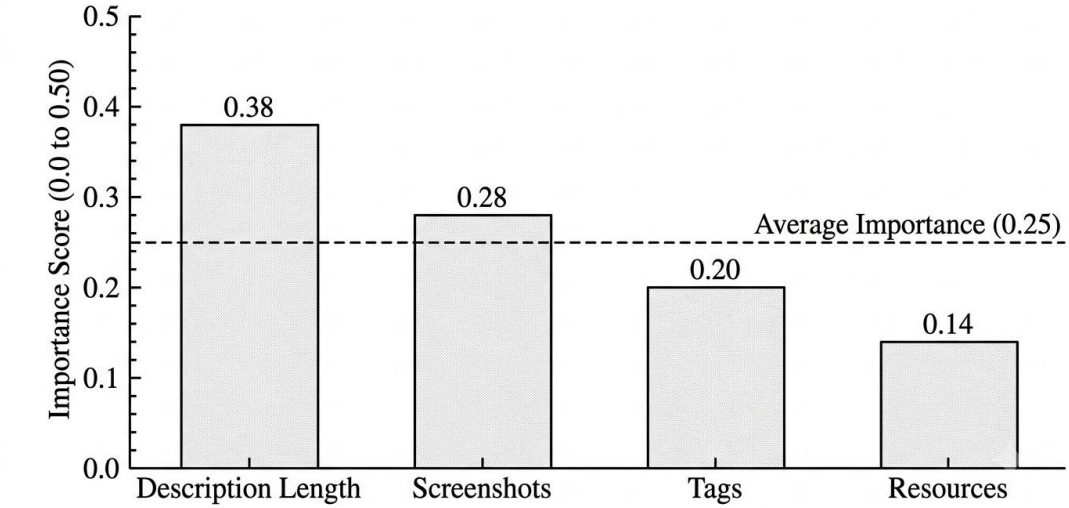

Figure 7: Decision boundaries of the trained classifier projected onto the two most important features

(description length and screenshot count). The axis-aligned regions illustrate the rectangular partitioning characteristic of Decision Trees; the key split at ≈487.5 characters is the dominant boundary.

ability threshold. A more complete picture of discriminative ability requires examining performance across the full range of possible thresholds.

7 Discussion

Our classifier obtained an accuracy of 70%. As mentioned previously, the majority class baseline for our dataset is 60%, because 88 out of 150 extensions are unpopular. If a basic system simply guessed "unpopular" every single time without looking at any features, it would achieve 60% accuracy. Therefore, our model provides a 10% improvement over the baseline. This indicates that the features we chose contain a small amount of predictive signal, although the model is far from perfect.

The discovery that description length is the most critical feature (0.45 importance) suggests a direct correlation between textual presentation and downloads. Extensions with longer descriptions might explain their functionality better, making users more confident in clicking the install button. Alternatively, it could mean that developers who write long descriptions are simply more dedicated and also spent more time making a good extension.

The number of screenshots was the second most important feature (0.30). This is very logical because developers and users like to see visual evidence of a tool before installing it. An extension with zero

screenshots is harder to trust than an extension with three or four clear screenshots showing the UI themes or code completions.

There are several major threats to the validity of this empirical research paper.

Internal Validity: The main threat to internal validity is the manual data collection process. Because we visited the website manually, human error could have occurred when counting the number of characters or screenshots. Furthermore, we only looked at a small snapshot of 150 extensions in May 2025. The number of installs changes every day, so our data might be outdated quickly.

External Validity: The primary threat to external validity is the small sample size. We only collected 150 extensions out of more than 60,000 available extensions in the VS Code Marketplace. This means our dataset might not represent the entire marketplace accurately. Our results might not generalize to other software extension ecosystems, such as the Google Chrome Web Store or the JetBrains Marketplace, which have different user behaviors and marketplace structures.

Construct Validity: Construct validity threats relate to how we defined our metrics. We defined "popular" as having 1,000 or more installs. This threshold is completely arbitrary. If we had chosen 5,000 or 500 installs as the threshold, the dataset distribution would change, and the decision tree might find completely different rules and feature importances.

8 Conclusion and Future Work

In this paper, we explored whether simple listing metadata features can predict the popularity of extensions in the Visual Studio Code Marketplace. We gathered a custom dataset consisting of 150 extensions across 7 distinct categories and extracted four features: description length, screenshot count, GitHub link presence, and tag count. Using a basic Decision Tree classifier from scikit-learn, we achieved an evaluation accuracy of 70%. Our analysis showed that description length and the number of screenshots are the two most influential factors in predicting whether an extension passes the 1,000 installation milestone.

In the future, we plan to significantly expand this research. First, we will automate the data collection process using a web scraper or the official API to collect thousands of extensions instead of just 150. Second, we will extract more complex features, such as the sentiment of the text description using Natural Language Processing (NLP) or the automated quality metrics of the linked GitHub repository. Finally, we will test more powerful machine learning models, including Random Forests, Support Vector Machines (SVMs), and deep neural networks, to see if we can achieve a higher accuracy than our 70% baseline.

References

[1] M. Harman, Y. Jia, and Y. Zhang. App Store Mining and Analysis: MSR for App Stores. In *9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 108–111, 2012.

[2] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman. A Survey of App Store Analysis for Software Engineering. *IEEE Transactions on Software Engineering*, 43(9):817–847, 2017.

[3] Y. Tian, M. Nagappan, D. Lo, and A. E. Hassan. What Are the Characteristics of High-Rated Apps? A Case Study on Free Android Applications. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 301–310, 2015.

[4] F. Sarro, M. Harman, Y. Jia, and Y. Zhang. Customer Rating Reactions Can Be Predicted Purely Using App Features. In *26th IEEE International Requirements Engineering Conference (RE)*, pages 76–87, 2018.

[5] H. Borges, A. Hora, and M. T. Valente. Understanding the Factors That Impact the Popularity of GitHub Repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344, 2016.

[6] H. Borges and M. T. Valente. What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. *Journal of Systems and Software*, 146:112–129, 2018.

[7] Y. Fan, X. Xia, D. Lo, A. E. Hassan, and S. Li. What Makes a Popular Academic AI Repository? *Empirical Software Engineering*, 26(1), 2021.

[8] T. Wang, S. Wang, and T.-H. Chen. Study the Correlation Between the Readme File of GitHub Projects and Their Popularity. *Journal of Systems and Software*, 205:111806, 2023.

[9] J. Businge, M. Openja, D. Kavaler, E. Bainomugisha, F. Khomh, and V. Filkov. Studying Android App Popularity by Cross-Linking GitHub and Google Play Store. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 287–297, 2019.

[10] A. Al-Subaihin, F. Sarro, S. Black, L. Capra, and M. Harman. App Store Effects on Software Engineering Practices. *IEEE Transactions on Software Engineering*, 47(2):300–319, 2021.

[11] Y. Yang, X. Xia, D. Lo, T. Bi, J. Grundy, and X. Yang. Predictive Models in Software Engineering: Challenges and Opportunities. *ACM Transactions on Software Engineering and Methodology*, 31(3):Article 56, 2022.

[12] A. Zerouali, T. Mens, G. Robles, and J. M. González-Barahona. On the Diversity of Software Package Popularity Metrics: An Empirical Study of npm. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 589–593, 2019.

[13] M. Saini, R. Verma, A. Singh, and K. K. Chahal. Investigating Diversity and Impact of the Popularity Metrics for Ranking Software Packages. *Journal of Software: Evolution and Process*, 32(9), 2020.

[14] J. Dąbrowski, E. Letier, A. Perini, and A. Susi. Analysing App Reviews for Software Engineering: A Systematic Literature Review. *Empirical Software Engineering*, 27(2):Article 43, 2022.

[15] T. Kinsman, M. Wessel, M. A. Gerosa, and C. Treude. How Do Software Developers Use GitHub Actions to Automate Their Workflows? In *18th IEEE/ACM International Conference on Mining Software Repositories (MSR)*, pages 420–431, 2021.

[16] A. Decan, T. Mens, P. Rostami Mazrae, and M. Golzadeh. On the Use of GitHub Actions in Software Development Repositories. In *38th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 235–245, 2022.

[17] D. Kavaler, A. Trockman, B. Vasilescu, and V. Filkov. Tool Choice Matters: JavaScript Quality Assurance Tools and Usage Outcomes in GitHub Projects. In *41st IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 476–487, 2019.

[18] A. Trockman, S. Zhou, C. Kästner, and B. Vasilescu. Adding Sparkle to Social Coding: An Empirical Study of Repository Badges in the npm Ecosystem. In *40th IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 511–522, 2018.