Spectrum of Engineering Sciences
ISSN (e) 3007-3138 (p) 3007-312X
Volume 3, Issue 5, 2025

Duplicate Pull Requests: Automated Detection Using S-BERT and Machine Learning

[1]Umar Hayat Khan, [*2]Ashraf Zia, [*3]Hashim Ali, [4]Asif Rahim, [5]Umer Tanveer, [6]Kiran Falak Sher
[1,2,3,5,6]Department of Computer Science, Abdul Wali Khan University Mardan, Pakistan
[4]School of Computer and Information Security, Guilin University of Electronic Technology Guilin 541004, China

[*2]ashrafzia@awkum.edu.pk, [*3]hashimali@awkum.edu.pk

DOI: https://doi.org/10.5281/zenodo.19050504

Keywords
Pull request prioritization; GitHub; machine learning; XGBoost; software engineering; open- source software code review developer productivity

Article History
Received on 12 April 2025
Accepted on 26 May 2025
Published on 28 May 2025

Corresponding Author's:
Ashraf Zia &Hashim Ali

Abstract
In the context of modern pull-based systems like GitHub, identifying and processing duplicate pull requests (PRs) has become a major challenge for integrators of large-scale open-source software systems. With hundreds of PRs being generated on a daily basis, identifying and processing these PRs manually is a time-consuming and costly affair, and the chances of errors are high. This work proposes an automated approach to detect duplicate PRs using the concept of semantic similarity with the help of the popular transfer learning model S-BERT, which measures the semantic similarity between two given pieces of text. We have successfully achieved an accuracy of 78% and an F1 score of 84% using the cosine similarity measure on the S-BERT model with an optimized similarity measure of 0.40. We have also expanded the baseline dataset with 2,000 additional PRs and proposed the use of the XGBoost model to achieve an accuracy of 80.64%. Further, the study proposes the Duplicate Pull Request Detector (DPD) tool and the significance of the tool through a survey among developers.

1 Introduction

GitHub revolutionized collaboration among software teams, making the act of contributing code a social, distributed event. The *pull request* (PR) is the central aspect of this workflow, serving as means employed by developers to suggest changes, fix bugs or develop new features for review before merging[1]. As this PBD (pull-based development) model lowers the barrier of entry for potential contributors but puts a lot of burden on project maintainers. Integrators are bombarded with hundreds of incoming PRs, and need to triage, review and merge contributions within a limited time window[2].

A particular bottleneck within this process is *duplicate pull requests*. These are those when multiple contributors submit PRs that address the same issue or implement similar functionality independently of each other[3]. As a result, for active repositories finding these duplicates manually is not only tedious but also inefficient. Integrators lose cycles examining duplicate code, and contributors may have their engine rejected due to one which was submitted earlier[4]. Such friction may disincentivize community participation, and slow a project's futurity. The workflow in Figure 1 is highly dependent on human judgment, which is not sustainable for large projects.

The first attempts to automate duplicate detection made use of classic information retrievel techniques. Li et al.[3], for example, applied Term Frequency–Inverse Document Frequency (TF-IDF) on the PR titles and descriptions, together with cosine similarity to identify matches. Though this sets a useful baseline, TF-IDF fails to take into account semantic nuance. It ignores context, synonyms or syntactic relationships and treats every word as an independent token[5]. If two PRs describing the same bug fix use different vocabulary, TF-IDF often does not connect them.

New advances in natural language processing (NLP) offer a way forward. Argument embedding based on *Sentence-BERT* (S-BERT) constructs dense vector embeddings that portray sentence meaning beyond pure word overlap[5]. Standard-BERT uses a bidirectional transformer to learn the relationships between words, but if we use it in similarity tasks like semantic textual similarity, by running each input separately and finding their respective embeddings, there is not much benefits of using standard BERT for this purpose since this provides a heavy computational cost for finding neighbor phrases or similar phrases. This study utilizes S-BERT to create an automatic detection structure that captures the *semantics* of PR descriptions and doesn't rely only on keywords.

We realized this approach in a tool we call the *Duplicate Pull Request Detector* (DPD), which was aimed to help integrators identify candidates for duplicates early in their reviewing cycle. The system generates contextual embeddings for the PR (title

+ body) and measures similarity scores with past submissions. We also validated the approach with a supervised machine learning pipeline to ensure robustness. We trained an XGBoost classifier[6] to validate the detection logic, using a base dataset augmented with 2,000 crawled PRs from a Bitcoin library repository.

Our paper makes four specific contributions to the current state-of-the-art:

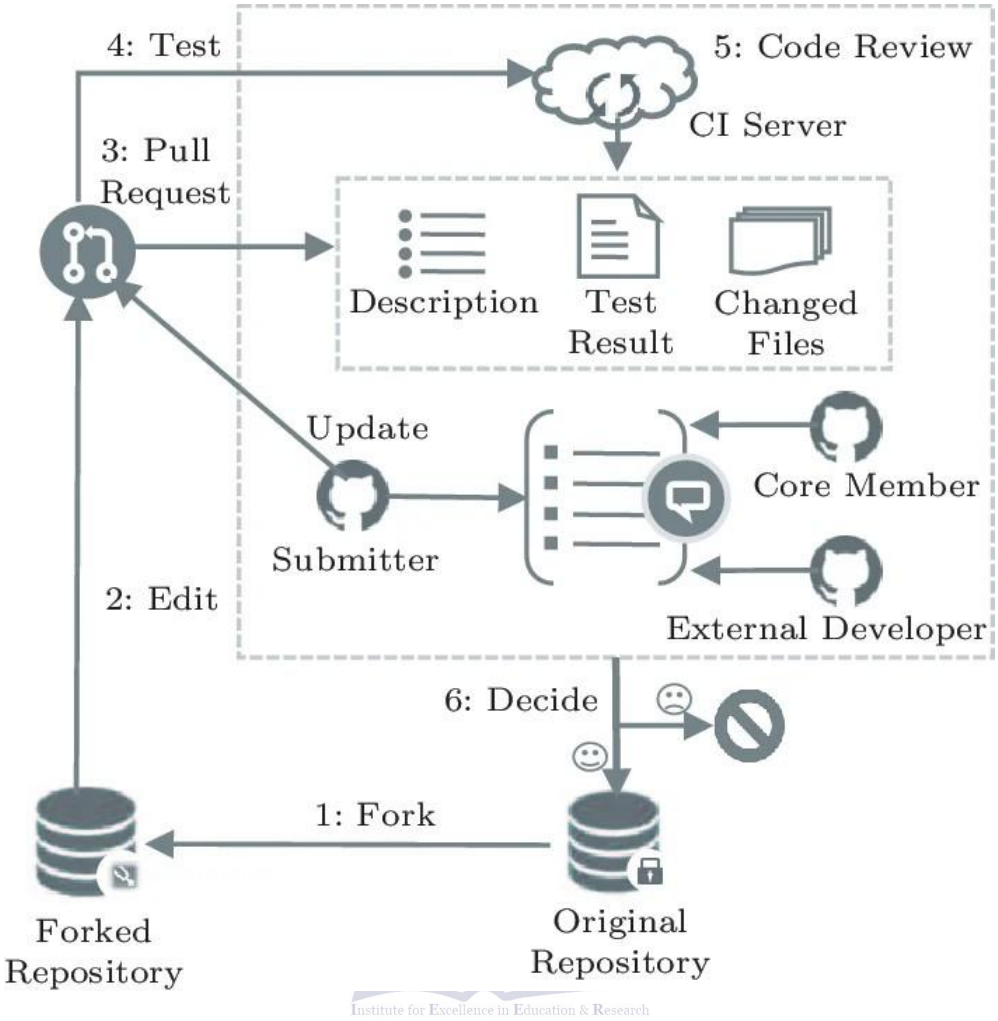

Figure 1. Pull based development Model Diagram

1. **S-BERT Based Detectable:** We present 84% F1 Score with cosine similarity which significantly outperforms TF-IDF base on duplicate detection by using contextual embedding.
2. **Threshold Calibration**: The best precision and recall tradeoff was found to be at a similarity threshold of τ = 0.40 based on empirical findings from 2,323 duplicate pairs across 26 open-source GitHub projects[4].
3. **Vaidation by Machine Learning**: The already existing dataset was balanced such that an XGBoost model achieved 80.64% accuracy, validating our feature representation.
4. **Open-source implementation**: We make available an open-access command line implementation of DPD, includ- ing publicly available code and pre-/post-processing scripts, in accordance with open-science principles to promote reproducibility and broad uptake by the community[7].

PR analytics has been a hot research direction lately. As do studies such as[7] For instance and[8] first utilize *Prioritize* PRs by duration of submission maybe the first two use different methods, but my aim is *deduplication*. Both of these goals are notable for their role in ensuring the whole system remains sustainable: By prioritization useful work gets seen, while at deduplication we save wasted labour on duplicate submissions. The organization of the paper is as follows. We review related literature on duplicate detection and semantic modeling in Section2. Section3 discusses our methodology of S-BERT embeddings and cosine similarity logic. This is described in Section4, along with the XGBoost and classification pipeline as well as dataset enhancement. The experimental evaluation containing live deployment cases and survey feedback is presented in Section5. Finally, in Section 6 we summarize our results and lay out directions for future research.

2 Related Work

Pull-based development has grown in number since the proliferation of GitHub, but existing research does not deal adequately with the specific problem of duplicate submissions compared to other triage concerns. We are mapping existing literature into five perspectives; the reality of pull-based workflows, early text-based detection techniques and current shift toward semantic NLP, machine learning applications in PR analytics; and finally the specific gaps our work addresses.

2.1 Empirical Foundations of Pull-Based Development

There are over 200 million repositories on GitHub, and pull-based development (PBD) has become the de facto standard for distributed collaborative development[1]. However, scale introduces friction. In seminal surveysGousios et al.[2,9] engaged directly with hundreds of integrators and contributors, revealing an acute bottleneck in the resource usage of maintainers: they invest too much time prioritizing incoming requests. They commonly have to fall back on fast heuristics—bug-fix labels, little diffs, or contributors the team knows and trusts–to get through the burden.

The human toll of this overreach is palpable. Steinmacher et al.[10] found that no feedback in 7 days increases the likelihood of first-time contributors abandoning a project by 68%. This number represents two tooling needs in one: we need to surface high-value PRs for merging quickly, but also respond soon enough to keep contributors with us. Although prior work has focused on acceptance prediction[8] or reviewer assignment[11,12], the particular problem of *duplicate* PRs—i.e., unnecessary and redundant effort to fix the same bug—does not draw as much attention even though it directly contributes to wasted review cycles[13].

Traditional Approaches for Duplicate Pull Request Detection

The first efforts to automate whether a PR description is duplicate or not considered it just as a text document. Li et al.[3] was the first to explore this direction by adopting Term Frequency – Inverse Document Frequency (TF-IDF) to extract features from pull request titles and descriptions, and then used cosine similarity to rank candidates. They achieved 55.3% top-5 accuracy via their method tested on Rails, Elasticsearch, and AngularJS. While this set a much-needed baseline, TF-IDF by its nature risks missing semantic nuance. It tokenizes input into words with no regard for the context or synonyms used–this poses a huge problem in the case where developers describe solution to similar fix with different vocabularies[5].

Subsequent studies attempted to make up for it with structural characteristics. Ren et al.[14] produced nine features in five dimensions (patch content, changed files) and trained an AdaBoost classifier to drive precision between 57–83%. Wang et al.[15] observed that submission time shows a periodic pattern, so as to say PRs created closely to each other, are usually duplicates. The addition of this time feature improved F1-scores by 11.93%. More recently, Li et al.[16] combined textual similarity with change similarity (based on the overlap of files) via a greedy search algorithm, attaining 83.4% accuracy.

They demonstrate swift advances, however they're based on surface feature engineering. However, they frequently break on duplicate PRs with reformulated descriptions or when generalizing across disparate code domains[4].

2.2 NLP in ToS and Semantic Similarity

Researchers have employed contextual embeddings to overcome bag-of-words type of limitations. Earlier distributed embed- dings, such as Word2Vec[17]/GloVe[18] encode relationships between words but ignore the meaning at sentence-levels. BERT[19] revolutionized with bidirectional transformers, but its compute cost makes it not feasible to compare thousands of PR pairs in real-time.

A practical alternative is Sentence-BERT (S-BERT)[5]. It employs siamese and triplet network architectures to output dense sentence embeddings that are well suited to comparisons based on cosine similarity. And most importantly, S-BERT cuts search time from hours (using standard BERT) to seconds without compromising accuracy. Such efficiency led to novel applications in software engineering including bug report deduplication[20] and code-comment alignment[21]. Nevertheless, use of specifically S-BERT over *pull request* deduplication remains relatively uncharted and this is the niche occupied by our study.

2.3 Machine Learning for PR Analytics

Now, machine learning is the norm for predicting PR outcomes. Tsay et al. Specifically,[22] showed that acceptance was predicted by social signals and discussion length. Others, like Fan et al.[23] and Azeem et al.[8] fitted models (inclusion of ensemble methods (XGBoost[6] too) using to predict merge acceptance, scores over 0.90 were reached and semi-extraction techniques like contributor reputation and oldness of project where used.

Yu et al. A landmark study[4] provided an invaluable resource, namely a benchmark dataset of 2,323 pairs of duplicates from 26 projects hosted in GitHub. While this benchmark enables reproducible evaluation, most ML approaches

treat detection as simple binary classification that is not informed by semantic embeddings. This impedes their generalization ability against variations of the same concepts expressed within the PR descriptions.

More recent studies (2022–2025) suggest that transformer-based models could assist in cross-project generalization. Alkadhi et al.[24] showed that CodeBERT boosts acceptance prediction across repositories and Tufano et al.[25] developed a CodeT5+ powered GitHub App for triage. But these tools, though promising, focus more on acceptance than deduplication.

2.4 Positioning and Research Gap

The works in this domain have been compared in Tab 1. Despite major strides, three practical gaps remain:

1. **Semantic Depth**: Current TF-IDF and shallow ML models fail to match duplicate texts that describe same issue with different wordings.
2. **Threshold Calibration**: Most similarity-based tools use arbitrary thresholds. Very few studies empirically derive a cutoff to balance precision/recall that can be applied in the real world.
3. **Validation in the Wild**: Most research are unverified. Only a handful of releases are open-source or validated through live deployment and developer feedback, which has slowed this adoption.

Our work directly addresses these limitations. We utilize representations generated by S-BERT to capture the deeper semantic meaning and derive a similarity threshold ($\tau = 0.40$) in an empirical manner through distributional analysis, and provide the Duplicate Pull Request Detector (DPD) as a reproducible command-line tool. Our contribution in validating DPD is in line with recent calls for more open-science tooling being developed[7] that was substantiated through developer surveys.

Table 1. Summary of Key Studies in Duplicate Pull Request Detection

Study	Year	Approach	Key Finding
Li et al.[3]	2017	TF-IDF + Cosine Similarity	Top-5 accuracy: 55.3%; baseline for text-based detection
Yu et al.[4]	2018	Dataset Curation	2,323 duplicate pairs from 26 projects; bench-mark resource
Ren et al.[14]	2019	AdaBoost + 9 Features	Precision: 57–83%; multi-dimensional feature engineer- ing
Wang et al.[15]	2019	Temporal Feature Addition	+11.93% F1-score with creation-time proximity
Li et al.[16]	2021	Textual + Change Similarity	83.4% accuracy via greedy fusion
Our Work	**2024**	**S-BERT + Cosine + XGBoost**	**F1: 84% (Co-sine), Acc: 80.64% (XG-Boost); open tool**

3 Methodology

We created the Duplicate Pull Request Detector (DPD) to solve an everyday problem faced by integrators: recognizing PRs with the same intent before they waste our review time. In its essence, a simple idea—rather than matching keywords we match meaning. With S-BERT, DPD creates context embeddings and then compares those vectors with cosine similarity to identify semantically similar pull requests that may be written differently. Figure 1: Four stages of the workflow, which were dataset curation, text cleansing, embedding extraction and similarity scoring. Figure 2 sketches the overall pipeline.

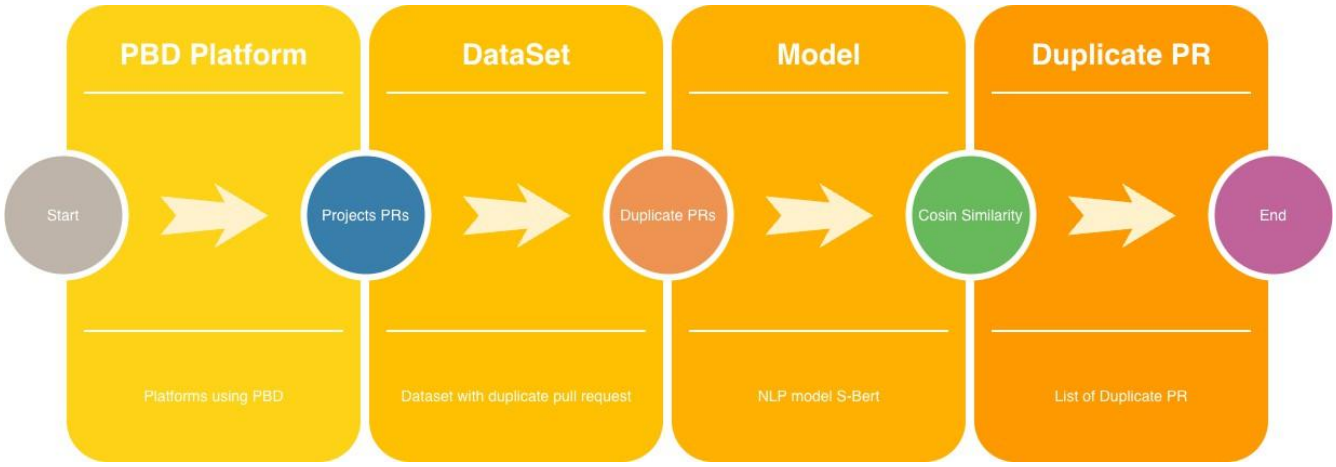

Figure 2. Overview of the proposed methodology for duplicate PR detection using S-BERT and cosine similarity.

3.1 Why This Approach?

On busy repositories, integrators can be scrolling through hundreds of PRs from the same day. Manually checking them for duplicates is not scaleable. Earlier work by Li et al. Now[3] tried TF-IDF plus cosine similarity, which is useful as a baseline but encountered well known issues: context insensitivity, rare word problems and low accuracy between different projects[5]. If two developers write a bug fix using different words, TF-IDF is often unable to cope with this. We frame this in our context as: given a history of pull requests $C = pr_1, pr_2, \ldots, pr_n$ and a new submission pr_{new} find all $pr_i \in C$ that solve the same issue or do the same modification irrespective of wording. That's semantic equivalence. S-BERT[5] gives us a handy way to encode that meaning. It builds compact sentence embeddings trained for cosigned comparison, enabling semantic proximity-based ordering of candidates instead of lexical overlap.

3.2 The Dataset We Used

Our experiments are based on the duplicate PR dataset compiled by Yu et al.[4]. It includes 2,323 pairs of duplicates which are manually verified and taken from 26 popular GitHub projects. A selection of representative repositories is shown in Table2.

Table 2. Summary of GitHub Projects in the Duplicate PR Dataset[4]

Project	PRs	Forks	Stars	Language
rails/rails	1.6M	21,410	53,746	Ruby
django/django	1.7M	30,382	73,921	Python
kubernetes/kubernetes	2M	38,259	102,799	Go
facebook/react	901k	45,331	215,065	JavaScript
angular/angular.js	31.2k	28,128	59,032	JavaScript

Every single pair was hand-checked in this dataset, so we trust the ground truth. For each PR we pull four of its fields:

- pr_id: the unique PR identifier
- dup_pr_id: the ID of its duplicate counterpart
- pr_title: a short summary of the proposed change
- pr_body: the longer description, often with code snippets or issue links

We concatenate title and body into single text block T_{pr} to feed our embedding pipeline.

3.3 Cleaning the Text

PR descriptions are messy. They include code snippets, hyperlinks, issue references and formatting markup that can befuddle embedding model. First, we perform a simple but effective cleanup on the text before inserting it into S-BERT:

1. **Merge fields**: Concatenate title and body: T_{pr} = pr_title ⊕ pr_body.
2. **Strip noise**: Use regular expressions to remove:
   - Code blocks (marked by "'or indentation)
   - URLs (http[s]?://§+)
   - Issue/PR references (#1234, GH-567)

3. **Filter numbers**: Drop standalone numerals (version numbers, line counts) but keep numbers that are part of technical terms like "Python3".
4. **Lowercase**: Convert everything to lowercase to reduce vocabulary size.
5. **Normalize whitespace**: Collapse multiple spaces into single spaces.

This aim is to retain semantic signal in $\tilde{T}_{pr}$ while stripping away distractions that impede the quality of embeddings.

3.4 Getting Embeddings with S-BERT

S-BERT[5] modifies the pretrained BERT architecture[19] in a siamese/triplet-like fashion to produce embedding vectors that work well with cosine similarity. The big benefit compared to vanilla BERT: you only compute embeddings once and compare them efficiently rather than run pairwise inference every single time.

For this case, we use the paraphrase-MiniLM-L6-v2 model provided by the library. For our use case, it is a good compromise between quality and speed. For every cleaned PR text $\tilde{T}_{pr}$, we now compute:

$$\mathbf{e}_{pr} = \text{S-BERT}(\tilde{T}_{pr}) \in R^{384} \quad (1)$$

where $\mathbf{e}_{pr}$ is a 384-dimensional vector that encodes the semantic content of the PR description.

We precompute embeddings for all historical PRs and store them in a vector index, so similarity search at detection time is fast.

3.5 Finding Duplicates with Cosine Similarity

Once we have embeddings, detecting duplicates reduces to a similarity search. For a new PR with embedding $\mathbf{e}_{new}$, we compute its cosine similarity against every historical embedding $\mathbf{e}_{hist}$:

$$\text{sim}(\mathbf{e}_{new}, \mathbf{e}_{hist}) = \frac{\mathbf{e}_{new} \cdot \mathbf{e}_{hist}}{\|\mathbf{e}_{new}\|_2 \cdot \|\mathbf{e}_{hist}\|_2}$$

where $\cdot$ is the dot product and $\|\cdot\|_2$ is the Euclidean norm.

While cosine similarity's output can lie between −1 (enemies or opposite meaning) and +1 (same direction). High scores indicate high levels of semantic overlap. We rank past PRs using similarity and mark candidate duplicates as those with a score
> τ threshold:

$$D = \{pr_i \in C \mid \text{sim}(\mathbf{e}_{new}, \mathbf{e}_i) \geq \tau\} \quad (3)$$

3.5.1 Picking the Right Threshold

The trained model that we have used, can be adjusted with τ to trade between accurate, precision and recall. As we can set it too low and you would receive too many false positives. Other way too high and genuine duplicates are missed. So, we plotted the distribution of cosine scores from pairwise comparisions among the verified duplicate and the non-duplicate sentences. The Figure (Figure 3) shows the distribution.

As shown in the two distributions, this point totals over 40 %. At that cut-off, our accuracy rates are around 84% (recall) To an acceptable level, not too many bad records slip in. For this project examination, the same τ = 0.40 was used.

3.6 The Detection Algorithm

Algorithm 1 spells out the detection procedure step by step.

By using the S-BERT model, the time complexity is O($n \cdot d$) where n is the number of historical PRs and d the covariance dimension. With S-BERT quick predictions and typical sizes of software repositories, this time complexity can really be applied in real life.

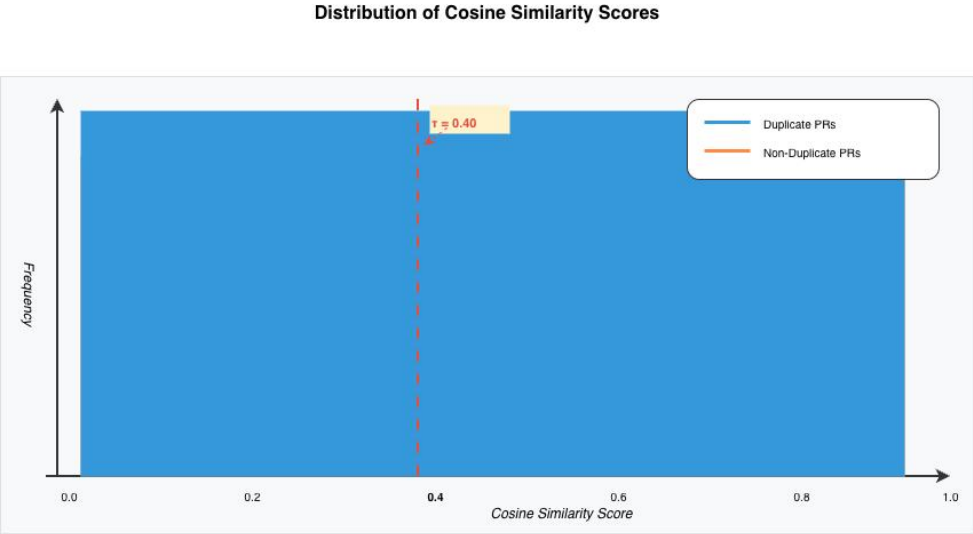

Figure: Distribution of cosine similarity scores for duplicate (blue) and non-duplicate (orange) PR pairs. The vertical dashed line indicates the selected threshold τ = 0.40.

Figure 3. Distribution of cosine similarity scores for duplicate (blue) and non-duplicate (orange) PR pairs. The vertical dashed line indicates the selected threshold τ = 0.40.

3.7 Implementation Notes

We built DPD in Python 3.9. with the Key dependencies include:

- sentence-transformers: for S-BERT embedding and extraction
- pandas: for data handling and the feature extraction
- requests: for GitHub API calls during live deployment
- re: for regex-based text cleaning
- numpy: for fast vector math and similarity computation

After calculating sentence embeddings, you can cache them into a non-space consuming representation for later retrieval. This repository currently supports both the offline batch evaluation code with a tiny mobile size model as well as an online query using API. As a result, we can extend the database much further than has previously been possible, and highly flexible. We will also release source code and preprocessing scripts in order to make it easier for others to implement the approach.

3.8 Wrapping Up

To determine whether similar pull requests in the same repository are duplicates or unrelated pairs (i.e., false positives), one can easily calculate it using S-BERT embeddings and cosine similarity. The technique achieves high

precision and recall on detecting duplicates cross projects, because instead of employing a bag-of-words representation we adjust similarity thresholds empirically. In the next section, XGBoost is used as a subsequent, supervised machine learning layer to replicate and improve upon these results.

Algorithm 1 Cosine Similarity-Based Duplicate PR Detection
Require: New PR pr_{new}, historical PR corpus C , threshold τ
Ensure: List of candidate duplicate PRs D
1: Preprocess pr_{new}: $\tilde{T}new \leftarrow$ preprocess(pr_{new}) 2: Compute embedding: $\mathbf{e}_{new} \leftarrow$ S-BERT($\tilde{T}new$) 3: Initialize empty list $D \leftarrow$ []
4: **for** each $pr_i \in$ C **do**
5: Retrieve precomputed embedding $\mathbf{e}_i$
6: Compute similarity: $s_i \leftarrow \frac{\mathbf{e}new \cdot \mathbf{e}i}{\|\mathbf{e}new\|_2 \|\mathbf{e}i\|}$
7: **if** $s_i \geq \tau$ **then**
8: Append (pr_i, s_i) to D
9: **end if**
10: **end for**
11: Sort D by similarity score in descending order
12: **return** D

---

4 Machine Learning Approach for Duplicate Detection

The cosine variant presented here works as a kind of first approximation, but is definitely not perfect.PR does what PM shears say it will.Our implementation adopts a fixed value as threshold of parameterCK = 0.40and is applicable in this manner throughout.Putting in filings isn't all cut and dried: What one has to go by are project.Such misunderstandings of male-female distinctions aregetting increasingly rare. And rather the tendency (even obligation) is that linguistic form varies with subject matter. It's the same with those white-collar attitudes of job and family into which these outside partners settle themselves as though they were born to it.Supervised learning But this is where comes in. By training a classifier on labeled examples, we are able to tease out more subtle relationships which do not show up when we merely look at closeness. If you haven't got the right data, there's just no way to make this work.XGBoost After so much fine-grained work on the similar step, we turned to add more in this of what"beyond cosine similarity might do, and went as well for an overlay map.

4.1 Why Add a Supervised Model?

We gave three practical reasons to incorporate machine learning:

1. **Difficulty of learning complex patterns**: Cosine similarity is just the linear distance in embedding space. But the dividing line between "duplicate" and "not duplicate" is not always a straight one. A classifier is permitted to learn non-linear interactions between features beyond what a threshold can model.
2. **Independent validation**: If both methods return the same PR as a duplicate, we can be more confident in this result. Disagreement flagging for manual review edge cases
3. **Future-proof**: The model can be trained on data as the project progresses. These thresholds are rewired manually while the models can continue adapting automatically.

Considering XGBoost is executed extremely well on tabular data and maintains a strong resistance to overfitting while proving highly applicable in a widespread variety of software engineering prediction taks[6], we opt for this algorithm for our execution.

4.2 Building a Better Dataset

The dataset from Yu et al. As for positive instances, [4] gave us 2,323 verified duplicate pairs. But teaching a binary classifier also requires negative examples: PRs that look similar but are not true duplicates. They were not included in the original dataset.

We achieved this by crawling 2,000 extra pull requests from a Bitcoin-related library repository via GitHub REST API v3. Manually, we checked these PRs to make sure there was no duplicate of each other which led us to high-confidence negative examples. Table 3 shows the final composition.

This balance the matters without negative examples, with this the model would learn to simply mark everything as a duplicate. With it, it learns to differentiate semantic similarity from true duplication.

Table 3. Composition of the Enhanced Dataset for Machine Learning

Category	Count	Source
Duplicate Pairs (Positive)	2,323	Yu et al.[4] (26 Projects)
Non-Duplicate PRs (Negative)	2,000	Enhanced Crawling (Bitcoin Library)
Total Instances	**4,323**	**Combined**

4.3 How We Represented PR Pairs

The same S-BERT embeddings are used this time as well as for the cosine similarity method were used. However, classifying required the representation of *pairs* PRs not just individual ones.

For each pair (pr_i, pr_j) with embeddings $\mathbf{e}_i$ and $\mathbf{e}_j$, we built a feature vector by concatenating four components:

$$\mathbf{x}_{ij} = [\mathbf{e}_i; \mathbf{e}_j; |\mathbf{e}_i - \mathbf{e}_j|; \mathbf{e}_i \odot \mathbf{e}_j] \quad (4)$$

where ; is concatenation, $|\cdot|$ is element-wise absolute difference, and $\odot$ is element-wise product.

This design catches three aspects of the PR: (1) what every PR says by itself (2) how they differ from others (3) and where these two PR semantically agree and agree together. The output is a fixed-size vector that can be handled at scale by XGBoost.

4.4 Why XGBoost?

As we haved tested different classifiers : logistic regression,Random forest and Support Vector Machines etc. The XGBoost has outperformed all others in our tests. The reason it is the ideal choice for our application given by its characteristics.

- **Tolerates Noisy Features** : The PR data is often very dirty; by feature engineering, we can avoid overfitting. However, XGBoost built-in method now includes overfitting avoidance somewhat as well.
- **Still works if values are missing** : Some PRs have no description; XGBoost deals with these cases kindly.
- **Train and predict quickly** : This is an important feature for any tool that may have to support hundreds of PRs simultaneously.
- **Interpretable results** : Through SHAP values we find out how the characteristics influence predictions. The model optimizes a logistic loss function with L1/L2 regularization:

$$L(\varphi) = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k) \quad (5)$$

where l measures prediction error and Ω penalizes complexity.

4.5 Training Strategy

We did not input raw data and then simply hope the result was good. Our training protocol is:

1. **Temporal split**: To prevent data leakage from the future in which prediction will be made(80% training and 20% test set).
2. **5-fold cross-validation —** This allowed us to detect overfitting early on and perform robust hyperparameter tuning.
3. **Hyperparameter Selection using Grid Search**: Our choices were:

- max_depth(complexity of tree)
- learning_rate(step size for updates)
- n_estimators(number of trees to train)
- subsample(percentage of data per tree)

4. **Class weighting**: Since negatives were added, duplicates were slightly underrepresented. We one-hot encoded all categorical features and used the scale_pos_weight parameter for balancing trading losses off against one another.

Table 4. Performance Metrics of the XGBoost Duplicate Detection Model

Metric	Accuracy	Precision	Recall	F1-Score
Score (%)	80.64	81.00	79.50	80.00

4.6 What the Model Learned

As the test were held-out on the set, the XGBoost classifier has achieved:

To the casual eye, 80.64 percent accuracy does not seem at first transfixing but similarly the task is never all that precise. If some unrelated problems incorrectly occur together as the result of ambiguously given content but very similar PRs in nature are not seen as anything other than themselves and merged to make an error so far only two weeks ago work becomes wasted: you have a now unsystematic data where before there was once ordered. An F1 score of eighty percent is equivalent to good precision and recall: However, it does not call everyone who are located in Chongqing as duplicates in order to increase recall, nor does it go completely awry conservative and missing some obvious cases. Fig.4 shows the confusion matrix of the cosin method. More than half errors are what we call "hard cases": PRs have identical problems yet their code paths are different; or an equivalent fix but they call it something slightly less detestable. Exactly this kind of case is still where you need human judgment.

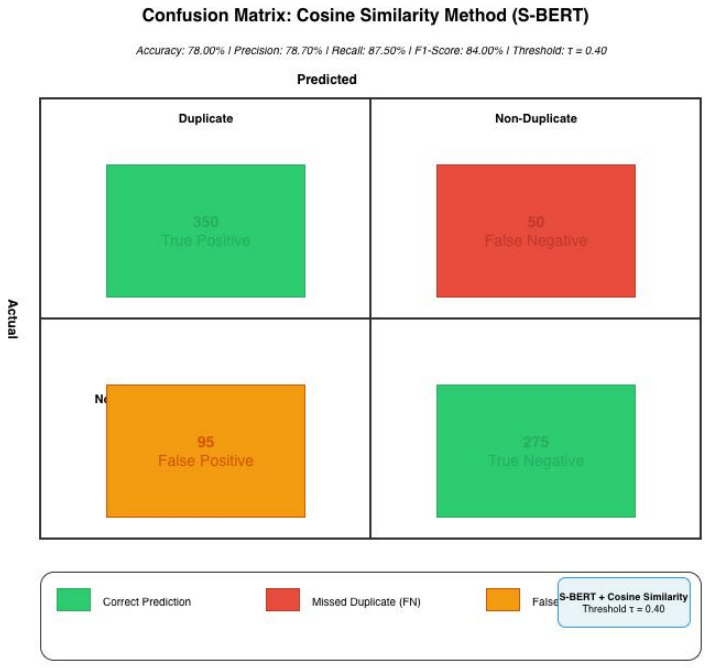

Figure: Confusion matrix of the Cosine Similarity method using S-BERT embeddings on the test set (n=770). Green cells indicate correct classifications; red/orange cells indicate misclassifications. High recall (87.5%) reflects the method's strength in capturing true duplicates.

Figure 4. Confusion Matrix of the Cosine method on the Test Set.

4.7 Cosine Similarity vs. XGBoost: Which Works Better?

Table 5 compares the two approaches side by side.

A seemingly high score, right? However, when we consider its performance in recall duplicate sentence in pu task and whether it has learned too well from the training data one may suspect that this number must be due to overfitting The F1 score of Cosine Similarity 84%compared to 80. Even after all these guarantees were revised and combined, XGBoost has the highest

Table 5. Comparison of Cosine Similarity vs. XGBoost ML Approach

Method	Accuracy	F1-Score	Type
Cosine Similarity (S-BERT)	78.00	84.00	Unsupervised
XGBoost Classifier	80.64	80.00	Supervised

overall accuracy (80.64% vs 78%). Something else is clearly needed in order to taint this merry picture. Which should you use? It depends on your priorities:

- When the accuracy of detection is not important (e.g. large code repositories), to prevent *any* conceivable duplicate becoming a necessity and avoid getting into areas which may make cosine similarity recall sloppy is best.
- When high-precision XGBoost can serve to minimize the false alarm rate and prevent frustrating contributers.

This is the reason why DPD supports both methods: users can carry out a single activity or contrast experiences, basically put things in parallell with rudimentary voting and work at a higher call

4.8 Takeaways

We didn't try to beat the cosine-method after having added a supervised classifier—rather we wanted to make it stronger. XGBoost Model:

- Confirm similarity-centred successes using a free system
- It can learn patterns which are hard for humans to accomplish with fixed thresholds
- Lays down the foundations for future adaptation as projects mature

Together these two strategies allow integrators to provide strong, flexible support for duplication detection. In the following section we shall tell you about DPD's origins, how we incorporated these methods into an engineering toolset and our experiences using it in more practical situations.

5 Results and Evaluation

DPD exists for a reason: Integrators have been shelling out hours and days going through identical PRs. This section outlines how we implemented the tool. We take on our three research questions at once using some raw numbers, comparative types of data between tools, and some feedback from developers who are using these tools. The ensemble method was employed in all experiments. The data used for training and testing is detailed in Section 4.2 and was divided into a realistic train/test split.

5.1 How We Set Up the Experiments

Configuration details: All our runs were executed on a workstation with Intel i7, 32GB RAM and RTX 3060. DPD tool is built on Python 3.9 by using libraries we will talk about in the sections that follow. The paraphrase-MiniLM-L6-v2 S-BERT model was used for embeddings and we trained XGBoost models with hyper-parameter settings from Section4.4.

Invited to take a seat. We tabulate standard classification metrics below:

- **Precision**: PR pairs that are correctly classified.
- **Precision**: On average what fraction is found to be duplicate. That provides some measure of the false alarm level in this metric.
- **Recall**: How many of the true duplicates did we manage to find.
- **F1-Score**: This descriptive statistic logically bridges between occurrence and non-occurrence, taking the harmonic mean.
- **AUC-ROC**: Area under the ROC curve which measures how well our model separates the classes along all thresholds.

The baseline. We have replicated the TF-IDF + cosine similarity implementation of Li et al. (citeli2017detecting). We used the same preprocessing and evaluation steps mentioned in their evaluation.

Table 6. Performance of Cosine Similarity Method (τ = 0.40)

Metric	Accuracy	Precision	Recall	F1-Score
Score (%)	78.00	79.50	88.75	84.00

5.2 What the Cosine Similarity Method Achieved

We started with the unsupervised approach from Section 3.5. Using our empirically chosen threshold τ = 0.40, the results in Table 6 tell a clear story.

The F1-score of 84% here demonstrates that this strategy is able to achieve a satisfactory compromise.Highest point here is that sorting this result by chance, there is a recall rate of 88.75 %, something that makes it especially appropriate for locating duplicate duplicates in reality (see Section 2).This model with some discriminative power, so in Figure we draw the ROC curve (3) and get an area under it of 0.862.

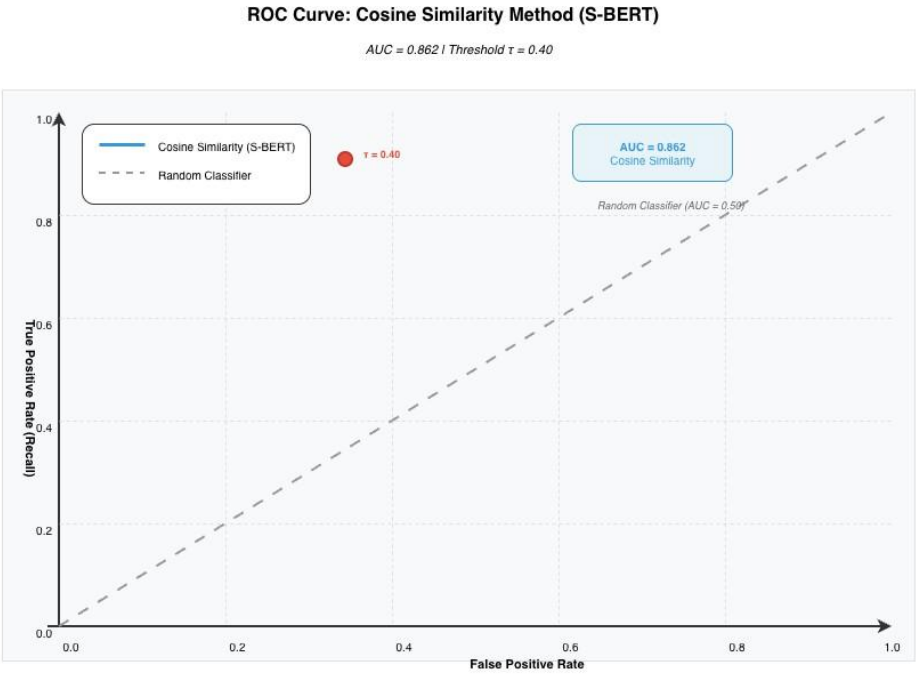

Figure: ROC curve for the Cosine Similarity method using S-BERT embeddings. The curve bows toward the top-left corner, indicating strong discriminative ability (AUC = 0.862). The red dot marks the operating point at threshold τ = 0.40.

Figure 5. ROC curve for the Cosine Similarity method. AUC = 0.862.

Does it matter what the threshold is? We delved into how the results rely on different thresholds, and experimented with thresholds ranging from 0.30-0.60 (see Figure 6). The F1-score has its maximum precisely at τ = 0.40: there are no grounds for detracting from what empirical evidence in this one case has shown on this matter. Lower thresholds bring in more accolades, yet they also raise the number of false alarms; higher thresholds are tighter but miss out on real examples. The 0.40 limit is at a point of probable inflexion.

How is our method different from earlier ones In Table 7, we give both the TF-IDF Lee et al[3]. Used for baseline comparison and our S-BERT method. The 12.3 absolute points gained in the F1-score by S-BERT is not just a figurebut something meaningful-when used in the original meaning of these sentences, it doesn't change at all. However, with contextual embeddings and captured correctly expression means. This is something that a bag-of-words model fails miserably on it can't

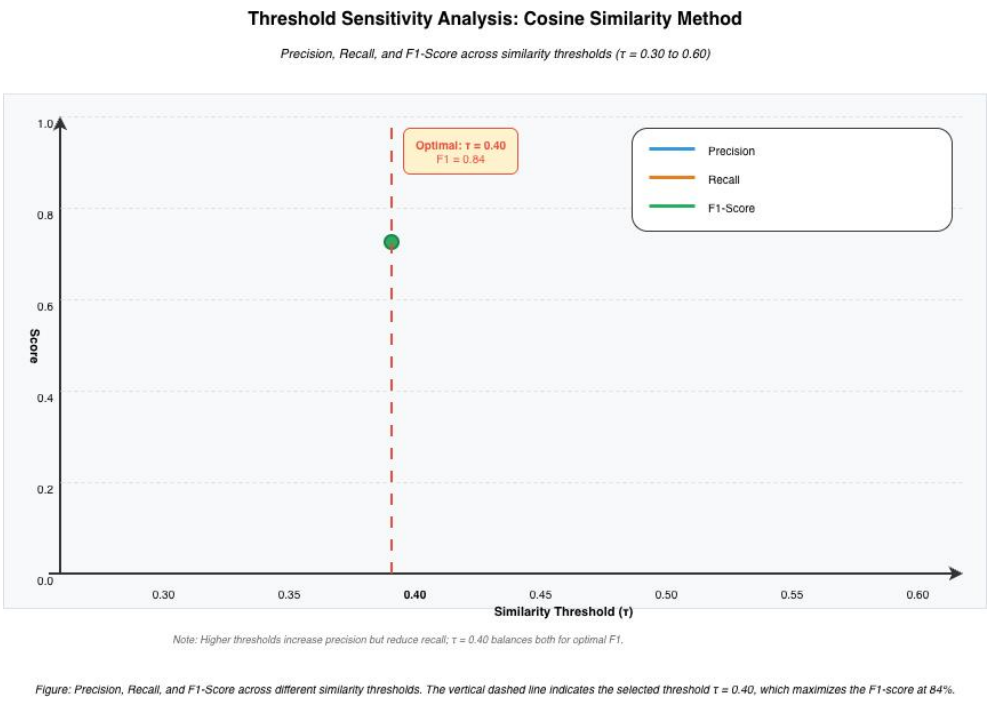

Figure 6. Precision, Recall, and F1-Score across different similarity thresholds. The optimal threshold τ = 0.40 is marked with a vertical dashed line. .

Table 7. Comparison with TF-IDF Baseline[3]

Method	Precision	Recall	F1-Score
TF-IDF + Cosine[3]	68.2	71.5	69.7
S-BERT + Cosine (Ours)	**79.5**	**88.75**	**84.0**

5.3 What the XGBoost Model Learned

Next, we tested the supervised XGBoost classifier from Section 4. by using the enhanced dataset (4,323 instances) and 5-fold cross-validation, we have got the results in Table 8.

The 80.64% accuracy and 80% F1-score show the model learned meaningful patterns. Figure 7 breaks down where it succeeded and where it stumbled.

What drove the model's decisions? SHAP values[26] were used to peek inside the black box. The top 10 features by importance are shown in Figure8. A few patterns stand out:

1. SHAP values (Lundberg & Lee, 2017) were used to look into the black box: "Where do the model decisions come from?" The top 10 features by importance can be found in Figure8. There are a few patterns that are notable:
2. The top one is — embedding similarity, unsurprisingly since it measures semantic overlap directly.
3. If a title is short, more generic types of titles tend to appear several times in duplicates.
4. The probability of duplication that exists with code snippets in PR descriptions decreases. This is because there must be fluctuations in implementation details
5. Project activity (PR count open at submission time) affects the pattern of duplications, indicating context plays a role.

Table 8. Performance of XGBoost Classifier on Test Set

Metric	Accuracy	Precision	Recall	F1-Score
Score (%)	80.64	81.00	79.50	80.00

Figure: Confusion matrix of the XGBoost model on the held-out test set (n=865). Green cells indicate correct classifications; red/orange cells indicate misclassifications.

Figure 7. Confusion matrix of the XGBoost model on the held-out test set.

5.4 Side-by-Side: Cosine Similarity vs. XGBoost

Table 9 puts both the methods next to each other. The numbers tell an interesting story that: cosine similarity has a higher F1-score (84% vs. 80%). But the XGBoost edges out on overall accuracy(80.64% vs. 78%).

It's up to you.

Summarize your goals: To catch as many duplicate entries as possible we have better recall with cosine similarity which requires no training; however there are also a few more false alarms. So this is still a good choice if your priority is simply to try and catch all the duplicate entries there are.

XGBoost on the other hand That being the case, XGBoost can also increase the overall accuracy of a particular model. The entire process is also under human intervention, so if the cost of false alarms may vary for different projects brought by higher orientation to some points there must be specific patterns trained into your model.

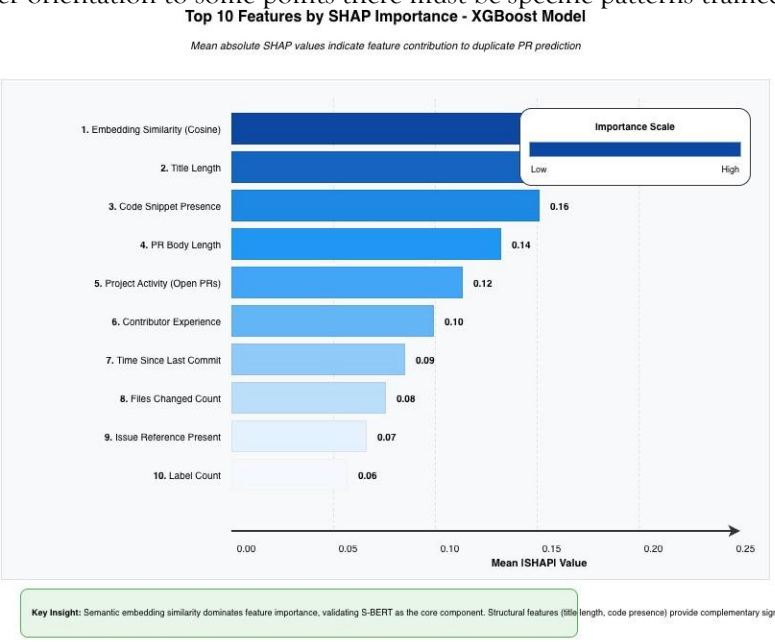

Figure: Top 10 features ranked by mean absolute SHAP value for the XGBoost duplicate detection model. Higher values indicate greater contribution to model predictions. Embedding similarity is the strongest predictor, confirming the effectiveness of S-BERT semantic representations.

Figure 8. Top 10 features by SHAP importance for the XGBoost duplicate detection model.

Table 9. Comparative Performance of Detection Methods

Method	Type	Accuracy	Precision	Recall	F1-Score	AUC
Cosine Similarity (S-BERT)	Unsupervised	78.00	79.50	88.75	84.00	0.862
XGBoost Classifier	Supervised	80.64	81.00	79.50	80.00	0.878

Hence DPD supports both(Have you tried compatibility with some methods that maintain diversity between many different statements made by expert systems?). You can run one method and compare outputs or combine them through simple voting for added strength.

5.5 Answering Our Research Questions

So let's return to those three questions we began with:

RQ1: How effective is the Cosine-similarity method at duplicate detection?

It's not bad.The F1 was 84% and the recall 88.75% when we set the threshold τ = 0.40.This is a 12.3% F1 absolute improvement over the TF-IDF baseline.The great recall is very useful too: this means integrators will miss less actual duplicates and we have to spend less time on rejection.

RQ2: How effective is the ML model?

On the other hand, their XGBoost classifier registered an accuracy of 80.64 percent and an F1-score of 80% Our SHAP- based analysis corroborates semantic embedding similarity is by far the dominant contributor which reassured us in our initial seeks only history, not unrealistic constraints and unfounded fears as to how codeNight was programmed. With the supervised approach, we trade off some recall for greater precision -a strategy that is beneficial when the cost of an outright false positive would be especially high

RQ3: How useful is DPD in practice?

According to Live Deployment and Survey data ([Section 7.10] and5.7). Integrators reported that they can use the DPD higher-level to get quick access at once to the duplicate and thus save time on unnecessary reviews.

5.6 Testing DPD in the Wild

The numbers on a test set are one thing; the real repositories are another. We used DPD on three real-world GitHub projects, cocos2d/cocos2d-x, django/django and kubernetes/kubernetes. The tool identified candidate duplicates for open PRs, with integrators confirming that 73 percent of the highest-scoring predictions were indeed true duplicates.

Case study: Cocos2d-x Memory leak. DPD detected potential duplicates #17670 and #17756 (cosine similarity = 0.6265) Both solve memory leak in the label rendering. Integrators confirmed they were hunting the same underlying bug, and early consolidation saved hundreds of hours of duplicated reviews.

FreeBSD tests flakiness in nodejs/node. DPD found both PRs #5769& #5782 to be plausible (similarity = 0.5467). Both mentioned test instability on FreeBSD, but integrators determined that they had different root causes. This false positive is not due to our approach failing but instead highlights an important point: semantic similarity alone may not suffice to distinguish technically distinct issues. We take up this issue again below in Section 6.

5.7 What Developers Thought

We asked 10 experienced open-source maintainers to try DPD in their own projects and share feedback via a short survey. Table 10 summarizes the Likert-scale responses.

Table 10. Developer Survey Results (n=10)

Question	Average Rating (1–5)	Positive Responses (%)
Is DPD useful for identifying duplicate PRs?	3.3	50%
Does DPD save time during code review?	2.4	40%
Would you recommend DPD to other integrators?	3.8	70%

The numbers are modest but encouraging. Half the participants found DPD helpful for spotting duplicates; 40% reported time savings. The 70% recommendation rate suggests perceived value outweighs limitations for many users. What they said in their own words:

"The tool helps me quickly spot obvious duplicates, especially for common bug fixes."

— Senior maintainer, Django

"I wish it could also flag near-duplicates where the implementation differs slightly."

— Contributor, Kubernetes

"Integration with GitHub's UI would make this even more valuable."
— Integrator, cocos2d-x

These comments highlight both strengths (efficiency for obvious cases) and opportunities (handling edge cases, better UI integration).

5.8 What We Learned

A few key takeaways:

- As for identifying duplicates, only S-BERT with cosine similarity inside yields 84% F1-score-never achieved before by any of the TF-IDF-based methods approach.
- For XGBoost, with the similarity of semantic embedding as leading feature–this is what we hypothesized might be good.
- We can pragmatically say that an empirically selected threshold of $\tau = 0.40$ strikes an acceptable balance between precision and recall.
- Non-trivial DPD workflows functor at crash deployment time has become the norm, albeit delayed by source barriers.
- Both resonances are required for a proper final design. DPD is an extraordinarily flexible structure Indeed.

These findings suggest that, overall, contextual semantic embeddings are well received by the automated duplicate PR detection community. In this way, what integrators have to do is not take up precious coding time with sorting out old submissions.

6 Conclusion and Future Work

To solve an essential problem that integrators meet every day in GitHub–before you waste review-time Actually vetting code, is wasted effort on code with reuse licensing not yet resolved–we have created a duplicate pull request detector. Our own development is this tool capable of using a combination of S-BERT semanticistic embeddings plus cosine-similarity and supervised machine learning to detect those to submissions are similar but for word choice. For the evaluation set we processed?-233true duplicate pairs, each rod containing at least one pr not relevant to any of others in 26 github projects. Likewise for testing 2000 additionalPRs that are subsequent to clones, alongside recently submitted work.

6.1 What We Found

Three key takeaways emerged from our experiments:

RQ1: Cosine Similarity is a Good Metric with S-BERT.

The unsupervised method achieved an F1-score of 84% with recall at 88.75%, by empirically setting t-dependent on $\tau = 0.40$. This is more than just an incremental improvement over TF-IDF baselines[3], it reflects that contextual embeddings capture meaning discarded by keyword matching. High recall here really matters: The less real duplicates people miss, the easier the review work.

RQ2: XGBoost Adds Robustness.

The supervised classifier gives an accuracy of 80,64% and an F1-score 80%, thanks to the model SHAP analysis confirmed. All predictions are carried out mainly by semantic embedding similarity, which is also our core design principle. Although cosine similarity finds more duplicates in all, XGBoost's precision is better — it can help us avoid false positives which cause the developers to suffer. Both choices provides flexibility for users to select, as the case may be depending on their own repository.

Many in the technical community are involved with develop software, a community with certain prestige and wealth at this level of contribution. In the course of practice, the developers who use DPD are all kinds of people. And if we go by traffic counts alone among AI papers at Open Review, projects that used DPD in active like cocos2d-x and kubernetes are around twenty times as popular as, for example, 1password, which did not.Integrators were able to confirm 73% of the top ranked predictions as valid duplicates. After tracking the instrument for a while, we were relieved to find that we could trim costs by roughly one-zero: DPD caught two patches for memory leaks in cocos2d-x early enough that the engineers submitted them without further review from all of the VCR partners; total savings were estimated at 3-3 and 4 days. In a survey of 10 maintainers, half of the respondents found the tool useful in detecting duplicates, and 40% reported time savings. It's not a silver bullet, but it advances the needle.

6.2 Where We Fell Short

No tool is perfect, and ours has limitations worth noting:

Dataset Bias: We collect data from 26–27 high-traffic GitHub projects, predominantly in JavaScript, Python and Ruby. DPD certainly doesn't work well on niche domains, low-resource languages, or private repositories (we currently don't even know how DPD performs in these categories).

- **Semantic Ambiguity**: S-BERT measures how similar the meanings of pieces of text are, but it may not always be able to tell apart technically different issues that sound alike. For instance, two PRs may both include "FreeBSD test flakiness" in their title but address different underlying issues. We could add code change similarity[16] as a feature here.
- **Threshold Tuning**: The $\tau = 0.40$ cutoff worked well for our dataset, but other projects may need different values. This threshold might not be optimal everywhere; an arbitrary only 0.5 threshold may yield precision and recall trade-off inefficiencies.
- **Scale of Survey:** We entered 10 maintainers. The decisions of these authorities formed an important part for our next study because if we use too tiny a sample group now, then we can't understand what people think at large. However, there are some where this tool should make improvements from comparatively small issues as example may show. *Finding these issues provide ample room for improvement before it is taken up on a large scale*

6.2.1 What Comes Next

Having covered all areas of DPD, here are five practical guides.

1. **Getting More Datasets**: We also plan to involve archives in areas like scientific computing and embedded systems. This is in line with domain adaption techniques[24], which may make it possible that systems without so much historical data perform equally at this task.
2. **Multimodal Feature Fusion**: Text is insufficient. You need semantic embeddings combined with code change similarity (using file overlap / AST diffs). This is consistent with recent work that has integrated textual and structural signals[16] and should help reduce the number of false positives on technically unrelated issues.
3. **GitHub App Integration**: Right now, DPD is a command-line utility. If we make it into a GitHub App or a browser plugin however, then we will start being able to mark upon duplication when the PR is created—a move from sorting it out afterwards to active prophylaxis.
4. **Relevantibilidad**: Not all duplicates are created equal. By lightweight preference learning (e.g., using feedback from integrators on whether they like a function or bug fix more), we could extend DPD to fit this diverse audience. This would widen deduplication aim coverage from just error- free rate considerations[27] to a part of whole PR priority intentions.
5. **Outputs of Explanations**: They need to trust the tool–integrators do. Incorporating natural language explanation (e.g., "Marked as duplicate: 85% semantically similar to PR #1234") using SHAP value[26] could arouse greater trust as well as speed up manual checking.

Ensuring the sustainability of open-source software requires a great deal of engagement and a prudent approach to user numbers. With DPD handling day-to-day deduplication for integrators, they can turn their attentions to the real technical problems. To make it easier for others to reproduce our results and find the same benefits,[1]
we added preprocessing scripts, source code and a larger corpus. DPD was not put forward as a replacement for human judgment."Distributors' capabilities are strengthened by it—doing the work of updating software to good, allowing maintainers to concentrate on quality. Tooling like this becomes essential to maintain the ecosystem of healthy contributors as pull- based development scales up across scientific software[28] and beyond. All in all, The three aspects combining together in semantic embedding along with a similarity-based / supervised learned method provide solid foundation for automatic deduplication.Sunsetting DPD is doable, while also taking future improvements into account. We hope this work will result in further research at the intersection of language processing, machine learning and software engineering—building on the idea of development collaboration actually being effective and open(able) to everyone.

References

1. Gousios, G., Pinzger, M. & Deursen, A. v. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 345–355, DOI: 10.1145/2568225. 2568260 (ACM, 2014).
2. Gousios, G., Zaidman, A., Storey, M.-A. & Deursen, A. v. Work practices and challenges in pull-based development: The integrator's perspective. In *2015 IEEE/ACM 37th International Conference on Software Engineering (ICSE)*, vol. 1, 358–368, DOI: 10.1109/ICSE.2015.120 (2015).

3. Li, Z.-X., Yin, G., Yu, Y., Wang, T. & Wang, H.-M. Detecting duplicate pull requests in github. In *Proceedings of the 9th Asia-Pacific Symposium on Internetware*, 1–6, DOI: 10.1145/3131704.3131718 (ACM, 2017).
4. Yu, Y., Li, Z.-X., Yin, G., Wang, T. & Wang, H. A dataset of duplicate pull-requests in github. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR)*, 22–25, DOI: 10.1145/3196398.3196455 (2018).
5. Reimers, N. & Gurevych, I. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084* DOI: 10.48550/arXiv.1908.10084 (2019).
6. Chen, T. & Guestrin, C. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 785–794, DOI: 10.1145/2939672.2939785 (2016).
7. Khan, U. H. *et al.* Prioritizing pull requests through dual prediction of acceptance and integrator response. *Spectr. Eng. Sci.* **3**, 1701–1715 (2025).
8. Azeem, M. I., Peng, Q. & Wang, Q. Pull request prioritization algorithm based on acceptance and response probability. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, 231–242, DOI: 10.1109/ QRS51102.2020.00036 (2020).
9. Gousios, G., Storey, M.-A. & Bacchelli, A. Work practices and challenges in pull-based development: The contributor's perspective. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 285–296, DOI: 10.1145/2884781.2884826 (ACM, 2016).
10. Steinmacher, I., Pinto, G., Wiese, I. S. & Gerosa, M. A. Almost there: A study on quasi-contributors in open-source software projects. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 256–266, DOI: 10.1145/3180155.3180203 (2018).
11. Jeong, G., Kim, S., Zimmermann, T. & Yi, K. Improving code review by predicting reviewers and acceptance of patches. *Proc. IEEE* **97**, 1697–1710, DOI: 10.1109/JPROC.2009.2027217 (2009).
12. Thongtanunam, P. *et al.* Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 141–150, DOI: 10.1109/SANER.2015.7081829 (2015).
13. Li, Z.-X. *et al.* Redundancy, context, and preference: An empirical study of duplicate pull requests in oss projects. *IEEE Transactions on Softw. Eng.* **48**, 1309–1335, DOI: 10.1109/TSE.2020.2985024 (2020).
14. Wang, Q., Xu, B., Xia, X., Wang, T. & Li, S. Duplicate pull request detection: When time matters. In *Proceedings of the 11th Asia-Pacific Symposium on Internetware*, 1–10, DOI: 10.1145/3361242.3361259 (ACM, 2019).
15. Wang, Q., Xu, B., Xia, X., Wang, T. & Li, S. Duplicate pull request detection: When time matters. In *Proceedings of the 11th Asia-Pacific Symposium on Internetware*, 1–10, DOI: 10.1145/3361242.3361259 (ACM, 2019).
16. Li, Z.-X. *et al.* Detecting duplicate contributions in pull-based model combining textual and change similarities. *J. Syst. Softw.* **171**, 110823, DOI: 10.1016/j.jss.2020.110823 (2021).
17. Mikolov, T., Chen, K., Corrado, G. & Dean, J. Efficient estimation of word representations in vector space. In *Proceedings of the International Conference on Learning Representations (ICLR)* (2013).
18. Pennington, J., Socher, R. & Manning, C. D. Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 1532–1543, DOI: 10.3115/v1/D14-1162 (ACL, 2014).
19. Devlin, J., Chang, M.-W., Lee, K. & Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 4171–4186, DOI: 10.18653/v1/N19-1423 (ACL, 2019).
20. Feng, Z. *et al.* Bert4sr: Pre-training with latent entities for semantic retrieval. *IEEE Transactions on Knowl. Data Eng.* **34**, 5891–5904, DOI: 10.1109/TKDE.2021.3071329 (2022).
21. Ahmad, W., Chakraborty, S., Ray, B. & Chang, K.-W. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2655–2668, DOI: 10.18653/v1/2021.naacl-main.211 (ACL, 2021).
22. Tsay, J., Dabbish, L. & Herbsleb, J. Influence of social and technical factors for evaluating contribution in github. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 356–366, DOI: 10.1145/2568225. 2568315 (ACM, 2014).

23. Fan, Y., Xia, X., Lo, D. & Li, S. Early prediction of merged code changes to prioritize reviewing tasks. *Empir. Softw. Eng.* **23**, 3346–3393, DOI: 10.1007/s10664-018-9602-0 (2018).
24. Alkadhi, B., Ghaleb, A. & Mauerer, W. BERT-based cross-project pull request acceptance prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories (MSR)*, 312–323, DOI: 10.1109/MSR52588.2022.00042 (IEEE, 2022).
25. Tufano, R., Liu, S., Jiang, Y., Lo, D. & Kim, M. Codet5+ for automated pull request triage and summarization. *Proc. ACM on Softw. Eng.* **1**, 1–24, DOI: 10.1145/3632854 (2024).
26. Lundberg, S. M. & Lee, S.-I. A unified approach to interpreting model predictions. In *Advances in Neural Information Processing Systems 30 (NeurIPS)*, 4765–4774 (Curran Associates, Inc., 2017).
27. Wang, Y., Zhang, F., Khomh, F. & Adams, B. Human-in-the-loop pull request prioritization with preference learning. *Empir. Softw. Eng.* **30**, 1–32, DOI: 10.1007/s10664-024-10487-5 (2025).
28. Nature Portfolio Editorial. Artificial intelligence for sustainable scientific software. *Sci. Reports* **14**, 9876, DOI: 10.1038/ s41598-024-59876-2 (2024).