Document-Level Embeddings for Duplicate GitHub Issue Detection

Kian Hosseinhani, Rohan Parmar, Mikhail Sinitcyn

1 Abstract

Document embeddings serve a purpose in a multitude of tasks including information retrieval, clustering, and pair classification, by enabling semantic similarity comparison between two documents via the cosine similarity operator. This property makes document embeddings useful for detecting duplicate issues in large repositories with backlogs of thousands of issues as an extension of the pair classification task. We evaluate 8 document embeddings models for the duplicate detection task within a repository of GitHub issues by computing pair-wise comparisons between issues and evaluating the pair-wise duplicate classification via the F1 metric across a number of thresholds. We find that the intended embedding task has the biggest effect on the model's performance, confirming the findings of Muennighoff et al (**?**).

2 Introduction

It is common for enterprise software repositories to have a large backlog of issues, many of which are duplicates of each other. For example, the financials team at Workday has a JIRA ticket backlog exceeding 5,000 tickets, whilst the Business Intelligence team at Telus has a backlog exceeding 30,000 tickets. A significant percentage of these tickets are duplicates - issues that have been logged by different customers for the same issue, often filed due to a lack of a streamlined reporting process that validates duplication for incoming issues. Labelling these tickets as pair-wise *'duplicates'* requires developer time and effort, often requiring manual investigation into the ticket base and the code base.

The duplicate pair detection problem is an instance of the Semantic Similarity Matching task for a given document, as we are comparing its similarity to every other document in a set. We will be breaking down this task to a pair classification task where we classify pairs of issues as duplicate and non duplicates.

In our pipeline, we create document-level embeddings of the concatenated issue title and description for each issue in the repository dataset and compute pair-wise cosine similarity scores as the primary measure of issue similarity, labelled as as a duplicate pair over a predefined threshold. Thus, for a given GitHub issue, the concatenated title and description are the model input and the classification label is the output. Our pipeline in summary gets a set of issues in CSV file as an input. Then generates embeddings for computing pair-wise cosine similarity scores between all pairs of issues. Finally, compares all pairwise cosine similarity scores with a similarity threshold, indicating potential duplicates.

We select the top 4 highest-performing models from the SBERT Fine-Tuned Models Leaderboard, evaluated on the semantic search and encoding tasks. Additionally, we select the top 4 highest-performing models from the Hugging Face Massive Text Embedding Benchmark (MTEB)(**?**) Leaderboard. The models are evaluated for the pair classification task by generating embeddings on a sample of the data, computing pair-wise cosine similarity scores, and evaluating the classification F1 score over a number of thresholds. Additionally, the ROC curve and AUC score provide information about the trade-off between true positive and true negative classifications, independent of the threshold value. Whereas the precision-recall curve visualizes the tradeoff between precision and recall over different threshold values.

We find that the intended task has the biggest effect on performance the pair classification task.

3 Prior Work

Prior to the advent of neural embeddings models, the duplicate bug report problem had been tackled using frequency-based static embeddings such as

TF-IDF, evaluated using cosine similarity scores. These earlier method evaluations showed that about 2/3 of the duplicates in a repository can possibly be found using the NLP techniques (**?**). Later, the BERT (**?**) model presented a breakthrough in dynamic word embeddings, allowing the language model to have different representations for a given token depending on its context. In BERT, the special [CLS] token is prepended to each input sequence and its representation in the final layer functions as the sentence embedding, representing the entire input sequence (**?**). However, this aggregate representation of the base BERT model is "unsuitable for document-level embeddings" per the Sentence-BERT paper (**?**). As a solution, the Sentence-BERT (**?**) model adds a pooling operation to the output of BERT/RoBERTa to derive a fixed sized document embedding, fine-tuned using contrastive Siamese (**?**) and triplet networks such that the produced sentence embeddings are semantically meaningful and can be directly compared by cosine-similarity (dot-product for normalized embeddings). This result is relevant to our work as all 8 of the evaluated models utilize this architecture as described in section 4.2.

In formulating the duplicate detection problem as a pair classification task, defining the threshold at which a pair of GitHub issues is classified as a duplicate pair is a challenge in itself. Runeson et al (**?**) provide an interesting observation that bug reports tend to fall under one of two categories: "those that describe the same failure", and "those that describe two different failures with the same underlying fault". They note that, in the former type, duplicate issues generally use the same vocabulary, while in the latter type, duplicates may use different vocabulary. As a result, they only address the first type of issues because frequency-based methods prove ineffective for the second type. This is a valuable heuristic that motivates the use of neural embedding models, as they are trained to create semantically representative embeddings.

Similarly, in the information retrieval domain, detecting near-duplicate web pages is a challenging problem "as there is no generally accepted definition of near-duplicate states" (**?**). Thus, their work borrows heuristics from information theory, tree-based algorithms, and computer vision for a robust definition of duplicate pairs. Our work does not explicitly define a duplicate metric. Rather, we use the labelled data in the Mozilla Firefox repository as provided for the grand truth labels in our model evaluations.

In this report, we evaluate document embeddings on the pair classification task within the Github issues domain. The Massive Text Embedding Benchmark (MTEB) (**?**) paper is a highly relevant work as it provides a thorough evaluation of 33 models across 8 embedding tasks and 58 datasets, drawing the conclusion that no one embeddings model dominates across all tasks.

4 Approach

Performing the Pair Classification task, our pipeline is formulated such that, given an input of two issues, the output is a binary duplicate/non-duplicate classification label. We first embed the pairs using the models then calculate the distances following the cosine similarity metric. We use standard binary classification evaluations the following evaluations: F1, ROC-AUC, and Precision-Recall.

4.1 Generating Embeddings for the Dataset

To generate the document-level embeddings, we first define "document" in our use case. Given that the natural language content of a GitHub issue within the given dataset is entirely defined by its title and description, we concatenate the two into a *Content* field, for which we compute the document-level embeddings as a single text document.

We selected the top 4 highest-performing models from the SBERT.net Fine-Tuned Models Leaderboard, evaluated on the semantic search and encoding tasks. Additionally, we select the top 4 highest-performing models from the Hugging Face Massive Text Embedding Benchmark (MTEB)(**?**) Leaderboard with model size around $1GB$.

4.2 Siamese Network Architecture and Contrastive Learning Objective

The Siamese architecture involves two identical networks, sharing the same parameters, that receive two documents as input and create two corresponding embeddings in parallel, then compute their similarity via the cosine similarity operator (**?**). Using the contrastive learning objective, this network learns to create document embeddings such that similar document pairs (positive pairs) have a higher similarity score and dissimilar document pairs (negative pairs) have a lower similarity score. Similarly, the triplet network extends this architecture by adding a third network to serve as the negative document embedding. These two architectural terms are used interchangeably in literature and model cards, thus no explicit distinction is made between the two further in this report.

Among the 8 embeddings models selected in this report, the 4 SBERT Leaderboard models are known to utilize the Siamese architecture per the open-sourced model code. The 4 MTEB Leaderboard models are implicitly assumed to utilize the architecture, as their configuration files indicate the use of the *BertModel* architecture, contrastive learning objective, and a pooling mechanism per the Siamese architecture.

4.3 Embedding Models

4.3.1 Semantic Search SBERT Leaderboard Models

These 4 models were selected from the SBERT.net leaderboard where they ranked among the highest performing on semantic search and various sentence encoding tasks. All four models were developed as part of the “Train the Best Sentence Embedding Model Ever with 1B Training Pairs" challenge to produce document embeddings (sentence to short paragraph) for information retrieval, clustering or sentence similarity tasks. They were trained on samples of the same dataset, except for Model 1 which notably excludes the Reddit comments and S2ORC (?) data comprising 84% of the dataset.

Table 1 provides an overview of the model specifications, where *Dimension* represents the embedding dimensions and *Size* refers to the model size. All 4 models produced normalized embeddings.

Table 1: Model Specifications

Model	**Size (MB)**	**Dimension**
multi-qa-mpnet-base-dot-v1	420	512
all-mpnet-base-v2	420	384
all-distilroberta-v1	290	512
all-MiniLM-L12-v2	120	256

Model 1. multi-qa-mpnet-base-dot-v1

This model uses mpnet-base as the base model (?), which is pre-trained on the MPNet language modelling task and fine-tuned on a variety of downstream tasks including GLUe and SQuAD, significantly outperforming MLM and PLM pre-training methods. Unlike models 2,3 and 4, which were developed for intended use as sentence and short paragraph encoders, this model is trained for semantic search, and utilizes CLS-pooling (?) rather than mean pooling.

This model notably excludes the Reddit and Wiki datasets from the fine-tune data. It samples from the remaining 215 million question-answer pairs including 300 thousand Stack Exchange duplicate questions (question body, title+body) and Stack Exchange (45 million title+body pairs). Note that this domain-specific training data is highly relevant to our use case, yet the training objective differs from our pair classification task.

Model 2. all-mpnet-base-v2

Similar to model 1, all-mpnet-base-v2 uses MPNet as the base model.However, the model is fine-tuned for sentence and short paragraph encoding using cross entropy loss with mean pooling instead of CLS pooling, averaging the embeddings of all tokens in the sequence excluding special tokens. Additionally, this model was fine-tuned on a higher variety of data, sampled from 1 billion sentence tuples. The dataset notably includes 700 milion Reddit comments and 100 million S2ORC Citation pairs mined from scientific papers. Consequently, the fine-tune data is general purpose (thus the *all-* prefix), unlike model 1. The training procedure was performed using a TPU v3-8 during 100k steps using a batch size of 1024, learning rate warm up of 500, and AdamW optimizer with a 2e-5 learning rate.

Model 3. all-distilroberta-v1

Using distilrobert-base (?) as the base model, all-distilroberta-v1 is trained on the same general purpose dataset as all-mpnet-base-v2. The training procedure was performed using a TPU v3-8 during 920k steps using a batch size of 512, learning rate warm up of 500, and AdamW optimizer with a 2e-5 learning rate. Note that the training procedure is similar to all-mpnet-base-v2, with the exception of a smaller batch size and significantly more steps.

Model 4. all-MiniLM-L12-v2

Using microsoft/MiniLM-L12-H384-uncased as the base model, all-MiniLM-L12-v2 is by far the smallest model in the lineup, yet it ranks at #4 on the SBERT Performance Sentence Embeddings evaluation, trailing closely behind the other 3 models. The base model distills the BERT base (uncased) teacher model into a "6-layer student model with 768 hidden size", which "retains more than 99% accuracy on SQuAD 2.0 and several GLUE benchmark tasks using 50% of the Transformer parameters and computations of the teacher model" (?). The training procedure and dataset of this embeddings model are the same as all-distilroberta-v1, key difference is the base model.

4.3.2 MTEB leaderboard models

The following 4 models are selected from (MTEB) Huggingface Leaderboard where they are ranked on their performance on the Massive Text Embedding Benchmark (?). This benchmark evaluates 33 models across 8 embedding tasks, including Pair Classification, Semantic Textual Similarity, Clustering, and Bitext Mining. The Pair Classification is explained in the Approach section, and other tasks that relate to future plans are discussed in the Appendix section.

We selected the top 4 highest-performing models on the overall MTEB evaluation with model size under 1.5GB. The selected models also rank among the highest on the semantic textual similarity and pair classification tasks, relevant to our use case.

Table 2: Model Specifications

Model	**Size (GB)**	**dimension**
mxbai-embed-large-v1	0.67	1024
multilingual-e5-large-instruct	1.12	1024
GIST-embedding-v0	0.44	768
ember-v1	1.34	1024

Model 1. mxbai-embed-large-v1
This embedding model is built for use in Retrieval Augmented Generation (RAG) applications using a custom dataset containing zero overlap with the MTEB train and test sets. This ensures that the model's results on the MTEB benchmark are unbiased. The model is trained on over 700 million pairs using contrastive training and tuned on over 30 million triplets using the AnglE loss (?). The AnglE loss mitigates the vanishing gradient problem caused by the saturation zones in the cosine function (?). As a result, this model is ranked at #7 on the overall MTEB performance despite its 73% reduction in size compared to the next smallest model in the top 6. It is also ranked at #3 on the STS task.

Model 2. multilingual-e5-large-instruct
Using xlm-robert-large as the base model, multilingual-e5-large-instruct is designed for ranking tasks such as text retrieval or semantic similarity. It follows the training procedure of weakly-supervised contrastive pre-training with 1 billion multilingual text pairs including data from Wikipedia, Reddit, S2ORC and Stackexchange. The model is trained on a large collection of unlabeled text pairs from a dataset named CCPairs, using in-batch negatives to distinguish relevant text pairs from irrelevant ones. This pre-training helps the model learn a generalized understanding of textual similarities. Then this is followed by supervised finetuning with 1.6 million downstream application text pairs including SQuAD and Quora Duplicate Questions (?). Additionally fine-tuned on $500k$ pairs of synthetic data generated by GPT-3.5/4 per Wang et al, 2024 (?).

Model 3. GIST-embedding-v0
Using the multilingual bge-large-en-v1.5 (?) as the base model, GIST-embedding-v0 is fine-tuned using the GISTEmbed framework (?) for improved contrastive fine-tuning on the MEDI (?) dataset.

This framework modifies the training batch by identifying and excluding negative examples that are too similar to the query-positive pairs, which can be mistakenly treated as negatives. "This is done through leveraging a guide model (G) to dynamically select the in-sample negatives for the contrastive training of embedding models (?)". The guide model functions as an agent capable of scoring relevance of text relative to a given query. This adjustment ensures that the model does not incorrectly categorize positive pairs (texts relevant to the query) as negatives (irrelevant texts). This is achieved by carefully selecting negatives that are sufficiently dissimilar to the query-positive pairs, thus removing unnecessary noise.

Model 4. ember-v1
Although the official research paper is yet to be published by the development team at LLMRails, the ember-v1 model card on Hugging Face states that the model has been trained on text pairs from a variety of domains using retrieval-oriented (?) and prompt-free (?) techniques for generating English text embeddings. The retrieval-oriented pre-training paradigm, RetroMAE, model modifies the input sentence with distinct masks for the encoder and decoder. The BERT-like transformer encoder generates an embedding from its masked input, which is combined with the single-layer transformer decoder's differently masked input. The model then reconstructs the original sentence through masked language modeling. Additionally,

ember-v1 uses the prompt-free SetFit (?) framework for few-shot fine-tuning, by creating detailed embeddings from a few labeled text samples.

4.4 Evaluation

Given the binary classification problem formulation of classifying issue pairs as duplicates for cosine similarity scores exceeding a threshold, we use standard metrics for evaluating binary classifiers. We start by defining the basic terminology:

- **True Positives (TP):** Duplicate issue pairs classified as duplicates.
- **True Negatives (TN):** Non-duplicate issue pairs classified as non-duplicates.
- **False Negatives (FN):** Duplicate issue pairs classified as non-duplicates.
- **False Positives (FP):** Non-duplicate issue pairs classified as duplicates.
- **Precision**:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

- **Recall**:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

1. **F1 Score:**

$$F1 = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

   The F1 score is a threshold-dependent metric for evaluating the performance of a binary classification model as a harmonic mean of the precision and recall.

2. **ROC-AUC:**
   The ROC curve plots the true positive rate against the false positive rate for various threshold levels.

   Any point located on the diagonal line illustrates a scenario where the proportion of accurately identified duplicates equals the proportion of non-duplicates mistakenly labeled as duplicates. Points situated to the left of the diagonal line indicate better performance, as they represent a higher proportion of duplicates correctly classified.
   Connecting every point calculated at different threshold forms the ROC curve, the area under which is referred to as the AUC (area under the curve). This threshold-agnostic metric is a measure of performance across all possible classification thresholds, interpreted as the percentage of correctly classified pairs.

3. **Precision-Recall:**
   Plotting the precision and recall together is particularly relevant to our evaluations, due to the imbalance between duplicate and non-duplicate pairs, as precision is unaffected by the number of true negatives.

For our final evaluations, we computed the above metrics across the threshold range $[0.6, 0.9]$ in iterations of 0.02.

4.5 Data Sampling Methodology

Due to compute limitations and the scope of the data, we employed sampling methods to evaluate the embeddings on an adequately representative subset of the data as an estimate of the true performance.

4.5.1 127 Issues Sampling

Initially, we used a manual sampling procedure to quickly test our streamlined pipeline during development on a smaller subset. We selected 100 issues from the data frame randomly and added one duplicate issue for 27 of the selected issues. This procedure preserved the original distribution of the data frame wherein 21% of the issues are duplicates of at least one other issue.

4.5.2 Large Set Sampling

We automated the pipeline by sorting our data by duplicate count and selecting 5000 entries from the top and 5,000 entries from the bottom. However, this resulted in a lower duplicate ratio of 10.5%.

On testing, we found that selecting the top 10,000 entries from the sorted dataset had a 24% duplicate ratio, which was much closer to the desired rate.

4.5.3 Stratified Sampling

We wanted to get a better representation of characteristics in the subset. More precisely, we wanted to reduce potential bias that could be introduced by assuming normal data. We also wanted to increase sub-group representation within the dataset.

A sub-group in this context refers to issues marked under a specific 'Issue-type'. Therefore, we stratified the sample based on 'Issue-type', creating unique sub-groups and sampling equally from these sub-groups as per the following method:

```
1. Convert 'Duplicated_issues'
   column in dataset_manager.df to
   tuples
2. Fill missing values in '
   Issue_type' column of
   dataset_manager.df
3. Perform stratified sampling on
   dataset_manager.df based on '
   Issue_type'
4. Initialize embeddings_df as a new
    DataFrame
5. Generate embeddings for
   stratified sample
```

4.5.4 Hard Negative Mining

We implemented hard negative mining to ensure that the selected models are evaluated on a nuanced dataset, sampling issues that are semantically close but are not classified as duplicates. Through this, we were able to get a finer distinguishing metric comparing the performance of the chose models.

```
1. Convert 'Duplicated_issues' in
   dataset_manager.df to tuples
2. Stratify dataset_manager.df by '
   Duplicates_count' and sample 20%
    from each group
3. Generate embeddings for the
   sampled groups
4. Calculate similarity matrix from
   embeddings
5. Find hard negative pairs using
   similarity matrix
6. Add hard negatives to sampled
   groups
7. Generate embeddings for the
   enhanced dataset with added hard
    negatives
```

Although this method produced better results in the manual testing phase, it would have required restructuring the dataframe carry-forward strategy established in our automated pipeline owing to which we had to forego this, leaving it as a future improvement strategy.

5 Experiments

5.1 Data

We sourced a dataset of $115,814$ Mozilla Firefox bug reports as GitHub issues from LOGPAI's BugRepo repository which collects bug reports for research purposes.

Each entry contains the following fields:

1. *Issue_id*: Unique identifier of the issue.
2. *Component*: Component where the issue was observed. Ranging from UI/DOM to Networking layer.
3. *Duplicated Issue*: Issue_id of duplicate issue. May be empty for issues with no duplicates. For issues with multiple duplicates, this field is limited to one duplicate reference.
4. *Title*: Brief description of the observed issue
5. *Description*: More detailed explanation of the issue on a technical level, often including console error messages and build specifications. Includes author of the issue and date of report, separated by semicolons.

The following fields are ignored because they contain no relevant natural language information: *Priority, Status, Resolution, Version, Created_time, Resolved_time*.

5.1.1 Extract-Transform-Load (ETL)

Starting with the data preprocessing step, we loaded the CSV file containing $115,814$ Mozilla Firefox repository GitHub issues as a Pandas dataframe.

5.1.2 Exploratory Data Analysis (EDA)

As a proxy for the linguistic diversity of the issues, we found that the Type-Token-Ratio of the concatenated title-description content is 0.0471, indicating a diverse vocabulary.

Investigating the structure of the issue descriptions, we find that they most often consist of brief descriptions (4-10 words), separated by semi-colons. Additionally, they occasionally include key-value pairs, for example *"Location: file_name.py; Line Number 313;"*, as structured language. Conversely, the issue titles are purely natural language summaries of the observed problem, often non-technical and colloquial.

Excluding punctuation and stop words, we computed the top 20 unigrams, bigrams, and trigrams most frequently used in the issue titles and descriptions. We observe that "actual results" and "expected results" have the highest bigram counts, confirming a frequently used format for the issue descriptions. Additionally, technical specifications such as operating systems, coding languages, and build id's rank among the top unigrams.

5.2 Implementation

We utilized standard Python Machine Learning libraries, including Pandas, Numpy, and SciPy. Additionally, we imported and employed the models using the Hugging Face Transformers library. All remaining code in this project is written by our team as described in the following subsections.

5.2.1 Data Configuration

We created a *configurations.py* file to centralise the configuration data for the pipeline to be built upon.

- Data paths (`DATA_PATH`, `DATASET_FILE`)
- List of models to evaluate (`MODEL_NAMES`) MODEL NAME
- Other model-specific settings

5.2.2 Extract-Transform-Load Data

Responsible for data preprocessing, transformation, and loading. Contains the `ETLProcessor` class with the following methods:

- Load raw data from CSV files.
- Clean and prepare the dataset.
- Save the processed dataset in a pickle file.

5.2.3 Embedding Generation

Handles the generation of text embeddings using various sentence transformer models. Defines the following components:

- `EmbeddingsGenerator` class: Handles embedding generation using a specified model.
- `DatasetManager` class: Loads and preprocesses datasets.
- `EmbeddingsPipeline` class: Orchestrates the process of embedding generation and saving.

5.2.4 Embedding Evaluation

Evaluates the performance of the generated embeddings. Contains the `EmbeddingEvaluator` class with the following methods:

- Calculate similarity matrix for embeddings.
- Compute F1 score, precision, and recall based on similarity scores and ground truth labels.
- Compute cross-entropy loss.
- Compute ROC-AUC (Receiver Operating Characteristic - Area Under the Curve).

5.2.5 Evaluating Multiple Models

The main Jupyter Notebook file that automates the entire evaluation process.

- `setup_folders`: Creates necessary directories for storing evaluation metrics and results.
- `create_evaluation_folder_and_csv`: Prepares the CSV file to store evaluation metrics and appends results for each model.
- `evaluate`: Core function that handles:
  - Setting up directories.
  - ETL (data loading and preprocessing).
  - Embedding generation.
  - Embedding evaluation.
  - Storing evaluation metrics in the CSV file.

5.2.6 Pipeline Flow

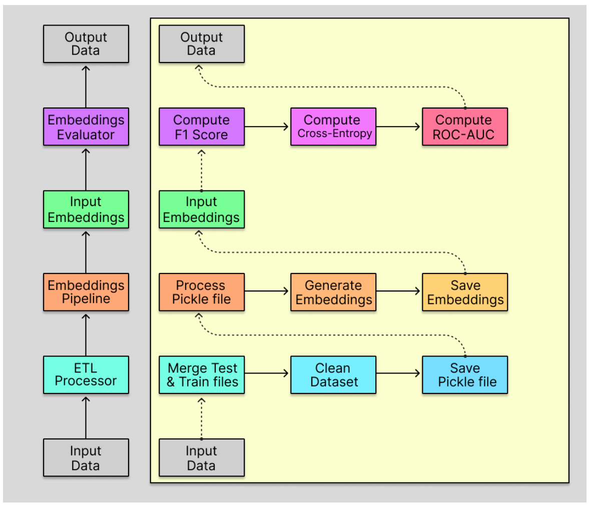

Figure 1: Pipeline Architecture

ETL: Main notebook invokes `ETLProcessor` to load, clean, and process the dataset.

Embedding Generation: Main notebook invokes `EmbeddingsPipeline` to generate embeddings using a specific model from `configurations.py`.

Embedding Evaluation: Main notebook uses `EmbeddingEvaluator` to compute F1 score, cross-entropy loss, ROC-AUC, and other metrics based on the generated embeddings.

Results: Main stores evaluation results into `all_evaluations.csv` for easy comparison across different models.

5.3 Results

5.3.1 Results from 127 issues sample

Sampling 127 issues per sampling procedure described in 4.5.1, we observe the results in Table 3.

Table 3: 127 sample results at optimal thresholds

Model	TP	FN	TN	FP	F1
all-distilroberta	34	34	7883	25	0.54
ember-v1	29	39	7896	12	0.53
mxbai-embed-v1	29	39	7893	15	0.52
all-mpnet-v2	30	38	7889	19	0.51
multi-qa-mpnet	35	33	7874	34	0.51
GIST-embed-v0	30	38	7880	28	0.48
multilingual-e5	28	40	7884	24	0.47
MiniLM-L12-v2	30	38	7803	105	0.30

5.3.2 Results from 10000 issues sample

Sampling 10000 issues per the sampling procedure in 4.5.2, we evaluated the models across the threshold range $[0.6, 0.9]$

Table 4: Model Performance at Individually Optimal Threshold Values

Model	TP	FN	TN	FP	F1
all-mpnet-v2	1362	3162	$5(10^7)$	1962	0.35
GIST-embed-v0	1285	3239	$5(10^7)$	1705	0.34
mxbai-embed-v1	1129	3395	$5(10^7)$	976	0.34
multi-qa-mpnet	1269	3255	$5(10^7)$	1802	0.33
ember-v1	1272	3252	$5(10^7)$	1808	0.33
all-distilroberta	1194	3330	$5(10^7)$	1813	0.32
MiniLM-L12-v2	808	3716	$5(10^7)$	1498	0.24
multilingual-e5	3329	1195	$5(10^7)$	$1.2(10^5)$	0.05

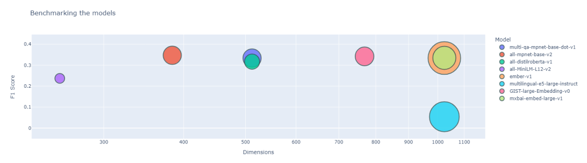

Figure 2: Evaluation Results Visualization

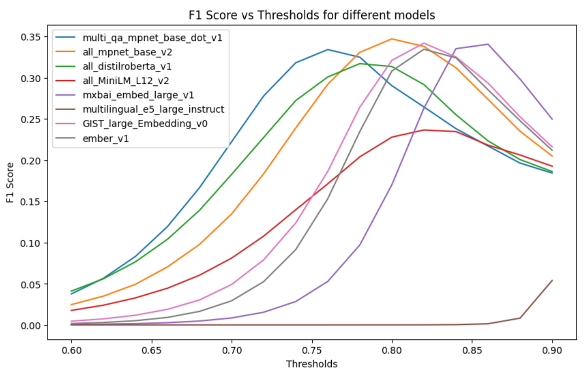

Figure 3: Model Performance Across Classification Thresholds

Per Table 4, each model performs significantly better than randomly classifying an issue pair as a duplicate with probability (Positive Pairs/Negative Pairs), which would yield F1 of 0.00009. Visually, we observe that the models perform similarly well, except for MiniLM-L12-v2 and multilingual-e5-large-instruct. The two highest performing models, all-mpnet and GIST-embed, perform similarly well. Note the similarity in model size (420 MB and 440 MB); yet the former model produces embeddings of half the dimensionality at 384 rather than 768. Furthermore, mxbai-embed-large-v1, multi-qa-mpnet-base-dot-v1 and ember-v1 performed similarly well. Note that, despite being trained specifically on domain-specific StackOverflow duplicate question pairs data for twice the number of training steps, the multi-qa-mpnet-base-dot-v1 model does not outperform its generalist all-mpnet-v2. This result confirms the observation that embeddings models, created for a given task, do not tend to dominate across unrelated tasks (?).

Per Figure 3, the multilingual-e5 model performs very poorly compared to the other models across the threshold range. Per the model card on Hugging Face, the model is designed for ranking tasks such as semantic textual similarity, where similarity is evaluated for relative order rather than score magnitude. Again highlighting the effect of the intended task on model performance in unrelated tasks.

For ROC-AUC and Precision-Recall metrics, refer to the Appendix.

5.3.3 Improvements over the baseline

Having evaluated the embeddings models as provided, we did not improve over the baseline results. However, the pipeline allows for improvements to the embedding generation and classification task. The embeddings models can be fine-tuned for the domain-specific pair classification task per the contrastive learning objective in section 4.2, as they are fully open-code models, available on Hugging Face. Furthermore, improvements to the classification task and implementation are detailed in Section 6.

6 Conclusion

We find that the intended task of the model has an effect on performance for the pair classification task, confirming the findings of Muennighoff et al, 2023 (**?**). Most notably, the multilingual-e5 model performs poorly on the classification task because it produces embeddings for the intended use in ranking tasks, where cosine similarity magnitude is less expressive. Per our evaluations, we find the all-mpnet-v2 model to be the most relevant to our use case as it achieves the highest F1 score, with a relatively low model size (420MB) and low embeddings dimensions (384). This allows the model to be easily deployed in production without LLM-scale compute and space requirements.

The thorough literature survey conducted in Section 3 provided a deep understanding of the history of document embeddings, natural language tasks, and relevance to our project goal. The model descriptions we wrote for each of the 8 embeddings models highlighted how embeddings models differ by training loss, objective, data, and base model.

Unfortunately, the nature of our problem formulation complicated the interpretability of the evaluations, as the pairwise comparisons created an unbalanced dataset that reduced the effectiveness of the traditional binary classification evaluation metrics. Given the pair classification nature of our problem formulation, we relied on binary classification evaluation metrics which resulted in lowered interpretability, and effectiveness.It may have been more appropriate to formulate the problem as a semantic textual similarity task and conducting evaluations using Spearman correlation based on cosine similarity. This threshold-agnostic metric evaluates how well a model creates embeddings that represent the semantic similarity of documents by cosine similarity, such that duplicate issue pairs have the highest cosine similarity and non-duplicate issue pairs have the lowest. Relating to the original task of detecting duplicate issue pairs in a repository, this problem formulation would have enabled the system to rank issue pairs by cosine similarity rather than creating binary classifications. However, this formulation may have required unique heuristics such as limiting the number of duplicates for a given issue.

The pipeline can be improved in the future by training and employing a classification head rather than using the cosine-similarity operator directly. We had initially considered doing so in this report, but the process introduces extra hyperparameters and is beyond the scope of our established evaluations.

A Appendix

A.1 ROC-AUC and Precision-Recall Curves

all-distilroberta-v1

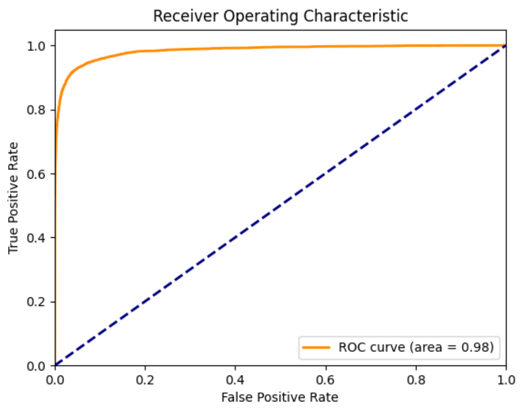

Figure 4: Distil Roberta ROC AUC

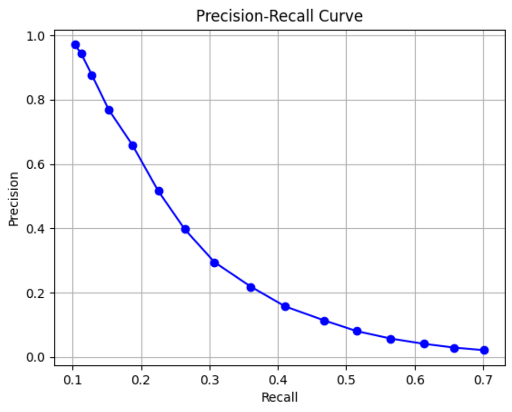

Figure 5: Distil Roberta Precision-Recall Curve

ember-v1

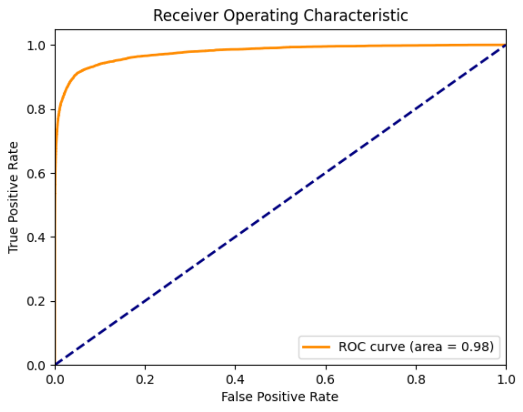

Figure 6: Ember ROC AUC

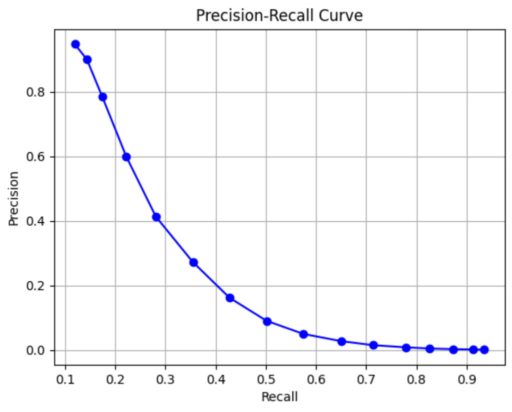

Figure 7: Ember Precision-Recall Curve

mxbai-embed-v1

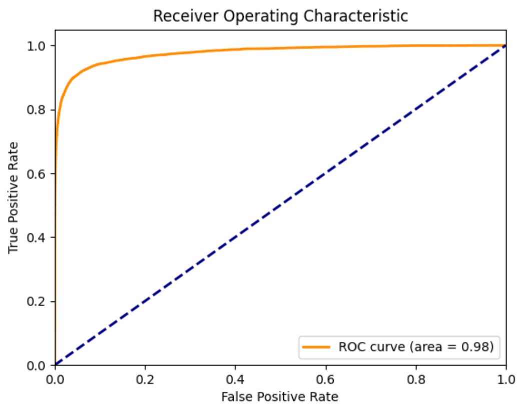

Figure 8: Mxbai ROC AUC

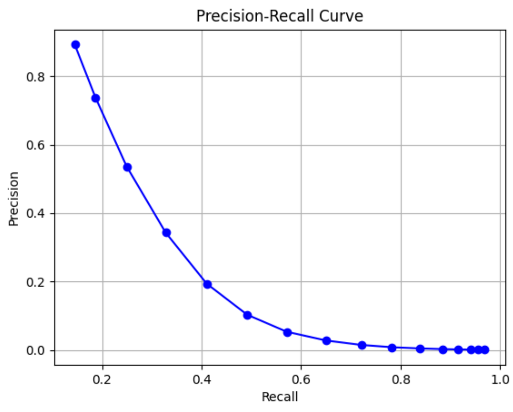

Figure 9: Mxbai Precision-Recall Curve

all-mpnet-v2

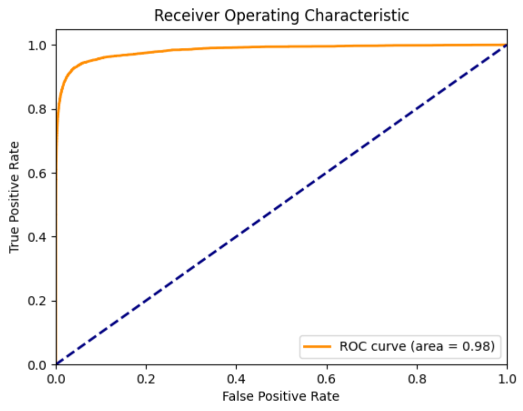

Figure 10: All-mpnet-v2 ROC AUC

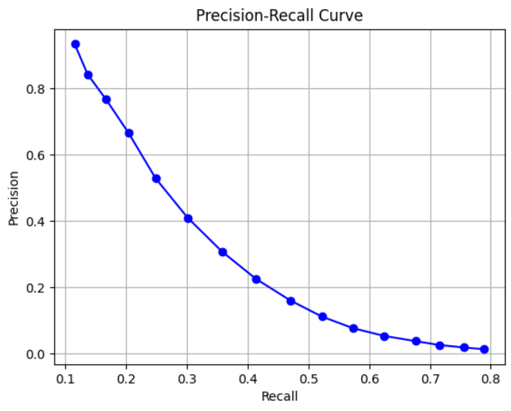

Figure 11: All-mpnet-v2 Precision-Recall Curve

multi-qa-mpnet

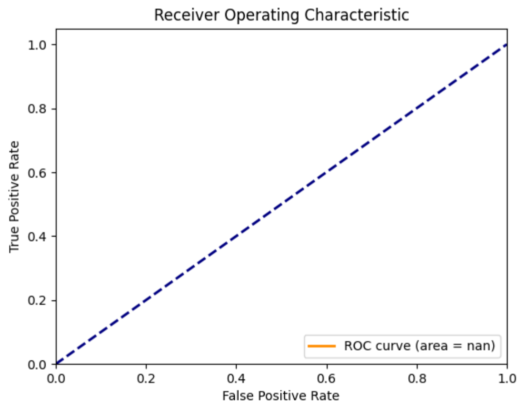

Figure 12: Multi-qa-mpnet ROC AUC

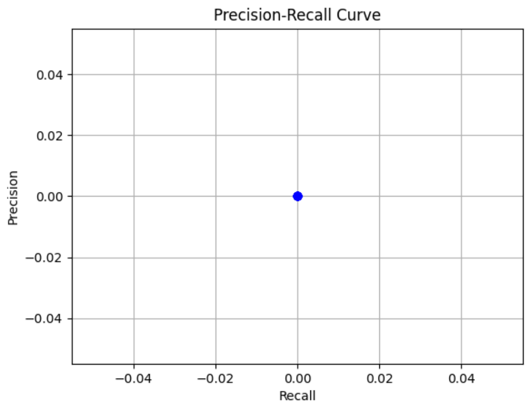

Figure 13: Multi-qa-mpnet Precision-Recall Curve

GIST-embed-v0

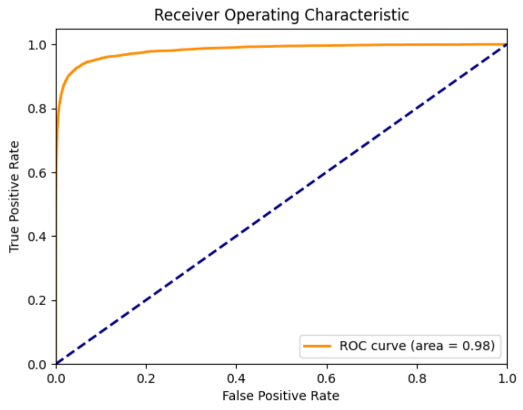

Figure 14: GIST-embed-v0 ROC AUC

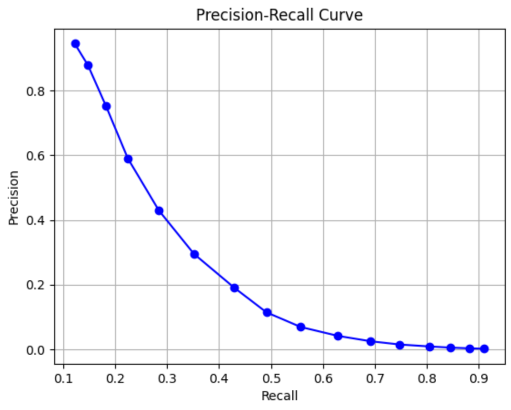

Figure 15: GIST-embed-v0 Precision-Recall Curve

multilingual-e5

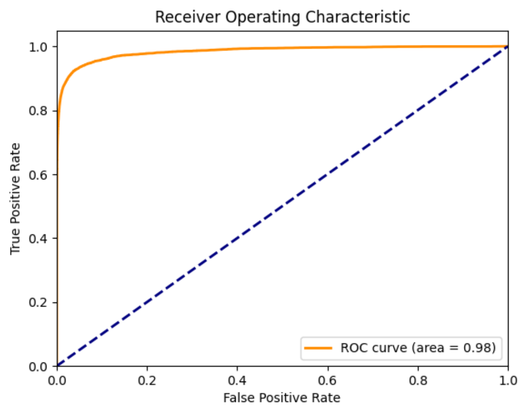

Figure 16: multilingual-e5 ROC AUC

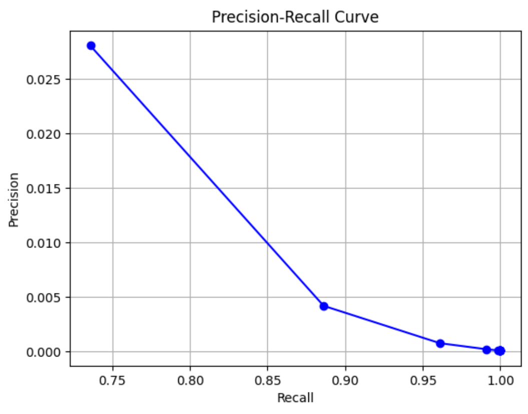

Figure 17: multilingual-e5 Precision-Recall Curve

MiniLM-L12-v2

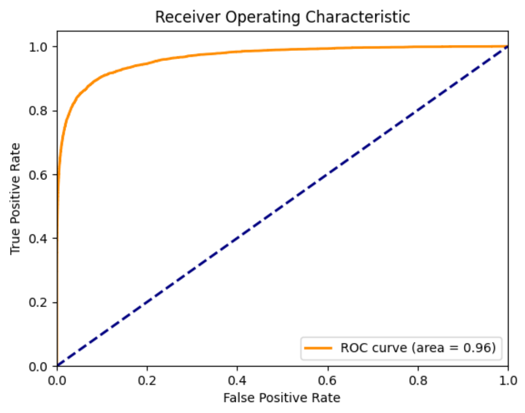

Figure 18: MiniLM-L12-v2 ROC AUC

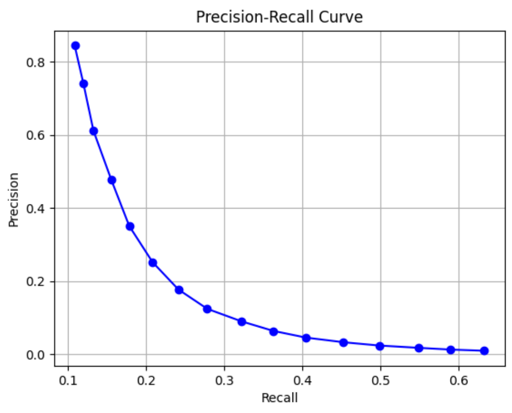

Figure 19: MiniLM-L12-v2 Precision-Recall Curve