1. Data Collection - found issues and PRs resolving them. For each issue's PR, blame and find the commits that change those lines in the PR
    a. How did we find PRs for issues?
    b. How did we run blame?
    c. Where did we collect data from, and how many data points were collected
    d. How long did data collection take?
    e. Please link to the raw dataset
2. Data cleanup
    a. TODO
    b. How did you quantify the risk and severity of the issue
3. Embedding generation
    a. What models did you try? Which one did you use?
    b. How did we handle code that does fit the context window?
    c. How long have embedding tools been around?
4. Forecasting - explain each model, any data pipeline steps or tuning, and accuracy

1.Data Collection - found issues and PRs resolving them. For each issue's PR, blame and find the commits that change those lines in the PR

a.How did we find PRs for issues?

Initially we downloaded the PRs from the github using GitHub API as Jsons in mongoDB then we locate the PR and issue by matching the issue number , github link by analysing
- Title
- Description
- Comments

and linking using Regex by mapping the keywords and retrieve the issue details from GitHub
  📄 AI-powered risk assessment before merging code
And then use events api in Github to link issues
  📄 AI-powered risk assessment before merging code

b.How did we run blame?

The **SZZ algorithm** is a technique used in software engineering to identify the commits in a version control system that likely introduced bugs.

1. **Start with a bug-fixing commit**—a change that resolves a known bug.
2. **Trace back** through the version history to find the lines of code that were modified to fix the bug.

3. **Identify the last commit** that touched those lines before the fix. That commit is flagged as a potential *bug-introducing* change

This method is widely used in **empirical software engineering research**

According to this research article
https://www.scribd.com/document/866959848/SZZ-Unleashed

c.Where did we collect data from, and how many data points were collected

From :https://github.com/ballerina-platform/ballerina-lang/issues
We have **125825** there **42043** was labeled as **is_bug_introducing** 33.41%
While prepare the data set we collect all the commit IDs and labeled with the dataset derived from **SZZ algorithm** Analisys

d.How long did data collection take?

Nearly it took 3 months to understand the project nature and optimize the data pipelines.

e.Please link to the raw dataset

All Major versions of the dataset
https://drive.google.com/file/d/1Jy1UhNqEig-qDHKlDLwtk6mxRIv52ib_/view?usp=drive_link

2.Data cleanup

a.TODO

Keyword matching from pull request to issue

- Key word mapping
  - Title `Completed` ▾
  - Description (body) `Completed` ▾
  - Comments body `Not Started` ▾
- Timeline event
  - Closed `Blocked` ▾
  - Referenced `Completed` ▾
  - Cross-referenced `Not Started` ▾

Also while transforming embeddings using attention based pooling would be very efficient to

capture the semantic meaning

b.How did you quantify the risk and severity of the issue

At this moment we didn't defined any risk level but we have some useful data like coupled fils , entropy , developer experience , (frequence of in 60 days),added lines and deleted lines kind of information using that we can prepare it

3.Embedding generation

a.What models did you try? Which one did you use?

For embedding I have used codeBERT(`microsoft/codebert-base`) with AutoTokenizer

Also I try RoBERT with sample data set (small chunk from main dataset) to Observe the duration of embeddings it's remain as codeBERT no significant difference

b.How did we handle code that does fit the context window?

Handling the code with the context window we transform our diff chunk as embedding vectors. There we use sliding window with 100 tokens overlap to get vector stack then reduce the dimension using mean pooling , attention based pooling is the efficient way due to computational complexity we choose mean pooling.The default size of codeBERT is 512 token so in every 512 stack we get Only CLS token to pass to the pooling function and finally represent as 768 dimensional vector
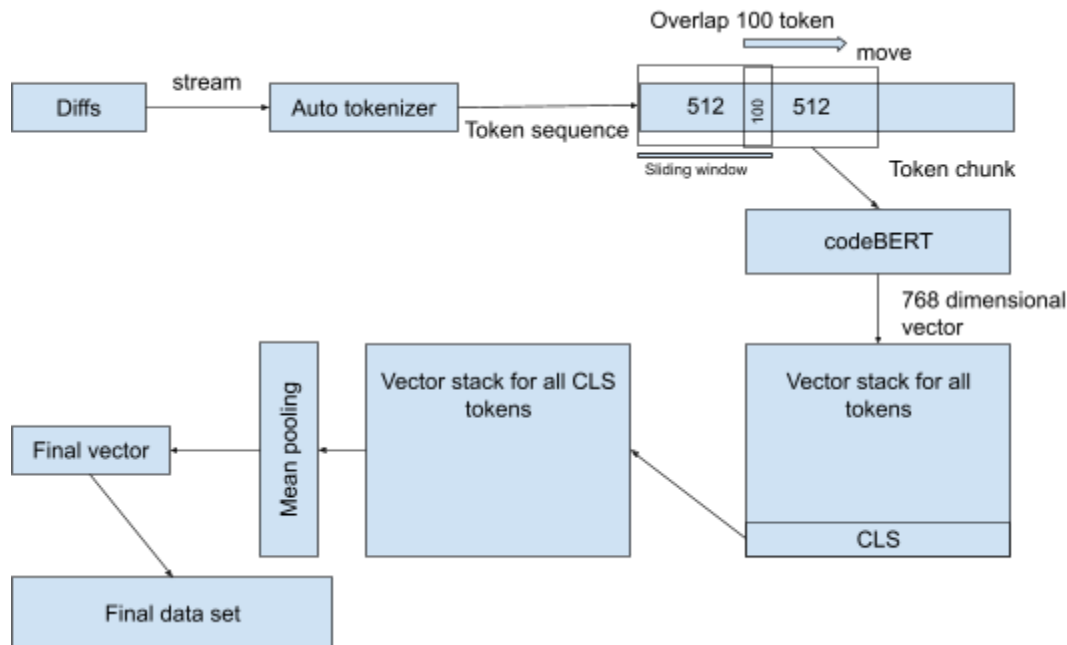
c.How long have embedding tools been around?

In the initial stage without any pooling and sliding window techniques, to process the **125825** data took days even multiple failures memory errors and several rounds of Data cleanup and segregation of comparatively small Diffs

After introducing Sliding window and mean pooling just for 10 records it took 13 minutes then I decided to go with GPU, in colab with their free computer units we can get embedding for 1000 to 1600 records it defer based on the diff chunk size

After getting runpod.io access I used RTX A5000 GPU in 3 hours I could manage to get nearly 7000+ data points but some long chung failed and I logged them due to exceeding memory limit

It would better to multi GPU setup rather than switch to a Biggest GPU

5.Forecasting - explain each model, any data pipeline steps or tuning, and accuracy

Most of the mode I used the technique to split the data set for train based on chronological order in some random case use random splitting

It several models but failed

I will list down some systemic experiment approach that we have followed

Full dataset with statistical feature

Model	Accuracy	Precision	Recall	F1-score	ROC_AUC
Random Forest Baseline	0.6712	0.4842	0.5874	0.5293	0.7182
Random Forest + SMOTE	0.6442	0.4632	0.7822	0.5656	0.7295
Random Forest + SMOTE + Tomek	0.6445	0.4609	0.7794	0.5652	0.7252
XGBoost Basline	0.6754	0.4897	0.6016	0.5373	0.7260
XGBoost + SMOTE	0.6548	0.4703	0.7819	0.5689	0.7307
XGBoos + SMOTEt + Tomek	0.6712	0.4842	0.5874	0.5293	0.7182

PCA

I perform PAC and UMAP Analysis
UMAP give poor performance in 2 dimension representation
While PCA tuned based on the cumulative Explained variance the result is 95% significance 51 components at 99% significance level 177 components we use 177

While performing PCA did 2 experiments
Data split randomly

Model	Dataset	Data split	recall	F1 score	precision
XGBoost	PCA-Reduced Embeddings Only	random	0.80	0.77	0.74
		chronological	0.85	0.27	0.16

Small dataset with statistical feature and embeddings

Model	Dataset	F1 score	Best parameters
XGBoost	Stats_Only	0.2961	{'learning_rate': 0.05, 'max_depth': 20, 'n_estimators': 200, 'subsample': 1.0}
	Embeddings_Only	0.2850	{'learning_rate': 0.1, 'max_depth': 10, 'n_estimators': 200, 'subsample': 0.7}

	PCA-Reduced Embeddings Only	0.2790	{'learning_rate': 0.1, 'max_depth': 10, 'n_estimators': 300, 'subsample': 0.7}
	Stats + Full Embeddings	0.3169	{'learning_rate': 0.1, 'max_depth': 20, 'n_estimators': 300, 'subsample': 0.7}
	Stats + PCA-Reduced Embeddings	0.3040	{'learning_rate': 0.1, 'max_depth': 10, 'n_estimators': 100, 'subsample': 0.7}

For tune the XGBoost model I use random search cv and in some case Grid search

```python
# --- Model Configuration ---
PARAM_GRID = {
    "n_estimators": [100, 200, 300],
    "max_depth": [10, 20],
    "learning_rate": [0.05, 0.1],
    "subsample": [0.7, 1.0],
}
```

Model	Dataset	F1 score	ROC AUC
ANN	Stats_Only	0.2609	0.6014
	Embeddings_Only	0.2996	0.6456
	PCA-Reduced Embeddings Only	0.2732	0.6308
	Stats + Full Embeddings	0.3209	0.6725
	Stats + PCA-Reduced Embeddings	0.2910	0.6517

For the MLP the architecture that we used is

```python
model = keras.Sequential([
        layers.Input(shape=(input_shape,)),
        layers.Dense(128, activation='relu'),
        layers.Dropout(0.3),
        layers.Dense(64, activation='relu'),
```

```
        layers.Dropout(0.3),
        layers.Dense(1, activation='sigmoid') # Binary
classification
    ])
```

Repository

https://github.com/Pradishan/wso2-AI-Tool

Project snapshot

🖻 AI-powered risk assessment before merging code