




Feature Extraction

Refer to this [research](#) they use 16 statistical features
Those are:

- ☒ Ft1 Lines of code added
- ☒ Ft2 Lines of code deleted
- ☒ Ft3 Files churned / Number of files
- ☒ Ft4 Lines of code in previous version 
- ☒ Ft5 Number of modified subsystems
- ☒ Ft6 Number of modified sub-directories
- ☒ Ft7 Entropy (spreading of changes)
- ☐ Ft8 Purpose of a change (e.g., bug fix) 
- ☐ Ft9 Number of previous committers
- ☒ Ft10 Time between committer's contributions
- ☒ Ft11 Number of unique changes 
- ☒ Ft12 Overall experience of committer
- ☒ Ft13 Recent experience of committer
- ☐ Ft14 Number of highly coupled files
- ☐ Ft15 Number of coupled files for all degrees
- ☐ Ft16 Number of non-modified coupled files

Feature extraction focuses solely on the **master** branch of this repository.

A code diff is extracted using GitPython. `commit.stats` is a native attribute of `Commit` objects in GitPython, returning a `Stats` object. `stats.total` provides a dictionary with the following structure:

```
None
{
  'insertions': 42,
  'deletions': 17,
  'lines': 59,
  'files': 3
}
```

Ft1 Lines of Code Added

This feature extracts the number of lines added in a code diff using GitPython. The `commit.stats` attribute of `Commit` objects returns a `Stats` object, and `stats.total` provides the 'insertions' value.

Ft2 Lines of Code Deleted

This feature extracts the number of lines deleted, similar to Ft1, using `stats.total` to retrieve the 'deletions' value.

Ft3 Files Churned / Number of Files

This feature extracts the number of files changed, derived from the 'files' value within the `stats.total` object.

Ft4: Lines of code in previous version

The current interpretation of "lines of code in previous versions" is ambiguous, as it's unclear whether it refers to the entire repository or only to **changed files**. Furthermore, accurately counting lines of code is computationally intensive.

The trade-off between accuracy and performance:

Calculating the exact line count is very slow, while using file size in bytes is extremely fast and provides a highly correlated proxy for the model. My choice was to prioritize performance by using the file size in bytes, as it offers nearly the same predictive value at a much lower computational cost. The Chosen Implementation (Fast Proxy)

This code calculates the total size in bytes of the files as they existed in the parent commit:

```
Python
previous_total_size = 0

if commit.parents:
    parent = commit.parents
    for diff in commit.diff(parent):
        # 'a_blob' is the file's state before the change.
```

```
        # '.size' gets the file size in bytes, which is very
    fast.
    if diff.a_blob:
        previous_total_size += diff.a_blob.size
```

If this approach does not yield good results during AI model training, we can change the approach to extract the lines of code before and after the commits:

```
Python
from git import Repo

repo = Repo("path/to/repo")
commit = repo.head.commit
parent = commit.parents

lines_before = 0
lines_after = 0

for diff in commit.diff(parent):
    if diff.a_blob:
        lines_before +=
diff.a_blob.data_stream.read().decode('utf-8',
errors='ignore').count('\n')
    if diff.b_blob:
        lines_after +=
diff.b_blob.data_stream.read().decode('utf-8',
errors='ignore').count('\n')

print(f"Total lines before: {lines_before}")
print(f"Total lines after: {lines_after}")
```

Ft5 Number of modified subsystems

The research paper does not provide an explicit definition for what it considers a "subsystem."

I considered a **subsystem** to be the **top-level directory** in a file's path.

This is a common and practical heuristic used in software repository mining when a formal definition isn't available. Top-level directories often represent major components of a project.

For example, given a file path like `src/main/java/io/ballerina/types/Atom.java`, the script would extract "`src`" as the subsystem. If a commit modified files in both the `src` and `tests` directories, it would be counted as modifying two subsystems.

Python

```
subsystems = {path.split('/')[0] for path in commit.stats.files.keys()}
```

Ft6 Number of modified sub-directories

Same as the Ft5

Python

```
modified_dirs = {os.path.dirname(f) for f in  
commit.stats.files.keys()}
```

This one grabs the full directory path (excluding the filename) for each modified file

Ft7 Entropy (spreading of changes)

Python

```
total_lines_changed = lines_added + lines_deleted  
entropy = 0.0  
if total_lines_changed > 0:  
    for file_path, file_stats in commit.stats.files.items():  
        file_lines_changed = file_stats['insertions'] +  
file_stats['deletions']
```

```
if file_lines_changed > 0:
    change_proportion = file_lines_changed /
total_lines_changed
    entropy -= change_proportion * math.log2(change_proportion)
```

This snippet calculates **entropy** to quantify how evenly code changes are distributed across files in a Git commit. It's a concept borrowed from information theory—Shannon entropy—to measure uncertainty or unpredictability. Here, it tells us whether the changes in a commit are

Concentrated (most lines changed in a few files → lower entropy)

Distributed (lines changed fairly evenly across many files → higher entropy)

It calculates the entropy of code changes in a Git commit — measuring how evenly the changes (insertions and deletions) are spread across the modified files. A higher entropy means the changes are spread out over many files; a lower entropy means they're concentrated in just a few. This helps assess the complexity or impact of a commit.

[Using entropy to measure the complexity of software changes](#)

Ft8 Purpose of a change (e.g., bug fix)

Currently, I am unable to find a logical way to extract this feature, as it requires manual annotation. Therefore, I am ignoring it for now.

If this approach doesn't work, we can use the commit message instead of the purpose.

However, since we are currently training the model with statistical features, encoding the purpose into classes, the commit message approach will not be effective at this stage.

Ft9 Number of previous committers

```
Python
previous_committers = set()
if commit.parents:
```

```

for diff in commit.diff(commit.parents[0]):
    file_path = diff.a_path or diff.b_path
    if file_path:
        previous_committers.update(file_history[file_path])

```

This code identifies all *previous contributors* to the files modified in the current Git commit. By examining the commit's diff and checking which files changed, it looks up those file paths in a `file_history` dictionary and collects the set of authors who previously edited them. This is useful for analyzing code ownership or suggesting relevant reviewers.

Ft10 Time between committer's contributions

Python

```

# Calculate metrics based on history *before* this commit
past_commits_timestamps = author_history[author_email]
time_since_last = (commit_time -
past_commits_timestamps[-1]).total_seconds() if past_commits_timestamps else 0

```

In the feature extraction script, I extracted the **time difference in seconds** between the current commit and the author's immediately preceding commit.

Ft11 Number of unique changes ,Ft12 Overall experience of committer

Our implementation for features **Ft11 ("Number of unique changes")** and **Ft12 ("Overall experience")** uses a single metric—the author's total previous commit count—due to ambiguity in the paper's descriptions. "Overall experience" is commonly measured by this proxy, and the most practical interpretation of "unique changes" is the number of unique commits. Since both logical interpretations converge to the same calculation, combining them is an efficient and defensible engineering decision that avoids inventing unsupported complexity. So alternatively I introduce **author_total_commits** as a new feature equivalent to experience. Other vice need integrate with personal information of the employee

Ft13 Recent experience of committer

Based on the research paper, there is **no specific research or justification** mentioned to support the choice of a 60-day window.

McIntosh et al. (2016) – The impact of code review coverage and code review participation on software quality

They found developer activity in the past 30–90 days to be strongly correlated with contribution quality.

Inactive or less-recent contributors were more likely to introduce bugs.

 DOI: 10.1109/TSE.2015.2506474

Rahman and Devanbu (2013) – How, and why, process metrics are better

Found that recent contribution count (within 30–60 days) performed well as a predictive feature in defect prediction models.

 DOI: 10.1109/ICSE.2013.6606615

To extract the remaining three features, we will utilize the open-source tool [code-maat](#).

Before processing the data, we need to perform data extraction using code-maat

1. Generate Git Log File

- First, clone the repository.
- Then, generate the git log file using Git Bash:

None

- `git log --pretty=format:'[%h] %aN %ad %s' --date=short --numstat --all > gitlog.log`

2. Perform Code-Maat Coupling Analysis

- Download the latest standalone JAR release.
- Place the JAR in the repository location.
- Run the tool with the following command:

None

- `java -jar code-maat-1.0.4-standalone.jar -l gitlog.log -c`

```
git -a coupling -o coupling.csv
```

This will give output as

Entity - The first file in the coupled pair

coupled - The second file that frequently changes with the first

Degree - Percentage of commits where both files changed together

Average-revs - Average number of revisions (commits) shared between the two entities

Ft14 Number of highly coupled files

Here we consider the Degree for this feature. The choice of **75%** threshold is considered a **hyperparameter**. The best value would be found through experimentation by training the model with different thresholds (e.g., 50%, 75%, 90%) and selecting the one that results in the highest prediction accuracy.

Ft15 Number of coupled files for all degrees

For this feature the explanation is there in the name itself. We get all the count for this coupled pair for all degree.

Ft16 Number of non-modified coupled files

This feature acts as a warning system by counting how many files in a commit have a known dependency on another file that was **not** changed in that same commit. For example, if `config.json` and `server.js` are almost always modified together, but a commit only changes `config.json`, this feature flags that commit. A high count for this feature is a strong "red flag" 🚩, suggesting a developer may have forgotten to make a necessary update in a related file, which is a common way bugs are introduced.

Our script calculates Ft16 per commit as follows

1. **Build a Coupling Map:** A ``coupling_map`` is created from the entire code-maat output, allowing instant lookup of coupled files.
2. **Analyze a Single Commit:** For each commit, the script identifies ``modified_files``.
3. **Find "External" Coupling:** The script iterates through ``modified_files``, checking if any coupled partners of these files are **NOT** in the ``modified_files`` set.

4. **Count the Files:** If a file has at least one coupled partner not modified in the commit, it's counted. Ft16 is the total count of such files within the commit.