Mozila dataset project ideas

Naser Ezzati-Jivan

November 2025

1 Phase 1: Regression vs. Noise Classification

1.1 Goal and Motivation

The goal of Phase 1 is to build an initial supervised learning model that predicts whether a Mozilla performance alert corresponds to a true regression or to noise. This phase is designed as the entry point for the student. It requires basic data science skills and produces a useful baseline for the later phases. The Mozilla dataset is ideal for this step because it already contains regression labels and rich alert-level metadata. A strong Phase 1 outcome is a model that reduces false positives relative to Perfherder's current rule-based alerts, while remaining interpretable.

From a research perspective, Phase 1 answers a simple but important question: can alert metadata alone separate meaningful regressions from noisy fluctuations in large-scale continuous integration testing. If this is successful, later phases can add time-series features and more advanced detection models.

1.2 Objectives and Research Questions

The first phase of the project focuses on understanding whether alert-level metadata is sufficient to distinguish true performance regressions from noise in Mozilla's continuous integration environment. The objectives combine practical goals, such as building an effective baseline classifier, with scientific goals, such as determining which features carry the most predictive power. The phase also provides a foundation for later work by establishing a reproducible methodology for data preparation, modeling, and evaluation.

1.2.1 Main Objective

The primary objective of Phase 1 is to develop a supervised learning model that predicts whether an alert corresponds to a true regression using only the information provided in `alerts_data.csv`. The goal is to determine whether readily available metadata signals, such as magnitude of change, test suite information, and platform characteristics, allow us to identify regressions with higher precision than the current threshold-based methods used in Perfherder.

1.2.2 Sub-Objectives

To support the main objective, Phase 1 has the following sub-objectives:

- Prepare and clean the alert dataset by selecting features, encoding categorical variables, and handling missing values without introducing label leakage.
- Train and evaluate several baseline machine learning models, beginning with Logistic Regression and extending to Random Forest and Gradient Boosting.
- Identify the most influential features and feature types through permutation importance or SHAP analysis.
- Evaluate cross-repository generalization by training on a subset of repositories and testing on a held-out repository.
- Conduct ablation studies to measure the contribution of magnitude-based features, context-based features, and workflow-based features.

1.2.3 Research Questions

The following research questions guide Phase 1:

- **RQ1:** Are alert-level metadata features sufficient to distinguish true regressions from noisy alerts in Mozilla's performance testing environment?
- **RQ2:** Which alert metadata features contribute the most to the accurate identification of regressions?
- **RQ3:** Does a supervised learning model achieve higher precision or lower false-positive rates than Perfherder's existing rule-based heuristics?
- **RQ4:** How well do models trained on alerts from certain repositories or platforms generalize to previously unseen repositories or platforms?
- **RQ5:** Do magnitude-based signals (such as percent change and t-value) dominate predictive performance, or do contextual features (such as platform or suite) also provide significant value?

Together, these objectives and research questions define the scope of Phase 1 and clarify the expected outcomes. They also motivate the design of the experiments described later, ensuring that the results contribute to a broader understanding of regression detection in large-scale continuous integration systems.

1.3 Data Used

Phase 1 uses only the curated alert file:

- `data/alerts_data.csv`: each row represents a single alert detected in Perfherder, along with properties of the alert, its signature, and its associated alert summary.

The target label is:

- `single_alert_is_regression`: Boolean label produced by Mozilla. In Phase 1 we treat `True` as *regression* and `False` as *noise or improvement*.

Phase 1 does *not* depend on raw time series or bug reports. Those are used in later phases.

1.4 Problem Definition

We define a binary classification problem:

$$f(\mathbf{x}) \to y, \quad y \in \{0, 1\} \tag{1}$$

where $\mathbf{x}$ is a vector of alert-level features derived from `alerts_data.csv`, and $y = 1$ indicates regression.

1.5 Feature Set

We divide the Phase 1 features into three groups. This grouping helps interpret which type of signal is most useful.

Magnitude and statistical signals (numeric). These features capture how strong the alert appears to be:

- `single_alert_amount_abs`
- `single_alert_amount_pct`
- `single_alert_t_value`
- `single_alert_prev_value`
- `single_alert_new_value`

Test and platform context (categorical). These features capture where the alert came from:

- `repository_name` or `repository_id`
- `framework_id`
- `machine_platform`

- `single_alert_series_signature_suite`
- `single_alert_series_signature_tags` (if available)
- `lower_is_better` (if present in alerts file)

Workflow hints (categorical or numeric). These features describe how the alert behaved inside Mozilla's workflow:

- `single_alert_manually_created`
- `single_alert_status`
- `alert_summary_status`

For Phase 1, we include these features only if they are recorded at alert creation time. If a feature reflects a later decision, it must be excluded to avoid label leakage. The student should confirm this by checking timestamps.

1.6 Implementation Steps

Step 1: Load and inspect the alerts data.

- Load `alerts_data.csv` using pandas.
- Print the number of alerts, number of unique signatures, and class distribution.

Example python snippet:

```
import pandas as pd

alerts = pd.read_csv("data/alerts_data.csv")
print(alerts.shape)
print(alerts["single_alert_is_regression"].value_counts(dropna=False))
```

Step 2: Label cleaning.

- Remove rows where `single_alert_is_regression` is missing.
- Map labels to integers: regression = 1, else = 0.

Step 3: Feature selection and leakage check.

- Start with the feature groups listed above.
- Remove columns that are clearly identifiers or post-triage decisions.
- Remove columns with nearly all missing values.

A practical rule: any column that uses terms like `triaged`, `fixed`, or `backedout` after the alert timestamp should not be used as input in Phase 1.

Step 4: Handle missing values.

- For numeric features, fill missing values using median imputation.
- For categorical features, fill missing values using `"Unknown"`.

Example:

```
num_cols = ["single_alert_amount_abs",
            "single_alert_amount_pct",
            "single_alert_t_value",
            "single_alert_prev_value",
            "single_alert_new_value"]

cat_cols = ["framework_id",
            "machine_platform",
            "single_alert_series_signature_suite",
            "repository_name"]

alerts[num_cols] = alerts[num_cols].fillna(alerts[num_cols].median())
alerts[cat_cols] = alerts[cat_cols].fillna("Unknown")
```

Step 5: Encode categorical variables. Two safe beginner-level options:

- **One-hot encoding** for low-cardinality features such as framework and repository.
- **Frequency encoding** for high-cardinality features such as suite if needed.

If suite has hundreds of values, frequency encoding avoids an extremely wide feature matrix.

Step 6: Train, validation, and test split.

- Random split 70% train, 15% validation, 15% test.
- Use stratification to preserve the regression ratio.

Example:

```
from sklearn.model_selection import train_test_split

X = alerts[features]
y = alerts["single_alert_is_regression"].astype(int)

X_train, X_temp, y_train, y_temp = train_test_split(
    X, y, test_size=0.30, stratify=y, random_state=42)

X_val, X_test, y_val, y_test = train_test_split(
    X_temp, y_temp, test_size=0.50, stratify=y_temp, random_state=42)
```

Step 7: Baseline models. Train a small set of interpretable baselines:

- Logistic Regression
- Random Forest
- Gradient Boosting (XGBoost or LightGBM if available)

The student should start with Logistic Regression to learn the full pipeline, then move to tree models.

Step 8: Hyperparameter tuning. Use simple grid search on the validation set:

- Random Forest: number of trees, max depth, minimum leaf size.
- Gradient Boosting: number of estimators, learning rate, depth.

Step 9: Evaluation. Compute metrics on the test set:

- Accuracy
- Precision and recall for the regression class
- F1-score
- ROC-AUC

Precision matters because false positives waste triage effort. Recall matters because missing a real regression is costly.

1.7 Experimental Design

Experiment E1: Alert metadata classifier. This is the main experiment. Input features are alert-level metadata only. The target is `single_alert_is_regression`. Models are Logistic Regression, Random Forest, and Gradient Boosting. The evaluation uses the metrics listed above.

Experiment E2: Cross-repository generalization. To test whether the model learns real patterns instead of repository-specific quirks:

- Train on a subset of repositories, such as autoland1 through autoland3.
- Test on a held-out repository such as mozilla-central or mozilla-beta.

This requires filtering the alerts by `repository_name`.

Experiment E3: Feature ablation. Measure the contribution of each feature group:

- Only magnitude features.
- Only context features.
- Magnitude plus context.
- All features.

This shows whether metadata provides value beyond pure magnitude thresholds.

1.8 Concrete Examples of Alerts

Example 1: High-magnitude clear regression. An alert with:

- high `single_alert_amount_pct` (for example above 5%),
- high `single_alert_t_value`,
- stable suite history on a low-noise platform.

Such cases should be easy for both Perfherder and ML to classify as regression.

Example 2: Noisy test false positive. An alert with:

- moderate change magnitude,
- low t-value,
- suite or platform known to be unstable,
- historically high variance in similar alerts.

These are typical "Invalid" regressions. A good model should learn to label many of them as noise.

Example 3: Downstream alert. An alert that appears shortly after another alert within the same summary, often with smaller magnitude. These alerts are frequently labeled "Downstream." Even without time-series features, metadata such as summary linkage and suite patterns may help the model reduce false positives.

1.9 Expected Outcomes

Phase 1 is expected to produce:

- A reproducible alert classification pipeline.
- A baseline classifier that achieves better precision than Perfherder heuristics.
- A ranked list of the most important metadata features.
- Initial evidence on whether alert context is critical for regression detection.

1.10 Reproducibility and Reporting

All experiments will be performed using fixed random seeds and saved configuration files. The student will provide:

- data preprocessing scripts,
- model training notebooks,
- evaluation plots and tables,
- a short technical report summarizing findings.

1.11 Risks and Mitigations

- **Class imbalance.** If regressions are rare, apply class weights or SMOTE on the training set.
- **High-cardinality categories.** Use frequency encoding or group rare suites into an "Other" bucket.
- **Label noise.** Cross-check with `alert_summary_status` for a sensitivity analysis, while avoiding leakage.

1.12 Phase 1 Deliverables

At the end of Phase 1 the student will deliver:

- A cleaned feature table derived from `alerts_data.csv`.
- Baseline regression classifiers with tuned parameters.
- A full evaluation report including ablations and generalization tests.
- A written summary of insights that motivate Phase 2 and Phase 3.