Software error grouping with machine learning

Arpad Jozsef Tamasi
Rollbar, Inc.
arpad.tamasi@rollbar.com

Abstract

Automated error classification is a cruicial service in software project management. The diversity of errors and the large amount of error attributes makes it computationally hard. This project is an attempt to reduce the dimensionality of this problem by finding and extracting the most relevant features that are specific to common error types.

1 Introduction

I work for Rollbar, an error monitoring provider. We monitor software crashes to help developers build software quickly and painlessly. We process more than 100 million events daily real time to help selecting the bug to fix next.

Software is created by the cooperation of multiple teams and runs in different environments. A single error made by a developer or an outage of an external service creates several software crashes.

A key feature of our service is error grouping. Grouping is responsible to aggregate crashes caused by the same bug. Its accuracy is crucial for the users. Narrow grouping (high variance) generates noise while broad grouping (high bias) hides important information. Error grouping has to be real time, unsupervised and identify errors that it has not seen before. It works by extracting the features from error reports that are common in the same bug's crashes. Unfortunately it is hard to tell these features.

In this project I take the stack trace of common errors and use unsupervised learning techniques to identify their relevant features. These features may generalize well since I take errors from several applications implemented by different teams. On the other hand they are specific to the common error types that helps error-specific grouping and improve accuracy.

2 Related work

Large software companies implement their own crash reporters and classifiers. I liked the sophisticated stack frame analysis but seem to ignore the messages since it's a harder problem. However, the context information that messages contain are cruicial in classification. They do not process errors from different software so those researches do not generatlize well.

Other projects focus on gathering more context information from the crash. From my experience it seems that it is important but cause overfitting. Different errors have different characteristics and it takes more work to define for which information is important for each type.

our approach is different by (1) taking samples from several software companies to find a model that generalize well, (2) focusing the error message since the usage data we see shows that - along with the tracess - it is cruicial to handle narrow and browad grouping issues.

3 Dataset and Features

The dataset consists of 100 million stack traces taken from Rollbar database, captured in September 2019. The errors were captured in applications of 11,644 different accounts, containing 11 million different messages and 495,543 exception types. 1,200 of them appears in multiple projects.

```
ActiveRecord::RecordNotFound: Couldn't find Availability with id=147603
  File "~/.rvm/gems/ruby-2.2.3/gems/activerecord-3.2.22/lib/active_record/relation/finder_methods.rb" line 344 in find_one
  ...
  File "~/.rvm/gems/ruby-2.2.3/bin/ruby_executable_hooks" line 24 in <main>
```

Figure 1: Sample stack trace

3.1 Attributes

- **Exception class name** (`ActiveRecord::RecordNotFound`) Defines the error category. Not analyzed, used as a categorical feature. Intended to identify an error type - ideally with a single message structure. In practice engineers use the same type for different errors with multiple message structures.
- **Exception message** (`Couldn't find Availability with id=147603`) Provides human-readable (but generated) context information about the error.
  Grouping cannot ignore it and often cannot use it as it is. In the example ignoring `Availability` would lead to broad grouping while considering `id=147603` to narrow grouping.

3.2 Features

- **Tokens with context**: since the messages are generated from templates, the exception type and the word positions are significant. To encode this I simple added this information to the tokens, e.g: `"ActiveRecord::RecordNotFound|1|Couldn't"`, `"ActiveRecord::RecordNotFound|2|find"` etc.
- **Message hash**: I used binary hashing term frequency on the tokens with context to represent each message as a single point in a high dimensional (parametrizable) space.

4 Methods

The project's source code is available at `https://github.com/arpadtamasi/ml-error-classification`

We want to (1) recognize the structure of messages, to use it as a categorical feature and (2) use the value in their placeholders as grouping or non-grouping features. Example: recognize the structure of `"Couldn't find Availability with id=147603"` and extract the following features:

- (`Availability`), categorical grouping feature
- (`147603`), non-grouping feature

4.1 Find common patterns using agglomerative clustering

Agglomerative clustering: bottom-up hierarchical clustering method. Each message starts as its own cluster, then we recursively merge the closest cluster pairs.

Metric: error messages are generated from templates with some placeholders to describe the context (e.g `Couldn't find '%s' with id='%d'`). We can define the **word substitution edit distance** of two different messages:

$$d_{subst}(m^{(i)}, m^{(j)}) := \text{the number of substitutions required to transform } m^{(i)} \text{ into } m^{(j)} \quad (1)$$

Given template t that contains k placeholders, the distance of message pairs generated from that template:

$$d_{subst}(m_t^{(i)}, m_t^{(j)}) = k \quad (2)$$

In the message hash space we can substitute it with the squared euclidean distance if the length of the two messages are equal.

$$d(m^{(i)}, m^{(i)}) = \begin{cases} \left\| m^{(i)} - m^{(j)} \right\|_2^2 & \text{if } \left| m^{(i)} \right| = \left| m^{(j)} \right| \\ \infty & \text{otherwise} \end{cases} \tag{3}$$

Considering the hash collisions, for message pairs generated from t template with k placeholders:

$$d(m_t^{(i)}, m_t^{(j)}) \leq 2^k \tag{4}$$

Linkage criteria: the distance between C_1 and C_2 clusters. I used maximum linkage:

$$d_{max}(C_1, C_2) = \max_{m^{(i)}, m^{(j)}} \{ d(m^{(i)}, m^{(j)}) : m^{(i)} \in C_1, m^{(j)} \in C_2 \} \tag{5}$$

4.2 Dimensionality reduction using PCA

We mapped the messages to points in a high-dimensional space. Let's see where the messages generated from the same template lie. Take a hashing function $h : text \to [1..d]$ and a word $w^{(i)}$ in this space. Since we generated binary term frequency on tokens, the word's vector is $[w_1^{(i)}, \ldots, w_d^{(i)}]$, where

$$w_j^{(i)} = \begin{cases} 1 & \text{if } h(w^{(i)}) = j \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

Now take the non-placeholder words W_t of the template t in the same space. Its vector is $[t_1^{(i)}, \ldots, t_d^{(i)}]$, where

$$t_j^{(i)} = \begin{cases} 1 & \text{if } \exists w \in W_t : h(w) = j \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

Now take a message $m_t^{(i)}$ generated from this template non-placeholder with $w^{(i)}$ word in the placeholder

$$m_{t_j}^{(i)} = \begin{cases} 1 & \text{if } \exists w \in W_t : h(w) = j \\ 1 & \text{if } h(w^{(i)}) = j \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

This means that the messages generated from a template lie in a $d - |W_t|$ dimensional affine subspace of our original space. If we project the dataset to the normal hyperplane of the template's vector, all messages will lie within a unit radius circle around the template on this hyperplane. This way we can represent these messages in a two-dimensional space while keeping the maximum variance to express the word in the placeholder.

The PCA algorithm **reduces the dimensionality** of a dataset by taking the top k eigenvectors of the $\Sigma = \frac{1}{n} \sum_{i=1}^{n} x^{(i)} x^{(i)T}$ empirical covariance matrix of the **normalized dataset**: the k **principal components**.

5 Experiments/Results/Discussion

5.1 Agglomerative clustering

I created separate batches for different exception types, enabling maximum 3 placeholders. Used the number of clusters and the number errors of as the main metric.

Since the algorithm guarantees that the length of messages in a cluster is the same (3) and they differ in limited number of words (5) I was able to recover the cluster's pattern by keeping the words that are the same in each message and replaceing the variable ones ones with a placeholder. It helped with the quantitative analysis by providing a very concise representation of the cluster.

To filter the noise I excluded the clusters that did not appear in different projects.

5.1.1 Javascript TypeError

TypeError is an exception type that covers many different errors. I used it as a first example for two reasons: (1) it has many different message structures (2) we already implemented some of its patterns manually and 80% of the users found the TypeError-specific grouping better than thee default one. I found 600 multi-project clusters and 3000 single-project ones. 3 The single-project ones overfitted to project-specific noise. The multi-project ones covered 86% of crashes, just as expected.

Some of the most frequent patterns:

- `^(\S+) is not an object \(evaluating (\S+)(\S+))\$`
- `^Cannot read property (\S+) of (\w+)$`
- `^(\S+) is not defined$$`

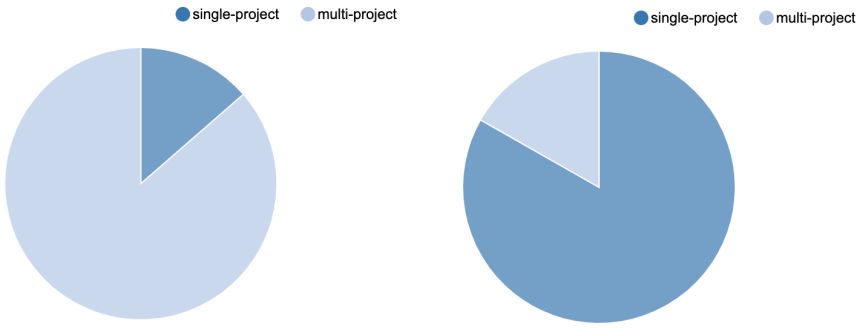

Figure 2: Single- and multi-project clusters and crashes recognized in Javascript TypeError

5.2 Dimensionality reduction using PCA

For PCA I used 2 metrics: (1) the number of prinipal components that explained more, than 1%% variance. (2) Silhouette of clusters selected by K-means in the reduced space, measured in the original high-dimensional space.

5.2.1 JavaScript TypeError

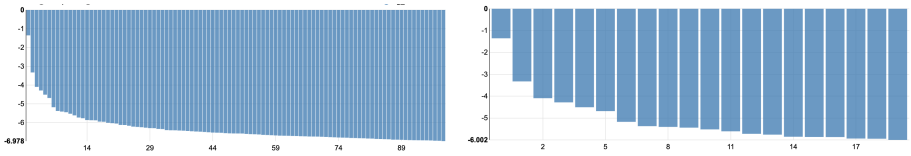

Figure 3: Explained variance of Javascript TypeError, reduced to 100 and 10 dimensions

The silhouette values were 0.56 and 0.54. For qualitative analysis I simply checked a random sample of the messages in the clusters:

- Unable to get property 'data' of undefined or null reference
- null is not an object (evaluating 'document.querySelector('#myContent').style')
- The certificate for this server is invalid. You might be connecting to a server that is pretending to be "api.xxxx.com", which could put your confidential information at risk.
- t[n].data is undefined
- Toimintoa ei voitu suorittaa. Ohjelmistosta johtuva yhteyden keskeytys
- I18n.t is not a function, null is not an object (evaluating 'e.length')

- Cannot read property 'index' of undefined
- Unable to get property 'variables' of undefined or null reference
- Cannot read property 'completeKind' of undefined)

The result was disappointing. Possible reasons:

- Hash collisions. On smaller datasets I got better results - still worse than with agglomerative clustering, but Spark cannot do PCA on really high dimensions
- Noise from outliers. As we found with the other method, the noise - outlier messages - create more clusters thus higher noise that looks too much for PCA.
- Skewed dataset. There is a huge difference between the cardinality of frequent and rare error types. The very frequent ones suppress the rare ones that look noise.

5.3 Comparison of the two methods

- **Accuracy**: the agglomerative method definitely wins in the current situation.
- **Interpretability**: the regular expressions generated from the agglomerative method let us use that context information in telling the users why we grouped an item. We can also refine grouping based on the cardinalities of those variables: ignore quantities, identifiers.
- **Implementability**: A single vector transformation is much easier than finding the right regular exception from hundreds.
- **Flexibility**: Once works on messages, PCA can seamlessly handle other attributes like stack trace frames.

5.4 Future plans

- Make a test with really high dimensions to avoid hash collisions
- Use the regular expressions identified by agglomerative clustering to augment and balance the dataset and try PCA again. On small test sets it looked very promising.
- Use the regular expressions identified by agglomerative clustering to give semantic labels to datasets and train a supervised model.

5.5 References

[1] Dang, Y., Wu, R., Zhang, H., Zhang, D., & Nobel, P. (2012, June). ReBucket: a method for clustering duplicate crash reports based on call stack similarity. In 2012 34th International Conference on Software Engineering (ICSE) (pp. 1084-1093). IEEE. Conference on Software Engineering (ICSE). IEEE, 2012.

[2] Apache Spark [Computer software]. (2019). Retrieved from `https://spark.apache.org`

[3] Apple, "Technical Note TN2123: CrashReporter," 2010, developer.apple.com/library/mac/#technotes/tn2004/tn2123.html.

[4] Moran, K., Linares-Vásquez, M., Bernal-Cárdenas, C., Vendome, C., & Poshyvanyk, D. (2016, April). Automatically discovering, reporting and reproducing android application crashes. In 2016 IEEE international conference on software testing, verification and validation (icst) (pp. 33-44). IEEE.

[5] Wu, R., Zhang, H., Cheung, S. C., & Kim, S. (2014, July). CrashLocator: locating crashing faults based on crash stacks. In Proceedings of the 2014 International Symposium on Software Testing and Analysis (pp. 204-214). ACM.

[6] Aggarwal, K., Timbers, F., Rutgers, T., Hindle, A., Stroulia, E., & Greiner, R. (2017). Detecting duplicate bug reports with software engineering domain knowledge. Journal of Software: Evolution and Process, 29(3), e1821.

[7] Seo, H., & Kim, S. (2012, September). Predicting recurring crash stacks. In 2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (pp. 180-189). IEEE.

[8] Wikipedia contributors. (2019, September 22). Hierarchical clustering. In Wikipedia, The Free Encyclopedia. Retrieved 07:04, December 14, 2019, from https://en.wikipedia.org/w/index.php?title=Hierarchical_clustering&oldid=917135536