Classifying applications based on API consumption. CS229, Autumn 2012

Arne Roomann-Kurrik

December 14, 2012

Is it possible to classify applications based off of records of their calls to an API?

Providers of public APIs take on the burden of supporting the ecosystem of applications which depend upon the API to function. Insight into the types of applications issuing queries into the system is valuable to the maintainers of the platform.

This paper is concerned with two use cases addressed by classifying applications into categories.

The first addresses identifying applications which are considered to be abusive users of the platform. This can be modeled through a binary “abusive” and “not abusive” classification.

The second is categorization of applications into distinct groups for the purposes of identifying application verticals. Such verticals of similar applications can be used to tune rate limiting, plan new features, and gain insight into the ways the platform is being used which may not be totally intuitive from an initial inspection.

Examining this problem required collecting and identifying characteristics of application usage logs, experimenting with ways to clean the dataset, and training classifiers to determine the potential success rates of applying machine learning toward application classification.

1. Data collection

Twitter’s API supports an ecosystem of millions of applications. This volume complicates even simple analysis. Such analysis may benefit from machine learning to expose inherent categories of application and highlight candidates which should be reviewed by policy teams.

As an employee of Twitter, I was able to run distributed queries against the API logs using a Hadoop cluster. The scale of the raw log data required a 30 minute job to pull records for a single hour. The final training dataset contained 124,916 applications which made requests to `api.twitter.com` during a period of October 8th, 2012. The final test dataset contained 89,166 applications from a period of October 15th, 2012- a week later.

A log line contains a request to a single path (e.g `/1.1/statuses/update`) on the server. Some normalization is performed (e.g. `/1/statuses/destroy/:id/`) but the set still contained 2,232 unique paths.

To assign a score to the entries in the test and train dataset, I used up-to-date lists of suspended applications. There were 147 applications which had made calls in October which had been suspended since.

The process which collected the log lines output a matrix where the columns corresponded to individual endpoint paths, and the rows represented applications. The $M_{i,j}$th element of the matrix represented the number of times application i made calls to endpoint j.

2. Properties of the dataset

A suspended application is the result of a manual review, so the list of suspended applications is a strong dataset for training a classifier to look for abusive applications. However, not being suspended is not an indicator that an application is non-abusive, just that it hasn't been reviewed and flagged. So while there are no false positives, there are potentially many false negatives.

Reviewing an application requires a non-trivial amount of manual work. It was not very feasible to build a large, accurate dataset by hand. I realized that a successful classifier would identify as many suspended applications as possible while keeping the false positive rate low enough as to be realistic to review by hand.

The ratio of suspended to non-suspended applications is very unbalanced. It is possible to achieve a 99.882% success rate simply by writing a classifier which says that no applications are abusive.

3. Clustering applications

My intuition was that an application of k-means clustering would separate applications into logical groupings which could be individually analyzed. However, running the k-means algorithm to convergence did not result in even groupings of applications - one cluster always gained the vast majority of applications, as shown in this grouping:

k	Size	k	Size	k	Size
0	31	7	9	14	4
1	2	8	15	15	6
2	38	9	4	16	9
3	256	10	2	17	124380
4	3	11	4	18	19
5	30	12	5	19	76
6	12	13	11		

Table 1: Cluster size for 20 k-means groups

There were some patterns in the groupings. For example, all of Twitter's official clients were clustered into various smaller groups. Many popular mobile clients also fell into the smaller clusters. Mobile clients typically display the same data and differentiate on presentation, so this seems intuitve.

Cluster 3 also showed some patterns. As the second largest cluster, it contained sequences of applications with almost-contiguous IDs and very similar naming patterns:

```
XXXXX3869 XXXXXXXXXXAPP
XXXXX3876 XXXXXXXXXXAPP
XXXXX3898 XXXXXXXXXX's App
XXXXX3913 XXXXXXXXXX's tweetApp
XXXXX3964 XXXXXXXXXX's tweetApplication
XXXXX3970 XXXXXXXXXX's testApp
XXXXX3990 XXXXXXXXXX'sSP app tweet awsome
XXXXX4007 XXXXXXXXXX's app
XXXXX4017 XXXXXXXXXX's app test
XXXXX4023 XXXXXXXXXX's app
XXXXX4028 XXXXXXXXXX'sapplication
XXXXX4050 XXXXXXXXXX's app
XXXXX4055 XXXXXXXXXX application
XXXXX4058 XXXXXXXXXX's app
XXXXX4099 XXXXXXXXXX's app
XXXXX4112 XXXXXXXXXX's application
XXXXX4117 XXXXXXXXXX's app
XXXXX4123 XXXXXXXXXX's app
XXXXX4145 XXXXXXXXXX's app
XXXXX4147 XXXXXXXXXX'sapp
XXXXX4241 XXXXXXXXXX's APP
```

Even some relatively distant apps (by ID) were obviously similarly registered:

```
XXX599713 FOO App 23 P
XXX599734 FOO App 23 Q
XXX600451 FOO App 23 B
XXX600463 FOO App 23 A
XXX731254 FOO App 33 T45
XXX754732 FOO App 33 N63
```

This would seem to indicate that these apps were registered and operated by some sort of automated system, and warrant investigation. That the clusterings were created by analyzing usage without regard to ID or application name is promising. The sheer size of cluster 17 means a different categorization approach would be needed for the majority of applications, however.

4. Logistic regression

Most of the work for this analysis was spent trying to classify abusive applications. To get an intuition about where to spend my time, I wrote a 'quick and dirty' logistic regression classifier using the following stochastic gradient ascent rule:

$$\theta_j := \theta_j + \alpha \left(y^{(i)} - h_\theta(x^{(i)}) \right) x_j^{(i)}$$

Where $h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$, the sigmoid function. Graphing the performance of this classifier using increasingly large subsets of the training set showed a slow convergence between the error rates for the test and training sets:

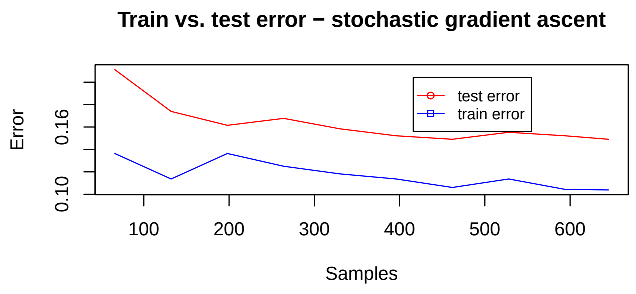

Figure 1: SGA performance

This was an indicator of high variance. General strategies to address this are to increase the data set size and to reduce the number of features used in the model. I pulled more data for my datasets, and planned strategies for reducing the dimensionality of the data.

5. Principal component analysis

When collecting the dataset, I really had no idea whether the suspended applications would be located in proximity to each other, or if they would be spread out throughout the dataset. I wanted a way to determine whether a classifier was even likely to find a separation between the two categories of application. Principal component analysis seemed like a good way to cast the data into two dimensional space, which could be used for a visualization.

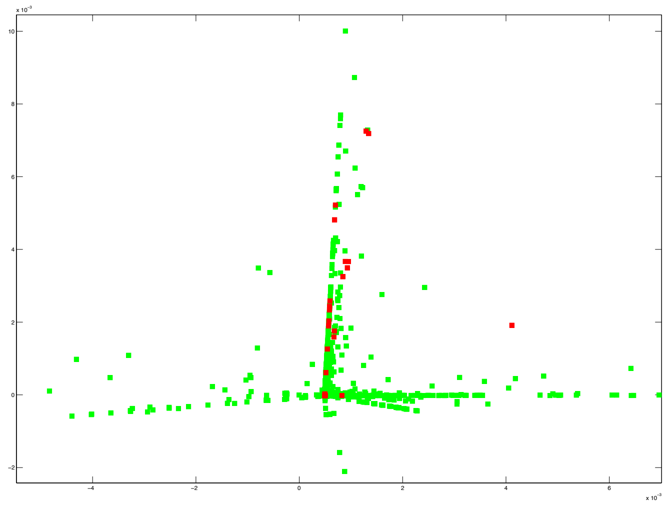

Figure 2: 2D plot of applications

By normalizing the data to zero out the mean and variance, I was able to find the corresponding eigenvectors and generate Image 2.

By plotting suspended applications in red, it seems that types of application fall into specific linear combinations of the two principal components. Applications which have been marked as abusive tend to vary along the second principal component (y-axis in the above graph) with little variation along the first principal component (x-axis).

6. Endpoint collection

A look at the features in the dataset made it obvious that many were redundant. During the time that the data was pulled there were two available versions of the API, and calls were distributed across each. A simple way to reduce features seemed to be to collect the endpoints which accomplished the same task (e.g. posting a Tweet) with different semantics (e.g. `update` vs. `update_with_media`) and combine the numbers of calls into a single number.

Endpoint	Mapping
/1/statuses/destroy/	statuses_destroy
/1/statuses/destroy/:id/	statuses_destroy
/1.1/statuses/destroy/:id/	statuses_destroy
/1.1/statuses/update/	statuses_write
/1/statuses/update/	statuses_write
⋮	⋮

Table 2: Logically grouping endpoints

This was best accomplished via a manual process. Reviewing 2000 endpoints by hand seems like a lot of work, but many contained IDs which had not been collected properly by the logs processor, many contained mistakes or typos made by the application, and some were otherwise restricted or internal endpoints made by official clients. This process reduced the number of features to 61.

7. Feature selection and forward search

Another approach for reducing the number of features was to identify which endpoints contributed most toward reducing the error of a classifier. To implement forward search, I generated one dataset per feature (at this point having culled out typos, unparsed IDs, and restricted endpoints) and saw which endpoint produced the best logistic regression score. Keeping that endpoint, I created datasets with 2 features, and ran the classifier again.

i	Endpoint	Error
1	/1/friendships/create/	0.088608
2	/1/followers/ids/	0.075949
3	/1/statuses/friends_timeline/	0.072333
4	/1/statuses/home_timeline/	0.065099
5	/1/users/show/	0.059675
6	/1/account/totals/	0.057866
7	/1/statuses/friends/	0.056058
8	/1/statuses/update/	0.054250
9	/1/account/rate_limit_status/	0.052441
10	/1/statuses/update_with_media/	0.050633

Table 3: Forward search for API features

The results, listed in Table 3 indicate the 10 endpoints which contributed the most to classifier accuracy. Intuitively, `/1/friendships/create/` is used for follower spam and appears to be the best single-endpoint feature for classifying abusive apps. The `/1/followers/ids` endpoint would also be useful for identifying possible targets for spammy follow requests.

8. Scoring

As mentioned earlier, it would be possible to write a classifier with 99.882% accuracy simply by asserting that every app is not abusive. My experiments with logistic regression were yielding 90% accuracy, so I investigated different scoring mechanisms which may give better insight into how well a given classifier was doing.

Precision and recall are useful for the unbalanced data set case:

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

I desired a single value metric, though, so looked into F_1 score:

$$F_1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

When investigating F_1, I also came across Matthew's correlation coefficient (C), which was supposed to be useful for ranking the performance of binary classifiers even in cases where classes were of very different sizes.

$$\text{C} = \frac{\text{TP} \times \text{TN} - \text{FP} \times \text{FN}}{\sqrt{(\text{TP}+\text{FP})(\text{TP}+\text{FN})(\text{TN}+\text{FP})(\text{TN}+\text{FN})}}$$

In practice, I found Matthew's correlation coefficient to be most in line with my intuitions about well-performing classifiers, in that classifiers which appeared to be doing a good job had higher coefficients than ones which appeared to be doing poorly.

9. Support vector machines

Using LIBSVM, classifiers were trained against each type of feature reduction.

I paid special attention to the number of false positives each classifier returned. By adjusting weights given to each category, the SVM-generated model could be tweaked to return more or fewer false positives as needed. It was important to develop a model which identified as many abusive applications as possible, while keeping the false positive rate below a quantity which would be appropriate for manual review. I estimated that a review might take 10 minutes for an experienced reviewer with appropriate tools. 500 reviews would cost 83 reviewer-hours, so over two weeks of work for a single person. I rejected models which returned significantly more false positives than this.

I was also sometimes able to tune the amount of returned results by changing the threshold of the SVM score past which a positive score would be awarded. Surprisingly the only model which really differed much here was the endpoint collection method:

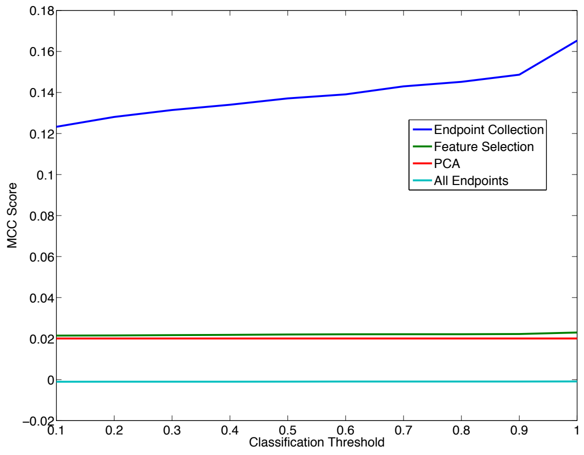

Figure 3: Threshold vs MCC score for SVM

Ultimately the endpoint collection method produced the best results out of all of the models. The result was a SVM classifier which identified 34% of the suspended applications in the test set of 89,166 applications, with 576 false positives.

Model	FP	C
Endpoint collection	576	0.165301030367
Feature selection	370	0.0229517324541
2d PCA	220	0.0200499603763
All endpoints	42	-0.000882154435157

Table 4: Comparison of models for SVM

Threshold	TP	TN	FP	FN
0.1	53	87888	1131	94
0.2	53	87970	1049	94
0.3	53	88023	996	94
⋮				
0.9	52	88272	747	95
1.0	51	88443	576	96

Table 5: Endpoint collection counts

Threshold	P	R	F1	MCC
0.1	0.044	0.000602	0.001189	0.123
0.2	0.048	0.000602	0.001189	0.128
0.3	0.050	0.000601	0.001189	0.131
⋮				
0.9	0.065	0.000588	0.001166	0.148
1.0	0.081	0.000576	0.001144	0.165

Table 6: Endpoint collection scores

10. Conclusion

Machine learning appears to have merit for classifying applications, particularly in the context of surfacing candidates for eventual human review.

While not perfect, the current SVM results represent a pool of applications realistically sized to be able to review by hand.

There are a few steps which could be used to improve classifier accuracy while reducing the number of false positives returned. More complicated kernels may be able to better categorize the data, and a system which took the output of human reviews (in particular applications which were marked as candidates and not suspended) in order to retrain its model would have a cleaner dataset to work with.