AimBrain Project

Max Chamberlin

September 25, 2016

Solution Overview

For each user, I trained a binary classifier that outputs 1 if a sequence of key presses belongs to that user and 0 otherwise.

To train this classifier, we first re-label the rows of the training data appropriately (1 if the sequence of key presses that comprise a row belongs to the classifier's 'owner'; otherwise 0) before running any particular inference algorithm.

Authentication requires the running of the classifier corresponding to a particular user. We will now discuss the following aspects of our authentication system.

- Pre-Processing
- Classifiers and Cross-Validation
- Scalability
- Unseen Attacks

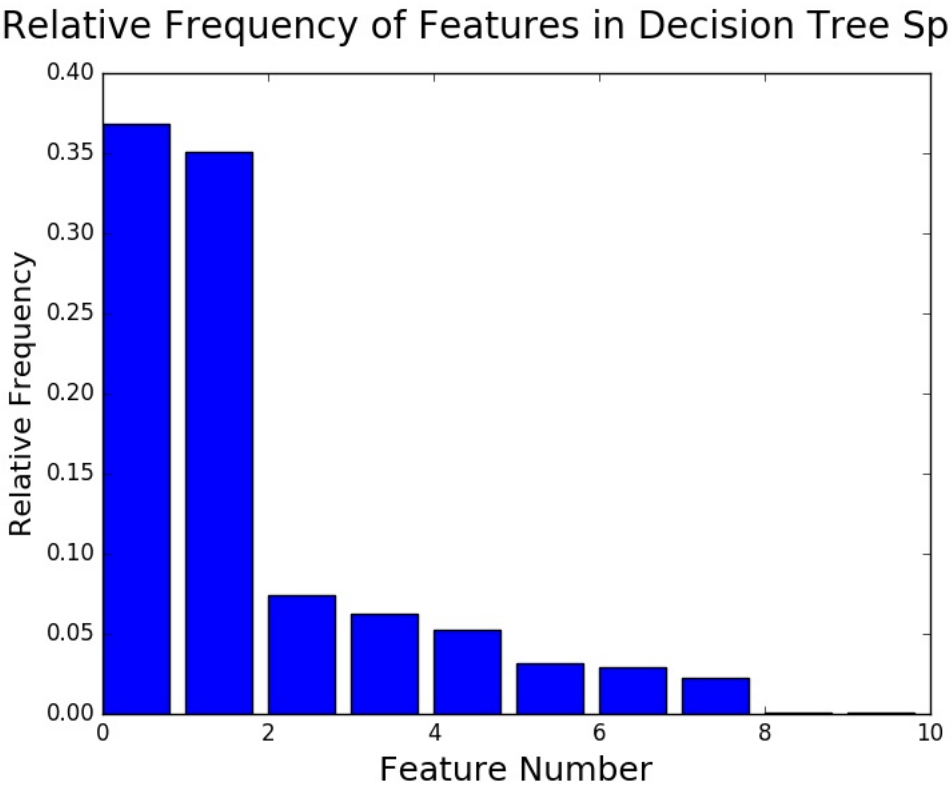
Preprocessing

The following bullet points are a description of some of the pre-processing steps I undertook.

- Removed redundant features.
  - Features like **screen size** are the same for all users, and thus contain no information that can be used to discriminate one user from another.
- Increased the size of the training data (Few additional benefits).
  - The data set only has 200 training points. This is small, and often machine learning algorithms benefit from training on larger data sets. However, artificially increasing the size of the data-set can be a difficult procedure.

- I explored the following methods: duplicating and adding random noise to the data-set; duplicating and re-weighting different touch events; duplication and permuting the touch events, e.g. so that features for 'touch1' and 'touch7' are swapped.
- In some instances, permuting the touch events helped the classifier reduce the error rate by about half on a validation set (log-likelihood dropped from 0.015 to around 0.008). This is a method that may be explored but not something that was used in the final evaluation system. Improvements were not seen from the other methods.

- Feature Engineering.
  - My exploration began by including aggregate features, like the average 'pressure' and the variance of the pressure for the same user across the 7 touch events.
  - The next step was to determine the importance of each feature. This was done by tallying the number of splits in a random-forest classifier that was trained on the data (See figure below for an example). I found that the variance of the pressure and the average size of a touch event were the two most commonly used features, but it varied per user. As an illustrative example, I present a graph and table for he feature importances of a particular user.



	Feature	Frequency		Feature	Frequency
1	avg(press)*var(size)	0.377	6	avg(size)	0.05
2	avg(press)*var(up_size)	0.349	7	avg(up_size)	0.03
3	avg(press)*avg(size)	0.075	8	avg(down_size)	0.02
4	avg(press)*var(down_size)	0.06	9	touch_5(up_time)	0.0007
5	avg(down_press)	0.05	10	touch_3(down_time)	0.0007

Figure 1: All displayed figures relate to the classifier for user 1

- The third step was to combine the most important features together. This was done with a simple multiplication of 'size' and 'pressure' (which ultimately gives us a measure of force F=PA), and after training a new decision tree classifier I discovered that this new feature was the most important to the classifier (as it appeared in most splits).
  - I then tested various other engineered features like 'area' and 'energy' which may have been useful for the classifier. Ultimately, the approach taken was to include as many useful features as possible, and to let the classifier for each user decide what was most relevant. However, choosing from a large number of features runs the risk of over-fitting, but would be mitigated by training on larger data-sets. Using a random forest classifier, we did not see a large degree of over-fitting when using many features.
  - I also changed the way in which touch times were recorded to ensure robustness to user mistakes on the key-pad. I will assume that when a user makes a

mistake, it will be possible for them to go back and correct the mistake (by pressing delete for instance). This means we ought to have the 7 touch events as before. In this case, we can expect the corrected touches to be just like other touch events, except for the fact that there will likely be a time delay since the corrected touch event would have followed deleted touch events. To account for this, one might only take the relative time between touches on a keypad, and drop the absolute time.

– It was remarked that in many respects we can expect the corrected touch events to be just like other touch events. However, if a 'corrected' key press has a different pressure/ features to a 'non-corrected' key press (e.g. if the user feels more cautious and types slower), then to some extent this will be mitigated by the fact that most of the features used were aggregates (averages or variances) across the different touch events, and so they should be fairly robust to individual mistakes. In addition to the classifier that performed best on the test-data (which used some non-aggregate features), I also built a classifier that used only aggregate features and saw almost as strong a performance on the test data: (98% accuracy vs 96%).

Classifiers

The classifiers that were considered for this problem were random forests, extremely random forests and gradient boosted decision trees (xgboost). The size of the training data mean that using neural networks would be a poor choice of classifier. The dimensionality of the data discourages us from using kernel based methods, although a fruitful avenue of future exploration would be to train a support-vector machine on the 4 or 5 most important features from the pre-processing stage. It was found that approximately 5 features accounted for 85% of splits in the random forest classifier for the first feature; and 8 features accounted for 99% of splits.

Random forests and Extremely Random Forests:

Random forests have very few parameters to tune (mainly the number of trees and the depth). To tune these parameter it's possible to use the 'out of bag' accuracy as a cross-validation framework. However, I found that parameter tuning had very little effect on the classifier's performance. To err on the side of caution, I used models with a fairly large number of trees (1000 or so).

My final model was an ensemble of two random forest with very different parameter settings that also made use of different features. These two models both performed fairly well individually, and through model averaging I hoped to reduce error arising from the variance of the models.

The other parameter that was fairly important was the penalty associated with the class weight. Because there is quite a heavy class imbalance between positive and negative labels, it is necessary to increase the weight of the penalty associated with mistakes from positive labels.

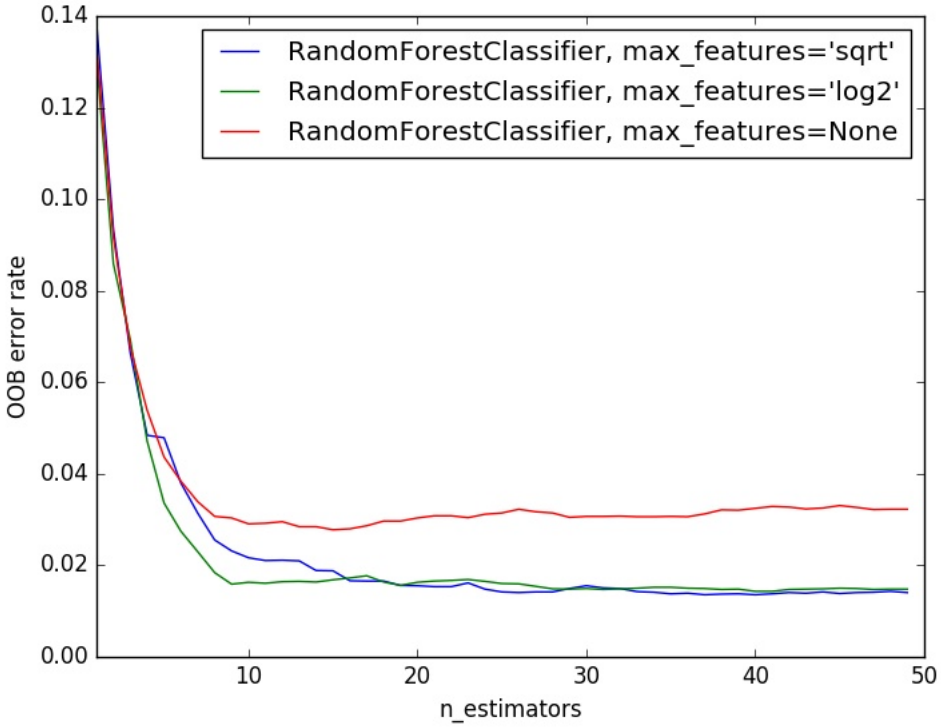


Figure 2:

XGBoost:

```
space = {
                'n_estimators' : hp.quniform('n_estimators', 50, 1000, 1),
                'eta' : hp.quniform('eta', 0.2, 0.9, 0.025),
                'max_depth' :  hp.choice('max_depth', [3,4,5]),
                'min_child_weight' : hp.quniform('min_child_weight', 1, 6, 1),
                'subsample' : hp.quniform('subsample', 0.75, 1, 0.05),
                'gamma' : hp.quniform('gamma', 0.1, 1, 0.05),
                'lambda' : hp.quniform('lambda', 1, 10, 0.1),
                'colsample_bytree' : hp.quniform('colsample_bytree', 0.1, 1, 0.0
                #'num_class' : 1,
                'class_weight':{1:hp.quniform('class1_weight',2.5, 6,0.5),0:1},
                'eval_metric': 'mlogloss',#'mlogloss',
                'objective': 'binary:logistic',
                'nthread' : -1,
                'silent' : 1
                }
```

Gradient boosted decision trees have many more parameters. To tune the parameters, I used a python library called hyperopt which uses Bayesian optimisation (kernel density

estimation) to efficiently search for new parameter settings.

Within the cross-validation framework devised, I used stratified sampling to ensure that the CV-folds did not have too few positive examples, owing to the strong class imbalance with only 20% being positive examples.

However, even with all of this machinery, the boosted decision trees performed worse than random-forests. This is most likely because with such small data-sets it's harder to find a good parameter setting that doesn't over-fit.

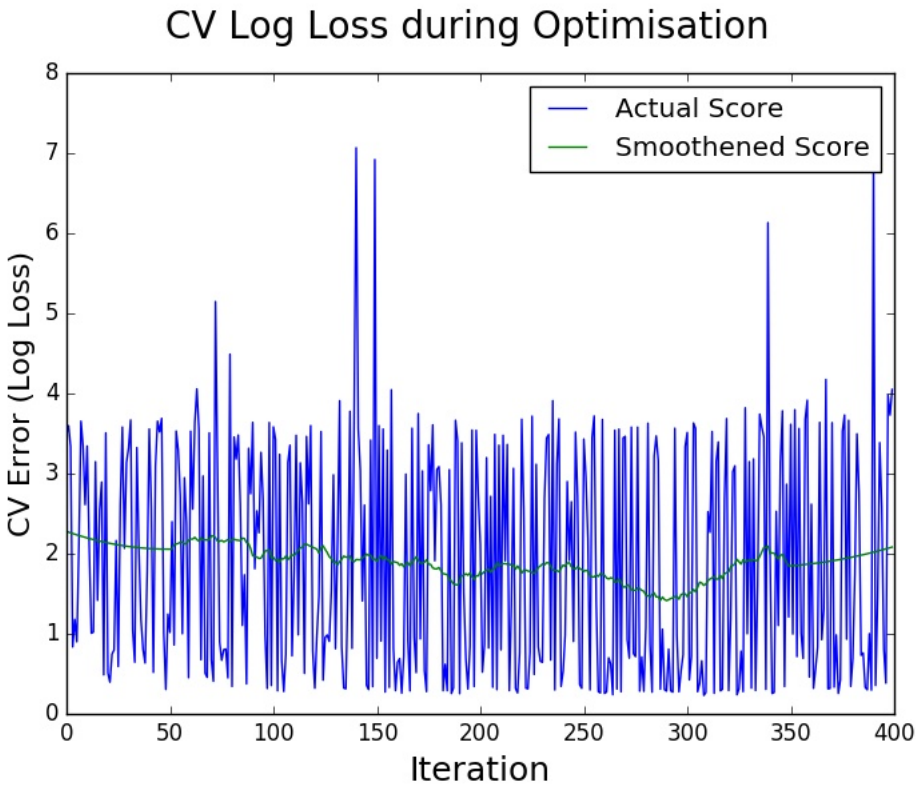


Figure 3: Hyper-parameter optimisation using gradient boosted decision trees

As we can see from the graph above, the Bayesian optimisation framework performs a little better than random sampling.

How the system could scale to millions of users

To scale the algorithm to millions of users presents no problems. Each user has their own classifier, which individually has a very modest memory requirement. And as was mentioned at the start of this report, authentication only requires one classifier to be run, which ensures a short compute time. The compute time is O(1) in the number of users.

To train a single classifier, we need not consider data from millions of users since it would be perfectly possible to train each binary classifier on a subset of the data. However, it would probably be preferable to train on the data for millions of users,

since more data-points would likely reduce generalisation error and would be feasible with the random forest classifier.

The one challenge that would occur: we would have many more negative training instances (hacker rows) than positive instances (genuine user authentications). To remedy this, we would be that we would need to ensure that the weights for positive class labels are given even greater weight. As an example, the weight of one class label could be scaled by the inverse of the frequency of the other.

How it deals with previously unseen attackers

The nature of the training data, where we see many more negative cases than positive (perhaps with a ratio of 1:1000) means that a random attacker is much more likely to fall into a negative case than a positive case. What's more, there is more in common to the positive cases (which are from the same user) than the negative cases (which are from different users). As such, the negative cases will have a much larger variance and this will be incorporated into the trained classifier's predictions, further increasing the likelihood that a new attacker is not authenticated as a particular user.

Ultimately, by casting this problem as a binary classification problem with a large class imbalance, each classifier is learning more about what differentiates the user from some randomly selected individual.