BotOrNot:
Finding Fraud Accounts using Patterns

Rece Coffin
David Henningsen
Christian McErlean
Ian Tobiason

June 20, 2017

Introduction

In the modern age most people have a significant online presence through email and social media, each of which have their own system of managing friends or contacts. FullContact takes on the challenge of gathering these contacts from all other locations and placing them into a single, more robust contact. They do this by providing an online platform for users to sign up for that allows them to import contacts and information from all over to build an easy to manage contact list. This platform also gives users tools to further populate their contacts such as the ability to scan in and transcribe business cards or parse information from email signatures. In addition to the tools build by FullContact, an API is provided for developers to use this platform to build their own tools. FullContact's goal is to give people the tools they need to effectively manage their contacts and interact with other people.

FullContact's online platform is publicly available with some features even being provided for free. One of the companies ideologies is to make their service easily accessible to new users, this coupled with the free features leaves the platform open to signups by fraudulent bot accounts. These bot accounts have a negative impact on the system as they increase cost and skew the data collected by the company. The purpose of this project is to develop an application to flag fraudulent accounts based on information given when the account is created. Additionally, the application uses a machine learning model in order to learn from past fraud events and detect an increasing number of fraud accounts the longer it is in place.

Requirements

Functional Requirements:

The Bot or Not detection tool utilizes machine learning algorithms to detect bot accounts on the Full Contact server and create an interface to allow a human to manually review flagged accounts and determine whether the account is fraudulent or not.

Specific Requirements:

1. Train an automated system to detect bot accounts across the FullContact database.
2. Train an automated system to detect bot accounts shortly after sign up.
3. Create a graphic interface to manually review a flagged account.

Non-Functional Requirements:

1. Code style is consistent with FullContact style guides.
2. Application is written in Java using Java machine learning libraries.
3. Application only uses data acquired from the time the account is created.
4. Application is flexible enough to be applied to other databases should we extend this project as a SaaS.

Risks:

Technology Risks

- Trouble receiving live data from the company's database
- Learning algorithm gives too many false negatives (Not detecting bots)
- Training the classifier models takes too long, limiting debugging

Skills Risks

- Poor choice in machine learning model, resulting in a model that does not improve as more data comes in
- Training the model takes too long, making the application hard to use.

Definition of Done:

The finished product is a fully integrated bot detection software that identifies high risk accounts at creation. The software then displays, through some sort of graphic interface, the accounts in question with the data used to classify the account as a likely bot in order for a human to make the final decision. The software is integrated with the existing FullContact server and be run on daily basis to analyze account provisions for that day.

System Architecture

Figure 1 below outlines the architecture of our application. The account information is taken from the FullContact database and exported to a text file in a format readable by the clustering library. After being clustered, each cluster in the training data set is analyzed and given one of three rankings: 2 for clusters that contain 100% human accounts, 1 for clusters that contain more than 70% fraudulent accounts, and 0 for clusters that contain less than 70% bots and not 100% humans.

These clusters are then sent into the featurizer which analyzes each account within the cluster, once it analyzes each account it scores the entire cluster and saves a number of cluster specific features. After the features are determined and stored, each cluster is passed into the machine learning model with the specific features as well as the cluster rank to be used for training. A small subset of the data collected is withheld in order to be passed into the model for evaluation. The evaluation is done by giving the model a small set of already scored data and comparing the output from the model to the ideal score given to that cluster manually. This is how we are able to test a variety of feature sets and determine what information is going to provide us with the most accurate predictions.

Once the model has been trained on known data, we can pass new data to be scored. The clusters from the new data that are identified as containing a large amount of bots are sent to a user interface in order to be manually reviewed and approved or rejected. From there, the accounts that were manually reviewed are given a score and appended to the existing training data to be used for future evaluations. This allows for the model to continually learn from new fraud events, in hopes of preventing similar patterns from occurring in the future.

Figure 1

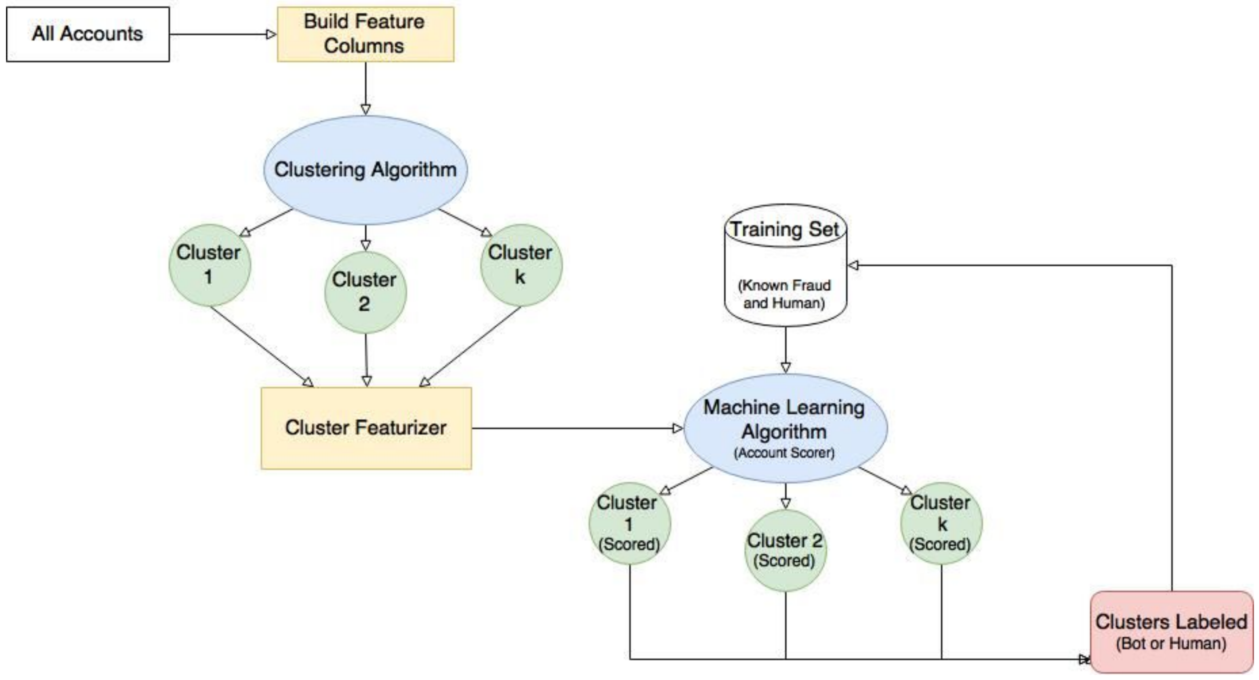
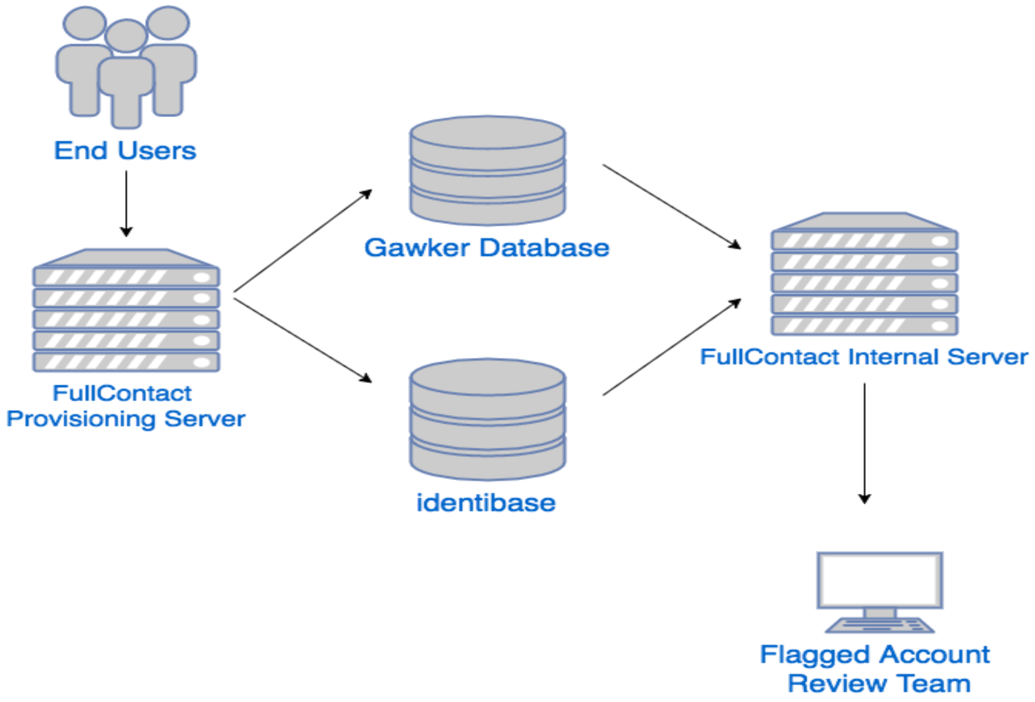


Figure 2 on the next page outlines the datapath implemented by the application. End users can create accounts using a portal on FullContact's website, the account's information is stored in the Identibase database and the account creation event itself is stored in the Gawker database. Our software is implemented on an internal server that makes calls to these databases. The initial call to the Gawker Database gives us the account number, time of account provision, and other information collected when the account is created. Using the account number, the application is able to cross reference Identibase, which is the FullContact data warehouse, and collect the email address, password commonality, and I.P. address associated with that account number.

This information is combined and condensed into a single file stored on the server. The file contains all of the information to be used to cluster the accounts. The file is read and processed by the clustering application. After our application runs we are left with a list of accounts that have a high chance of being bots and these are saved and sent to a team to manually verify if they are fraudulent accounts or not.

Figure 2

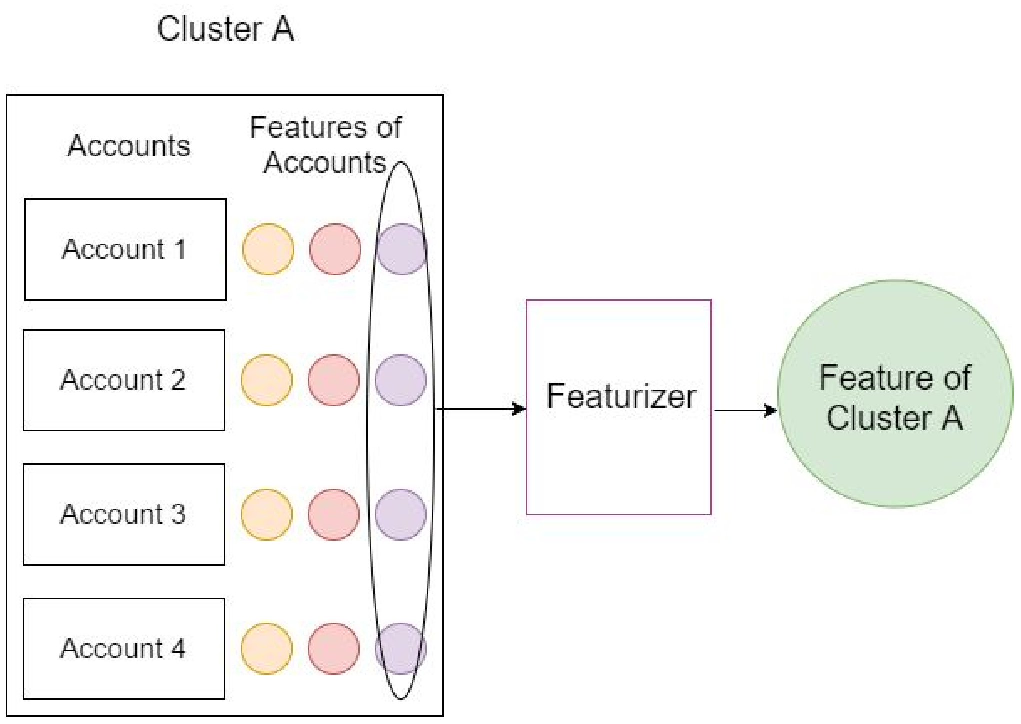

End Users
Gawker Database
FullContact Provisioning Server
identibase
FullContact Internal Server
Flagged Account Review Team

Technical Design

When first approaching the problem we were given a large set of data with the task of creating a machine learning model to classify new account signups as regular or fraudulent bot accounts. We had access to information about these accounts such as the time they signed up or the email address they used. The problem is that when looking at these fraudulent accounts in a vacuum it is hard to tell if they are bots or not. To actually tell with the information available at signup we had to compare them to accounts made around the same time. Previously an exploratory attempt was made by our client to feed different features on an account by account basis that was meet with mixed results. To get features that we knew would be more adept at determining whether accounts being made were part of fraudulent events we knew that we would have to find a way to incorporate patterns between signups into our model.

We decided the best approach was to first cluster these accounts together based upon information collected at the time of sign up. This technique allowed us to analyze accounts with similar attributes and determine patterns that could distinguish accounts as fraudulent. Once these accounts were grouped we lost most of our ability to use their individual features in the next step of our model, classifying clusters as humans or bots. Losing this presented a challenge and an opportunity as we were forced to have features that were unique to clusters instead of individual accounts. This meant we were able to look for patterns within the clusters themselves and give these patterns numeric scores based on their presence and how prevalent they are. The process is shown in **Figure 3** were the featurizer takes in data from every account in the cluster and outputs a single feature for the cluster. The thought behind this is that if the accounts have similar signup activity but then also show patterns that would be unlikely in for humans who act differently then it would be a grouping of accounts made by one person most likely using a script.

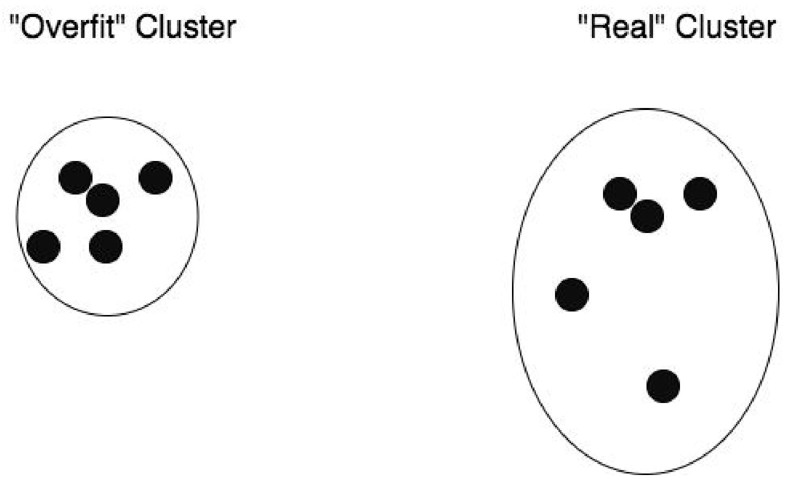
Figure 3



When designing our model and overall system we had to keep in mind how to make it run smoothly and regularly in a production environment. Clustering requires us to look at sums of events at a time so we realistically couldn't run each new account through our model at the provisioning stage. To properly handle this we decided to go with a daily run where we take all of the new sign ups for the day and cluster them to run through our model for classification. FullContact usually sees around a thousand signups on a daily basis so we designed the way we handle the data with this in mind. We cluster with the goal of having around 10 new accounts per cluster meaning with the actual number of clusters being based on the number of accounts created that day.

The problem of clustering these accounts by the day means that they are going to not be as accurate as clusters made by clustering bigger datasets. Using the larger dataset results in very tight clusters represented by by the "overfit cluster" in **Figure 4.** Looking at a day's worth of data resulted in the “real cluster”. When clustering hundreds of thousands of events at once, we received much more accurate specific clusters. The problem was, we found that clusters from only 1000 events were less accurate and harder to score based on the training data that was clustered in larger numbers. We combated this problem by breaking apart the training data into sets of around 1000-1500 accounts, and clustering only these small subsets at a time. The thought process behind this was that it would lead to clusters similar to the one we would get from a daily run, that way the data we used to train the model would look much more similar to the data we hope to classify daily. An unforeseen benefit of this change was that our application now runs considerably faster. The clustering of the data in small groups made that portion of the application run in just a fraction of the previous time, as well as improved the accuracy of our machine learning algorithm.

Figure 4



Real clusters may be less accurate than the overfit, but they more closely model the data our application will receive when it is run on a daily basis.

<u>Decisions</u>

Language - We are using Java for our solution because the existing code already in place was also written in Java. We also feel that as a team, we are most comfortable using Java. There are also a great number of machine learning libraries available in Java for use to choose from.

Weka Library for clustering - We decided to use the Weka library for clustering after experimenting with other libraries such as Encog, Deeplearning4j, and Java-ML. Of all these libraries Weka was the easiest to use and understand. It also gave us the best results of any of the libraries we used. Using the same features with the Encog and Java-ML clustering algorithms we get significantly lower percentages of good quality clusters. These results, and the ease of use, were what drove us to choose the Weka library for our clustering algorithm.

Encog Library for machine learning - We decided to use the Encog library for machine learning because the previous code already had a model using Encog in place. Our hopes were that we could apply our clustering model on top of the existing machine learning without much modification. We ended up writing our own machine learning model, but it was very similar to what was already in place. After much time spent deriving a set of features to pass into the model, we started getting very good accuracy results. Because of our good results and the limited time we had for this whole project, we did not explore many other machine learning libraries.

Machine learning model - The machine learning model we are using currently is a Support Vector Machine (SVM) model. The SVM model is a supervised learning machine learning model, this means that we provide the model with already classified data in order to train before providing it with new data to score. Using a supervised model like this allows us to continually change and update the data we are training with. This gives us the ability to update it in the event we discover a fraud event that was not previously detected, by constantly updating the training data we can prevent attackers from using the same approach for multiple fraud events.

Results

The main goal for the project was to build an automated system that detects fraudulent accounts both across the entire database and in real time as accounts are created, then to have those accounts available for review in a user interface. This goal is accomplished by clustering account events based upon the time the account was provisioned, its IP address, its email prefix rate, and its email domain rate. Then email domain frequency, email domain popularity, email pattern frequency, average rate of provisioned accounts for the previous hour, the average length of the domain and prefix of the accounts within the cluster, and the average rate a specific prefix appears within a cluster are extracted as cluster specific features. A machine learning model is trained with these cluster features and a known “cluster specification” (all genuine, all/mostly fraudulent accounts, or unknown mix). The model will then be able to, based upon the training data, give unknown clusters a specification. The logic behind the feature sets is that if one clusters the data using a feature set that forms mostly homogenous clusters, then one can design another feature set, for the clusters, that will be used for the machine learning algorithm to detect the cluster specification. The account events in each cluster specified as all/mostly fraudulent accounts or as an unknown mix can then be viewed and manually labeled in a user interface.

Depending on the data it was found that around 85% of clusters were either 100% genuine accounts or >= 70% fraudulent accounts. Using some validation data sets we were able to determine that around 99% of genuine clusters and around 96% of all/mostly fraudulent clusters are labeled correctly. Overall about 90% of fraudulent accounts are predicted to be caught by this program.

Future work can always include updating, adding, and changing the two feature sets used for the clustering and machine learning, better feature sets is the main avenue to improve the program. As clustering is working fairly well in the current version, future improvements would most likely occur by designing new cluster specific features to better train the machine learning model with. Had there been more time to develop this project, we could have derived better feature lists and possible achieved much higher accuracy, in both the clustering and machine learning.

It was learned throughout this project that communication and research is key before starting anything, or time will likely be used inefficiently or wasted entirely. It was also learned that the outcome of a machine learning program relies heavily on the quality and quantity of data available. If too many features are used to cluster it will likely result in overfitting, where the machine learning algorithm will be unable to properly specify a cluster because it looks too unique. Overall we learned a great deal about machine learning and how dependent it is on the data you are using.