IDENTIFYING POPULARITY MANIPULATION OF LIVESTREAMS ON TWITCH.TV

by

COLLIN ANDREW WATTS

(Under the Direction of WALTER D. POTTER)

ABSTRACT

Livestreaming has grown from a theoretical concept to a multibillion dollar industry in the span of less than 10 years. This industry couples entertainers with audiences, and derives its revenue primarily from subscription and advertisement. As with many such fast growth industries, the growth of livestreaming has left it vulnerable to exploitation and manipulation. Using Deep Ensemble Recurrent Artificial Neural Networks, as well as a variety of other techniques, we attempt to identify this manipulation such that it might be properly protected against.

INDEX WORDS: Livestreaming, Viewbots, Artificial Neural Networks, Fraud Detection, Twitch.TV, Deep Learning, Botnets

IDENTIFYING POPULARITY MANIPULATION OF LIVESTREAMS ON TWITCH.TV

by

COLLIN ANDREW WATTS

B.S., The University of Georgia, 2016

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment

of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2016

IDENTIFYING POPULARITY MANIPULATION OF LIVESTREAMS ON TWITCH.TV

by

COLLIN ANDREW WATTS

Major Professor:	Walter Potter
Committee:	Khaled Rasheed
	Shannon Quinn

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Overview

This introduction will attempt to provide an overview of all of the relevant techniques used in this paper, as well as the domain knowledge required to understand the application of these techniques. They include the following: Artificial Neural Networks, Genetic Algorithms, and the Real-time Web Technologies used to support Livestreaming. Furthermore, we will provide the historical and present context of livestreaming as both a technology and an industry, in order to properly frame the necessity of this work. This background is disjoint from the following papers, and are thus not necessary to understanding the technology presented, however it is strongly recommended that this primer not be skipped.

Background

Livestreaming (*verb*) – the act of real time broadcasting a computer screen to viewers, typically for profit, via the internet.

As an industry, livestreaming has existed for just under a decade. While there was no official commencement, the three largest companies (TwitchTV, Livestream, and Ustream) were all founded in 2007.[1][2][3] Since then, there have been many other entrants into the market, most notably Google via their subsidiary company YouTube and their associated product, YouTube Live. The largest of these companies, TwitchTV, was sold for over $1,000,000,000 in 2015 to Amazon Inc.[4]

Originally designed to be a general purpose media platform, designed to supplant and expand upon traditional media networks such as live cable TV or radio, livestreaming quickly found a niche in symbiosis with the budding electronic sports ("eSports") industry. eSports is a growing phenomenon wherein professional video game players play games in the same way professional Football players might play Football. The teams practice, and the advertising and prize money are enough to support this growing industry. The *League of Legends Championship Series* finals, an event similar to the Super Bowl for the popular game League of Legends, boasted 14 million concurrent viewers in 2015.[5] If compared to the most popular TV shows of 2015, this event would rank 11th, surpassing hugely popular shows such as Game of Thrones. In a recent press release, Turner Broadcasting and TBS joined with WME/IMG, (a large broadcasting conglomerate) announcing that they would be partnering with the video game publisher, Valve, to feature Valve's flagship eSports game, Counter Strike: Global Offensive (referred to as CS:GO) in a new traditionally broadcast league.

This follows upon the models formed in other countries, most namely South Korea, where games such as StarCraft: Brood War, published by Blizzard Entertainment, have been broadcast on television for over a decade.

However, despite these impressive numbers and partnerships, the vast majority of livestreaming viewers are watching independent livestreams. This fast growth industry thrives on the symbiotic relationship between content producers and content consumers. These content producers are individuals, or small teams of individuals, who work to produce content on a regular basis in a similar business model to that of YouTube. Unlike traditional media, anyone can choose to become a livestreamer by downloading and using freely available technology.[6] As an example, Sean Plott, better known as "Day[9]", has been hosting a 5 day-a-week show on

TwitchTV for over 5 years. He has developed both a dedicated community following, and a successful business, as an entertainer through this medium. This community following is the most important aspect of livestreaming. Compared with traditional media, livestreams earn very little money from advertising. Instead, they follow a subscription model, where subscribers gain extra benefits in the community, such as access to recordings of past videos and the ability to talk in the livestream's associated chat room. Another popular streamer, Steven "Destiny" Bonnell II, publicly disclosed that of his roughly $100,000 per year income, only $1,000 per month comes from advertising. This is in part due to the fact that the demographic most likely to use livestreams is also the demographic most likely to use some form of ad blocking software. Mr. Bonnell has previously discussed that he has conducted informal studies of his stream, and found that at least 60% of the viewers are seeing zero ads. The rest of his revenue is generated through subscriptions and donations from his viewers.[7]

Because of the near non-existent barriers to entry, competition for viewers among unpopular or new streamers is fierce. If the broadcaster (also referred to as a "streamer") is able to capture a dedicated fan base similar to Sean Plott or Steven Bonnell, they will be able to leverage this into the acquisition of more viewers, which results in more subscribers, and thus more revenue as described in the next paragraph.

Motivation

Due to the previously mentioned structure, one of the primary business goals for streamers is to both capture new viewers and convert those users into subscribers. This creates a system whereby there is a strong economic incentive to be noticed by the general body of users. One of the most common ways to do this is to already be popular – that is, to have an already established group of fans that regularly follow and are interested in your body of work. TwitchTV offers the ability for users to browse for streams by both topic -- generally a videogame or eSport of some fashion, which is then further subdivided into individual streamers -- and by streamer, organized with the streams with the largest number of viewers at the top. At the time of writing, in order to appear above the fold, that is to not have to scroll down to view the streamer, a stream must have above four thousand viewers.

This is one of the few means of discovery for new streamers. If a streamer was somehow able to cheat this system, they would have a strong advantage versus their competitors in a very competitive market.

Streamers can accomplish this by using a system called “viewbots.” Viewbots are automated fake viewers that make it appear as though there are more viewers on a stream than there are in reality. In traditional internet media, this type of system referred to as a clickbot is used to click ads and generate revenue. This cannot happen in the livestreaming domain due to the fact that ads are viewed and not clicked, and the vast majority of income is earned from subscriptions rather than advertisements. However, what this system does do is exploit the popularity system implemented by providers such as TwitchTV. By having a large proportion of fake viewers, the stream is pushed above the fold, where it then receives much more exposure than it traditionally would. Over an extended period of time, this exposure will result in increased interest and viewers resulting in increased subscribers, which ultimately results in increased revenue.

Because of this structure, the general process used by unscrupulous streamers is to purchase viewbots in order to appear massively popular in a comparatively short period of time. This does not strike legitimate users as uncommon, as there are other means of garnering large amounts of interest quickly. If a much more popular streamer endorses a less popular livestream, this can generate a massive influx of short-lived attention. Similarly, if a video of a streamer becomes popular on a link aggregation website such as Reddit.com, a similar mechanism can occur. Thus, it is not sufficient for users to say “this streamer is suddenly popular, something is suspicious.” In all of these situations, legitimate users are attracted to the stream via the popularity rankings.

Assuming that the malicious livestreamer is producing content of at least average quality, some percentage of those legitimate new viewers become followers, which translates into actual subscribers over time.

After a sufficient user base has been established, the viewbots no longer serve a purpose. With a dedicated, legitimate following, a streamer can use his or her newfound fan base to continue to promote his or her brand. At this point, the viewbots become a risk in that being discovered would have strong negative public relations implications for the streamer. Thus, their usage is phased out in conjunction with the rise of legitimate users. Over time it appears that the streamer has a consistent viewing, where in reality they are able to leverage this position to increase users until a point where after the viewbots are no longer necessary.

There is at least one major recorded incident of this being discovered by viewers, however there have been many instances of accusations of foul play.[8]

Current Work

The viewbot and unscrupulous streamer system outlined in the previous paragraphs appears to be rather obvious, given a basic understanding of the market forces and domain involved. This would lead to the obvious conclusion that there is some method of detecting and preventing this behavior. However, as of the time of writing of this paper, no published materials could be found on a system designed to detect or prevent this behavior. It is entirely possible that livestreaming providers have designed a system, and chosen to keep it private, in order to help obfuscate the detection method, thereby making countermeasures less effective. However, a survey of the media surrounding livestreaming seems to imply that viewers still believe it to be happening, and are still discontent at the company's apparent lack of interest.

There are, however, many similar domains from which we can draw inspiration as to the current state of fraud detection. We can first consider the primary method of detection for *botnets* as a whole. There are many papers published on the subject, all revolving around a singular central methodology: that of detecting command and control (C2) traffic for the botnet as a whole, and using that to identify the source and scope of the botnet attack.[9][10][11][12] While the exact method used varies from paper-to-paper, they generally involve some method of clustering the data into valid and invalid traffic, using one of several statistical undirected classifiers. The most common method used in our research was K-Means[10] and in the case of BotMiner[10] and BotSniffer[9] X-Means[11]. This methodology gives us a clue to how we could potentially classify viewbot traffic, as undirected classification would also be an ideal candidate for detecting viewbots. Next, we considered the domain of currently available online fraud detection techniques. The most common advertisement available on the internet is called a *banner ad.* While the name used to be autological in the early days of the internet, the term has since grown to include any static ad that takes up a portion, but not the entirety of the screen. These ads pay the host, most commonly a website, for every click they receive which usually redirecting the user to some store or similar promotional material.

Within this domain we found one paper that showed very promising results derived from a competition where participants competed to detect most efficiently the fraudulent clicks among the data set provided.[12] The authors summarize in their abstract most plainly:

> “The mobile advertising data are unique and complex, involving heterogeneous information, noisy patterns with missing values, and highly imbalanced class distribution… Our principal findings are that features derived from fine-grained time series analysis are crucial for accurate fraud detection, and that ensemble methods offer promising solutions to highly-imbalanced nonlinear classification tasks with mixed variable types and noisy/missing patterns.”

Our initial findings strongly corroborated this claim. The winning approach for this competition used gradient boosting, a machine learning technique that yields a decision tree for classification of future data. The authors also noted that the winning teams attempted a wide variety of other methods, including support vector machines, and neural networks. These techniques will be elaborated on in later parts of this paper.

Finally, we considered the domain most similar to our own. Video content portals, most specifically YouTube, generate millions of dollars in revenue every year.[14] These ads are paid per-view, and share the most similar domain space with TwitchTV viewbots. While there are key differences, namely that TwitchTV viewbots are not used to generate fraudulent revenue, methods used to detect fake viewing activity have strong cross-application utility with respect to our domain problem. Unfortunately, while Miriam Marciel[15] et al. explore the ability to monitor, and independently verify the proprietary detection systems of video content portals, they do not attempt to design a system themselves.

Thusly, the availability of this attack cannot be overstated. A simple Google search for “TwitchTV viewbot” yields a site where users can purchase viewbots as a service, from a professionally designed website signed by a root certificate authority.[16][17] While this certification is certainly not a mark of reputability per se, other less scrupulous websites such as ThePirateBay have never been able to receive such certifications.

Due to the immediate availability of the method of attack, as well as the lack of current literature on this specific subject, viewbotting could easily become commonplace without counter measures in place. As with all high growth industries, there are strong incentives to game the system where possible. Thus, we have set out to create a system capable of identifying viewbotting in real time.

CHAPTER 2

AN EXPLORATION OF CLASSIFIERS TO IDENTIFY POPULARITY MANIPULATION OF LIVESTREAMS ON TWITCH.TV

[1]An Exploration of Classifiers To Identify Popularity Manipulation of Livestreams on Twitch.TV. To be submitted to *Open Science in Big Data 2016, a workshop of IEEE BigData 2016 Conference, Dec 5-8, 2016*.

Abstract

Livestreaming is a growing industry with millions of viewers and thousands of content producers. These content producers compete for top spaces in rankings in order to be discovered by viewers. This ranking system is easily susceptible to fraud via false viewers. Using a variety of machine learning techniques, we attempt to identify the most efficient method of detecting these fraudulent viewers.

Introduction

Livestreaming is a fast growth industry wherein thousands of new content producers and millions of viewers have within ten years developed a strong ecosystem of media. An evolution of video-on-demand services such as YouTube, and livestreaming websites such as TwitchTV, provide real-time media produced by small business content producers and delivered to millions of viewers. However, as is common with many fast growth industries, there is a growing fraud problem among the livestreaming communities. Using fraud tools called viewbots, livestreamers are able to fake the reputation system governing this industry, and artificially inflate their viewer numbers. This causes the fraudulent streamer to gain recognition, be noticed by real viewers, be more likely to gain sponsorship deals with large companies, etc. Obviously, this poses a major problem to the growth of the industry.

Fundamentally, identifying fraud is a classification problem. The viewers are the feature space, and there are two distinctive groups: real viewers, and fake viewers (viewbots). If a system can be developed to identify to a high degree of accuracy when viewbots are being used on a livestream, corrective measures may be taken in order to guarantee only real content is rewarded.

Classifiers

These analysis tools were chosen for their dual purpose use of being very common, very successful machine learning tools (with the possible exception of Naïve Bayes, which was included as a baseline), and being implemented either in Spark or in our previous work. The five chosen were:

1. Naïve Bayes
2. Support Vector Machines
3. K-Means
4. Random Forests
5. Deep Ensemble Recurrent Artificial Neural Networks (DERANN)

We believed that these would provide a large variety of different types of tools, allowing us to more completely explore the potential analysis space, and refine our tool selection from the results gathered.

Naïve Bayes

A Naïve Bayes classifier is a probabilistic classifier that functions on a variation of a Bayesian statistics equation. The most common equation, is as follows:

Given a discrete set of independent features described by $x = (x_1, \ldots, x_n)$ where x_m is a singular feature, probabilities $p(C_k|x_1, \ldots, x_n)$ can be assigned to each of K possible outcomes. Then, using Bayes' theorem, the probability of a given outcome can be decomposed as:

$$p(C_k|x) = \frac{p(C_k)p(x|C_k)}{p(x)}$$

This classifier has been in use for nearly 50 years, and has been extensively studied as a classification technique.[25] It maintains a strong performance characteristic during training at O(n), and can reasonably be used as a baseline for many other more advanced classification methods.

Support Vector Machines

Support Vector Machines are a supervised learning binary classifier. That is to say the standard construction only allows for classification in one of two domains, and they require training data in order to be effective.

Specifically, a support vector machine creates a hyperplane of *n=feature count* dimensions within the feature space. Then, through the use of the training data and weight models, this hyperplane is manipulated such that the error rate is as low as can be achieved given the training time. An error in this system is any training data point that is on the wrong side of the hyper plane within the feature space. Like all other classifiers, this method can suffer from overfitting if performed for too long.,

K-Means

K-Means is a clustering algorithm that attempts to group *n* objects into *k* categories or clusters. To do this, the algorithm selects *k* centroids within the feature space, and then assigns all points within the feature space to their closest centroid. The centroid is then recomputed using the mean of the points assigned, and then the process is repeated. Once the centroids settle, the algorithm is complete and the data is grouped. This process can be very computationally expensive as it is an NP-hard problem, however there exist a variety of powerful heuristic variants that allow for a much quicker processing time.

Random Forests

Random forests leverage ensemble learning to generate a large number of decision trees. Each decision tree represents a classification of the feature vectors into the possible outcomes. These decision trees are then polled en masse, and their results compiled to provide the classifier output.

Random forests have the dual purpose utility of both being very refined along the classification boundary, due to their ensemble nature, as well as efficient at classification after training has been conducted, as a given tree search is an O(log(n)) operation.

Artificial Neural Networks

An artificial neural network is an approximation and estimation algorithm that has shown great promise in a wide variety of problem solving situations. From Google's Alpha-Go's recent victory against Lee Sedol in the game of Go[18], to guiding autonomous vehicles[19], they have been effective in a myriad of different artificial intelligence applications.

Just as genetic algorithms emulated the natural emergent process of evolution, so to do artificial neural networks emulate a biological process; namely, they model the structure of a living brain, by connecting artificial neurons together in long chains. These neurons, referred to as nodes, each contain several properties that govern their function: an activation function, and an activation weight.

Activation functions accept inputs from all of the neurons connected to the node, and then returns its own output to the node it is connected to. These functions can vary in exact design, but most generally they serve to say if the input is sufficiently large, provide an output of one, otherwise provide an output of zero, or near zero. The most common activation function is the hyperbolic tangent, given an input system of continuous values from [0, 1].

Activation weights are controlled by a learning process. This process is a parameter to the network, picking one of several possible processes that modify the activation weights in different ways. As a general rule, the learning process finds the error between the output and the known quality of the input, and uses that to adjust the weights throughout the network. Over many trials, referred to as iterations, this causes the network to be more accurate with respect to its prediction of its training data. This highlights one of the central challenges with artificial neural networks in specific and machine learning on the whole. If the training data is not properly representative of the data set as a whole or if the training process is allowed to continue for too long the algorithm can overfit. Overfitting is a common machine learning error where the algorithm will learn from a small sample set for too long causing it to incorrectly classify anything that is not identical or nearly identical to the training data as the not-goal group, even if it belongs to the goal group. If a classifier overfits it will be unable to handle normal variations within the data set.

This would seem to incorrectly imply that all artificial neural network learning must be done with training data. While this is true in some use cases, it is worth noting that unstructured learning, where a classifier network will learn to group data without any prior training or input, are also widely successful in a variety of applications. This can be key in many situations where it is impractical or impossible to collect a large corpus of training data. As an example, many image classifiers combine structured and unstructured approaches in order to better classify very similar structures such as human faces.

Returning to the overall architecture, the previously described nodes are organized around groupings called layers. There are two special layers, the input layer and the output layer, which as their names suggest govern the inputs and outputs to the system. These special nodes then connect to any number of processing layers, traditional networks have only one processing layer, while more modern artificial neural networks could have many processing layers. An artificial intelligence that has more than 3 layers of nodes belongs to a class of algorithms called “deep learning.” Deep learning networks can serve to identify higher order structure unavailable to more simplistic designs. This emergent property is still the subject of large bodies of research that explore the nature of how these systems work, though the fundamentals are underpinned with a strong statistical framework.

Modern artificial intelligences use a wide variety of supplementary functions, just as genetic algorithms do. These can include any number of special properties and designs that allow for specialization. While we could not possibly provide an exhaustive list in this paper, we want to introduce the specializations used in our network.

Recurrences

Recurrent artificial neural networks are neural networks that share the property of having nodes form a directed cycle. A directed cycle allows nodes to activate each other in a sequence that eventually arrives back at the original node, effectively creating a closed loop within the system. These cycles allow for specialization and memory to be exhibited by the system, which allows for highly effective optimizations with respect to time series inputs – inputs where each data point is not independent, but rather dependent on the previous input or inputs.

Standard fully recurrent networks, and to a lesser extent deep learning artificial neural networks in general, face a problem called the vanishing gradient problem. If there are too many layers or nodes within a system, the errors that are backpropogated through the system tend towards 0 or ∞, which can ruin the learning potential of the system. Backpropogation is a method of updating the weights within an artificial neural network that has a diminishing return from layer to layer. These errors can be mitigated in a number of ways that we discuss later in this paper.

Ensembles

Many neural networks suffer from challenges revolving around overfitting, or having an imperfect decision boundary. Small variations in training data can lead to very minute differences in the decision boundary, which leads to an overall less effective classifier. Ensemble neural networks ameliorate that problem by implementing ensemble voting.

Ensemble neural networks create several different neural networks. They differ either by the training data received or the hyperparameters used to govern their learning, which cause small changes in the decision boundaries, leading to several implications. Firstly, when presented with input data, the system polls each member of the ensemble and then averages their outputs. Because these variations are small with respect to the input space, the ensemble should return the same result as a singular network in nearly all situations that don’t border a decision boundary. With inputs that do border a decision boundary, the ensemble serves as a fuzzy decider, averaging results from networks that have the input on both sides of the respective boundaries. This minimizes the chance that learning errors will carry over into the result, due to the fact that a single network is susceptible to overfitting or incomplete training errors, whereas for an ensemble to be similarly affected, the entire ensemble would need to contain the same error.

Genetic Algorithms

A genetic algorithm is an optimization and searching algorithm, designed to find an optimal result within a given search space. A genetic algorithm models the concept of genetic drift and evolution observed in biology, by performing three distinct steps: mate selection, crossover, and mutation. Following along the basic ideas of evolution that underpin the core concepts of biology, a genetic algorithm takes the idea of survival of the fittest, and applies it to an algorithmic problem. It does so by having a given solution to a problem, termed an individual, and a function to evaluate the quality of that solution. These individuals make up a population, and this function is referred to as the fitness function. A genetic algorithm randomly generates a population, and then performs the following actions on the population of individuals.

Mate selection: pairs two individuals to be potentially acted upon by the crossover function. This can be done by any number of random or guided techniques, depending on the problem set.

Crossover: randomly crosses the genetic makeup of two paired individuals, resulting in two new individuals. The theory of crossover is that over a long timescale stochastically combining strong (as defined by the fitness function) individuals will result in even stronger individuals.

Mutation: randomly modifies the genetic structure of an individual. Note that the genetic structure in this instance is any number of variables that define an individual within the system. In order for mutation to function properly, these variables must have a finite number of possibilities. Mutation serves as a pathfinding function within the search space, allowing the algorithm to escape from local optima and premature convergence. Premature convergence would result in a suboptimal output, as the algorithm had more optimization available within the search space.

These elements are then combined with a number of secondary aspects that supplement the core loop of the genetic algorithm. These secondary aspects are designed to specialize the genetic algorithm, allowing it to more effectively model a given problem. At times, this can mean a stronger search function that explores more of the search space, or a stronger convergence function that hones in on even minimal optimizations.

Parameter Selection

Each of these machine learning tools utilize between one and five parameters, not all of them discrete, as potential inputs affecting the results. We arrived at a potential 320,3400,000 possible parameter options, after discretizing the smooth parameters. If execution of each parameter possibility took one tenth of one second, this would take approximately 10 years to complete. That is obviously completely impractical, and as such we chose to guide our parameter selection using another machine learning algorithm: the genetic algorithm. The genetic algorithm executes a guided search through a parameter space, using a fitness function – in this case our analysis tool above serves as our fitness function – to guide it by reducing the error produced by this function. We provided the genetic algorithm the following parameters when conducting our search. These same parameters were implicitly also used in the trials of our analysis tools.

1. Naïve Bayes:

   λ (smoothing rate) – [0, 2], quantized by 0.02

2. Random Forests:

   numTrees – [5, 100], quantized by 5

   maxDepth – [3, 10], quantized by 1

3. K-Means:

   Iterations – [1, 100], quantized by 5

   Ɛ (convergence threshold) – [1, 10], quantized by 1

4. Support Vector Machines:

   Iterations – [100, 400], quantized by 10

   Regularization - [0, 1], quantized by 0.01

5. Deep Ensemble Recurrent Artificial Neural Networks

   Hidden Layers – [1, 4], quantized by 1

   Hidden Nodes* – [10, 100], quantized by 1

   Learning Rate – [0, 1], quantized by 0.01

Learning Decay Rate – [0, 1] quantized by 0.01

Default Weight – [.5, 1], quantized by 0.01

Epochs – [10000, 10000000], quantized by 10,000

* Note that this is technically not one allele but several, as the number of nodes per layer may vary independently from each other, however architecturally the genetic algorithm considers this item as a single allele for purposes of determining crossover, mutation, etc.

It is worth noting that iterations were considered for two reasons. Firstly, as a measure for whether a given configuration would begin to overfit and secondly in both cases execution time is also passed to the genetic algorithm as a floating parameter – it cannot be modified as an allele directly as it is a first order result of another allele, but is nonetheless used in the fitness function.

Feature Selection

We chose a variety of features designed to both identify whether a viewbot was being used, and mitigated one of the primary stealth features used by certain classes of viewbot. The primary set of features relies on the instantaneous and symmetric nature of most viewbot implementations. Most viewbots all connect to the stream at the exact same time, and do not join the chat. Of the implementations that do join the chat, most use symmetric naming conventions such as "light[0-99]" or similar usernames that are able to be generated programmatically. Of those, a vast majority of viewbots do not participate in the chat.

Thus, the following feature vectors were chosen:

1. Time interval in sampling rate intervals since the listening started. This is necessary to structure the time series inputs as ordered.
2. Chat-to-viewer ratio. Anyone can watch a livestream, without requiring an account. If you have an account, you are connected to the associated chat room for that livestream. Most

viewbot approaches don't register accounts, thus an abnormally high viewer to chat ratio (and conversely a low chat to viewer ratio) can be indicative of viewbots.

3. Viewer-to-follower ratio. Similarly to above, for a given livestream the conversion of viewers to followers should be a constant over time. If it decreases significantly this indicates there is a large proportion of anonymous viewers that are not exhibiting the statistically "standard" behavior.
4. Moving average of number of viewers. Viewership patterns, while not static, are normal with respect to the livestreamer. If this average is higher than its historical precedent without a steady increase (see next parameter) this is indicative of viewbot activity.
5. Derivative of number of viewers. Growth spikes are rare in livestreaming, and can be captured by looking at the rate of change of the number of viewers. The only legitimate way growth spikes happen is being referred to or hosted by another streamer, as this redirects the viewership of an entire other streamer to the original. This type of growth spike has other telltale signs in the 6th and 7th feature vectors and can be accounted for. A growth spike without these signs indicates that there has been an increase in viewership without cause. This is the strongest indicator of viewbot activity, while controlling for other factors.
6. Names of chatters. Many similar names in a chat can be indicative of viewbots programmatically generating accounts. A measure of clustered similarity of chat names identifies this trend. Due to email verification this is unlikely, but this parameter allows for advanced viewbots that are attempting to avoid detection methods to be identified.
7. Moving average of chat volume per chatter. For a given livestreamer, the amount of conversation per-sample-rate per-viewer should be constant over time. If this drops

significantly over time, then we have an indication that a proportion of chatters are not making use of the only function of the chat – to converse with others. This could be indicative of viewbots connected to the chat, and this measure serves to detect advanced viewbots in conjunction with feature vector 6.

The first three are used to detect abnormal viewer behavior, and the last three are used to detect abnormal chat behavior. Both behavior types are indicative of the fact that viewbots may be in use. To arrive at these features, we considered a number of other features as well, including whether the viewers were subscribers of the channel, number of followers of a given viewer, number of followed channels for a given viewer, and whether they had produced any content. While some of these features, particularly if they have a high number of followers or have ever produced any content, would be incredibly useful in filtering out false positives – a viewbot has never broadcast a video to a near 100% certainty – they present too large of a performance hit to be reasonable features. To collect this data the profile of each viewer would need to be loaded. The bandwidth to execute this search exceeds even the most powerful gigabit internet connections, and would had far too long of a round trip time to have any hope of building a real time system. Thus, the features chosen are all available from just the streamer's livestream webpage, and the chat associated with the livestream.

Training Design

Before our detection system could detect the viewbots, the classifier at the heart of the processor had to be trained. (Past tense needed for consistency.) The guiding principle behind the training stage was to minimize false positives, at the expense of false negatives. We decided this due to the fact that falsely accusing, or implicating a livestreamer with an accusation such as this, could be potentially damaging to their career. Only in situations where we have a high degree of certainty do we want the system to make the claim that a livestreamer is using viewbots to artificially inflate their viewer numbers.

In order to effectively both: train the classifiers and connect them to the genetic algorithm, a connection structure between the two needed to be created. Fortunately, the genetic algorithm implementation is our own, and thus allowed for a great degree of flexibility in connecting these two learning methods.

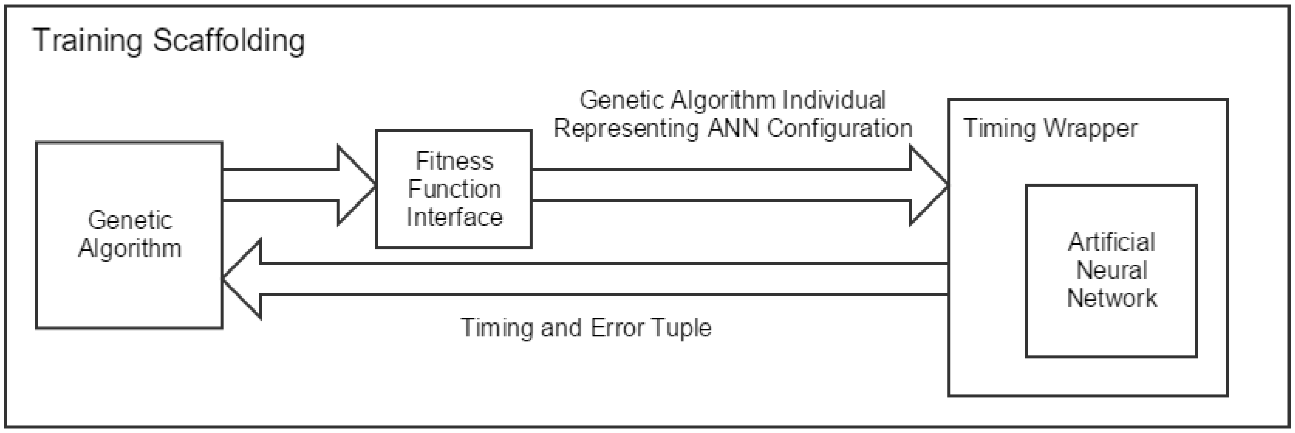
Training Scaffolding
Genetic Algorithm
Fitness Function Interface
Genetic Algorithm Individual Representing ANN Configuration
Timing Wrapper
Artificial Neural Network
Timing and Error Tuple

Figure 1

At the core of the system is the classifier that needs to be trained. Wrapped around this is a layer of timing and execution functions. This layer is distinguished from a layer inside a neural network that describes the number of nodes in that this layer indicates a loosely coupled interface, designed to mitigate coupling between large collections of related software. These functions are designed to provide a single entry-point to pass in all of the relevant parameters needed to execute the classifier, as well as provide back not only the final error rate, but the execution time – both of which are required by the genetic algorithm. These tuples, or key-value pairs, are returned to the genetic algorithm, which then continues its process of optimization. Using this structure, the scaffolding effectively separates the genetic algorithm from the classifier, using it purely as an objective function.

At the end of the genetic algorithm's execution the top *n* individuals in the population, where *n* is the size of the ensemble to be used, are provided as the output of the genetic algorithm. These individuals not only contain the parameters used for their creation, but the structure of the classifier itself. This classifier is then connected to the processor at runtime, and used for the verification step.

Meta Adaptive Genetic Algorithm

As previously mentioned, a genetic algorithm was used to perform a search through the feature space. This meta-genetic algorithm (GA, MGA, or MAGA) has a variety of design properties and its own hyper-parameters that affect the results.

The GA is a subtype of GA referred to as adaptive genetic algorithms, due to their ability to relax and expand their search parameters proportionately to the convergence rate – in our case measured as the absolute difference of the error generation over generation – of their fitness function. This practice is controversial as in some ways it violates the principle of not interfering with the natural selection process of a GA. However, it can be beneficial in some cases, especially when the topography of the search space is unknown. In our specific use case, we have no prior information as to what the proportion of viewers to viewbots will be. This will change from fraudulent streamer to fraudulent streamer. Moreover, because our feature space for our DERANN are all second order approximations of the first order information, our understanding of the search space is very limited. Thus, having an adaptive GA provided much better results than the use of a typical out-of-the-box GA implementation. Our MAGA accomplishes this in several different ways provided below.

1. As the convergence rate decreases, indicating an increase in convergence, the mutation and convergence rates are proportionately expanded and contracted respectively to a cap.
2. After a certain threshold, the crossover method is changed from one point to two point.

3. A predetermined "island" population is randomly generated and set aside before execution begins. After a certain threshold, a random culling of the population is performed, and new individuals are introduced from the island.

In some ways, the adaptability of this GA is in and of itself a machine learning algorithm, as it alters parameters on the fly in order to try to prevent a premature convergence, something where researchers would previously tweak settings in order to accomplish. However, what settings are most appropriate at what point in the process is not constant, thus the need for the algorithm to be able to adapt during execution. Because of our use of n-elitism, or the retaining of n individuals within the population from generation to generation, we suffer very little loss results due to the increased variation within the GA.

Hyper Parameters

The following hyper-parameters were used for our MAGA.

Population Size: 100

Island Size: 20

Crossover Type: Adaptive [1 point, 2 point]

Starting Crossover rate: .06

Starting Mutation Rate: .02

Mating Criteria: Adaptive [Tournament, Roulette]

Elitism: Adaptive

These hyper-parameters were chosen after some experimentation showed that the adaptive nature of the MAGA changes in the starting parameters yielded little to no improvement in MAGA performance. These parameters were chosen based on previous research into optimization of artificial neural networks using a genetic algorithm.[27]

Fitness Function

The fitness function for the genetic algorithm was the two term equation given below.

$$F = \frac{10}{E_N} * \frac{1}{T_N}$$

Where F is the overall fitness, E_N is the normalized error rate, and T_N is the normalized time of execution. Note, this execution time is not the training execution time, but rather execution against a single time series data point after training has finished. This happens as a post-execution step during the fitness function of the GA. Any execution time greater than the sampling rate will automatically result in a fitness of 0, as the system has violated the real time constraint of the design.

<u>Training Livestream Selection</u>

While mocked data could be used, there is no way to verify whether a system is working, or whether the data is constructed specifically so the system could identify it. As such, we decided that the most efficient way to collect this data would be to simply attach viewbots to an already running livestream, and monitor the results.

Due to the ethical ambiguity of using a livestream, we made several decisions in order to reduce any impact this would have on the livestreamer. Most significantly, we chose a livestreamer that already had a dedicated following and was by any measure of livestreaming "successful." We selected a stream that would have no need for viewbots, as their following was already well established. Furthermore, we performed our test in a very limited scope, and only during peak hours when the user was already "above the fold", in order to reduce any discovery impact the viewbots would have. We collected data from this livestream with viewbots being executed for 90 minutes, in one second intervals to generate our bot dataset. The data was only collected once, with the interval chosen to be roughly equivalent to one livestreaming session, so as to not unduly influence the livestreamers as aforementioned. Additionally, six hours of this stream were collected without viewbots being run. This gives us 27,000 time series data points, each one consisting of the features described in the *Feature Selection* section to use as training data.

Results

Verification was performed in two distinct phases. Firstly, we executed viewbots as described in the *Training Design* section, and recorded the results. This data is closest to our training data, and acts as a second control to verify the system is working as intended. For this test our viewbots represented roughly on average half of the viewing population.

Our results were very promising for certain classifiers. The table below gives our error lowest error rates for each of the respective classifiers found by the Meta Adaptive Genetic Algorithm.

Analysis Tool	**Error Rate – [0, 1] – Lower is Better**	**Execution Time (ms*)**
Naïve Bayes	0.4426	> 1000 ms
Random Forests	0.2885	< 100 ms
K-Means	0.2503	> 400 ms
Support Vector Machines	0.1001	~ 200 ms
DERANN	0.0654	< 100 ms

Please note that due to the nature of the performance tests run that these execution times are not exact. They are given on the order of 100 ms in order to provide a relative scale between analysis tools, but not exact measurements.

It is obvious that the ensemble neural network outperformed all other during the execution step, however it was the longest to train, having both the largest number of parameters for the MAGA to analyze, and the longest execution time due to the number of potential epochs. That said, training happens only when the data set changes, in our application once it has been trained it will be used in primarily a testing format. As such, we believe this will be ideal for our future work. The parameters identified by the MAGA for use in the DERANN are as follows:

Parameter	**Setting**
Hidden Layers	3
Hidden Nodes	87
Learning Rate	0.23
Default Weight	0.64
Epochs	10,000,000

The features of note in this design are the large number of epochs and the relatively low learning rate. Whether this is a flaw within the fitness function of the MAGA, weighting the execution time too low relative to the error rate, is unclear as the execution time remained well within what we considered acceptable. The learning rate is also inexplicably low, we speculate that this is perhaps due to the large number of epochs, and the rather steep penalty for overfitting assigned by the fitness function.

Secondly, using our own livestream* we performed four tests. After identifying the DERANN as the ideal candidate for our use case, using volunteer viewers as well as viewbots, we monitored normal livestream activity with viewbots making up 0%, 10%, 25%, 50%, and 100% of the viewing population. These tests used a total viewing population of 100. This was done in order to further identify a reasonable thresholding value above which it is reasonable to claim that a user's popularity is being inflated from the use of viewbots. The results of the DERANN are given on the following page. The outputs have been truncated for ease of reading.

Proportion of Population	**Processor Confidence [0-1] – Higher Indicates a higher likelihood of viewbots**
0	.04
.10	.03
.25	.78
.50	.95
1.0	.99

These are very promising results. While we were unable to detect the 10% proportion of the population, in fact it shows a lower result than no viewbots at all, all other proportions show a strongly identifiable result. We speculate that the 10% proportion is too small to identify within the noisy data that has been collected. However, this does not, in our view, represent a failure of the system. One of the original goals stated above was to minimize false positives at the expense of true negatives. A 10% viewbot rate is insufficient to push a streamer "above the fold" in nearly any category. Furthermore, a streamer who has a 90% true user participation rate is, in our estimation, not the intended target of detection such as this. While their fraud is still impactful it does not carry the same weight and challenge as a streamer who is able to displace the well-known streamers for a given game using viewbots.

Conclusions

We are able to conclude a number of positive claims from these tests. Firstly, it is possible to identify whether a viewbot is being used to artificially manipulate livestream traffic. Furthermore, we were able to identify that this can be done in a variety of ways, but that a properly trained neural network classifier is most suited to both the real-time constraints of our system and a need for high confidence in the claims of the system.

We believe that future research focusing on implementing this system, in a performant and scalable way, will yield a robust fraud detection system capable of protecting this burgeoning industry, and helping it grow into a media platform capable of driving future interest in a fast growth industry.

REFERENCES

[18] Google DeepMind, March 2016, https://deepmind.com/alpha-go

[19] https://www.google.com/selfdrivingcar/

[25]

https://www.researchgate.net/profile/Irina_Rish/publication/228845263_An_Empirical_Study_of

_the_naive_Bayes_Classifier/links/00b7d52dc3ccd8d692000000.pdf

CHAPTER 3

IDENTIFYING LIVESTREAM POPULARITY MANIPULATION ON TWITCH.TV AT SCALE

[1]An Exploration of Classifiers To Identify Popularity Manipulation of Livestreams on Twitch.TV. To be submitted to *Open Science in Big Data 2016, a workshop of IEEE BigData 2016 Conference, Dec 5-8, 2016*.

Abstract

Livestreaming is a fast growth industry connecting viewers with content producers via a popularity ranking system. There are hundreds of thousands of viewers and This system is susceptible to fraud via a method called viewbotting. We have developed a system of detecting this fraud method using Deep Ensemble Recurrent Artificial Neural Networks. In this paper, we scale this system to hundreds of simultaneous livestreams viewed by hundreds of thousands of users.

Introduction

Designing system architectures that scale is one of the most important and challenging problems faced by modern day computing. Systems that work at 100 users do not work at 100,000 or 100,000,000 users. Many times distinct design steps need to be taken in order to be able to extend out software systems by adding more hardware to support the system. The gold standard of scalability is linear scaling – a 100% increase in hardware results in a 100% increase in the number of operations that can be performed or users served.

Our specific use case revolves around the following: for a given livestream on the website Twitch.TV, we need to be able to monitor a certain set of features over time, and in real time determine whether they are using viewbots. A form of automated interaction similar to a botnet, these viewbots can be difficult to detect, and require advanced artificial intelligence methods in order to be detected.

Our design is broken into two distinct and separable parts. Firstly, there is the system designed to collect data from the livestreaming website itself. This is a simplistic, if tedious, piece of code designed to continuously read data from the target stream, and transform it in a way that it can be readily accepted by the second part of the system. This second part is a system used to analyze the data collected, utilizing the artificial intelligence methods described in Chapter 2. An overview of the system is given in *Figure 1* on the following page.

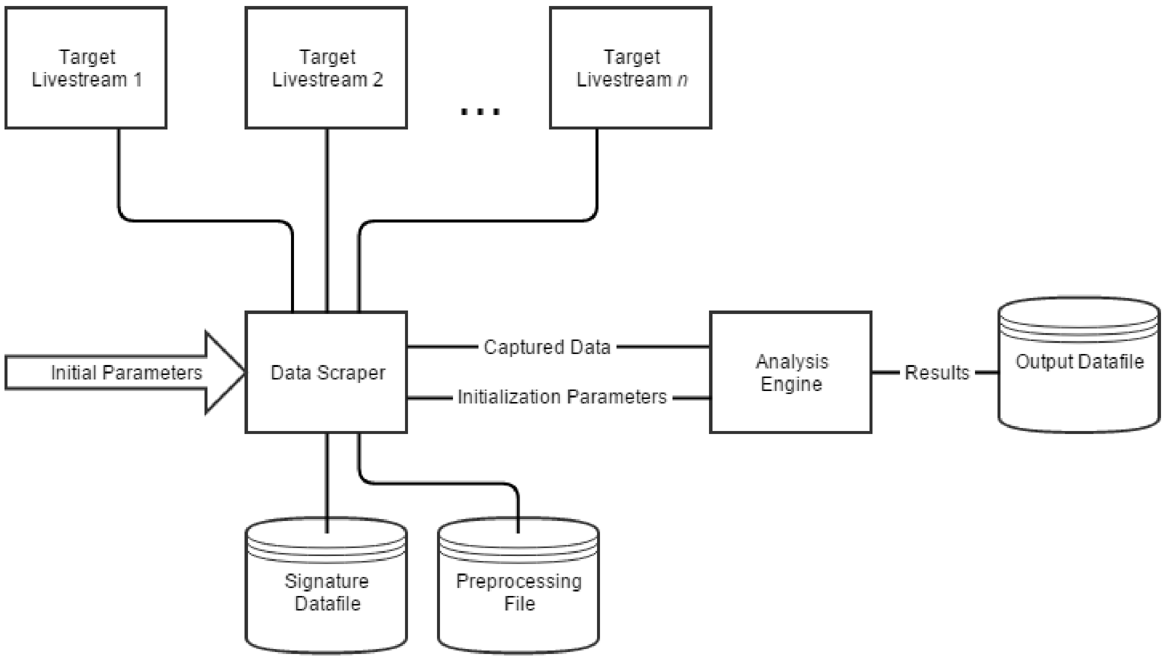

Figure 2

<u>Data Scraper</u>

The data scraper is designed to capture any number of feature vectors to be used by the analysis engine. Rather than fix the scraper to any specific target data from the stream itself, we chose to have it target any number of signatures as provided in its target file. This allows for easy extensibility as the front-end design of TwitchTV changes over time, as well as the introduction of new feature vectors simply by including their signature in the target file.

The data scraper accepts a number of parameters that control the data being retrieved from the site. These parameters are as follows:

1. Sampling rate – this parameter is lower bounded at 500 ms. We found this to be a good compromise between granularity of results and processing requirements. Below that, the processing requirements increase significantly, while having more finely sliced data suffers from diminishing returns.
2. Signature datafile location
3. Output datafile location
4. Preprocessing file location
5. Target livestream(s)

In addition, the data scraper collects all traffic that passes through the chat channel associated with the stream. These channels are implemented via the IRC protocol[20], and thus it is trivially easy to connect and record the chat. Blocks of text are passed to the preprocessing file in blocks the size of the sampling rate. The IRC protocol also communicates all state changes - such as someone entering or leaving the chat, for example - as text. These meta properties can also theoretically be used for analysis, and be analyzed in the same fashion as the chat itself.

For all other signatures, the scraper will poll the target livestream and cache the entirety of the http response. This is added to a queue, and then scanned for the signatures provided. In this way, the data retrieval and the processing of the data are loosely coupled, so as to remain performant.

The signature file is a JSON, or JavaScript Object Notation[21], file containing an array of objects with the following properties. JSON objects are a human readable key value pair data format designed to both easy to use and highly compressible for use in web technologies. In our use case, this allows for hand designing of feature vectors, while being very performant during application startup. Each of these objects is considered a single feature vector.

1. Type : an integer corresponding to which data source to use. 0 corresponds to the http response, and 1 to the chat.
2. Target : a JSON object containing “start” and “end” keys that correspond to the two sides of the signature containing the relevant information.
3. Method : the method name in the preprocessing file that will be used to process the data. This method should return whatever transformed feature vector is required by the analysis engine.

Once it has been loaded, the scraper opens its connections to the provided streams and begins parsing data. For the purposes of this paper, the scraper was implemented in Java 8.[22] Java as a language provides a much more performant base than similar web development languages such as C#[26], while maintaining access to the machine learning libraries most prevalent in that domain including MLLib, Spark, and DeepLearning4J.

Analysis Engine

The analysis engine is a two-step pipeline consisting of the preprocessor and the processor. The preprocessor’s purpose is to convert the raw data collected by the data scraper into feature vectors usable by the processor. The processor is a wrapper around the machine learning implementation used to analyze the data for possible viewbot activity. Both the preprocessor and processor are built on top of Apache^{TM} Spark 1.6[23], and implemented in Java 8, for compatibility and interoperability with the Data Scraper. Spark is a high performance high scalability data processing tool developed by Apache and released under their Commons license. Spark solves many of the programming challenges in distributed computing and pipeline processing allowing us to focus on building out the actual useful code we want to execute on top of this library. For every livestream being processed, the system generates a separate Spark processing thread, as shown in *Figure 2*.

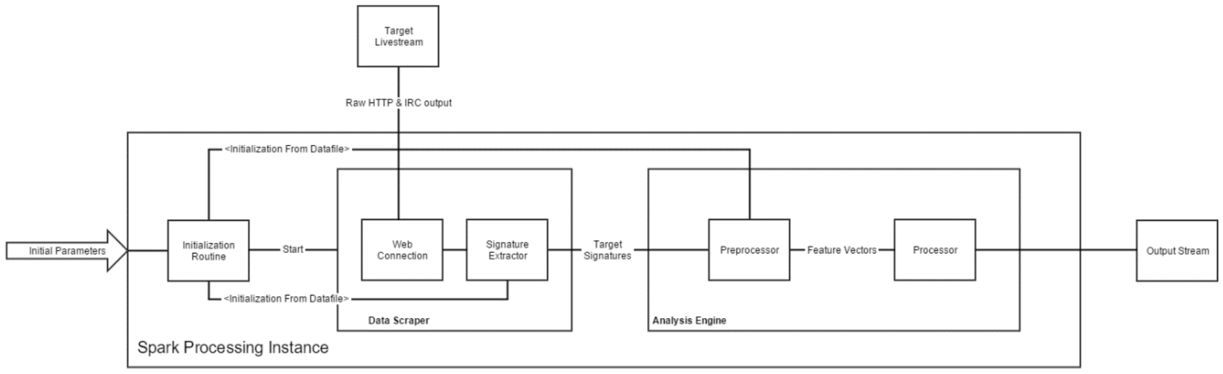

Figure 3

Preprocessor

No proper data analysis tool will be able to accept the raw data collected by the data scraper. It contains large amounts of extraneous html and css markup, and in some cases needs to be mathematically transformed in order to be useful to the processor. This is the job of the preprocessor.

The original design of the preprocessor called for a file accompanying the signature file used by the data scraper that would map each signature to its necessary transformation. However, we quickly realized that the language required to describe the transformations would be as complex if not more so than the code itself. Instead, we have a dynamically loaded Java class provided as part of the parameters to the data scraper. This file is a JAR, or Java Archive file, that contains all of necessary Java code for the Preprocessor's transformations of the data. In this design the user has the full capabilities of the Java language to transform the raw data into useful feature vectors, while not requiring any implementation details on dependencies on our part. These transformations result in the feature vectors passed to the processor, so that they may be interpreted as appropriate.

It is not recommended that a raw input ever be passed due to the signal to noise ratio in raw data collection, however a pass through method would acceptably accomplish that goal.

Processor

The processor is designed to be a wrapper around any number of analysis tools, either custom designed or provided in accompanying libraries such as MLLib within Spark. This is provided merely as an example. Additionally, it provides a small surface area API, providing minimal methods to provide feature vector input, destination stream output, and any configuration settings needed by the internal analysis tool.

As with many other sections in the processor and preprocessor, the processor is controlled by a configuration parameter passed to it at runtime. Rather than strongly linking the processor to a specific implementation, it loads at runtime an implementation capable of performing its needs. This means that in the future should it be necessary the neural network discussed in the following paragraphs would be easily replaceable with another system.

Deep Ensemble Recurrent Artificial Neural Network

Previous research into this subject (see Chapter 2) explored a wide variety of different classifiers to be used inside the processor. These included decision trees, naïve bayes classifiers, support vector machines, and others, including a deep ensemble recurrent artificial neural network (DERANN). We decided previous to this work that the DERANN was the most effective core for the processor, give the restrictions on efficiencies and scalability.

Our DERANN was implemented using the open source engine DeepLearning4J[24]. This library is interoperable with Spark and Java 8, our two other primary tools. Additionally, this library is horizontally scalable with respect to processors allowing it to scale with Spark and the rest of the Processor scaffolding.

A deep ensemble recurrent artificial neural network has the following properties:

1. Deep – At least three hidden layers between the input and output layers, and generally many hundreds of nodes per layer.
2. Ensemble – An ensemble of RANNs is created in order to facilitate ensemble voting, using the genetic algorithm discussed in the following sections.
3. Recurrent – This system uses a long-short term memory implementation of a Recurrent Artificial Neural Network. This implementation allows for the preservation of gradients through cycles and deep node chains, allowing for more effective memory and time series based learning.

In some ways, the DERANN could be classified as an ensemble of ensembles. DeepLearning4J allows for the distribution of sharded training data across cores. That is, data that can be broken into independent pieces and processed without dependencies to the other data. In a time series this implies a cut-off where after a certain number of steps a data point is considered unrelated to the points before the steps. This allows for independent processing that then is averaged together at intervals to inform the main model. Our ensemble is then built of these averaged models to run simultaneously and perform ensemble voting in real time against the data collected.

Once the ensemble was trained, we built a wrapper class around the ensemble that provided the interface as if it was a single classifier. By polling all ensembles and then averaging their responses, this is the input and output point for the processor to leverage in order to provide its results.

Testing

In order to test the scalability of our system, we first determined what reasonable bounds of operation should be. Trivially, one connected livestream is the minimum. Maximally, we surveyed Twitch.TV during peak usage times and found that there are on the order of one hundred livestreams with more than one thousand viewers. We believe this is an appropriate limit as livestreams below that threshold are not likely to be resulting in any appreciable positive utility from using viewbots.

However, our system does not only scale with number of livestreams. To a lesser extent, the number of users connected to the livestream also has a performance impact. To this end we collected both the number of livestreams and number of users in order to control for this impact.

We connected the system to livestreams in groups of five, starting from one for a period of five minutes (600 time series data points) and recorded the execution time of the pipeline per time series data point analyzed and then took the median execution time.

Results

We found that the system performed very well with respect to scaling up to our required numbers. On average, execution time for a livestream to be processed was 602 ms. This does not include the load time from the server to the data scraper for a number of reasons. Firstly, we cannot affect the load, and secondly if this technology were implemented in a co-located server to the Twitch.TV content distribution network, the load time would be on the order of 10ms, as large portions of the load time are due to the round trip time of communication between our testing location in Georgia and the Twitch.TV content distribution network. For completeness, the load time per stream averaged 2,230 ms. The data collected can be found in Appendix 1, and is represented in the graph on the following page.

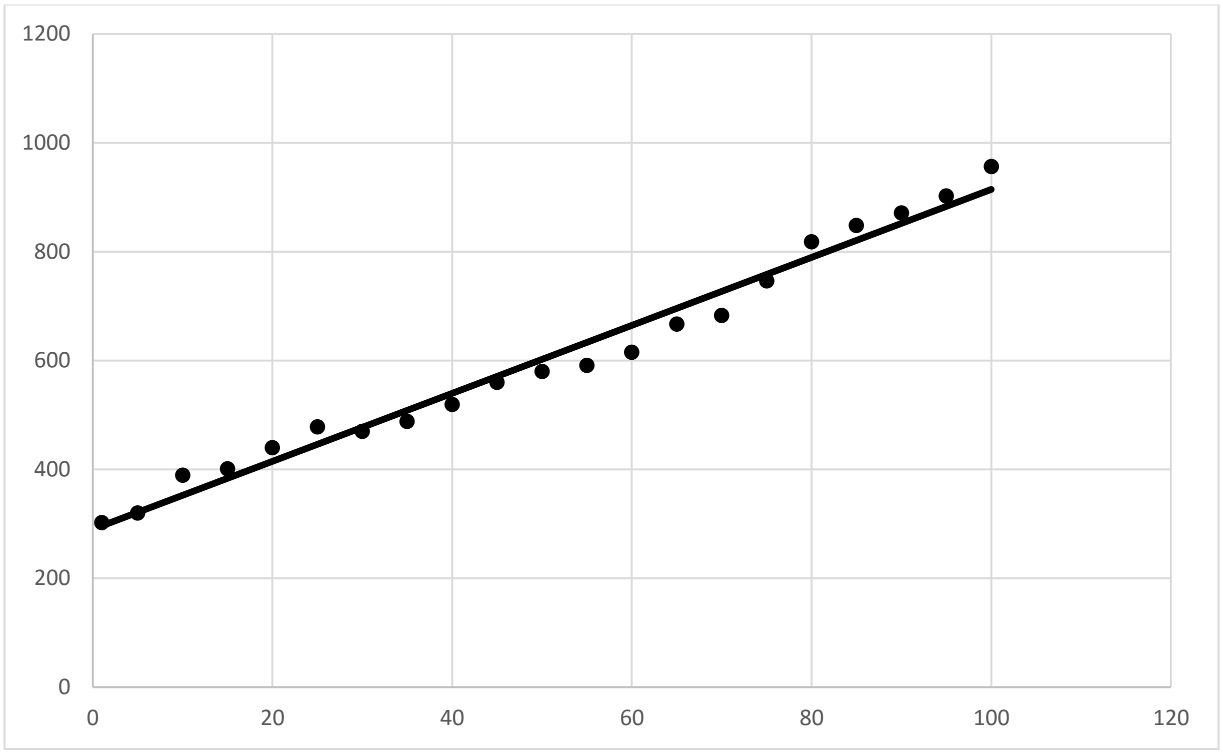

Figure 3

We can see that the system takes full advantage of the parallel processing capabilities available, increasing at significantly less than linear time per livestream added.

Conclusions

With additional hardware and colocation of hardware within the TwitchTV system, we believe that this system will be able to effectively monitor all streams of sufficient science. While staying under our real-time processing cap of 500 ms, the system is able to monitor 64% of the 1.09 million TwitchTV users. Giving a more generous one second window, which granted significantly reduces the granularity and to some degree the effectiveness of the monitoring system, it is able to monitor over 75% of users. Additionally, we can conclude that while the number of users significantly impacts livestreaming traffic, we see no strong correlation with respect to number of users, rather the number of livestreams being the strongly dominating parameter with respect to scalability.

REFERENCES

[20] Oikarinen, J., and D. Reed. "Internet Relay Chat Protocol." Request For Comment. Internet Engineering Task Force, May 1993. Web. Mar. 2016. <https://tools.ietf.org/html/rfc1459>.

[21] http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[22] http://www.oracle.com/technetwork/java/javase/overview/java8-2100321.html

[23] http://spark.apache.org/

[24] DeepLearning4J.org

[26] https://benchmarksgame.alioth.debian.org/u64q/java.html

APPENDIX A

Livestream Scalability Testing Data

Number of Livestreams	Number of Viewers	Average Execution Time (ms)
1	99679	302
5	517711	320
10	574460	389
15	614138	401
20	645644	440
25	669568	478
30	688675	470
35	704110	488
40	718068	519
45	730377	560
50	742212	580
55	752524	591
60	762024	615
65	770940	667
70	779147	683
75	786550	746
80	793359	818
85	800172	848
90	806875	871
95	812920	902
100	818800	956

CHAPTER 4
CONCLUSIONS

Put simply, the experiments were a success. We have shown that a properly scaffolded artificial neural network is capable of being used to detect viewbot traffic over livestreams.
The data collected indicates that a thresholding value of 0.75 would be sufficient to warrant the claim that a stream is being artificially inflated by viewbots.

This of course has not exhaustively covered all possible viewbot attack vectors, and even more importantly, at time of publishing the techniques involved will be available to be considered when designing countermeasures. This is an ever evolving process between the bot designer and the detection designer. However, there is future work that we believe could further improve this system as currently designed.

The data scraper suffers from some scalability issues for large numbers of livestreams or high polling frequencies. As previously indicated it faces challenges effectively operate at more than 500 ms, and while we don't consider this an impediment to the current function of the system, it is possible that future work might depend on a more precise measurement. However, as with any large scaling system, eventually more hardware will need to be introduced. The data scraper does not currently have a load balancing layer, but supports the extension of one architecturally due to the underlying libraries and systems it is built upon. Furthermore, we found that our system supports a reasonable degree of scalability with respect to the number of livestreams presented.

Finally, we believe that the technology created to facilitate this research could be applied to a number of different domains. The flexibility of the Data Scraper, Preprocessor, Processor, pipeline as well as the architecture, namely Spark, DeepLearning4J, and the custom implementation of the genetic algorithm, allows for this system to be reused, simply by modifying the structure of the target data and the preprocessing functions. Any time series based system where the information is publicly available in real time but not properly transformed could benefit from the architectural structure presented here.

REFERENCES

[1] Ewalt, David M. (December 2, 2013). "The ESPN of Videogames". Forbes (paper).

[2] “What is Livestream”. https://livestream.com/about.

[3] “CrunchBase Ustream Company Profile" CrunchBase, April 19, 2013

[4] "Amazon to Buy Video Site Twitch for More Than $1 Billion". The Wall Street Journal. 25 August 2014. Retrieved 9 December 2015.

[5] M. (2015, December 9). Worlds 2015 Viewership. Retrieved March 5, 2016, from http://www.lolesports.com/en_US/articles/worlds-2015-viewership

[6] (2016). Retrieved March 5, 2016, from www.xsplit.com

[7] Egger, Jay (April 21, 2015). “How exactly do Twitch streamers make a living? Destiny breaks it down”. The Daily Dot (magazine).

[8] NickHotS (March, 2015) “WinterGaming uses viewbots…” Reddit (website).

[9] Gu, G., Zhang, J., & Lee, W. (n.d.). BotSniffer: Detecting Botnet Command and Control Channels in Network Traffic. Retrieved March 8, 2016, from http://faculty.cse.tamu.edu/guofei/paper/Gu_NDSS08_botSniffer.pdf

[10] Gu, G., Perdisci, R., Zhang, J., & Lee, W. (n.d.). BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection. USENIX. Retrieved March 8, 2016, from http://static.usenix.org/events/sec08/tech/full_papers/gu/gu_html/

[11] Leyla Bilge, Davide Balzarotti, William Robertson, Engin Kirda, and Christopher Kruegel. 2012. Disclosure: detecting botnet command and control servers through large-scale NetFlow analysis. In Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC '12). ACM, New York, NY, USA, 129-138. DOI=http://dx.doi.org/10.1145/2420950.2420969

[12] Basil AsSadhan, José M.F. Moura, An efficient method to detect periodic behavior in botnet traffic by analyzing control plane traffic, Journal of Advanced Research, Volume 5, Issue 4, July 2014, Pages 435-448, ISSN 2090-1232, http://dx.doi.org/10.1016/j.jare.2013.11.005.

[13] A. K. Jain, M. N. Murty, and P. J. Flynn.
Data clustering: a review.
ACM Computer Survey, 31(3):264-323, 1999.

[14] M Marciel, R. Cuevas, A. Banchs, R. Gonzalez, S. Traverso, M. Ahmed, A. Azcorra (2015, July). Eprint ArXiv:1507.08874. Retrieved March 9, 2016, from http://arxiv.org/abs/1507.08874

[15] Richard Oentaryo, Ee-Peng Lim, Michael Finegold, David Lo, Feida Zhu, Clifton Phua, Eng-Yeow Cheu, Ghim-Eng Yap, Kelvin Sim, Minh Nhut Nguyen, Kasun Perera, Bijay Neupane, Mustafa Faisal, Zeyar Aung, Wei Lee Woon, Wei Chen, Dhaval Patel, and Daniel Berrar. 2014. Detecting click fraud in online advertising: a data mining approach. J. Mach. Learn. Res. 15, 1 (January 2014), 99-140.

[16] Google, March 2016, https://www.google.com/search?q=viewbot

[17] ViewBot.Net, March 2016, https://viewbot.net/

[18] Google DeepMind, March 2016, https://deepmind.com/alpha-go

[19] G. (n.d.). Google Self-Driving Car Project. Retrieved May 21, 2016, from https://www.google.com/selfdrivingcar/

[20] Oikarinen, J., and D. Reed. "Internet Relay Chat Protocol." Request For Comment. Internet Engineering Task Force, May 1993. Web. Mar. 2016. <https://tools.ietf.org/html/rfc1459>.

[21] The JSON Data Interchange Format. (2013, October). Retrieved April 6, 2016, from http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[22] Java 8 Overview. (n.d.). Retrieved April 15, 2016, from http://www.oracle.com/technetwork/java/javase/overview/java8-2100321.html

[23] Apache Spark. (n.d.). Retrieved February 6, 2016, from http://spark.apache.org/

[24] DeepLearning4J (n.d.). Retrieved June 6, 2016, from http://deeplearning4j.org/

[25] An empirical study of the naive Bayes classifier. (2001, January). ResearchGate. Retrieved June 19, 2016, from https://www.researchgate.net/profile/Irina_Rish/publication/228845263_An_Empirical_Study_of_the_naive_Bayes_Classifier/links/00b7d52dc3ccd8d692000000.pdf

[26] Java programs versus C gcc. (n.d.). Retrieved June 15, 2016, from https://benchmarksgame.alioth.debian.org/u64q/java.html

[27] Tuning of the Structure and Parameters of Neural Networks using an Improved Genetic Algorithm. H.K. Lam, S.H. Ling, F.H.F. Leung and P.K.S. Tam. IEEE. Retrieved July 16, 2016, from https://opus.lib.uts.edu.au/bitstream/10453/15148/1/2010004238.pdf.