Implementing best practices for fraud detection on an online advertising platform

Master of Science Thesis

MARTIN HÄGER
TORSTEN LANDERGREN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden, 2010

Title: Implementing best practices for fraud detection on an online advertising platform

MARTIN HÄGER
TORSTEN LANDERGREN

Examiner: ERLAND JONSSON

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering Göteborg, Sweden, 2010

Abstract

Fraud against online advertisements, most notably *click fraud*, is a problem that in recent years has gained attention as a serious threat to the advertising industry. In 2007, Google Inc. estimated[3] that 10 percent of clicks on advertisements in their AdWords program were not legitimate user clicks, which translates into a one billion USD yearly revenue loss after filtering out these clicks. Click fraud detection is not an unaddressed problem, but could largely be considered a "secret art" as little open research exists on the topic.

In this thesis, the application of general outlier detection and classification methods to the problem of detecting fraudulent behavior in an online advertisement metrics platform will be explored. Furthermore, the development of a fraud detection system based on such methods will be described. In the process, several different data mining algorithms will be evaluated based on prediction accuracy and performance.

We empirically show that satisfactory detection accuracy can be achieved by introducing *supervised machine learning* into the detection process - given that an appropriate set of *training data* can be constructed. Such a solution would also be able to benefit from the extraction of training data across a large customer base. We design and implement a system based on a three-step feedback process and find that it scales well in a distributed setting.

Keywords: data mining, machine learning, classification, click fraud, impression fraud, outlier detection, scalability, MapReduce, Hadoop, cloud computing

Sammanfattning

Bedrägeri mot Internetreklam, i synnerhet klickbedrägeri (eng. *click fraud*), är ett problem som på senare år har uppmärksammats som ett allvarligt hot mot reklamindustrin. Under 2007 uppskattade[3] Google Inc. att klicktrafiken mot annonser i deras AdWords-program innehöll ungefärligen 10 procent falska klick, motsvarande en årlig intäktsförlust på en miljard USD efter att dessa klick filtrerats bort. Detektering av klickbedrägeri är ej ett obehandlat problem, men kan på många sätt ses som en "hemlig konst" då det existerar lite öppen forskning inom området.

I detta examensarbete utforskas hur generella metoder för avvikelsedetektering och klassificering kan användas för att detektera bedrägeri mot reklam i en plattform för mätning av nätbaserade reklamkampanjer. Examensarbetet beskriver även utvecklingen av ett system för detektering av bedrägligt beteende baserat på dessa metoder. Under arbetets gång har ett flertal data mining-algoritmer utvärderats utifrån deras precision och prestanda.

Vi visar empiriskt att fullgod detektionsnoggrannhet kan åstadkommas genom att introducera *övervakad maskininlärning* i detekteringsprocessen - givet att lämplig *träningsdata* kan konstrueras. En sådan lösning skulle även kunna dra nytta av att träningsdata kan extraheras över en stor kundbas. Vi designar och implementerar ett system i form av en trestegsprocess med återkoppling och finner att det skalar väl i en distribuerad miljö.

Nyckelord: data mining, maskininlärning, klassificering, klickbedrägeri, impression fraud, avvikelsedetektering, skalbarhet, MapReduce, Hadoop, cloud computing

Preface

Martin Häger and Torsten Landergren both study Computer Science at Chalmers University of Technology, Gothenburg. This thesis is part of their master's degree in Software Engineering and Technology. The thesis work was carried out at the Gothenburg offices of Burt AB during the spring of 2010.

The Authors would like to thank the supervisors - Erland Jonsson at Chalmers and John Sjölander at Burt - for the guidance they have provided during the course of the project.

Contents

1 Introduction

1.1 Background

Burt AB develops tools for creating better online advertisement. Their current product, Rich, is an advertising campaign measurement application that collects data about banner ads (visibility, click-throughs etc.) and calculates the campaign's impact. Advertisement agencies use Rich to analyze the relative successes of their campaigns. To provide even more accurate and reliable data, and to be able to certify this to their customers, Burt sees the need for Rich to detect and filter out fraudulent behavior in their data, e.g. fraudulent clicks. Such clicks could originate from web bots or even humans imitating legitimate users with a genuine interest in the ad content, with the purpose of generating revenue from pay per click advertising (so called "click fraud" behavior).

1.2 Purpose

The purpose of the project is to develop an adaptive and scalable fraud detection component for Rich. This component shall be able to handle the large amount of data that flows through the system and provide output that can be used to improve the accuracy of produced reports.

1.3 Paper organization

The paper is organized as follows. In Section 2 we describe the project workflow. In Section 3 we present a rigorous survey of the areas of outlier detection and classification, and also introduce concepts central to the thesis. In Section 4 we present the system requirements and a model for partitioning user behavior. In Section 5 we describe the design of the developed system. In Section 6 we carry out an evaluation of a number of classification algorithms with respect to accuracy and performance, and discuss the results. Finally, in Section 7 we discuss future work.

2 Method

This section describes the structure of the project work. The project follows an agile approach, and is divided into five different phases which are performed iteratively and with a degree of overlap between them (with testing completely integrated into the implementation phase).

Phase 1: Literature survey

The initial project phase is concerned with studying research in the areas relevant to the thesis topic to determine the current definition of "best practice". This includes looking at existing frameworks and tools that could be used in the construction of the system.

Phase 2: Requirements specification

The second project phase is concerned with formalizing the problem statement into a set of requirements which will guide the design, implementation and evaluation of the system.

Phase 3: Design

The third project phase is concerned with suggesting a suitable design based on the requirements and the findings of the literature survey.

Phase 4: Implementation

The fourth project phase is concerned with implementing the suggested design as a component within Rich.

Phase 5: Evaluation

In the fifth project phase, the implementation will be tested and the results will be evaluated with respect to theory and the requirement specification.

3 Survey of theory

This section describes topics of existing research that are relevant to the thesis. In subsequent chapters of the report, we will discuss their applicability for our implementation. Although there exists some open research in the area of fraud detection, most of the knowledge is not openly available. Companies rarely disclose their fraud detection strategies and for this reason the bulk of this survey covers general approaches which are well-known in public research.

In our application, it is of utmost importance to employ an algorithm that scales well in a distributed setting. This requires that the algorithm can process individual fractions of the entire data set independently, without the need for a global state. We will take this into account when discussing various approaches in the following sections.

3.1 Terms and concepts

This section describes the various terms and concepts that are central to the project and that will be used throughout the report. Note that the terms *instance*, *row* and *data point* will be used interchangeably throughout the report. Similarly, the terms *label* and *class* are both used to describe different categories of data. We also use the terms *impression* and *exposure* interchangeably when describing a single event of a user being exposed to an ad.

3.1.1 Click-through and click-through rate

A *click-through* is the event of a user clicking on an advertisement and being taken through to the advertiser's site. The *click-through rate (CTR)* of an online advertisement is defined as the ratio of the number of times the ad was clicked to the number of exposures. Thus, a click-through rate of 100% means that each exposure yielded a click-through. Typically, a high click-through rate signifies a successful advertisement, but might also indicate that the advertisement has been subject to fraud.

3.1.2 Conversion and conversion rate

An advertisement typically has the intention of making a viewer perform a certain action, e.g. purchasing something on the advertiser's site. Such an action is called a *conversion*. From this we can define the *conversion rate* of an advertisement as the ratio of the number of conversions that occurred to the total number of times the ad was viewed.

3.1.3 Pay-per-click

Pay-per-click (PPC) advertising is a cost model where an advertiser, having one or more ads published on different websites, pays each ad publisher a certain amount of money every time a click is performed on its ads (known as the *cost-per-click*, or *CPC* for short). Advertisers place bids on how much they are willing to pay for each click, and publishers select the highest bidder. PPC is a commonly used cost model for online advertising, and the primary revenue source for a number of large search engines. For instance, Google Inc. employs PPC in its AdWords program, where advertisements are presented along with the search results based on keywords in users' search queries. In AdWords, the CPC for an ad is dependent on the set of keywords the advertiser chooses to associate with the ad. This is because keywords are priced according to attributes such as their historical click-through rate and search volume. The AdWords CPC also factors in the targeted language and geographical area [2].

3.1.4 Pay-per-impression

Analogous to pay-per-click, the *pay-per-impression (PPM)* cost model charges an advertiser for every exposure of an advertisement.

3.1.5 Pay-per-action

Another cost model follows a pay-per-action scheme, where advertisers are billed according to the number of conversions that an advertisement has generated.

3.1.6 Click fraud

Click fraud is a fraudulent activity against the PPC cost model where fake clicks are placed on advertisements to increase their total cost. These clicks are fake because they do not represent a genuine interest in the ad content and thus have a zero-probability of conversion. Fake clicks can either be generated automatically by a computer system (so called "click bots") or input manually by hiring cheap labor for clicking ads (so called "click farms").

There are two general scenarios of click fraud:

- A dishonorable ad publisher generating fake clicks on the ads it hosts to boost its own revenue
- A malicious competitor or other party generating fake clicks to deplete the advertiser's budget

By definition, click fraud inflates the CTR. This has a negative impact for PPC advertisers, as the CPC increases proportionally with the CTR.

3.1.7 Impression fraud

Impression fraud is a similar concept to click fraud, but instead concerns generating fake impressions. In the case that the advertiser is billed according to the pay-per-impression cost model, impression fraud has the same intentions and effects as click fraud has for the PPC model. However, in the more common PPC cost model (where advertisers are not charged per impression, but per click), the incentives for direct monetary gain is not present. Instead, the main intent of the perpetrator is to lower the CTR of an advertisement and subsequently lower its ranking in PPC search engine results. In the event that the CTR declines below a certain threshold, some advertising networks (most notably Google AdWords) will automatically disable the advertisement. Since the CPC is proportional to the CTR, another beneficial outcome for the perpetrator is that impression fraud enables him/her to buy the affected keywords at a lower price.

3.1.8 Click bots

A click bot is a piece of code designed to launch a click attack against PPC advertisements. There are two general classes of click bot. The first is an actual, standalone bot that can be bought and set to run on one or more servers. A click bot can also be distributed as a piece of malware that infects individual computers and participates in distributed click attacks under the coordination of some central entity (the "bot master"). Such an attack achieves high IP diversity, which (especially when performed slowly over time) makes it significantly harder to detect. For instance, Clickbot.A[9] operates in this manner.

3.1.9 MapReduce

MapReduce [10] is a model for performing scalable computation on large amounts of data. As the name suggests, the model is built around computation using two operations: *map* and *reduce*, commonly found in functional programming languages. Each map operation applies computation to a key-value pair, and the result is one or more key-value pairs that are fed as input to the reduce step. Each reduce operation receives a list of key-value pairs which share the same key, and aggregates (reduces) these pairs into one or more values for this key. The type signatures of both computation steps are shown in (1) and (2) [10].

Take for instance a MapReduce application that counts the occurrence of words in a piece of text. Given a line of text, each map operation would emit the pair $(w, 1)$ for each occurrence of the word w on that line. Each reduce operation i receives results from the map step that share the same key w_i - that is, $(w_i, \{1, 1, ...1\})$ - and sums the list of 1:s to yield the total number of occurrences of the word w_i in the text. Furthermore, since map and reduce operations amongst themselves are independent of one another, MapReduce allows for a highly distributed system where individual map and reduce operations run on different nodes in a potentially very large cluster of commodity hardware.

$$map(k_1, v_1) \rightarrow list(k_2, v_2) \quad (1)$$
$$reduce(k_2, list(v_2)) \rightarrow list(v_2) \quad (2)$$

As a performance optimizer, MapReduce also introduces the concept of *combiners*. A combiner is essentially an intermediate reducer that performs a local reduction of key-value pairs that would eventually get transferred to the same

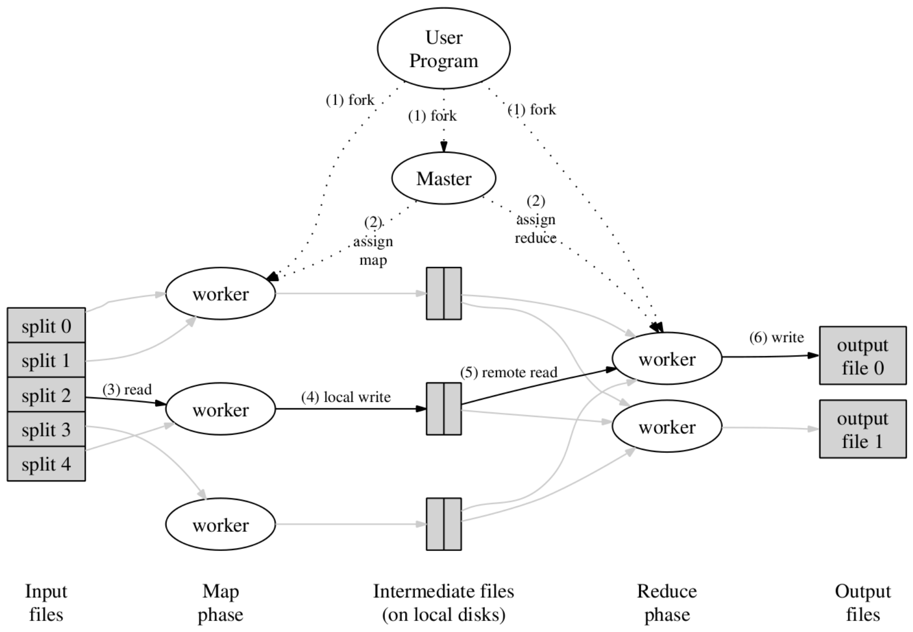


Figure 1: MapReduce overview (taken from [10])

reducer. In case the supplied reduce function is both associative and commutative, it can be used as a combiner. By performing this reduction locally, we can lower the amount of data that need be transferred over the network.

3.2 Classification approaches

If data describing fraudulent behavior can be found by manual search, then this data can be used to identify similar data points that should also be seen as fraudulent. A *classifier* builds a model from a set of training data, in which individual data points have been correctly identified as belonging to a certain class. This model is subsequently used to classify new data points [6].

One advantage with a classification approach is that once a model for classification has been built, the classification of instances is usually quite fast. Instance classification can also be easily distributed, as classifier instances on separate nodes in a cluster can work on copies of the model and subsets of the data set. A more detailed description of this process will be given in Section 5.

3.2.1 Unsupervised or supervised classification?

Classification algorithms can be categorized according to what they expect the training data to look like, and subsequently how they build classification models and classify instances. *Unsupervised* classification algorithms work without any training data at all, and thus are especially useful in applications where prelabelled data is hard to produce. One can view any clustering algorithm as an unsupervised classification algorithm.

Supervised classification algorithms rely on being supplied with a set of prelabelled data from which a model can be derived. *One-class classifiers* is a special case of supervised classification algorithms, where training data for a single label is used to partition the input into two sets: data that aligns with the label and data that doesn't (thus, a one-class classifier performs outlier detection). The training data for one-class classifiers often describes "normal" behavior patterns, as data for such patterns per definition constitutes a significantly greater portion of a real world data set than outliers and is thus easier to produce.

In Section 6, we will evaluate algorithms from each of these three categories.

3.2.2 Neural networks

Neural networks simulate the activities of neurons within a nervous system. A neural network consists of several layers of *artificial* neurons, with the first and last layers being referred to as the *input* and *output layers*, respectively. The layers between the input and the output layers are often referred to as *hidden layers* [27].

Each neuron has an activation function, which in its basic form is a weighted sum of the neuron's inputs. The activation function determines whether that neuron should emit a value or not. A common way to train neural networks is to adjust the weights of the activation function through the use of a back propagation process. This process compares the output of the network to the output that was expected and calculates an error for each of the output neurons. The weights of the entire network are subsequently updated to minimize the error.

Classifiers based on rudimentary neural networks (consisting of one or two layers and a linear activation function) cannot handle data that is not linearly separable. For this, classifiers based on *multilayer perceptrons* can be used instead. A multilayer perceptron is a form of neural network that encompasses more than two layers of neurons and a non-linear activation function, which allows this type of neural network to work on data that is not linearly separable [27].

3.2.3 Classification trees

A classification tree [22] is a specialization of a decision tree, where each leaf denotes a class which data instances can belong to and each inner node represents an attribute of the data. The number of children of a given node is equal to the number of values that the attribute for this node can take on (in the case of continuous attributes, this depends on the discretization of the attribute). The data instances belonging to a certain class can be described by the path from the corresponding leaf to the root node (see Figure 2). Conversely, after building a classification tree from a set of training data, further data instances can be

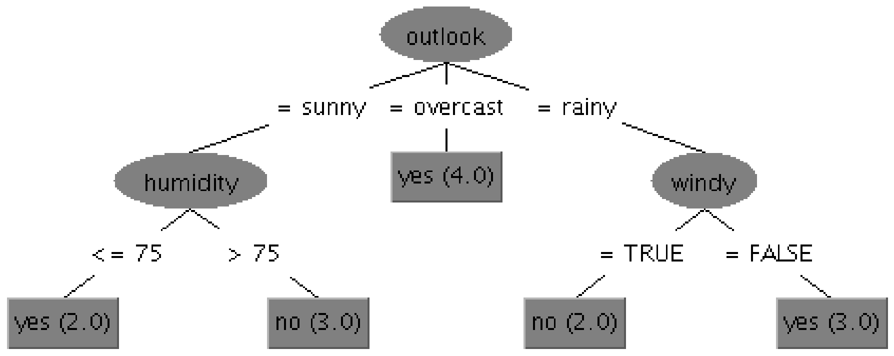


Figure 2: A classification tree

classified by following edges based on the attribute values of each instance until eventually ending up in a leaf node.

3.2.4 Association rule classification

Association rule classification [23] applies association rule learning to the problem of classifying data. Association rule learning is concerned with finding rules that define correlations between features in a data set D. Note that a feature is not equivalent to an attribute; rather, it is a single Boolean statement concerning a particular attribute (e.g. *outlook* is an attribute, while *outlook* = sunny is a feature). By producing rules that define correlations between a set of features and a set of classes, we can subsequently use these rules to classify unseen data.

Given a set of features $F = \{f_1, f_2, \ldots, f_n\}$ in our data set, we can generate a set of rules $R = \{r_1, r_2, \ldots, r_m\}$ where $r_i = (A_i \Rightarrow C_i)$ for two mutually exclusive sets of features A_i and C_i ($|C_i| = 1$), referred to as the *antecedent* and the *consequent*, respectively. To explain how this is done, we first need to make two important definitions.

Definition. Given a data set D with k data points and a set of features F, we define the *support* $S(X)$ of $X \subseteq F$ as the proportion of data points in D that exhibit the features in X. More formally,

$$S(X) = \frac{|\{d_i \in D : X \subseteq f(d_i)\}|}{k}$$

where $f(d_i)$ is defined as the set of features exhibited by data point d_i. Subsequently, we can define the support of a rule $r = (A \Rightarrow C)$ as the proportion of data points that satisfy both the antecedent and consequent feature sets, that is,

$$S(r) = S(A \cup C)$$

The first step in constructing association rules entails selecting the feature sets whose support is above a given minimum s_{min}. Let us denote this set of frequent feature sets by $F*$. Note that we need not consider all $2^{|F|} - 1$ possible non-empty subsets of F in this step since it holds that $X \subset Y \Rightarrow S(Y) \leq S(X)$, that is, if Y has an above-minimum support, so does all subsets of Y [23]. This observation allows us to reduce the size of the search space considerably. We start by building subsets X_i containing a single feature from F (thus, initially $1 \leq i \leq |F|$). We then discard all sets X_i for which $S(X_i) < s_{min}$, and combine the remaining sets into feature sets of size 2. This process is repeated for increasing set sizes until no new feature sets with support greater than s_{min} are found (that is, we discard all X_i).

Definition. Given a rule $r = (A \Rightarrow C)$ with support $S(r) = S(A \cup C)$, we define the *confidence* $C(r)$ as a measure of how well r reflects the entire data set.

$$C(r) = \frac{S(r)}{S(A)}$$

For instance, if $C(r) = 0.75$, three out of four data points that exhibit all the features in A will also exhibit all the features in C.

The confidence measure allows us to select a set of rules that best reflect the entire data, given by a minimum confidence level c_{min}. Specifically, for each feature set $X \in F*$ and every subset A of X that contains all except one of the features in X, we construct a rule $r = (A \Rightarrow X \setminus A)$ if and only if $C(r) \geq c_{min}$.

Once a rule set is generated, we can apply it to unclassified instances. The application of the rule set can be done in a number of different ways. For instance, we can consider all rules and classify instances by popular vote. We can also consider the rules in some order of relevancy and apply the first rule for which the antecedent is satisfied.

3.2.5 Support vector machines

The model of a support vector machine consists of an $n-1$-dimensional plane separating two sets of data points (each set corresponding to a class label) in an n-dimensional attribute space. This $n-1$-dimensional plane is referred to as a *hyperplane*. The separating hyperplane is constructed in such a way that the distance to the closest adjacent data points is maximized. Figure 3 depicts two hyperplanes, H_1 and H_2, that both separate the sets of white and black points. However, H_2 also maximizes the distance to adjacent data points.

After constructing a model based on training data, classification of further data points is performed by determining which side of the hyperplane each point is located on and assigning it the class label that corresponds to that side.

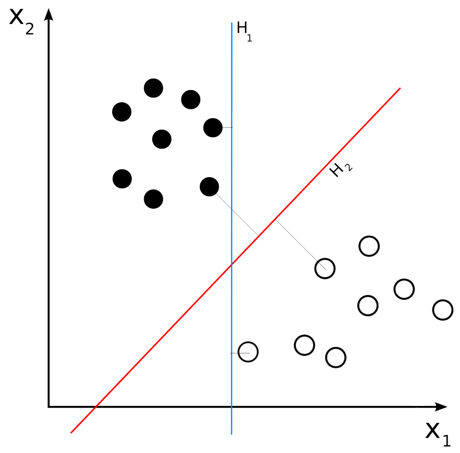


Figure 3: Two hyperplanes, H_1 and H_2 separating data instances in a two-dimensional attribute space

In the event that the training set is not linearly separable due to a few oddly placed data points (for instance, a black point somewhere within the set of white points in Figure 3), the placement of the hyperplane becomes a trade-off between maximizing the distance to adjacent points and minimizing the distance to the oddly placed points. In the case that the entire training set is laid out in such a manner that linear separation does not produce a meaningful model for classifying further data points (e.g. if one set of points completely encircles another, as depicted in Figure 4), we can instead employ a *kernel function* to map the data into higher dimensions [12]. The *Radial Basis* kernel function used in Figure 4 introduces an additional dimension (the distance from the origin) to produce a three-dimensional, linearly separable data set.

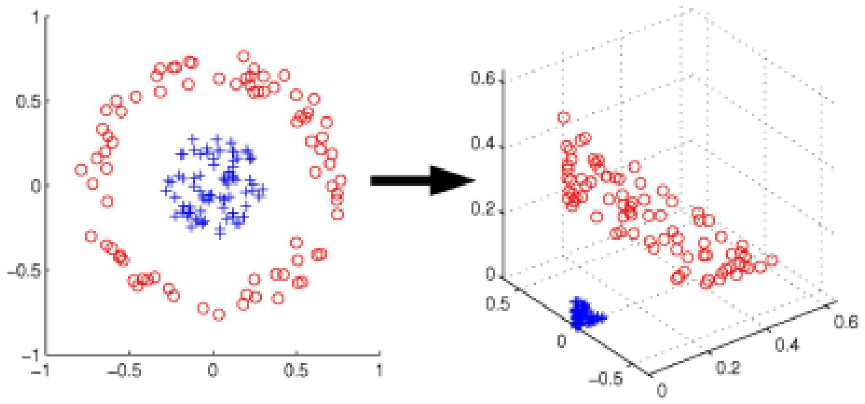


Figure 4: Mapping data to higher dimensions using a *Radial Basis* kernel function (taken from [17])

3.2.6 Bayesian classification

Bayesian classifiers apply Bayes' Theorem [21, 25] to the problem of classifying data instances. To introduce Bayes' Theorem to the unfamiliar reader, a short example follows.

Example. Assume that we have a data set from which we can derive a frequency distribution for the occurrence of a certain feature (and hence a probability distribution for said feature by normalizing the frequency curve). As an example, let this feature be the color of cars as we observe them on the highway from a distance. We can expect that some colors (such as red and black) are more frequent than others (such as purple); specifically, we know that two out of ten cars are yellow. From previous experience we also know that one out of twenty cars are taxis (assuming that the reader is not based in New York, NY). This is also known as the *prior probability*. We also know that half of the taxis (belonging to a certain cab company) are yellow. This is also referred to as the *likelihood function*.

Let Y denote the event that a given car is yellow, and T that a given car is a taxi. From the data given above we have:

$$\begin{aligned} P(Y) &= 0.2 \\ P(T) &= 0.05 \\ P(Y \mid T) &= 0.5 \end{aligned}$$

Since we are observing the cars from a distance, we are only able to make out their color. Thus, we now want to calculate the probability that a car is a taxi, given that it is yellow. That is, we wish to calculate $P(T \mid Y)$ (also known as the *posterior probability*) and to do this we can apply Bayes' Theorem directly:

$$P(T \mid Y) = \frac{P(Y \mid T) \cdot P(T)}{P(Y)} = \frac{0.5 \cdot 0.05}{0.2} = 0.125$$

Thus, the probability that a yellow car also is a taxi is 12.5%.

Thus, given an instance exhibiting the features $f_1, f_2, \ldots, f_n$, a Bayesian classifier can predict its class membership by finding the class C_i for which the posterior probability $P(C_i|f_1, f_2, \ldots, f_n)$ is maximized.

In real world applications, we practically always work with multi-dimensional data. Thus, we have more than one feature to take into account in our classification algorithm. But this situation grows rather complex, as the features have interdependencies. If we can assume that all features are independent, we can calculate $P(C|f_1, f_2, \ldots, f_n)$ by simply applying the multiplication rule for probabilities [21]:

$$P(C|f_1, f_2, \ldots, f_n) = P(C|f_1) \cdot P(C|f_2) \cdot \ldots \cdot P(C|f_n) = \prod_{i=1}^{n} P(C|f_i)$$

This is the assumption made by the *naïve* class of Bayesian classifiers, that the presence of feature A has no effect on the probability of the feature B also being present. In other words, the probability of occurrence of each feature X is assumed to be *independent* of the presence of every other feature $Y \neq X$. This assumption is generally false; if we encounter the word "Chalmers" in a piece of text, we can expect it to be more probable that the words "university" and "Gothenburg" are also present in the text than words like "pigeon" and "xylophone" occurring. Still, naïve Bayes classifiers have been proven to perform very well in practice [24].

3.2.7 Cluster-based

Cluster-based algorithms work according to a simple concept: similar data points can be grouped together in clusters and data points that do not belong to a cluster are identified as outliers.

A simple clustering algorithm is k-Means, which works as follows. First, k *centers* are chosen either randomly or (in more elaborate variants) using some heuristic. Thereafter, each data point is assigned to the nearest center (thus, forming a *cluster* of points around each center). When all data points have been assigned to a center, each center is relocated to the centroid of the corresponding cluster. This process is repeated iteratively for a predetermined number of iterations, or until the location of each center remains unchanged between two iterations. Often, outlier identification will be performed in a final iteration marking all data points farther than a maximum distance d from its designated center.

At first glance, it would seem that k-Means is not suitable for distributed implementation due to the distance calculations performed. However, distances between points are never calculated, as it is only necessary to calculate the distance from each point to a relatively few number of centers. [4] describes how a distributed version of k-Means was implemented for the Apache Mahout machine learning framework.

k-Means can also be used as a supervised classification algorithm, where a training set is used to determine the location of the cluster centers (each cluster represents a single class). Classification is subsequently performed by assigning a point to the nearest cluster center.

3.3 Outlier detection

Outlier detection is concerned with finding data points that have some have some discerning quality that make them deviate significantly from a given norm. Given that fraudulent behavior can be seen as deviating data points in a set of recorded activities, it should be possible to apply outlier detection to identify such behavior. There are a number of different approaches to outlier detection, which will be described in this section.

3.3.1 The distance-based and density-based approaches

The distance-based approach uses the distance between data points to define outliers according to one of three similar definitions [6]:

- An outlier is located further than a distance d away from at least a fraction p of the other data points.
- An outlier belongs to the n data points with the furthest distance to its k:th nearest neighbor.
- An outlier belongs to the n data points with the furthest average distance to its k nearest neighbors.

Density-based algorithms work according to the same principles, but use the relative density of the objects (i.e. how many objects are within a range r) rather than pure distance. Note that both approaches only work on data where the interpretation of distance is meaningful, i.e. for numerical data. Also, this approach scales poorly in a distributed setting. The reason for this is that to calculate the distance from a data point to every other data point, each distributed node needs to keep a local copy of the entire data set.

3.3.2 The greedy approach

The greedy approach [15] works towards minimizing the *entropy* of the data set, iteratively removing data points that provide the most significant contribution to the overall data entropy. Thus, such data points are identified as outliers. However, since calculating the entropy change requires complete knowledge of the entire data set, the greedy approach is not suitable for a distributed setting.

3.3.3 Attribute Value Frequency

Attribute Value Frequency (AVF) [20, 13] is a simple approach to outlier detection which ranks each data point by the frequency of its attributes. That is, AVF computes for each data point X_i a *score* which is the averaged sum of the number of times the value of each attribute X_{il} occurs in the data set. If X_i has m attributes, the score calculation formula can be described as:

$$s(X_i) = \frac{1}{m} \sum_{l=1}^{m} c(X_{il})$$

where $c(X_{il})$ is the number of times the value of the l-th attribute of X_i occurs in the data set. Data points with low AVF scores are subsequently identified as outliers.

AVF works well in a distributed setting, as the score of each data point X_i can be calculated independently of all other data points. Koufakou et al. [20] demonstrate an AVF implementation using two MapReduce iterations - the first iteration calculates $c(X_{il})$ for each data point attribute and the second iteration calculates $s(X_i)$ as given above (we observe that the authors omit the $\frac{1}{m}$ normalization step, since m is constant). The reduce operation of the second operation simply sorts the scores ascendingly.

3.4 Related work

This thesis is concerned with means of handling online advertising fraud by identification of fraudulent behavior through data mining. This section describes a number of other, proactive approaches to deal with fraud.

3.4.1 "Premium" clicks

Juels et al. [19] presents an alternative to detection and filtering in which only legitimate clicks are accepted by requiring that each client authenticates its clicks by providing an attestation of legitimacy, which the authors refer to as a "coupon". The user is awarded with such a coupon whenever he or she performs a conversion action (e.g. a purchase), and the coupon is subsequently stored in a browser cookie. Such an approach has its drawbacks. For instance, the coupon is lost once the user clears its browser cookies, and the coupon does not transfer between different browsers on the same machine.

3.4.2 Client-side validation of advertisement embedding code

Gandhi et al. [14] describes the concept of validating that the code used for embedding advertisements does not contain malicious code that generates spurious clicks. Ads are commonly embedded dynamically using JavaScript code, which also provides additional functionality such as tracking. However, such implementations are also susceptible to script injection attacks. There are a few useful techniques for combating this. Commonly, ad delivery systems can avoid putting their JavaScript code in global namespace where it can be called by malicious code, and instead declare and call functions anonymously. Furthermore, by embedding ads in `<iframe>` elements, browser security policies will prevent external code from accessing their content.

3.4.3 Alternative cost models

There has been development of alternative cost models that are more resistant to click fraud than the pay-per-click model. As previously mentioned, the *pay-per-action (PPA)* cost model only charges advertisers for every conversion that occurs. Thus, in a PPA advertising platform the publisher needs to be able to track conversions in order to correctly charge the advertiser. Since the conversion process is much more complicated than a simple click-through, the perpetration becomes significantly harder to automate. Furthermore, if the only conversion action is to make a potentially costly purchase, the incentive of monetary gain disappears. However, the PPA model does not entirely eliminate click fraud [8]. For instance, if a conversion action entails e.g. signing up for a free trial or newsletter, a perpetrator would simply perform false signups in order to generate conversions. Employing *captchas* (a type of challenge-response control field) in the signup process will eliminate traffic from bots, but if the monetary incentives are large enough a perpetrator might be motivated to hire people to perform signups manually.

4 Problem analysis

This section begins by listing the requirements on the system and describing the pre-existing fraud detection within Rich. We continue by describing a partitioning of user behavior and conclude this section by describing potential limitations of the way user identification is currently being performed within the system.

4.1 Requirements

Since Burt processes large amounts of data on a daily basis, the fundamental requirement on the system is scalability. Burt estimates that the system will be handling up to 30 million ad exposures per day on large campaigns. Thus, this restricts us from performing computationally expensive operations within the system.

Furthermore, Burt strives towards following the guidelines of the Interactive Advertising Bureau (IAB) for all their products and has mandated that the click fraud detection system should adhere to IAB's click measurement guidelines [18].

These were the only two mandatory requirements given by Burt. A number of additional requirements were subsequently identified. The complete set of functional and non-functional requirements is given in Table 1 and 2, respectively. Since the project was carried out according to an agile approach, a complete set of requirements was not elicited up front but rather emerged as the project progressed.

ID	Requirement	Class
F1	The system shall detect fraudulent user behavior	Mandatory
F2	The system shall be able to use previously acquired data to accurately detect fraudulent behavior	Desired
F3	The system shall provide an administration interface that provides performance measures	Desired
F3.1	The provided administration interface shall provide the ability to add data that defines patterns of fraudulent behavior	Desired
F3.2	The provided administration interface shall allow for modification of previously added behavioral data	Optional
F3.3	The provided administration interface shall present additional data, such as frequency of hits from IP ranges	Optional

Table 1: Functional requirements

ID	Requirement	Class
NF1	The system shall be able to process large amounts of data in a scalable manner	Mandatory
NF2	The system shall accept input in a format provided by preceding Rich components and output data in a format that consecutive Rich components accept	Mandatory
NF3	The system shall comply with IAB's click measurement guidelines	Mandatory

Table 2: Non-functional requirements

4.2 Existing fraud detection within Rich

Rich provides a rudimentary fraud detection mechanism built into the log parser that works according to a gray-list approach similar to rate-limiting techniques implemented on SMTP servers. If the parser detects several sessions in rapid succession originating from the same user, it will keep this user gray-listed until a predefined amount of time has passed before the next session from this user is seen. All sessions that originate from gray-listed users are marked as *spurious* in the output from the parser to the analysis step (see Section 5.2 for further description of the Rich system architecture). Thus, it is not detecting click fraud but impression fraud.

4.3 Partitioning user behavior

The core of the problem analysis is choosing a definition by which user behavior will be partitioned into "good" and "bad" according to their *traits*, e.g. "frequent clicker" or "initiates sessions on an irregular basis". In this section, the definition chosen for this thesis will be described and motivated. Note that such a definition is not constant, but will change over time with alterative user behavior. For this reason, we opted for a very dynamic system design as will be described in Sections 5.3 and 5.

Tuzhilin [26] covers a number of different definitions of click behavior, and the definition used for the purpose of this thesis is similar to the definition that Tuzhilin indicates as being used by Google at that time. As the existing data set contained insufficient data about user click-throughs, we opted for a broader view of fraud by including impression fraud as well as "test traffic" generated by advertisement agencies themselves. The lack of click-through data is caused by the fact that click-through tracking in Rich must be manually enabled by agencies and that not all ads have a click-through action. This broader view of fraud allows our system to also cover fraud on ads that are billed according to the PPM cost model.

We partition behavior into three categories: *legitimate, benign* and *fraudulent.* Legitimate behavior is generated by a human being with a genuine interest in the content of the advertisement. A legitimate click-through may thus eventually lead to a conversion. Benign behavior occurs "by accident" (e.g. a

Legitimate	**Benign**	**Fraudulent**
• Lasting sessions which occur infrequently • Click-throughs occur even less frequently • Notable variance in observed behavior	• A couple of click-throughs in rapid succession within a single session (unusual, but highly repetitive) • Multiple click-throughs within a single session (unsual and non-repetitive)	• Short sessions • Low variance in observed behavior, e.g. time between sessions • High click-through rate, or... • ...very high session count but almost no click-throughs (in a short period of time)

Table 3: Expected user behavioral traits, grouped by category

user habitually double-clicking on an advert), while fraudulent behavior occurs through deliberate means of causing harm. The partitioning into these three categories is not clear-cut. Suppose a genuinely interested user clicks on an advertisement several times during the same browsing session to review the content on the advertisement's landing page, perhaps to compare it to other offerings (in more extreme cases, this is referred to as "compulsive comparison shopping"). If we reduce ourselves to partitioning only into either legitimate or fraudulent behavior, in which partition does this user belong? Clearly, the user has exhibited a genuine interest in the contents of the advertisement, but has also cost the advertiser a larger amount of the advertisement budget with no higher probability of conversion. For the system described in Section 5, we have opted for a two-partition solution which attempts to classify benign behavior as legitimate.

Table 3 attempts to group observable user behavioral traits into the three categories described above. We can expect sessions from non-fraudulent users to occur at most a handful times a day (and that mostly last at least a few minutes) and that click-throughs seldomly occur. Furthermore, notable variance in observed behavior (such as duration of sessions and interactions with advertisements) are more likely to be exhibited by non-fraudulent users than those whose intent is to repeatedly click an ad or refresh a web page.

Contrarily, we expect fraudulent users to have short sessions occurring frequently and often if not always generating a click-through (thus perpetrating click fraud), or an extremely large number of sessions in rapid succession without any click-throughs at all (thus perpetrating impression fraud). In either case, we expect these sessions to be regular in their nature - e.g. have similar durations and temporal spacing.

However, the gray area in between these two extremes is more complicated. We have previously discussed two kinds of benign users: “the double-clicker” and “the compulsive comparison shopper”. The duplicate clicks of the former can easily be filtered out in real-time without any further consideration, but the behavior of the latter obviously borders on being fraudulent. However, benign users usually do not exhibit the excessively repetitive behavior of fraudulent users.

4.3.1 Attributes of interest

Based on Table 3, we enumerate a set of attributes of interest for fraud detection, which are aggregations of output from the Rich log parser. A number of these attributes were not originally in the parser output, and thus had to be introduced by extending the parser's functionality. Finally, a few of the attributes given below are simply not possible to extract from the data collected by Rich at the moment.

The data rows outputted from the Rich log parser describe individual sessions. However, building distributions for various attributes of sessions initiated by individual users allows us to identify repeating fraudulent behavioral patterns.

What follows is a full list of elicited attributes.

Total number of sessions: Allows us to detect high-volume users.

Total number of click-throughs: Allows us to detect peaks in click-through rate. This is one of the most important attributes due to its direct link to click-fraud. However, the main issue with this attribute is the lack of click-through data on many Rich campaigns, since (i) click-through tracking needs to be enabled by advertisers on a per-campaign basis, and (ii) some advertisements lack a click-through action.

Distribution of time between sessions: Allows us to detect how often sessions are initiated and if they appear regularly or irregularly.

Number of sessions marked as spurious: The Rich parser flags sessions that occur in a rapid succession as spurious, which is an indicator of impression fraud.

User IP address(es): Since NAT configurations, proxies and dynamic IP assignment using DHCP are commonplace practices, IP-addresses cannot currently be trusted as computer identifiers on the Internet and as such has little direct value for our application. However, we can cross-reference recently logged user IP addresses against various IP blacklists as a heuristic for detecting botnet membership and flag them. The motivation is that a common source of fraudulent clicks (and also one of the hardest to detect) are so called collusions, where clicks are generated by bot nets consisting of malware-infected personal computers. As previously mentioned, this is how Clickbot.A operates - it first compromises individual computers and gives the attacker control over them to carry out a distributed attack. Daswani and Stoppelman [9] suggest that such computers might also be used for other malicious activities (such as sending spam) and hence that there exists a strong correlation between IP addresses involved in performing click fraud and those that have previously been reported as spammers. Therefore, as a part of the analysis we cross-reference user IP addresses with spam blacklists, but also lists of public proxies which perpetrators frequently use to achieve IP diversity and to mask their own identity.

Distribution of session, engagement and visibility times: We produce distributions of session time, engagement time (that is, the amount of time that a user spends interacting with advertisements) and visibility time (the duration during which the advertisement was visible). Out of these three distributions, session time is the most interesting as it allows us to detect sessions of uniform length (which is an indicator of scripted access). Engagement time is useful for detecting possible user interaction in the case that click-throughs aren't being registered, and both engagement time and visibility time can be used in combination with click-throughs to detect whether the ad was visible or engaged upon before it was clicked.

Distribution of time to first click-through and mouse-over: Allows us to detect whether a user frequently and consistently begins interaction with an advertisement at a certain point in time during the sessions (in the case of potential click fraud, most likely at an early point in time).

Number of invalid sessions: An invalid session could e.g. contain a click-through event was sent before an exposure was made or a malformed request. This could indicate fraud perpetrated through direct HTTP requests to the log server.

4.3.2 Attributes omitted due to unavailability of data

In addition to the ones given above, we elicited two more attributes for which data from Rich is unavailable.

Conversion rate: Since click fraud generates clicks but no conversions, a declining conversion rate (that is, the ratio of the number of conversions to the number of click-throughs) can be used to indicate possible click fraud. However, tracking conversion rate requires that individual advertisers report back the number of conversions as a result of the ad campaign - data which is currently not available.

Premium clicks: The concept of premium clicks was introduced in Section 3.4.1. Instead of only registering clicks from authenticated users, we can use the availability of a "premium clicker"-token as a describing attribute for legitimate users. However, this attribute was omitted due to lack of feedback data from advertisers about user conversions.

4.3.3 Attributes omitted for other reasons

User mouse movement patterns: If click fraud is performed by scripting mouse movement, we can assume that the movement will follow some pattern. This requires a bit of extra client-side JavaScript logic for logging mouse movements and pushing them as a sequence of x- and y-coordinates to the server once an advertisement has been clicked, and some backend code for comparing the similarity of the sequences (a simple $O(n)$ algorithm would be to calculate the average distance between pairs of points in both sequences). However, within the time span of the project it was deemed infeasible to be able to develop, deploy and gather enough mouse movement data from a live production system to be able to analyze whether this approach would provide a useful indicator of fraudulent behavior. Hence, this attribute was omitted.

Distribution of activity hours (user's local time): A normal usage pattern would likely imply higher degrees of activity during daytime. We can compare the distribution of users within a timezone with the expected temporal behavior to identify activity peaks during off-hours. However, this attribute was omitted partly due to lack of theoretical justification of its usefulness, and partly due to the lack of the needed data within Rich.

Geographic origin: Detect if a large amount of users are originating from a country outside the geographic area targeted by the campaign. This attribute was omitted since it did not fit in to the model used for detection of fraudulent behavior, as it would require aggregation of data on at least an campaign-per-campaign basis.

4.3.4 Subsequent analysis

Given data for the attributes described in Section 4.3.1, we can also ask a number of follow-up questions:

- Does a lot of fraudulent activity originate from a narrow range of IP addresses and/or from a certain geographical region?
- Is a certain user agent frequently occurring in connection with the fraudulent activity?

The answers to these questions could possibly help us to further improve the quality of our detection system.

4.4 User identification limitations

Each user is identified in Rich using a 12-character uppercase alphanumerical string (in practice, it is encoded in a base-36 integer to for reasons of space efficiency). Thus, the user identifier can take on $36^{12} \approx 4.7 \cdot 10^{18}$ different values. The identifier is generated using the pseudo-random number generator provided by the ActionScript run-time in combination with a platform-dependent method for adding some extra randomness to the resulting value. As previously mentioned in Section 3.4.1, the user identifiers are subsequently stored in Flash Local Shared Objects (LSOs). LSOs (like HTTP cookies) are created on a per domain basis and thus cannot be accessed by Flash applications served from other domains. Typically, this entails that a new user ID will be generated per advertising network that serves Rich-enabled ads. As a consequence, this prevents us from tracking users across different advertising networks. Furthermore, although LSOs are more persistent than HTTP cookies, they are unlikely to be persistent over longer periods of time. This leads to the same user appearing under multiple user IDs, which fragments the user attribute aggregation if performed over longer periods of time (we also need to consider the possibility of user ID clashes in this case). Thus, while aggregating over long periods of time is preferable, these limitations restrict us to aggregating over shorter periods of time.

5 System design and implementation

The following section describes the design of the fraud detection system in a top-down fashion, starting with the architecture of Rich and how it interfaces with the fraud detection system. We then describe the overall system architecture of the fraud detection system and continue by describing each of the system's components in greater detail. Finally, we describe how a training data set was constructed from the existing Rich data.

5.1 Technologies

This section describes the set of technologies that were used in the system.

5.1.1 Hadoop

Hadoop is an open source Java framework for large-scale distributed computations, including an implementation of MapReduce as well as a distributed file system, HDFS. The architecture of Hadoop MapReduce is depicted in Figure 5. Hadoop MapReduce computations take the form of *jobs*, which consist of:

- an XML configuration file specifying e.g. the path to the input file on HDFS
- a *job split* file, which specifies the size of the chunks which the input data should be split into and passed to individual map workers
- implementations of the map and reduce functions

In control of execution is the *job tracker*, which receives job submissions from clients and assigns tasks to a number of *task trackers*, each performing a number of map and/or reduce tasks. The output from the map step is serialized, and a partitioner takes care of assigning the key-value pairs to reduce tasks. The default implementation uses the `hashCode()` definition of the key to partition the pairs, and the key's implementation of the `Comparable` interface to perform intermediary sort.

Hadoop provides three modes of operation: *stand-alone*, *pseudo-distributed* and *fully distributed*. The first two modes are only applicable for single-node setups. In stand-alone mode, Hadoop jobs run within a single process. In pseudo-distributed mode, each Hadoop daemon runs as a separate Java process on a single machine - thus effectively simulating a multi-machine cluster on a single computer. This multi-process setup allows Hadoop to take advantage of multi-core CPUs. However, the intended mode of operation for Hadoop is the fully distributed mode where daemons run on different nodes in a cluster.

5.1.2 Dumbo

Dumbo is a Python interface to Hadoop, allowing developers to write Hadoop jobs using Python rather than Java. Dumbo was developed at Last.fm and is currently used for data analysis within Rich.

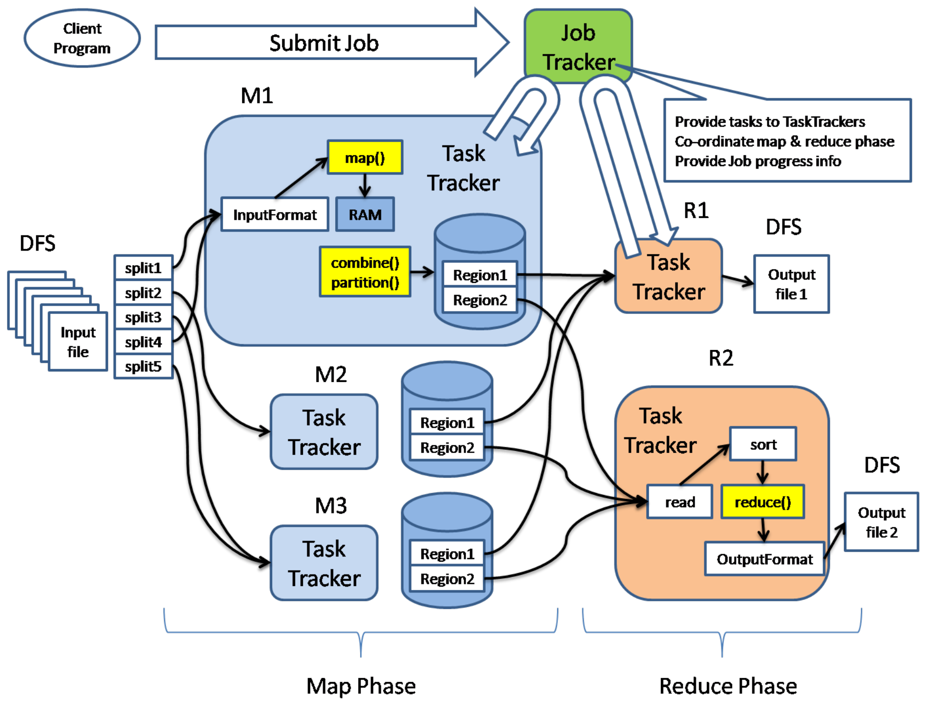


Figure 5: Hadoop MapReduce (taken from [16])

5.1.3 Sinatra

Sinatra is a minimal Ruby web framework with which a simple web application can be defined by writing handlers for different HTTP request verbs (GET, POST, PUT and DELETE) on different routes (the path component of URLs).

5.1.4 jQuery

jQuery is a general purpose JavaScript framework that, among other things, provides a simple and cross-browser compliant API for DOM and AJAX operations.

5.1.5 Highcharts

Highcharts is a JavaScript graph rendering framework that is built on top of jQuery.

5.1.6 Weka

Weka is a machine learning framework developed at the University of Waikato, New Zealand. It features a large variety of machine learning algorithm implementations. Weka also provides a graphical workbench for data analysis and algorithm evaluation.

5.1.7 Implementation languages

Both Hadoop and Weka provide Java APIs for client programs, and therefore Java was chosen as the implementation language for the system's classification component. In addition to its Java API, Hadoop can also interface to almost any executable using the Hadoop Streaming API. For the data transformation tasks that take place during the system's preprocessing phase, it was decided that Python was more suitable than Java due to its compactness and rapid prototyping abilities.

The server-side implementation of the web-based administration application is written in Ruby, due to ease of use and good availability of web frameworks for the language.

5.2 Integration with Rich

Rich closely resembles a narrowing pipeline (depicted in Figure 6), where each intermediate step performs aggregate computation on the data. Thus, the amount of data that need be processed decreases further down the pipeline. Each Rich-enabled advertisement banner contains a Flash component (henceforth referred to as the *Flash agent*) that emits HTTP requests to indicate the status of the ad (e.g. whether it was clicked or the mouse cursor was hovering the ad) and other environment variables (such as client time). These HTTP requests are sent to a server which produces log files. These log files are then processed by a parser that extracts interesting information about each *session* (that is, a unique ad impression) and sends it along the pipeline. Thereafter, the session information is further processed on Hadoop to yield relevant metrics about the advertising campaign. These metrics are subsequently stored in a database and presented to the end-user through a web application.

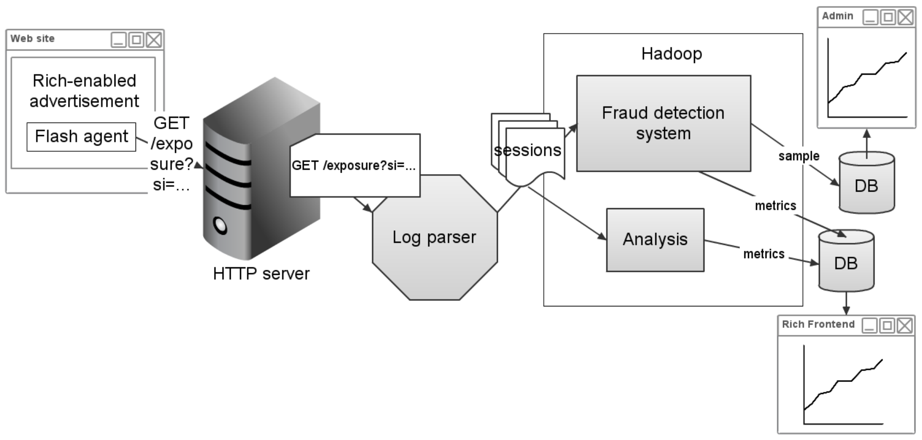


Figure 6: System integration

The fraud detection system runs alongside the analysis step on Hadoop and takes as input the session information outputted by the parser, as this data is presented at a suitable level of granularity. Introducing the system at an earlier stage leaves us with the task of extracting data from raw web server logs (which is the parser's duty) on top of performing analysis on this data, all within the high throughput requirements that are put on this part of the pipeline. This is simply infeasible. Likewise, placing the click fraud detection system after the analysis step would make little sense, as too much of the necessary attributes described in Section 4.3 have been lost in aggregation at this point.

5.3 Choosing classification as detection approach

Section 3 describes a number of approaches that could be used for detecting click fraud. Of these possible approaches, it was determined that classification would fit our system best. The main reasons for this were that (i) it's possible to scale by performing classification of each user in a distributed setting and (ii) since the definition of fraud might change over time it's important that the system can adapt to finding new behaviors. This is something for which classification algorithms are better suited than outlier detection algorithms. Thus, we opted for the classification approach when designing the system. In order to determine which classification algorithm was most suitable for our application, we chose to evaluate a number of algorithms on real application data. The results of the evaluation are presented in Section 6.

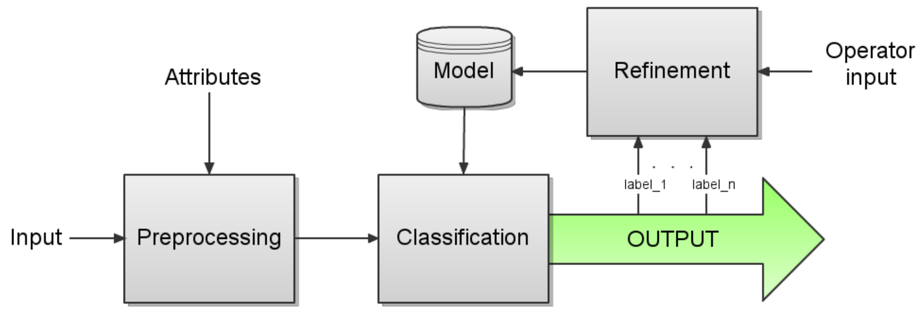


Figure 7: Overall fraud detection system architecture

5.4 System architecture

The fraud detection system works as a three-step process: preprocessing, classification and refinement. The preprocessing step aggregates user data and the classification step performs classification of this data. The last two steps form a feedback loop where the refinement step provides the ability to make improvements to the model used for classification. The overall system architecture is depicted in Figure 7.

5.4.1 Preprocessing

The preprocessing step takes the output from the parser and aggregates a number of useful attributes describing individual users, as given in Section 4.3.1. The preprocessing is implemented as a MapReduce job using Dumbo. Given tab-separated session data, each map task emits the user ID as key and a list of session attributes as value. Because of how MapReduce works, the session attributes belonging to a certain user ID will be passed to the same reduce task where they are aggregated.

5.4.2 Classification and Model

The classification is based on algorithms from the Weka framework, which have been applied in a MapReduce context. In order to determine the most suitable algorithm, several of them were evaluated on data sets of varying sizes. The results of this evaluation are provided in Section 6.

The classification step takes as input the aggregate user information from the preprocessing step and classifies individual users as either fraudulent or legitimate based on a preconstructed model. The model is constructed from a set of *training data*, which was prepared according in a manner described in Section 5.5. The training data is given in the Attribute-Relation File Format (ARFF, see Listing 1), which is the default data format accepted by Weka. Each data row consists of comma-separated columns corresponding to an attribute described in Section 4.3.1, accompanied by an additional label $l \in \{\text{fraudulent, legitimate}\}$

which indicates whether this data row describes fraudulent or legitimate behavior.

The model is built centrally at the initiating job tracker, and is subsequently serialized and distributed to each map task in the cluster through Hadoop's *distributed cache* mechanism. Each of the map tasks subsequently deserializes the model and uses it to classify the data instances that have been allocated to that map task. An individual map task outputs the data instance and a class prediction, and the following reduce phase partitions data instances of different class prediction to different output files. In the final system, the reduce phase is completely omitted for efficiency reasons. This will be further described in Section 6.

```
@RELATION user

@ATTRIBUTE session_count                   NUMERIC
@ATTRIBUTE tot_session_time                NUMERIC
@ATTRIBUTE avg_session_time                NUMERIC
@ATTRIBUTE session_dev                     NUMERIC
@ATTRIBUTE tot_visibility_time             NUMERIC
@ATTRIBUTE avg_visibility_time             NUMERIC
@ATTRIBUTE visibility_dev                  NUMERIC
@ATTRIBUTE avg_time_to_first_mouseover     NUMERIC
@ATTRIBUTE first_mouseover_dev             NUMERIC
@ATTRIBUTE tot_engagement_time             NUMERIC
@ATTRIBUTE avg_engagement_time             NUMERIC
@ATTRIBUTE engagement_dev                  NUMERIC
@ATTRIBUTE spurious_count                  NUMERIC
@ATTRIBUTE click_thru_count                NUMERIC
@ATTRIBUTE invalid_count                   NUMERIC
@ATTRIBUTE avg_time_between_sessions       NUMERIC
@ATTRIBUTE time_between_sessions_dev       NUMERIC
@ATTRIBUTE avg_time_to_first_click_thru    NUMERIC
@ATTRIBUTE time_to_first_click_thru_dev    NUMERIC
@ATTRIBUTE click_count                     NUMERIC
@ATTRIBUTE class                           {legitimate,fraudulent}

@DATA
444,8021,18.06,6.22,...,35.39,19.05,0,0,0,fraudulent
8,224,28.0,23.22,...,3021.14,7205.19,0,0,0,legitimate
```

Listing 1: Rich user data in ARFF format (a few columns in the middle left out for space reasons)

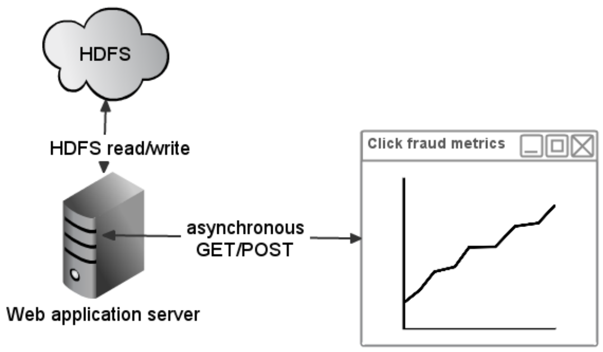


Figure 8: Administration and metrics interface

5.4.3 Refinement

The system provides a web interface with system performance metrics, giving system administrators an overview of the results of the classification and the ability to conveniently manipulate the data set used for training. For reasons of lucidity as well as browser rendering limitations, the data presented is based on a sample of the classifier output containing instances for which the *degree of confidence* lies below a predefined threshold. The degree of confidence describes the probability with which instances were predicted to belong to a certain class. Thus, the administration interface exposes data points which the classifier was unable to confidently classify due to inadequacies in the training set, and consequently the areas of the training set that need improvement. The administrator can thereafter investigate individual data points of interest, and select a set of points to include in the training data as either fraudulent or legitimate behavior.

The administration interface is a Sinatra-based web application. Communication between the client and server is implemented asynchronously using jQuery's AJAX API. The client first queries the server for classification results. Upon receiving this request, the server fetches a sample of the classifier output from an HDFS instance, adds meta-data (e.g. attribute names) and transforms the result into JSON, which is the data format preferred by the graph rendering component. The graph rendering component uses Highcharts to render an interactive scatter plot in which a user can select and review individual data points and request that these be added to the training set as either fraudulent or legitimate. Upon receiving this request, the client pushes JSON data to the server, which appends it to the training set stored on HDFS. The entire data flow is depicted in Figure 8.

5.5 Constructing initial training data

Based on the partitioning of click behavior described in Section 4.3, the next major problem was the initial construction of a set of behavioral data (corresponding to this description) to be used as a training set for the classifier. The construction of training data by generating raw HTTP logs was investigated, but was rejected since it was judged that generated training data inherently lacks the ability to completely reflect the nature of the real data set. Conversely, extracting training data from a data set requires extensive analysis and manual inspection. Rich has collected a large amount of data - in total approximately 150.2 million sessions from 49.8 million users going back to March 2009 (depicted in Figure 9). Thus, manual review of the entire data set is unfeasible. Instead, having the ability to ask arbitrary questions about the data and its aggregations in a quick and straightforward fashion would allow us to extract a significantly smaller subset of the data for manual review. For this, *Hive* proved to be a useful utility. Hive provides an SQL layer on top of Hadoop MapReduce. Listing 2 shows two Hive queries. The first query maps a Hive table to a location on HDFS containing columnar data on tab-separated form, while the second lists the 150 users with highest CTR.

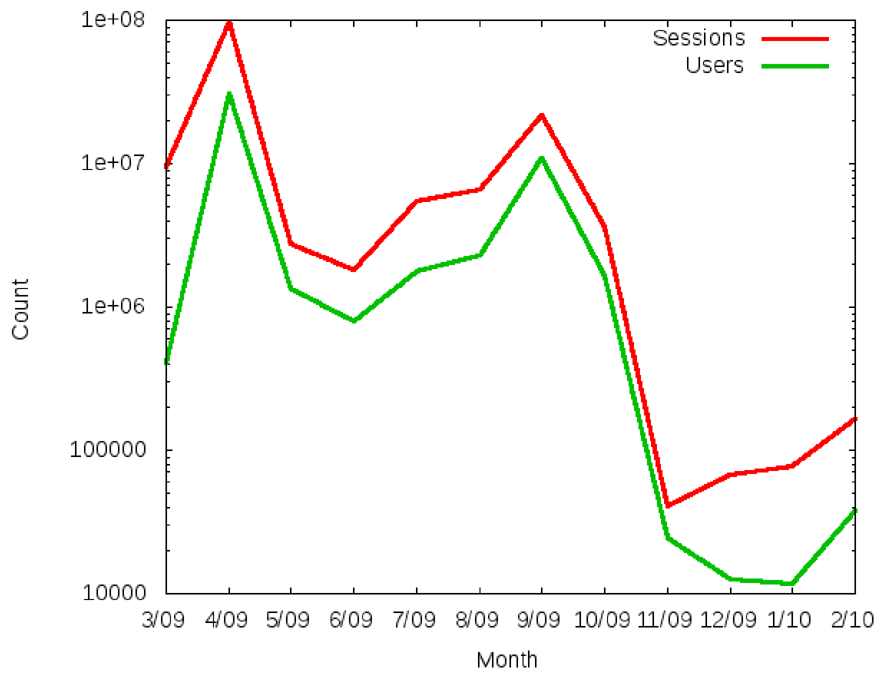


Figure 9: Total session and user count per month

```
CREATE EXTERNAL TABLE users(user_id STRING,
                   session_count INT,
                   total_session_time INT,
                   average_session_time INT,
                   session_time_deviation INT,
                   total_visibility_time INT,
                   average_visibility_time INT,
                   average_time_to_first_mouseover INT,
                   total_engagement_time INT,
                   average_engagement_time INT,
                   spurious_sessions INT,
                   click_throughs INT,
                   invalid_sessions INT,
                   average_time_between_sessions INT,
                   average_time_to_first_clickthrough INT)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE
LOCATION '/output/userdata' ;

-- users with highest ctr
SELECT user_id, SUM(click_throughs), SUM(session_count),
SUM(click_throughs)/SUM(session_count) AS ctr FROM users
GROUP BY user_id
ORDER BY ctr DESC
LIMIT 150;
```

Listing 2: Hive SQL

Once a small initial training set was constructed, a larger training set could be built iteratively by using the current training set to classify a set of unclassified data and extract new training data from the results of the classification.

6 Evaluation of classification algorithms

In order to judge the suitability of the different classification approaches for our use case, we decided to perform a three-phase evaluation of the classifier implementations provided by Weka. This was also done to overcome the lack of openly available studies of classification in the domain of our problem. The results of the evaluation are presented in this section.

In the first evaluation phase (Section 6.1), we evaluate one candidate algorithm from each approach described in Section 3.2 on a small set of prelabelled data to determine their accuracy (i.e. percentage of correctly classified instances). The candidates were chosen by briefly running the majority of classifiers available in Weka and weeding out those that provided too poor of a result (note that not all classifiers were applicable to the type of data we are working with, e.g. those that require strictly nominal input). The following algorithms were chosen:

- Bayesian: NaiveBayes
- Decision trees: RandomForest
- Rule-based: RIDOR (RIpple-DOwn Rule learner)
- Support vector machines: LibSVM (provided by an interface to the libsvm [7] library)
- Neural networks: MultilayerPerceptron
- Clustering: k-Means

Furthermore, in addition to the more common multi-class supervised classification, we also test both unsupervised and single-class supervised classification for the algorithms that support it (k-Means and LibSVM, respectively).

In the second phase (Section 6.2), we test the performance on data sets of varying sizes for the algorithms that yielded the best results in the first evaluation phase. Thus, we eliminate yet more algorithms simply because their accuracy was unsatisfactory.

In the third phase (Section 6.3), we evaluate one of the best-performing algorithms overall (that is, taking into account the results from phase one and two) on varying cluster sizes, given a data set of fixed size.

6.1 Accuracy of classification algorithms

6.1.1 Test setup

As the size of the prelabelled data set is significantly smaller than the data sets used for performance evaluation, accuracy tests were run on a single machine running the Weka workbench. This workbench provides access to detailed information about the classifier's performance. Each algorithm requires its own set of parameters, which have been adjusted in an attempt to maximize the prediction accuracy for this data set.

The following software versions were used:

- Linux 2.6.32
- Sun Java SE Runtime Environment 1.6.0_17
- Weka 3.7.1

The evaluation was performed by partitioning a set of data, prelabelled as either fraudulent or legitimate, into two disjoint sets used for training the classifier and evaluating the outcome of the classification with respect to the prelabelled data, respectively. Instead of performing a simple percentage split, a so called *n-fold cross-validation* procedure was used to improve the accuracy of the test results. In n-fold cross-validation, a model is built using $n-1$ equally sized partitions of the data set. The model is subsequently evaluated on the remaining partition. This process is repeated n times, that is, until every partition has been used for evaluation exactly once. The cross-validation algorithm is outlined in Listing 3. For the tests presented below, $n = 10$ was used.

```
Require: A set $D$ of data points prelabelled with a class
  $P = \{p_1, p_2, \ldots, p_n\}$, a set of equally sized partitions of $D$
  for $i = 1$ to $n$ do
    $S = \{p_i\}$
    $T = D \setminus S$
    Build a classifier $c$ using $T$ as the training set
    Let $r_i$ be the result of evaluating $c$ on test data $S$
  end for
  return The average of all results $\{r_1, r_2, \ldots, r_n\}$
```

Listing 3: Cross-validation

6.1.2 Test results

Before presenting the results, we give a few definitions that will be used in the presentation. In the following text, a *positive* is equivalent to an instance classified as fraudulent, while a *negative* is equivalent to an instance classified as non-fraudulent. A *true positive* is a reported positive that actually is a positive, while a *false positive* is a negative that was reported as a positive by the classifier. Similarly, a *true negative* is a reported negative that in fact is a negative, while a *false negative* is a positive that was reported as a negative. The following terminology is also used:

- TPR (True Positive Rate)= $\frac{\text{true positives}}{\text{true positives + false negatives}}$
- FPR (False Positive Rate)= $\frac{\text{false positives}}{\text{true negatives + false positives}}$
- TNR (True Negative Rate)= $\frac{\text{true negatives}}{\text{true negatives+false positives}}$

- FNR (False Negative Rate)= $\frac{\text{false negatives}}{\text{true positives + false negatives}}$

- ACC (Accuracy)= $\frac{\text{true positives+true negatives}}{\text{all instances}}$

- A_{ROC} (Receiver Operating Characteristic Area): the area under the *ROC curve*. An ROC curve describes how well-balanced the classifier is as a function of the *confidence threshold* $t \in [0, 1]$, that is, the minimum probability (for non-discrete classifiers) required for an instance to be reported as a positive. A perfect classifier would have $A_{ROC} = 1$; that is, $TPR = 1$ and $FPR = 0$ for all values of t. The ROC curves for all classifiers are presented in Figure 10.

- PAP (Portion of Actual Positives): the portion of actual positives in a data set.

- PFA (Portion of False Alarms): the portion of all reported positives which are false alarms.

For our application, we wish to keep the FPR as low as possible (thus maximizing the TNR while still achieving the best possible TPR). The motivation for this is simply that an overreported amount of fraudulent behavior has a direct detrimental effect on the credibility of other measures provided by Rich.

	TPR	FPR	TNR	FNR	ACC	A_{ROC}
NaiveBayes	95.4%	14.3%	85.7%	4.6%	89.4%	0.959
RandomForest	94.4%	7.6%	92.4%	5.6%	93.2%	0.975
RIDOR	95.4%	9.1%	90.9%	4.6%	92.7%	N/A[2]
LibSVM (one-class “fraudulent”)[1]	69.0%	69.0%	31.0%	31.0%	45.7%	N/A[2]
LibSVM (one-class “legitimate”)[1]	2.8%	76.9%	23.1%	97.2%	15.2%	N/A[2]
LibSVM (two-class)	93.5%	8.2%	91.8%	6.5%	92.5%	0.962
MultilayerPerceptron	93.5%	7.6%	92.4%	6.5%	92.8%	0.975
k-Means (supervised)	91.7%	49.4%	50.6%	8.3%	66.5%	N/A[2]
k-Means (unsupervised)	8.3%	0%	100%	91.7%	64.5%	N/A[2]

Data set size: 558 instances (216 “fraudulent”, 342 “legitimate”)

[1]Since one-class classification only uses training data from one class (e.g. either "fraudulent" or "legitimate"), cross-validation over the entire two-class test set is not possible. Instead, each one-class test was split into two sub-tests, which used the "fraudulent" and "legitimate" data as test sets, respectively. The results presented here are the combined results of the two sub-tests.

[2]The ROC Area has no useful interpretation for this classifier.

Table 4: Classifier accuracy

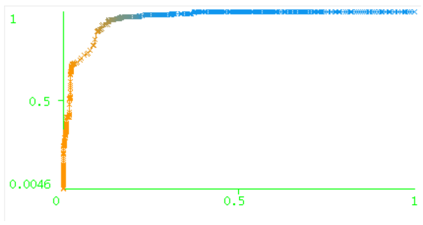
Note that the one-class LibSVM classifier trained with the “legitimate” data set (i.e. in which “fraudulent” instances are seen as outliers) performs significantly worst, surprisingly even worse than unsupervised k-Means! How can a classification algorithm that possesses prior knowledge about its input yield lower accuracy than an algorithm with no prior knowledge at all? This is most likely due to high diffusion in the "legitimate" data set, which causes a large number of false negatives to be reported. Note also that both one-class classifiers are in fact worse than purely random classification of instances! This is in accordance with our intuition that two-class classification should give the most accurate prediction, due to its ability to harvest knowledge from counter-examples in the training data. The majority of the two-class classifiers (RandomForest, RIDOR, LibSVM and MultilayerPerceptron) have very similar accuracy with slight skews towards either higher FPR or higher FNR. Based on these test results, RandomForest seems best suited for our application given its lower FPR.

A low FPR does not necessarily imply a satisfactory result, since the FPR has to be considered in relation to the portion of actual positives in the data. If the portion of actual positives is relatively low compared to the FPR, we can expect a lot of the reported positives to be false alarms. For instance, assume that we have a data set of 10 000 users of which 100 have actually exhibited fraudulent behavior (that is, $PAP = 1\%$) and the remaining 9 900 do not exhibit fraudulence. Using the measures for RandomForest from Table 4, the system will on average correctly report $0.944 \cdot 100 = 94.4$ users and incorrectly report $0.076 \cdot 9900 = 752.4$ users as fraudulent. As can be seen, the number of incorrectly classified positives is significantly greater than the number of correctly classified ones - simply due to the low portion of actual positives in the data. From this, we can conclude that $\frac{752.4}{94.4+752.4} \approx 88.9\%$ of all reported positives are false alarms!

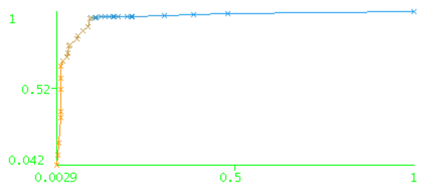
More generally, the portion of all reported positives which are false alarms can be described by the following equation:

$$PFA = \frac{\text{false alarms}}{\text{reported positives}} = \frac{FPR \cdot (1 - PAP)}{PAP \cdot TPR + FPR \cdot (1 - PAP)} \tag{3}$$

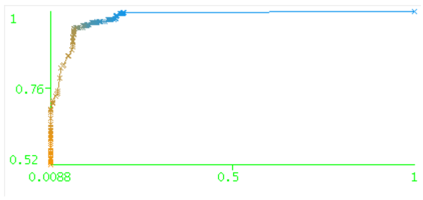
This describes a general hardship of fraud detection: we are looking for a very small needle in a very large haystack. However, all is not lost. By rigorous tweaking of each algorithm's parameters it should be possible to approach more satisfactory results. We can also expect the accuracy to increase as the training data becomes more refined. Given that the data set used for the tests likely has a higher density of points within the "gray" area between the two classes, a lower FPR can be expected when classifying real data. However, the PFA is still likely to be an issue since real data contains significantly less fraudulent instances than the data set used for these tests.



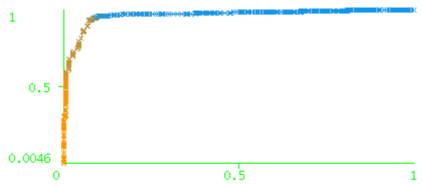
(a) NaiveBayes



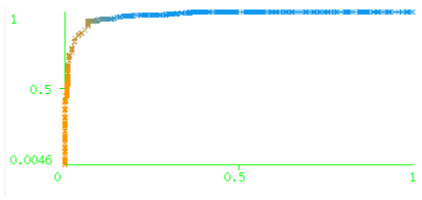
(b) J48



(c) RandomForest



(d) LibSVM (two-class)



(e) MultilayerPerceptron

Figure 10: ROC curves for classification algorithms in Table 4

6.2 Scalability with size of data set

6.2.1 Test setup

All performance tests were run on a cluster of four small Amazon EC2 instances with 1.7 GB of main memory. The cluster setup follows a common pattern for smaller Hadoop clusters: one node (the *master*) acts as both a MapReduce job tracker and an HDFS *name node* [5], while each of the remaining nodes (the *slaves*) acts as both a MapReduce task tracker and an HDFS *data node* [5]. Since EC2 is virtualized on top of a large number of commodity PCs, an instance might be powered by many different hardware configurations over time. Hence, it is not possible to give exact hardware specifications. For this reason, Amazon has introduced the *EC2 Compute Unit*, which according to [1] is equivalent to a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor. A small instance is guaranteed to be allocated 1 EC2 Compute Unit on a single virtual core.

The following software versions were used:

- Linux 2.6.21.7
- Hadoop 0.20.1

- Dumbo 0.21.24
- Python 2.5.2
- Sun Java SE Runtime Environment 1.6.0_14

Data for the test was gathered from parser output for the month of September 2009. Four data sets of varying sizes were used:

- Data set A: 1 million users (approximately 2 million sessions)
- Data set B: 2.5 million users (approximately 5 million sessions)
- Data set C: 5 million users (approximately 10 million sessions)
- Data set D: 10 million users (approximately 20 million sessions)

To account for variances in the test environment (e.g. network load), each test was run twice and the results shown below are averages.

6.2.2 Test results

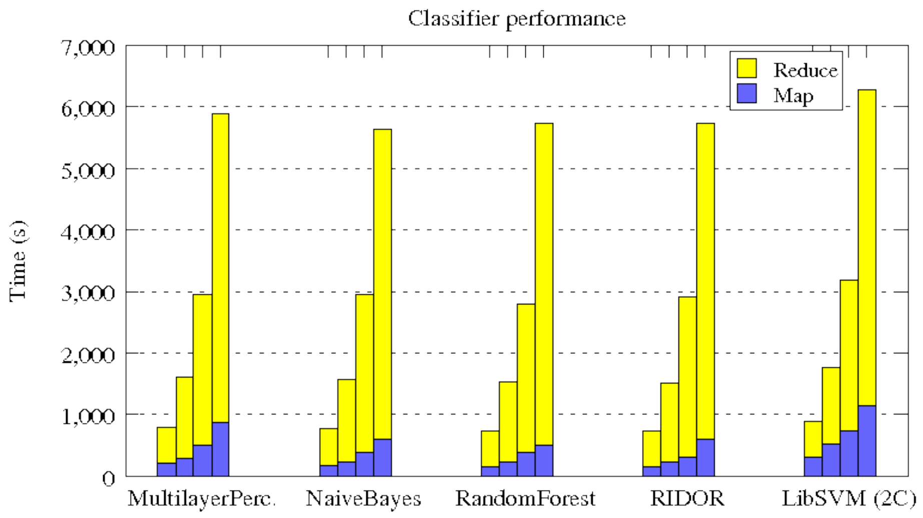


Figure 11: Classifier performance (each bar within a group of four corresponds to data sets A-D, in left-to-right order)

Results (Figure 11) show that all tested classifiers have similar performance, but that the results get spoiled by a very large overhead from the reduce phase (recall the map and reduce phases from the description of MapReduce in Section 3.1.9). This overhead includes the shuffling and sorting of mapper output values before they are handed to the reducer, followed by the actual execution of the reduce tasks which do nothing more than to write the classified data to different files based on the probability distribution.

One might then question why it is even necessary to have a reduce phase in our case, as it only serves the purpose of shuffling data through. And quite rightly so, as it isn't! We can take care of writing output to files from directly within the map phase, and thus eliminate the redundant copying and sorting. However, we cannot simply omit a reducer definition from our job code, as Hadoop in that case would provide an identity reducer for us. Instead, Hadoop provides a parameter for its job configuration that allows us to set the number of reduce tasks to zero, simply by saying `job.setNumReduceTasks(0)`. Now we have effectively eliminated the overhead of the reduce phase depicted in Figure 11. The improved results are shown in Table 5 and Figure 12. From this figure it is apparent that RIDOR, RandomForest and NaiveBayes has a performance advantage in our tests.

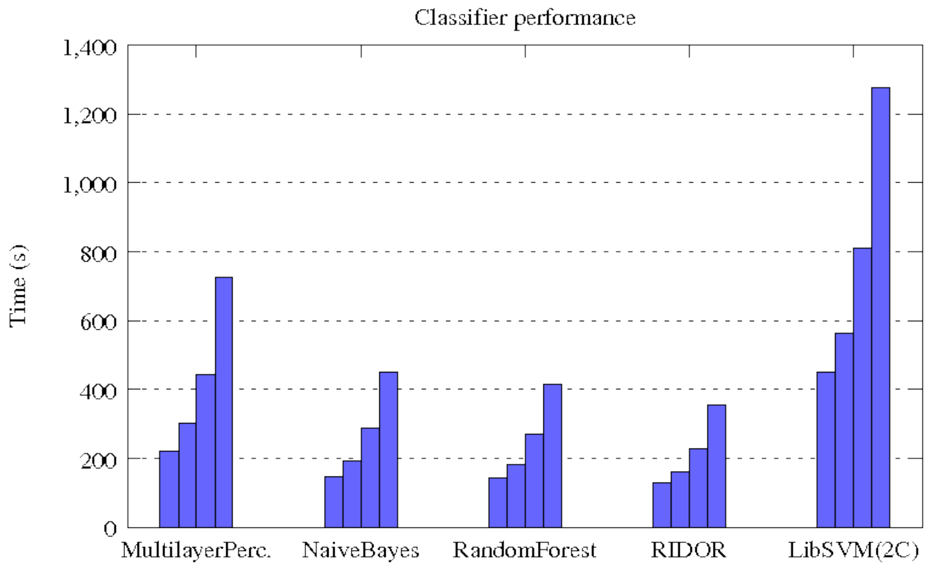


Figure 12: Classifier performance with reduce phase eliminated (each bar within a group of four corresponds to data sets A-D, in left-to-right order)

	A	B	C	D
User data aggregation	18min 56s	34min 9s	77min 28s	158min 23s
Classification				
- MultilayerPerceptron	3min 43s	5min 1s	7min 25s	12min 6s
- NaiveBayes	2min 28s	2min 13s	4min 47s	7min 29s
- RandomForest	2min 22s	3min 2s	4min 31s	6min 57s
- RIDOR	2min 8s	2min 41s	3min 47s	5min 55s
- LibSVM (two-class)	6min 31s	9min 23s	13min 32s	21min 15s

Table 5: System performance

6.3 Scalability with size of cluster

6.3.1 Test setup

The tests in Section 6.2 were all run on a cluster of four nodes. In order to evaluate how well the system scales up and down with the number of machines it runs on, tests were run on data set D with varying cluster sizes on Amazon EC2. The single-node setup was run in pseudo-distributed mode, while the remaining tests were run with Hadoop in fully distributed mode. All clusters were set up identically to those used for the tests in Section 6.2.

6.3.2 Test results

The results are presented in Table 6 and Figure 13. Note that due to MapReduce overhead (e.g. job coordination and copying data between machines in the cluster) we do not halve the time it takes to complete a job by doubling the number of machines in the cluster. Furthermore, the results show that the performance of the classification step does not improve mentionably with the sixteen-machine setup for a data set of this size. However, we still expect the classification step to scale close to linearly if the data set was to further increase in size, as Hadoop start-up and job scheduling would constitute a smaller part of the entire job. It should also be possible to make considerable improvements to the user data aggregation by reducing the amount of data that is being transferred between mappers and reducers - something that can be achieved by the use of a combiner (as mentioned in Section 3.1.9).

The scaling between the one- and two-machine clusters is worse than one might expect. This is mainly due to the way the cluster is configured. In accordance with the test setup described in Section 6.2.1, the two-machine cluster uses a dedicated master node and thus only has a single slave available in computation. The performance gained from this setup comes from the fact that some of the MapReduce overhead can be offloaded to the master node.

	1	2	4	8	16
User data aggregation	416min 9s	388min 32s	158min 23s	134min 40s	92min 51s
Classification	16min 39s	15min 24s	6min 57s	4min 45s	4min 41s
Total	432min 48s	403min 56s	165min 20s	139min 25s	97min 32s

Table 6: Scalability with size of cluster (column headings indicate cluster size in number of machines)

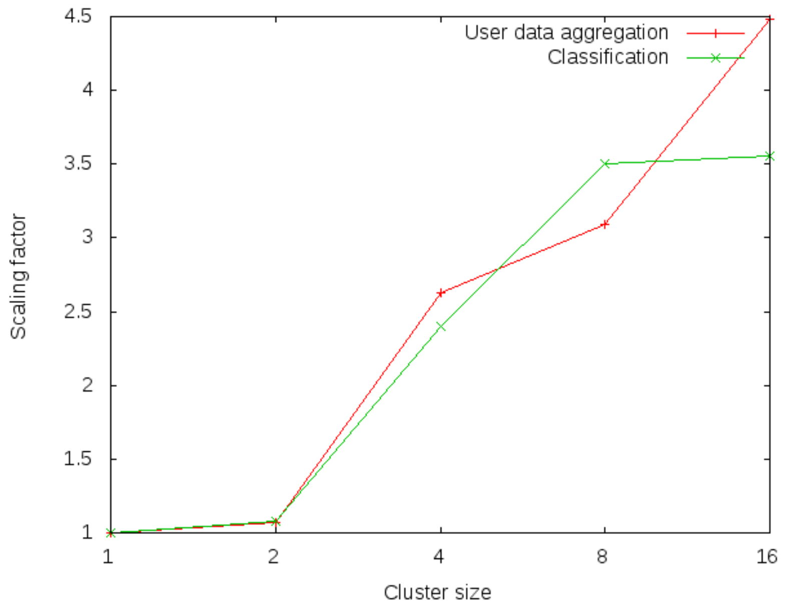


Figure 13: Scalability with size of cluster

6.4 Summary

Of the tested algorithms it was shown that MultilayerPerceptron, NaiveBayes, RandomForest, RIDOR and two-class LibSVM yielded the most satisfactory classification accuracy, with the RandomForest algorithm yielding the best accuracy and lowest false positive rate of them all. Further testing of these algorithms showed that while NaiveBayes, RandomForest and RIDOR had a performance advantage over the MultilayerPerceptron and LibSVM implementations, the overall performance of the classifiers was overshadowed by the time required to perform user data aggregation.

When evaluating the system on increasing cluster sizes with a fixed-size data set, results showed that the horizontal scaling capabilities of the system were adequate for our needs.

7 Future work

This section describes future improvements that can be made to the developed system. The adaptive nature of the system means that it will continuously improve as the training data is refined. However, there are further ways to improve the accuracy of the classification process. In this report, we have covered a fairly large range of classification algorithms with the purpose of determining which one would yield the best results for our application. However, as briefly mentioned in Section 6.1.1, there is room for more calibration of the classification algorithms' parameters (which is beyond the extent of this thesis).

There are also possibilities for performance improvements. As shown by the test results in Section 6, the current system bottleneck is the user data aggregation step (which currently is at least 7x slower on the largest data set). Thus, future efforts to improve overall system performance should focus on this part of the system.

In Section 4.3 we described a number of ideas that were investigated but left out due to inability to acquire the necessary data (e.g. as in the case of "premium clicks" or the mouse movement pattern similarity analysis). Some attributes were not applicable in the current classification process, such as geographical location of users. Using this would require that training data was built for each campaign, so that if most of the viewers for an ad suddenly comes from a new location a warning flag is raised. We believe that these ideas should be further explored (when data can be acquired) as they could potentially provide useful input attributes for the classifier.

Yet another interesting area, not directly related to the functionality of our detection system but to the usefulness of its results, is to research whether the current way of uniquely identify users (or rather, computers) can be improved. One such approach is using *browser fingerprinting* [11] in an attempt to uniquely identify a user's browser based on attributes of its configuration and the user's environment. Such attributes could include the HTTP User-Agent header field that the browser emits and the set of fonts and plugins that are installed on the user's computer. Naturally, such an approach works under the assumption that each browser instance has a unique configuration (and for the purpose of being a stable identifier also doesn't vary over foreseeable periods of time). This of course depends on what attributes are included in the fingerprint and how much entropy they yield. Thus, given enough entropy from attributes such as those described above we can in theory uniquely identify a user. However, browser fingerprinting has yet to be proven to work in practice.

8 Conclusions

In this report, we have studied a number of different data mining approaches to fraud detection in an online advertising platform and applied them in a distributed environment. We have shown that an implementation based on supervised classification algorithms can give satisfactory detection accuracy and that our solution scales reasonably well. Since the design of our system separates the domain-specific operation of extracting data attributes for building a model from the general operation of classifying data according to this model, it is easy to adapt the system to detect new types of behavior. This adaptiveness has subsequently opened up possibilities for Burt to use the system for other purposes. Furthermore, since our tests show that a large number of classification algorithms have similar accuracy, we conclude that the most important challenge is not in finding the optimal classification algorithm but rather in constructing a good set of training data.

List of abbreviations

ARFF	Attribute-Relation File Format
AVF	Attribute Value Frequency
CPC	Cost-per-click
CTR	Click-through rate
HDFS	Hadoop Distributed File System
IAB	Interactive Advertising Bureau
PPA	Pay-per-action
PPC	Pay-per-click
PPM	Pay-per-impression
ROC	Receiver Operating Characteristic

References

[1] Amazon EC2 Instance Types. Retrieved March 18, 2010, from `http://aws.amazon.com/ec2/instance-types/`.

[2] Google AdWords Traffic Estimator. Retrieved February 1, 2010, from `https://adwords.google.com/select/TrafficEstimatorSandbox`.

[3] Invalid Clicks - Google's Overall Numbers. Retrieved May 10, 2010, from `http://adwords.blogspot.com/2007/02/invalid-clicks-googles-overall-numbers.html`, February 2007.

[4] Apache Lucene Mahout: k-Means. Retrieved April 6, 2010, from `http://cwiki.apache.org/MAHOUT/k-means.html`, November 2009.

[5] Dhruba Borthakur. HDFS architecture. Retrieved April 29, 2010, from `http://hadoop.apache.org/common/docs/current/hdfs_design.html`, February 2010.

[6] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly Detection: A Survey. *ACM Computing Surveys*, 41, 2009.

[7] C.C. Chang and C.J. Lin. LIBSVM: a library for support vector machines, 2001.

[8] D. Chang, M. Chin, and C. Njo. Click Fraud Prevention and Detection. *Erasmus School of Economics e Erasmus University Rotterdam*, 2008.

[9] N. Daswani and M. Stoppelman. The anatomy of Clickbot. A. In *Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, page 11. USENIX Association, 2007.

[10] J. Dean and S. Ghemawat. Map Reduce: Simplified data processing on large clusters. *Communications of the ACM-Association for Computing Machinery-CACM*, 51(1):107–114, 2008.

[11] Peter Eckersley. A primer on information theory and privacy. Retrieved April 28, 2010, from `https://www.eff.org/deeplinks/2010/01/primer-information-theory-and-privacy`, January 2010.

[12] Tristan Fletcher. Support vector machines explained. 2009.

[13] D. Foregger, J. Manuel, R. Ramirez-Padron, and M. Georgiopoulos. Kernel similarity scores for outlier detection in mixed-attribute data sets. 2009.

[14] M. Gandhi, M. Jakobsson, and J. Ratkiewicz. Badvertisements: Stealthy click-fraud with unwitting accessories. *Journal of Digital Forensic Practice*, 1(2):131–142, 2006.

[15] Z. He, S. Deng, X. Xu, and J. Huang. A fast greedy algorithm for outlier mining. *Advances in Knowledge Discovery and Data Mining*, pages 567–576, 2005.

[16] Ricky Ho. Hadoop Map/Reduce Implementation. Retrieved May 10, 2010, from `http://horicky.blogspot.com/2008/11/hadoop-mapreduce-implementation.html`, November 2008.

[17] R. Holbrey. Dimension reduction algorithms for data mining and visualization, 2006.

[18] Interactive Advertising Bureau. *Click Measurement Guidelines Version 1.0 - Final Release*, 2009.

[19] A. Juels, S. Stamm, and M. Jakobsson. Combating click fraud via premium clicks. In *Proceedings of 16th USENIX Security Symposium (Security'07)*, pages 1–10, 2007.

[20] A. Koufakou, J. Secretan, J. Reeder, K. Cardona, and M. Georgiopoulos. Fast Parallel Outlier Detection for Categorical Datasets using MapReduce. 2008.

[21] J. Susan Milton and Jesse C. Arnold. *Introduction to Probability and Statistics*. McGraw-Hill, 4th edition, 2003.

[22] J.R. Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann, 1993.

[23] Dass Rajanish. Classification using association rules. IIMA Working Papers 2008-01-05, Indian Institute of Management Ahmedabad, Research and Publication Department, January 2008.

[24] I. Rish. An empirical study of the naive Bayes classifier. In *IJCAI 2001 Workshop on Empirical Methods in Artificial Intelligence*, pages 41–46, 2001.

[25] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 2nd edition.

[26] A. Tuzhilin. The Lane's Gifts v. Google report, 2006.

[27] M. Wahde. *Biologically Inspired Optimization Algorithms, An Introduction*. WIT Press, 1st edition, 2008.