CHALMERS

Browser Fingerprinting

Master of Science Thesis in the Program Computer Science: Algorithms, Languages and Logic

ERIK FLOOD
JOEL KARLSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, May 2012

Browser Fingerprinting

ERIK FLOOD,
JOEL KARLSSON,

Examiner: DEVDATT DUBHASHI

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

Department of Computer Science and Engineering
Göteborg, Sweden May 2012

Acknowledgements

Firstly, we would like to express our sincere gratitude to our academic supervisor, Associate Professor Chiranjib Bhattacharyya, for his commitment and enthusiasm to our work. He has been eager to help and has provided us with excellent feedback in all aspects of the thesis work; may it be in experiment design or in presentation preparations.

Besides our academic supervisor, we would also like to thank the company at which the thesis work was conducted, Burt AB, and especially our company supervisor; Frida Nero. Everyone at the company has been extraordinary helpful and has provided us with everything we have needed throughout the course of the thesis work.

Abstract

Tracking Internet users have several purposes; for example preventing online fraud or measuring and analysing online advertisements. The most common current methods for tracking users involve storing unique identifiers locally on the user's device, a method which may be restricted by law in the future. This thesis examines the possibility to replace such methods with the method of browser fingerprinting.

A browser fingerprint is a composition of information gathered from a web browser. Using data from several sources, we have analysed which features to extract to create a unique fingerprint. Within the scope of machine learning, we have designed online algorithms which can be used for telling Internet users apart by their browser fingerprints. The results from these algorithm show that if the data is preprocessed and partitioned adequately, users can be identified with high accuracy over time periods spanning over several days, even weeks in some cases.

Based on our analysis, we can conclude that machine learning is a promising approach for solving the problem of telling Internet users apart and that extremely naïve solutions are sufficient, in certain applications.

Keywords: *browser fingerprinting*, *browser uniqueness*, *Internet tracking*

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

This thesis investigates browser fingerprinting as an alternative to conventional identification methods used on the Internet.

Analysing Internet traffic is the core business for many companies and tracking users is an essential part of such analysis. However, forthcoming laws may restrict companies and organisations from using methods that store identifiers on client devices. Such restrictions does not, in any way, reduce the demand for reliable methods for tracking website users.

A browser fingerprint is a set of properties of a device and its software which is gathered from a web browser. This thesis asks the question if a method based on on browser fingerprinting can be used instead of the current methods for tracking users.

A difficult part of the problem is to identify what information is possible to collect and how to devise methods for collecting it. Once the information has been collected, the next great challenge is to be able to determine which users the fingerprints belongs to; fingerprints are highly dynamic which implies that a previously unseen fingerprint might very well belong to a known user.

This thesis aims to propose how these types of complex data can be retrieved from online devices, and present a thorough analysis of the temporal behaviour of browser fingerprints. Following the analysis, different machine learning methods are applied in an attempt to solve the problem of telling Internet users apart, especially focusing on methods from the domain of heuristic online learning.

1.1 Background

The ability to identify Internet users is a critical feature in many online businesses, as stated in Tene & Polonetsky (2012). Among the many applications are enhancing

user experience, analytics and measurements, targeting advertisements and fraud prevention; each of which may be more or less crucial to an online business.

For online advertisements agencies, the ability to track users and thereby target advertisements to specific users, will increase the value of their service, which in turn allows them to charge 2.68 times more for advertisements on average (Tene & Polonetsky 2012). The online advertisement industry had a yearly revenue of $31.7 billion for the year 2011 in the US alone according to Internet Advertising Bureau (2012), which gives an indication of the vastness of the industry.

The most common way of tracking online users is by using HTTP cookies, which is stored on the client's device (Eckersley 2010). In theory, storing a unique identifier on a client's device, which is then attached to each subsequent server request would be a perfect identification method. However, both Eckersley (2010) and Tene & Polonetsky (2012) states that many Internet users nowadays are aware of the privacy threats that HTTP cookies might pose, and therefor either reject HTTP cookies from certain hosts or disables such browser features entirely. As a result of the recent Internet privacy awareness, EU legislators are working on directives which will restrict the ability to store information on client's devices (Directive 2009/136/EC).

The importance of online user identification and the growing disadvantages of using HTTP cookies for that purpose calls for another, less intrusive, identification method. A web browser exposes several pieces of information about itself and the device, which when combined may constitute an identifier or *browser fingerprint* (Eckersley 2010). Many of these properties are subject to changes, why the identification method must be highly adaptive to be able to match the accuracy and persistence of HTTP cookies.

1.2 Problem Formulation

As online user identification is a absolute necessity for many online businesses and upcoming directives are restricting the use of conventional identification methods, alternative methods are needed. This thesis aims to examine such alternative methods and propose a solution to the identification problem, which does not store any information on the clients' devices.

There are two major challenges in seeking a solution to the problem; collecting relevant data from web browsers and designing a well performing identification algorithm.

No existing data sets with enough information to form unique fingerprints can be found, why a collection platform have to be developed. This can be divided into two parts; one that collects data in a research environment, without limitations or requirements on performance, in which all possible parameters can be considered, and another that collects data in a real-world environment, where there are strict restrictions on what can and cannot be done, which would be more of a proof-of-concept. When designing a such collection platform, the choice of which parameters to collect is highly critical, not only to the collection process itself, but also to the ability to use the data in future identification algorithms.

Since properties of devices and web browsers are dynamic (Eckersley 2010), an identification algorithm must be flexible enough to be able to recognise the same user despite changes in its fingerprint, without being too flexible so as to confuse users with each-other. In this thesis, machine learning methods are applied to achieve such adaptivity.

Designing an identification algorithm requires a deep understanding of the underlying system, why much focus in this thesis is on data analysis. The temporal behaviours and other important properties of browser fingerprints have a direct impact on the performance of any identification algorithm, and must be addressed.

The goal for this thesis is thus to design a data collection software which is able to collect relevant attributes from users' web browsers that have requested some Internet resource, which can be combined into a browser fingerprint, and later use this to distinguish and recognise these users, by proxy of their web browsers. Any proposed method must perform at least as well as those currently used in the industry, which rely on the ability to store information on a client's device.

There exists almost no literature on the subject of tracking users using browser fingerprinting techniques, even though the current methods used by companies that analyse Internet data may be restricted by upcoming laws. The fact that few, or no, companies have published any research on the subject could indicate that the problem is challenging and complex, and the fact that there are few academic publications on the subject indicates that the problem is of practical, rather than theoretical, nature.

1.2.1 Scope and Limitations

While identifying browser instances is an interesting problem in itself, the ultimate goal is to be able to identify Internet users. Even though identifying users lies outside of the scope for this thesis, identifying a browser instance could, by proxy, be the same

as identifying its user. This is obviously not always the case, as the same browser instance can be used by any number of users, but could give a decent indication.

The artificial intelligence theories of machine learning may be a plausible approach for solving this problem. A machine learning algorithm learns from previous observations and attempts to predict the true label of new observations, this approach seem to fit the problem of telling what user a specific browser fingerprint belongs to, if any. The aim of this thesis is, in the extended scope of these theories, search for a solutions to the problem of tracking and identifying browser instances.

Moreover, evaluating a solution for identifying browser instances requires knowledge of the real identities of these browser instances, which is only known by the identifiers set through storing information on the client's device. This makes it impossible to evaluate whether the solution's performance exceeds that of those methods which involves storing information on the client's device.

1.2.2 Definitions

This section describes specific terms which are used throughout the thesis.

Browser Instance

Browser instance is a very important term in this thesis, as it is what should be determined given a browser fingerprint. Browser instance refers to a specific installation of a specific web browser on a specific device. A browser instance remains the same even if the browser is upgraded or downgraded, plugins are installed or removed or settings changed.

A browser instance is not to be confused with an *instance*, which is a machine learning term denoting a single object or feature vector from the data used in the model.

Browser Fingerprint

A set of features used to identify a browser instance is referred to as a *browser fingerprint* or just *fingerprint*. These features are usually described as a feature vector.

The Company

The company at which the thesis work was conducted will throughout the thesis be referred to as *the Company*. The Company is in the online advertisement sector,

and provides follow-up and real-time analysis of Internet advertisement campaigns for both advertisers, media agencies and publicists.

1.3 Thesis Outline

The thesis work was divided into three separate categories; collection of data, analysis of data, and modelling and classification. These are all described in separate chapters, followed by a general discussion about using browser fingerprinting methods for Internet identification and the most important conclusions drawn from the research. The thesis also comprises a chapter presenting some theoretical concepts as well as previous research in browser fingerprinting.

1.3.1 Theory and Related Work

This chapter presents a concise summary of the theory around the concepts and methods used in this thesis, as well as a short survey of previous research on the uniqueness of certain browser features. The chapter is intended to be used for reference rather than for intensive reading.

1.3.2 Building a Collection-Platform for Internet Tracking

This chapter aims to summarise and describe how the software for collecting Internet tracking data was developed. The software was deployed on a dedicated website and collected data during a three-month period. As a proof-of-concept, a stand-alone version of the collection script was created. The portable script could with small means be implemented on any website. As part of the portable script, a compression algorithm for fingerprints was developed.

1.3.3 Analysing Internet Tracking Data

Many, if not all, of the considered features are subject to changes and their behaviours and properties are more or less complex. A good understanding of these behaviours is crucial to being able to build a well performing classifier, why several measurements were designed to highlight and explain important properties of the data. The analysis presented in this chapter aims to provide the data required to make a sound feature selection and adjust the model's responsiveness to changes as well as to constitute a solid base for further research.

1.3.4 Designing a Heuristic Online Classifier for Internet Tracking Data

The browser fingerprints are dynamic and new browser instances are encountered frequently, why an adaptive classifier is required to be able to respond to these continuous changes. Therefor, *online* classifiers are the main focus of this thesis.

This chapter aims to describe the design process and the choices therein, as well as the results obtained when testing the designed algorithms.

1.3.5 Is Browser Fingerprinting a Viable Alternative to Conventional Methods?

The extensive analysis of Internet data does not only bring clues on how different fingerprint feature compositions affect the results of the classification algorithms, but also how some information differs between different data sources and different devices. We believe that this type of analysis is essential both for improving classification algorithms, but also for better understanding the algorithmic behaviours when partitioning the data or when the data is retrieved from different sources. The results in terms of these aspects are discussed in this chapter.

1.3.6 Conclusions

This chapter tries to illustrate and conclude our most important experiences and results. Our analysis indicates that machine learning algorithms is a promising solution to the problem of how to tell users apart by their browser fingerprint. In some cases, more straight-forward algorithms may be a better choice in terms of computational speed versus algorithmic accuracy. Thorough analysis and processing of the data is a significant part of constructing good algorithms and collecting relevant features.

Chapter 2

Theory and Related Work

This chapter presents a concise summary of the theory around the concepts and methods used in this thesis, as well as a short survey of previous research on the uniqueness of certain browser features. The chapter is intended to be used for reference rather than for intensive reading.

2.1 Machine Learning and Classification

> A computer program is said to **learn** from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E. (Mitchell 1997, pp. 2)

Machine learning can be divided into two major areas; unsupervised and supervised learning. The former involves learning key properties about a system by finding clusters in data collected from that system. The "correct answer" is not available during any part of the process. The latter area is the main focus of this thesis, in which there is a "correct answer" known to a supervisor or teacher. There is a variety of such methods, but this thesis is only concerned with learning categorical *labels* for *instances*. An instance is a vector of properties from an observation of the modelled system, and each instance is associated with a label. The true label of an instance, x, is denoted $c(x)$, which is known as the *concept function* for the model. The goal for the learner is to learn a *labelling function* $l : X \to L$, where X is the set of all possible feature vectors and L the set of all labels, which is as close to the concept function as possible. A classifier is an algorithm or method for deriving such a function from a set of instances and their corresponding labels, which can then be used to

classify (or label) instances whose labels are unknown. The process of training and evaluating a classifier can be divided into two distinct phases; training and testing.

2.1.1 Training and Testing

When developing a classifier, labelled data from the real system must first be collected. When collecting data, extensive knowledge about the system at hand is required to be able to make adequate choices regarding the selection of which features to collect and how to collect these. When a large enough sample of labelled data, $D \subset X \times L$, has been collected, the classifier can be trained and tested.

During the training phase the classifier is fed with pairs of instances and their corresponding labels, or *training instances*, $(x, c(x)) \in D$. Hopefully, the supplied training instances have enabled the classifier to derive a well performing labelling function, which is then evaluated during the test phase. During the test phase, the classifier is asked to classify instances without knowing their labels and the results are then compared with the true labels to evaluate the classifiers performance.

Prior to training and testing, two subsets of D must be selected; the training and the test set. A common way of doing this is by *percentage split*, in which two complementary subsets are selected to be used for training and testing, respectively. For example, in a 70-30 percentage split 0.7 of D is used for training and the remaining 0.3 for testing. Another method is *test set performance*, in which the data used for training is also used for testing. There are several other methods for dividing the data set, which are not covered in this thesis.

2.1.2 Online Learning

In online learning, the distinction between training and testing is not as clear as in classic machine learning. An online classifier is cumulative and learns instances one at a time. When fed with a new instance, the classifier first predicts the label as usual, and is then given the true label from some supervisor and can use this information to refine itself to produce more accurate predictions in the future. Not all systems are thus suitable to model using online classifiers, since the true label has to be available at some time after the prediction. Note that there is no need for the true label to be known at the time of the prediction, which would essentially render the value of using the classifier futile.

2.1.3 Naïve Bayes

Naïve Bayes is, as the name suggests, a simple classifier. The naïvity of the classifier lies in the assumptions about the modelled system; complete independence between features must be assumed. It is a probabilistic classifier which assigns probabilities to each label given an instance, and returns the most likely label.

Given an instance $\mathbf{x} \in X$, the conditional $P(l|\mathbf{x})$ is sought for all known labels $l \in L$. The instance $\mathbf{x}$ is a vector of feature values $(x_1, x_2, \ldots, x_n)$. The assumed independence between features and Bayes' theorem implies that this probability can be written as

$$P(l|x_1, x_2, \ldots, x_n) = \frac{P(l)}{P(x_1, x_2, \ldots, x_n)} \prod_{i=1}^{n} P(x_i|l). \tag{2.1}$$

The most likely label is sought, and $P(x_1, x_2, \ldots, x_n)$ is constant, which means that the problem can be expressed as

$$\arg\max_{l \in L} P(l) \prod_{i=1}^{n} P(x_i|l). \tag{2.2}$$

The probabilities $P(l)$ and $P(x_i|l)$ are calculated as frequencies in the data set D (which is a multi-set, since identical instances can occur more than once);

$$P(l) = \frac{|D_l|}{|D|} \tag{2.3}$$

$$P(x_i|l) = \frac{|D_{x_i,l}|}{|D_l|} \tag{2.4}$$

where D_l is the set of observed instances with label l and $D_{x_i,l}$ is the set of instances with label l and feature value x_i for feature i.

Smoothing

The probability in (2.4) will be zero if an instance with label l has never been seen with feature value x_i. If this happens, the label will be regarded as impossible rather than implausible, which might not be the desired behaviour. *Laplacian* or *Dirichlet smoothing* can be used to ensure that the nominator never becomes zero with minimal influence on the final result.

As described by Smucker & Allan (2005), Dirichlet smoothing is a parametrised method which, for a certain feature i, can be expressed as:

$$P(x_i|l) = \frac{|D_{x_i,l}| + qp}{|D_l| + q} \tag{2.5}$$

where p is the prior probability for a certain feature value and q is a parameter, or free variable.

Laplacian smoothing is a non-parametrised smoothing and can be written as:

$$P(x_i|l) = \frac{|D^{F_i}_{x_i,l}| + 1}{|D_l| + |F_i|} \tag{2.6}$$

where F_i is the set of possible values for feature i. The Laplacian smoothing is a special case of Dirichlet smoothing when $q = |F_i|$ and $p = \frac{1}{|F_i|}$.

2.1.4 k-Nearest Neighbour

The k-nearest neighbour (kNN) algorithm is a classification algorithm which classifies instances based on the k closest training instances in a feature space (Cunningham & Delany 2007). In the most basic kNN, the instance to be classified is assigned with the most common class among the k nearest neighbours, but more advanced methods may be applied to improve the performance. One common modification is to perform a majority vote among the k training instances, in which the votes are weighted with the inverse of the distance to the instance to be classified (see Figure 2.1).

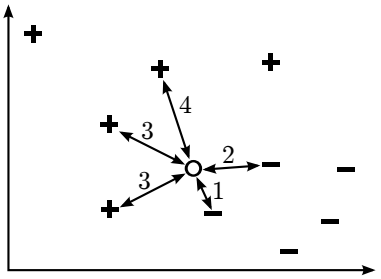

Figure 2.1: This example demonstrates a simple 2-dimensional 5NN classification. If simple majority vote is applied the instance would be classified as (+), whereas if the votes are weighted with the inverted distances to the corresponding training instances it would be classified as (-) since $\frac{1}{3} + \frac{1}{3} + \frac{1}{4} < \frac{1}{1} + \frac{1}{2}$.

The distance between two feature values can be calculated using any distance function, $d : T^2 \to \mathbb{R}$, where T is some set of feature values. However, a proper distance metric is to be preferred to an arbitrary distance function, since this allows

for some performance optimisations (Duda, et al. 2001). The formal definition of a metric states that the following criteria must be satisfied:

$$\begin{aligned}
\forall x, y \in T \,.\, d(x, y) &\geq 0 && \text{(non-negativity)} \\
\forall x, y \in T \,.\, d(x, y) = 0 &\rightarrow x = y && \text{(identity)} \\
\forall x, y \in T \,.\, d(x, y) &= d(y, x) && \text{(symmetry)} \\
\forall x, y, z \in T \,.\, d(x, z) &\leq d(x, y) + d(y, z) && \text{(triangle inequality)}
\end{aligned}$$

To calculate the distance between two complete instances, $\mathbf{x}$ and $\mathbf{y}$, the L_k-norm can be used. This is defined as

$$L_k(\mathbf{x}, \mathbf{y}) = \left(\sum_i d_i(x_i, y_i)^k \right)^{\frac{1}{k}}. \tag{2.7}$$

where k is any integer larger than 0 and d_i the distance metric for feature i (Duda et al. 2001). Some of the most common L_k-norms have their own names; the L_1 norm is called the *Manhattan distance* and the L_2 norm the *Euclidean distance*.

Since features can be of any type (continuous or discrete, numerical or non-numerical) different features calls for different metrics. The following sections aim to describe a few common metrics for features of different types.

Levenshtein Distance

Measuring the distance or dissimilarity between two strings can be done in several ways. One common method is the *Levenshtein distance*, also known as the edit distance, which is the minimum number of edit operations needed to transform one string into the other (Duda et al. 2001). The allowed edit operations can differ slightly from implementation to implementation, but insertion, substitution and deletion should be included.

1.	**s**tep	$\rightarrow$	tep	(deletion of **s**)
2.	t**e**p	$\rightarrow$	t**a**p	(substitution of **e** to **a**)
3.	tap	$\rightarrow$	t**r**ap	(insertion of **r**)

Table 2.1: This table shows how to calculate the Levenshtein distance between *step* and *trap*, which gives $d(\text{step}, \text{trap}) = 3$. Note that there exist several solutions which makes this transformation in three steps.

		Actual	
		Positive	*Negative*
Predicted	*Positive*	True Positive	False Positive
	Negative	False Negative	True Negative

Table 2.2: A *confusion matrix* can be used to visualise the performance of a classifier.

Tanimoto Metric

To determine the difference between two sets, without needing to have any method for determining the distance between their elements, the *Tanimoto metric* can be used (Duda et al. 2001). This defines the difference between two sets, X and Y as

$$d(X, Y) = 1 - \frac{|X \cap Y|}{|X \cup Y|}. \tag{2.8}$$

2.1.5 Evaluating Classifiers

Measuring the performance of a classifier is not always straightforward; Table 2.2 shows how true positive (TP), true negative (TN), false positive (FP) and false negative (FN) answers are defined. These are important terms, which can be combined to describe important properties of a classifier. Some of the most common measures derived from those terms are described in this section, along with a description of how those can be applied to an online classifier.

Accuracy

The *accuracy* of a classifier is defined as

$$\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN}\,, \tag{2.9}$$

that is the fraction of true or correct answers. This measure is easy to understand, but has the drawback that it may fail to discover some important flaws of the classifier. Consider if a classifier gives ten positive answers, of which only five are correct, and 90 negative answers, all of which are correct; then the accuracy would be 0.95, which could be considered to be good, but the fact that only half of the positive answers were correct remain undiscovered.

Precision

The *precision* of a classifier is defined as

$$\text{precision} = \frac{TP}{TP + FP} , \tag{2.10}$$

that is the fraction of correct answers among the positive answers. The precision for the example in the previous section would be 0.5, which would be alarming.

Recall

The *recall* of a classifier is defined as

$$\text{recall} = \frac{TP}{TP + FN} , \tag{2.11}$$

that is the fraction of answers that were positive among those that should have been positive. In the example in the previous sections, the recall would be 1.0, since all of the positive-labelled instances were classified as such.

F_1-Score

The F_1-score is defined as the harmonic mean of precision and recall,

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} . \tag{2.12}$$

The F_1-score is a single measure which allows for a quick evaluation of a classifier's performance, especially in combination with the accuracy.

Evaluating Online Classifiers

All of the measures described above concern a single well-defined classifier. An online classifier is continuously updating itself, why these measures become meaningless. For an online classifier, the order in which the classifier processes the instances affects the outcome greatly, meaning that it is highly likely to receive different results from the same data set depending on the ordering.

When comparing online classifiers, it is therefor important to process the data set in the exact same order for all classifiers. Moreover, to emphasise that it is a measure of an online classifier, all such measures will be prefixed with *online* (online accuracy, for example) throughout this thesis.

2.2 Entropy

The entropy of a data set is a measure of the level of disorder within the set, or more precisely the expected number of binary questions needed to classify a randomly picked instance from the data set. The entropy for a data set D with labels L is calculated as

$$\text{Entropy}(D) = -\sum_{l \in L} p(l) \log_2 p(l) \,, \tag{2.13}$$

where $p(l)$ is the fraction of $x \in D$ with $c(x) = l$. The entropy is maximal when the distribution of labels is as uniform as possible.

2.2.1 Information Gain

The information gain is a measure of the amount of information a specific feature gives about the labelling of a data set. The information gain is measured as the difference in entropy from knowing a feature value. For a given feature F, the dataset D is partitioned by the different values of F. The parts of the partition are denoted D_v^F, which is the set of all instances in D with value v for feature F. The information gain is then calculated using the following formula

$$\text{Gain}(F, D) = \text{Entropy}(D) - \sum_{v \in F} \frac{|D_v^F|}{|D|} \text{Entropy}\left(D_v^F\right) \,. \tag{2.14}$$

2.3 Previous Research – Project Panopticlick

Eckersley (2010) present a study which is similar to that in this thesis. The author have implemented a website, called `http://panopticlick.eff.org`, for collecting a set of web browser and device properties from consenting users, which was then analysed and used for designing a simple heuristic for identifying users. The purpose of the study was to highlight the privacy concerns associated with Internet tracking, by showing that browser fingerprints are unique enough to identify a large fraction of Internet users.

Through analysing the data collected at the research website, Eckersley (2010) have found that the following features are the most identifying: browser plugins, system fonts, User-Agent string (UA), HTTP Accept-headers and screen dimension. It is important to note that Internet Protocol Address (IP) was not included in the study as a feature. In addition to these five features, three more were collected, but

they proved less informative. By combining all eight parameters into one browser fingerprint, the author claims that 0.836 of all collected fingerprints were unique. When isolating computers in the statistics, around 0.90 is said to be unique.

The algorithm proposed in the article was "clearly very crude, and no doubt could be significantly improved with effort" (Eckersley 2010). The algorithm was run on the collected data set, which was preprocessed to remove noise, and resulted in 0.65 correct guesses, 0.0056 incorrect guesses and no guess in 0.35 of the cases (these figures were taken directly from Eckersley (2010), and appear to include some rounding error). From this, the author draws the conclusion that 0.991 of all guesses are correct, but pays no attention to the fact that the algorithm failed to return any guess in 0.35 of the cases.

Still, the results are intriguing and leads Eckersley to the conclusion that browser fingerprinting is a powerful technique, even if the studied method leaves room for improvement.

Chapter 3

Building a Collection-Platform for Internet Tracking

This chapter aims to summarise and describe how the software for collecting Internet tracking data was developed. The software was deployed on a dedicated website and collected data during a three-month period. As a proof-of-concept, a stand-alone version of the collection script was created. The portable script could with small means be implemented on any website. As part of the portable script, a compression algorithm for fingerprints was developed.

3.1 Building a Website for Internet Tracking

The research website was constituted of two large components; a set of scripts which collects information from devices who visit the research page and a web server which serves resources and stores client data in a database.

3.1.1 Client side

Apart from the HTML document, the relevant part of the client side software, with regards to collecting data, was a JavaScript script called `Fingerprint.js` which retrieved information about the client's web browser and device using different techniques. The information was stored in a JSON data structure and transmitted to the server using an HTTP POST request.

`Fingerprint.js` was able to collect information automatically as soon as a device visited the web page. However, due to ethical reasons, no information about a browser should be transmitted to the server without the users explicit consent, why a simple button on the website was implemented. When a user clicked the button, an event

in `Fingerprint.js` was triggered to send the actual HTTP request containing the fingerprint to the server.

3.1.2 Fingerprint.js

A complete list of all the information, or features, that is collected by the script is available in Appendix A. Most features could be collected directly from the Document Object Model (DOM) using JavaScript, an operation that is both easy to implement and cheap in terms of execution time.

More complex methods were used for those features not collected from the DOM. The following sections gives a detailed description of how those features where collected.

Browser Plugins

For all supported browsers, except Internet Explorer, the details about an installed browser plugin could be retrieved using the same method. For such browsers, information about the plugins was stored in the object `navigator.plugins` in the DOM. From each plugin object, the script collected the fields `name`, `version`, and `description`. All of the fields were string valued which could be of zero-length. In addition, each plugin was associated with one or more *mime-types*, which is a descriptive label containing information about what type of data the plugin is supposed to handle. This information was also collected by `Fingerprint.js`.

Internet Explorer does not store the information about the installed plugins in the DOM as the other major browsers does. Instead, the collection script performed queries in order to find out if a plugin is installed or not,

A large list containing names of known ActiveX-plugins was used for querying. For each plugin, the script tries to instantiate the corresponding ActiveX-object. If the instantiation was successful, the plugin was in fact installed. In addition to the plugin name, information about the version number may also be retrieved from the instantiated object.

An example of how a querying for Adobe Flash Player 9 can be done using JavaScript is presented in 3.1.

Fonts Retrieved via Adobe Flash

The information about which fonts are installed on a specific device can not be gathered directly using JavaScript. The information could however be retrieved for those

Algorithm 3.1 An example of how querying for Adobe Flash Player 9 can be done through JavaScript.

```
try {
    obj = new ActiveXObject('ShockwaveFlash.ShockwaveFlash.9')
} catch (e) {
    // The plugin was not installed
}
if (obj) {
   // The plugin was installed
}
```

browsers which had a plugin installed that supported Adobe Flash.

An Adobe Flash object was created in ActionScript in which the font information was retrieved using the `flash.text.Font` package. `Fingerprint.js` creates a hidden `embed`-tag containing the object and, when all fonts have been collected in Adobe Flash, the information was retrieved using a JavaScript callback.

Fonts Retrieved via CSS Querying

For devices without Adobe Flash support, an alternative method for retrieving system fonts was examined.

The examined method was based on an assumption that a specific letter in a specific font takes up a specific height and width in pixels. Fonts belong to certain font-families, such as *serif*, *monospace* and *sans-serif*. For compatibility reasons, web browsers have fallback fonts for each font-family, meaning that if a web page is formatted in a specific font which is not installed on the client's device, a fallback font is used instead.

The collection script queries a device for installed fonts by generating hidden elements on a web page. The hidden element contains a predetermined character string and for each queried font, the height and width of the hidden element is compared to the height and width of an element containing the same string but formatted in the fallback font. If the height and width differs, it is be assumed that the font is installed. However, if the height and width does not differ, it cannot be confirmed whether the queried font is installed or not.

Using this method for querying for fonts is a time-consuming process. On the research site, a query for about 1,500 fonts takes 5-15 seconds depending on the specifications of the queried device.

Round-Trip Times

For a specific device, it was assumed that the Round Trip Time (RTT) between the device and a specific web server only varies slightly over time under certain conditions. These conditions include constant web server load, that the device is more or less stationary and that the device's Internet bandwidth does not vary over time.

As such, RTT was examined as a possible information source for creating a true and reliable browser fingerprint. The communication between the research site and a web browser is shown in the figure Figure 3.1. The device sends a request at time t_1 and waits for the server response which arrives at time t_2. The RTT is simply the difference between the two timestamps, i.e $RTT_k = t_{k+1} - t_k$. `Fingerprint.js` collected k requests where experimental values for $k = 1, 2, \ldots, 10$. The collection script did not calculate any aggregate of the RTTs but instead a complete list was stored in the fingerprint.

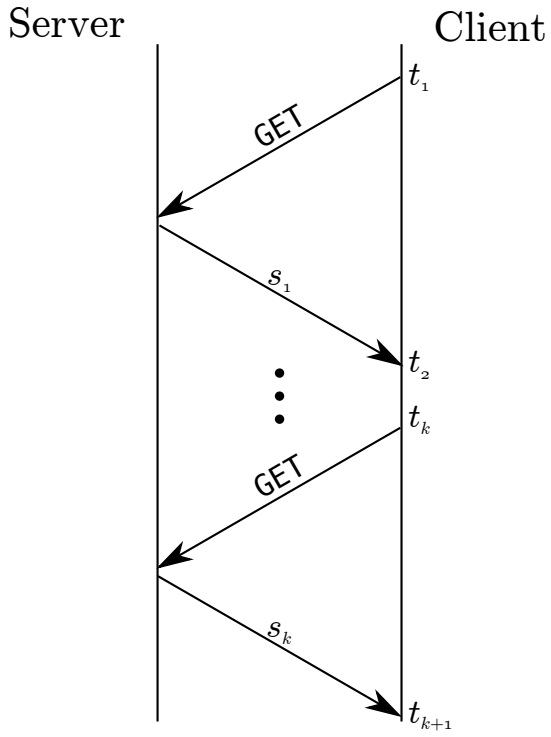

Figure 3.1: Assessment of RTTs and Client Clock Error over the HTTP protocol

Client Clock Error

The client's internal clock might differ from the web server's clock. A scheme to measure this difference, called client clock error, was developed. The collection of

necessary information is performed in the same requests as for the RTTs. As we can see in Figure 3.1, for each request the server responds with its own time, s_k.

Without access to both server and client it is not possible to know exactly when the server responds, it might be almost instantly after t_k but it can also be almost instantly before t_{k+1}. The effect of this knowledge is that we can only make an approximation of the clock error. If we use same notations as in Figure 3.1 and let ϵ be the true difference between server and device we know that

$$t_0 - \epsilon < s < t_1 - \epsilon \Leftrightarrow t_0 - s < \epsilon < t_1 - s \,. \tag{3.1}$$

In other words, the RTT is a bound for the size of the client clock error interval. The collection script stores both the maximum and minimum error.

3.1.3 Event-Driven Data Collection

Many of the complex collection operations can differ in execution time, especially those operations which depends on several server requests. By using an event queue, the collection script could be developed in such a way that neither the collection process nor the page load stalled because of a possible delay in the retrieval of a specific feature.

3.1.4 Server side

The server software had a rudimentary set of features:

- Serving web resources, with dynamic translations of text
- Serving the server time
- Retrieving and storing client fingerprints in a database
- Setting, or extending expiration date on, HTTP cookies on the devices

3.1.5 Take a Bite of the HTTP Cookie

The first draft of the research website had a very functional approach and it was tested internally on the company. As a part of reaching a broad audience, a more campaign-like approach was devised which was called “Take a bite”, the campaign was deployed on the website `take-a-bite.org`.

Over a three month period, a total of 1753 devices visited the website as shown in Figure 3.2 `take-a-bite.org` website.

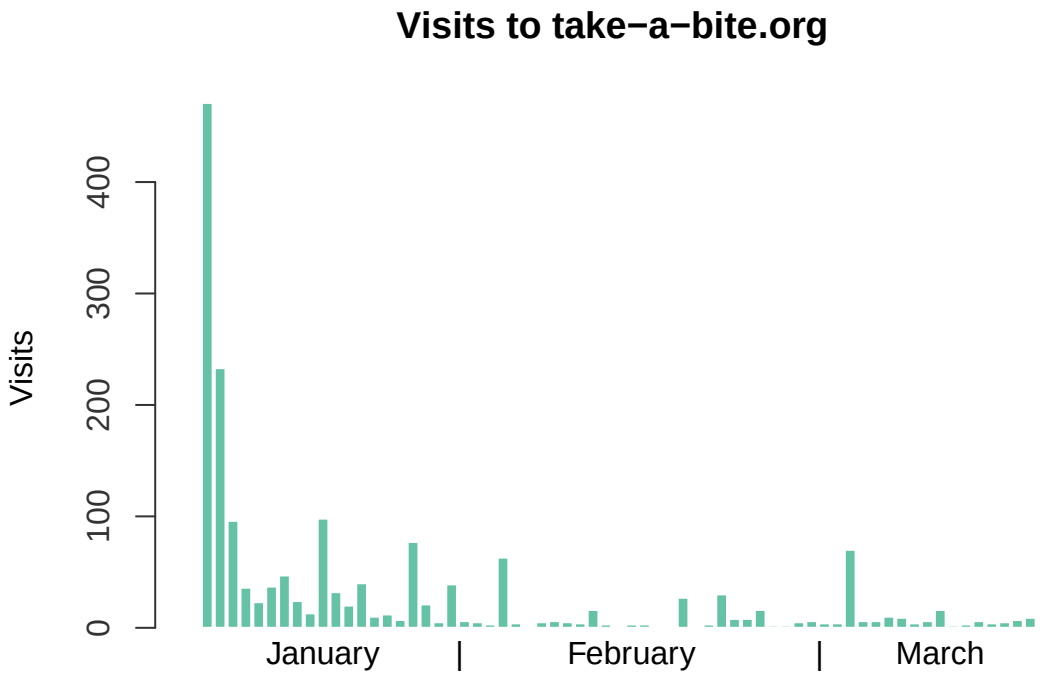

Figure 3.2: Visits to `take-a-bite.org`

3.2 Building a Portable Tracking Script: FP.js

A portable tracking script, called `FP.js` was developed as a proof-of-concept. This was done to demonstrate how a feasible real-world solution to the problem of collecting fingerprints could be devised.

On the research website, a large number of features were collected. Because of the implied time-constraints in a real-world environment, only a subset of those features were chosen to be collected by `FP.js`. The collected features are listed in Appendix B.

It was important that the script could be implemented on any website without installing any server software, why the actual collection is performed on separate servers. The script can be implemented on a website by just adding a single `script`-tag. All necessary alterations to the DOM needed for the collection to work properly is then added by the script itself.

If the script is installed on a website which does not belong to the same domain as the collection server, the fingerprint cannot be sent using HTTP POST requests because of security restrictions in the web browsers. Because of this limitation, the fingerprint had to be transmitted using a HTTP GET request instead, that is as part

of a URL. When all necessary data about a fingerprint has been collected, the script generates a `script`-tag linking to a URL to the collection server with the information passed as argument in the URL. Below is an example of a request in which the key-value pairs (k_1, v_1) and (k_2, v_2) is sent using the described method:

```
<script type="text/javascript"
        src="http://fp.example.com/example.js?k1=v1&k2=v2" />
```

After the generated script tag have been requested by the browser, the element is removed from the DOM.

One limitation with passing information using HTTP GET is that most browsers have a limit on the maximum size of the URL string. Internet Explorer have the smallest limit of 2048 bytes meaning that all URLs with a larger size must be split up in several smaller requests.

Since `FP.js` collects features such as fonts and plugins which yield large amount of data, it was necessary to split the requests. Initial analysis showed that a typical user generated between 10 to 20 requests. A compression scheme were developed to decrease the total number of requests.

3.2.1 Compression Scheme

The compression algorithm needed to be fast, accurate and the compressed data must only consist of allowed URL characters. Only the large features that contained information about fonts and plugins was compressed. Based on the data collected on the research site, a look-up table of the most common fonts and plugins was generated and each font and plugin was associated with a unique integer.

Because of the large number of possible feature values for fonts, each font which was defined in the look-up table were converted to a fixed-length number on base 64. As an example, a base-64 number with fixed length 3 can at most represent the decimal number $64^3 + 64^2 + 64 = 266404$. The benefit of using a fixed length was that no separation was needed between each data point in the request.

A URL-safe base64-encoding algorithm was used for encoding those elements which were undefined in the look-up table.

3.2.2 Testing FP.js

`FP.js` have been deployed on the Company's websites and has been tested on a large variety of devices and browsers with a good overall performance. Deploying the

script on these sites was essential to prove that the fingerprints could be collected in a real-world scenario and not just in a research environment.

Compression Rate

Data were collected from the Company's websites over a period of 3 months. A total of 2100 fingerprints where collected. If no compression algorithm would not have been used, the data would have been transferred as a base64-encoded request. The size of the uncompressed data in base-64 encoding yielded a total size of 111,047,994 bytes, whereas the compressed data only resulted in a total of 7,497,854 bytes of data. Hence the total compression rate in this case was 93.13%.

Chapter 4

Description of the Different Data Sets Used

The data sets analysed in this thesis can be divided into real world data and research data. The research data was collected from consenting users of a site whose sole purpose was to provide data for this thesis whereas the real world data was collected from users of sites with different purposes, without asking for their explicit consent. A summary of all the data sets can be found in Table 4.1.

4.1 Real World Data

Three sources of real world data was used in this thesis; these have different sets of features and originate from sites of different genres.

4.1.1 Multi-Site Advertisement Campaign

The, by far, largest data set was collected from advertisements in a campaign visible over multiple websites for three weeks. The sites which hosted the advertisement belonged to different genres, and the most common genre was online news papers. The data was collected by the Company with no intent of being used for the purpose of this thesis, and is thus lacking many interesting features. Aside from labels, the only features available in this data set were IP, UA, screen dimension and window dimension. However, the size of the data set was just over 22 million instances, which made it relevant to investigate. Over these 22 million instances only 1.7 million labels, of the total 3.3 million labels, occurred more than once.

Another raison d'être for this data set, was the fact that the true labels were based on Flash LSO which are more persistent than HTTP cookie values. However, the usage of Flash LSO requires that the client's device has got software able to run

Adobe Flash applications, which limits the range of browsers present in the data set as not all browsers has such capabilities. In particular, neither the Apple iPhone nor the Apple iPad had Adobe Flash support at the time of the data collection.

This data set will be referred to as *campaign data* throughout the remainder of this report.

4.1.2 Special Interest Website

As an experiment, the company collected data with two additional features for a short period of time. The two additional features were collected from the HTTP Header and did hence not require any modification to the code running on the clients' devices. The extra features were HTTP Accept-Language and HTTP Accept-Charset, which tells the server which are the client's preferred languages and character sets, respectively.

From this data, all traffic to one certain special interest website was extracted for a number of reasons:

- To enable testing and analysing of the complete data without sampling, which removes the risk of testing on an non-representative sample.
- To validate or invalidate the hypothesis stating that site genre is an important factor to the user behaviour, and thus to the choice of classifier.
- To maximise the number of recurring visitors, which in turn enables testing of how well those are recognised.

The data set contains 673,112 instances, collected over one week, and has the same four features as the campaign data plus two additional; HTTP Accept-Language and HTTP Accept-Charset. There were 160,712 labels in the data, and 74,319 of these occurred more than once. The true labels are based on HTTP cookies which may subject to deletion, but since the data is from only one week deletion should not be a significant issue. Since the data collection was not dependent on Adobe Flash, all devices that visited the site during the collection period should be present in the data.

This data set will be referred to as *single-site data* throughout the remainder of this report.

4.1.3 Company Websites

The FP.js script, described in section 3.2, was deployed on all sites owned by the Company. These sites belong to the genres company websites and web application sites. The collected features constitute a small subset of those collected from `http://take-a-bite.org`; IP, UA, fonts, screen height, browser plugins, HTTP Accept-Language and HTTP Accept-Charset.

The amount of traffic to these sites are relatively small, and the size of the data set was 5,652 instances collected over a time period of 72 days. These instances were generated by 2,604 unique users of which 933 were seen more than once.

This data set will be referred to as *company data* throughout the remainder of this report.

4.2 Research Data

The data collected from `http://take-a-bite.org`, as described in section 3.1, was collected with the sole purpose of being used in this thesis. Each instance in the data set consists of 49 features, excluding the label, but in order to make a comprehensible analysis only a subset of these was used. The features not considered were either deemed uninformative or redundant in an early phase of the thesis. The selected subset of 13 features consists of: IP, UA, browser plugins, browser language, screen width, screen height, timezone, minimum clock error, fonts (retrieved via both Adobe Flash scripts and a Cascading Style Sheet (CSS) exploit as separate features), RTT, HTTP Accept-Language and HTTP Accept-Charset.

Since the collection site did not offer any value to the visitors, other than that in helping in this research, the amount of visitors was small. The majority of the visitors were either employed by the company or acquainted otherwise, thus not constituting a representative sample of Internet users in general. There were a total of 1,753 visits to `http://take-a-bite.org` by 1,129 unique users of which only 195 where recurring during a collection period of nearly three months.

This data set will be referred to as *research data* throughout the remainder of this report.

Name	Features	Instances	Labels	Recurring	Days
Campaign data	4	22,248,326	3,133,761	1,748,446	21
Single-site data	6	673,112	160,712	74,319	7
Company data	7	5,652	2,604	933	72
Research data	13 (49)	1,753	1,129	195	80

Table 4.1: Summary of the data sets used for testing and analysing.

4.3 Partitioning Data

Some features have values which directly rule out all labels seen with certain other values for the same feature. This allows for partitioning the data based on the values of these features, since the labels in each part are distinct from those of the other parts. Creating such a partition has a number of advantages:

- Minimising the number of candidate labels makes the classification process faster.
- Minimising the number of candidate labels also makes the probability of classification error smaller.
- By separating the data, the classification process can be tailor-made to fit the different parts.

Three such partitions, and combinations thereof, were made to investigate a potential difference in classification performance as well as in feature behaviour for different parts, as described below:

Device type:

- Handheld
- Computer

Operating System (OS):

- Microsoft Windows
- Apple Mac OS X

Web browser:

- Internet Explorer

- Mozilla Firefox
- Google Chrome
- Safari

The most common OSs and web browsers were chosen so as to minimise the number of instances not fitting into any of the parts of the partitions. The partitions were based solely on information found in the UA reported by the clients. Different browsers include different kind of information in the UA and there is no standard method for extracting specific information from it. Therefor, specialised filtering criteria had to be developed in order to create the partitions. The lack of a standard format for the UA makes it difficult to create a complete partition, that is a partition in which the complete dataset is represented, without knowing beforehand all possible values the pivot feature can have or by having a part for all instances not meeting any filtering criterion.

Chapter 5

Analysing Internet Tracking Data

Many, if not all, of the considered features are subject to changes and their behaviours and properties are more or less complex. A good understanding of these behaviours is crucial to being able to build a well performing classifier, why several measurements were designed to highlight and explain important properties of the data. The analysis presented in this chapter aims to provide the data required to make a sound feature selection and adjust the model's responsiveness to changes as well as to constitute a solid base for further research.

5.1 Designing Measurements

Choosing adequate phenomena and/or entities to measure is key to describing any system. This analysis focused on three such phenomena: feature value changes, information gain and functionality. These will be described in detail in this section.

After choosing what to measure, the actual measurements had to be designed such that important properties of the phenomenon were made as clear as possible. For all measurements, all possible partitions of all four data sets described in chapter 4 were considered to enable the identification of differences between different data sources as well as between different parts of the partitions.

5.1.1 Temporal Feature Value Changes

Feature value changes are common for almost all of the collected features; IP changes if the device is connected to a new network, UA changes when the browser is upgraded, screen dimension changes when the device is connected to an external monitor or rotated (handheld devices), and other features display similar behaviours. Some changes might be reversible, as is the case for IP changes, or irreversible, as is the

case for UA changes. A classifier must hence be adaptive to be able to recognise a browser instance even if it has changed some feature value.

A measurement was developed attempting to measure to which extent feature value changes occur, in which the temporal rate of change was considered. The desired point of measurement was the fraction of browser instances who had been seen with a changed feature value within a given time, t, from their first occurrence. But since the data consisted of discrete samples of browser instances in time, there was no way of knowing exactly at which time a change occurred. If a browser instance is seen at time t with a certain feature value and at time $t' > t$ with another value for the same feature, this change can have occurred at any time between t and t'. Assuming that the change occurred at t gives an upper bound for the change rate, whereas assuming that it occurred at t' gives a lower bound. Rather than producing bounds for the change rate, the actual rate of change was approximated by dividing the time axis into a series of consecutive intervals and calculating the fraction of browser instances displaying changes of those recurring within each such interval. That is, for each interval $[t_1, t_2[$, only browser instances seen at time $t_1 \leq t < t_2$ from their first occurrence was considered. Moreover, all changes are considered irreversible, meaning that a browser instance seen with a new feature value is considered to be changed in all of its subsequent occurrences in the data, regardless of its feature value in these occurrences.

5.1.2 Information Gain and Entropy

The information gain for a feature is, as described in section 2.2, a measure of the information about the label, which is a specific browser instance in this case, gained by knowing the value for the feature at hand. This is useful for comparing the relevance of features when selecting which features to consider when collecting and classifying data.

5.1.3 Degree of Functionality

Functionality is a property of a binary relation. A binary relation, R, between two sets, A and B, is a subset of the Cartesian product of the two sets, $R \subset A \times B$. Then $a \in A$ is R-related to $b \in B$ if $(a, b) \in R$, which is commonly denoted as $R(a, b)$ or aRb. The two sets, A and B, are called the domain and co-domain, respectively, for

R. A binary relation, $R \subset A \times B$, is said to be functional if

$$\forall a \in A \,.\, \forall b, c \in B \,.\, R(a, b) \wedge R(a, c) \rightarrow b = c \,. \tag{5.1}$$

The relation considered in the measurement is $R_i(v, l) =$"feature value v for feature i has been seen in an instance with label l". Thus, the domain for the relation is all possible values for the given features and the co-domain is all known labels, or more precisely;

$$R_i = \{(v, l) \in F_i \times L \mid \exists \mathbf{x} \,.\, (\mathbf{x}, l) \in D_v^{F_i}\} \,, \tag{5.2}$$

where i is the given feature. For this relation, the degree of functionality, that is to which extent the feature i makes the relation functional, was measured as the expected number of labels associated with a feature value, L_{F_i}. The relation is truly functional if there is exactly one label per feature value.

5.2 Results from Data Analysis

As mentioned in chapter 4, there are four different data sources, three partitions with two or four parts which may be combined into larger partitions with up to 16 parts. In total, this results in 176 different data sets to be analysed. This section presents a small subset of these results, which have been selected for being the most interesting and relevant.

Some of the investigated features are not available from all data sources, in which case the choice of data set was simple. Even though the number of recurring visitors was much greater in the company data than in the research data, the number of recurring visitors after more than two weeks is actually greater in the research data, why this data set has been prioritised in the analysis.

5.2.1 Temporal Feature Value Changes

The results from the analysis of feature value changes are presented in this section, by feature. Only those features that produced especially interesting results are considered. For the research data, only changes per week was measured, since the majority of the recurring visitors visited to the site once a week (see subsection 3.1.4).

IP

In Figure 5.1, the IP changes over time are visualised for different device types. The graph shows a very distinct difference between the rate of change for handheld devices versus computers; handhelds are much more prone to change IP than computers. The values in the graph was calculated from the campaign data, but the other data sources displayed similar behaviour.

The fraction of recurring browser instances with changed IP grows very fast the first day, and tapers thereafter. Moreover, there seems to be a component which oscillates periodically with period 24 hours for both handheld devices and computers.

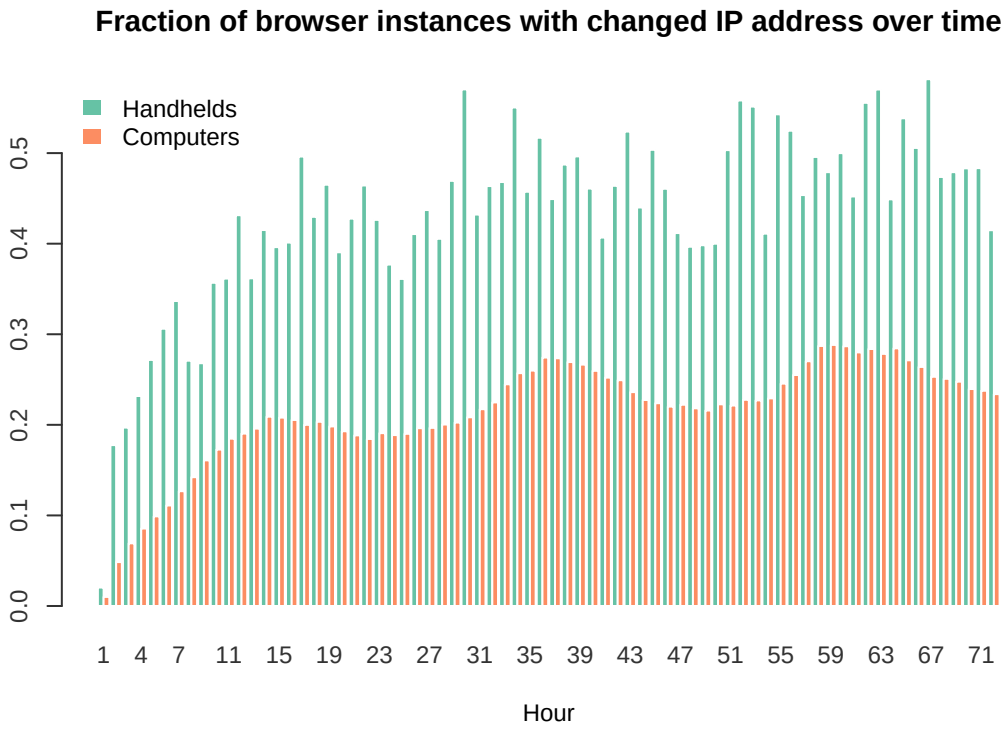

Figure 5.1: The graph shows IP changes per device type for campaign data. The bars represent the fraction of recurring browser instances who showed a changed IP in the xth hour from their first occurrence.

To investigate how common changes between different C-class networks were, that is changes in other octets of an IP than the least significant, the same measurement was applied to IP addresses with the least significant octet removed. The results from this measure showed a slightly lower rate of change than for the full IP, but no less than to be expected when removing information prior to analysis. The decrease

was thus not significant enough to draw any conclusions regarding whether changes between C-class networks are exceptionally uncommon or not.

Screen and Window Dimension

For screen dimension changes, the difference between handheld devices and computers is even more clear than for IP changes. As seen in Figure 5.2, the fraction of instances displaying screen resolution changes for handheld devices is more than four times larger than that of computers. Screen dimension changes are however much less frequent than IP changes; after two weeks only about 0.05 of all computers and just over 0.20 of all handheld devices have changed their screen dimension. This graph is also generated from campaign data.

In an attempt to further investigate the more frequent changes of screen resolution for handheld devices, which was assumed to be due to the possibility to rotate the screen, the change rate for the total number of pixels on the screen ($\text{width} \cdot \text{height}$) was also investigated. However, the results for this measure was nearly identical to those presented in Figure 5.2.

The window dimension displayed a very high change rate and no apparent pattern in these changes could be found, which makes it ineligible for using as an identifier.

RTT, Client Clock Error and Timezone

The RTT feature was actually a vector of RTTs (see subsection 3.1.2) measured in milliseconds, making it extremely unlikely that the exact same feature value is seen twice. To remedy this, the mean RTT was calculated before analysing the change rate, but it turned out that even the mean value was too sensitive. Rounding the mean RTT lead to a lower change rate, but in turn also led to a much lower uniqueness since many browser instances shared the same RTT when rounded. Several rounding schemes were applied, but neither gave the desired result.

The client clock error was represented as an interval, in which the true clock error resided. This was also measured in milliseconds and proved to be much too sensitive. The same methods as those for RTT were applied, but proved to be fruitless for the client clock error as well.

Changes in the timezone feature were, not surprisingly, very uncommon, making it a very stable feature. Some changes were seen, however insignificant, for computers but none for handheld devices.

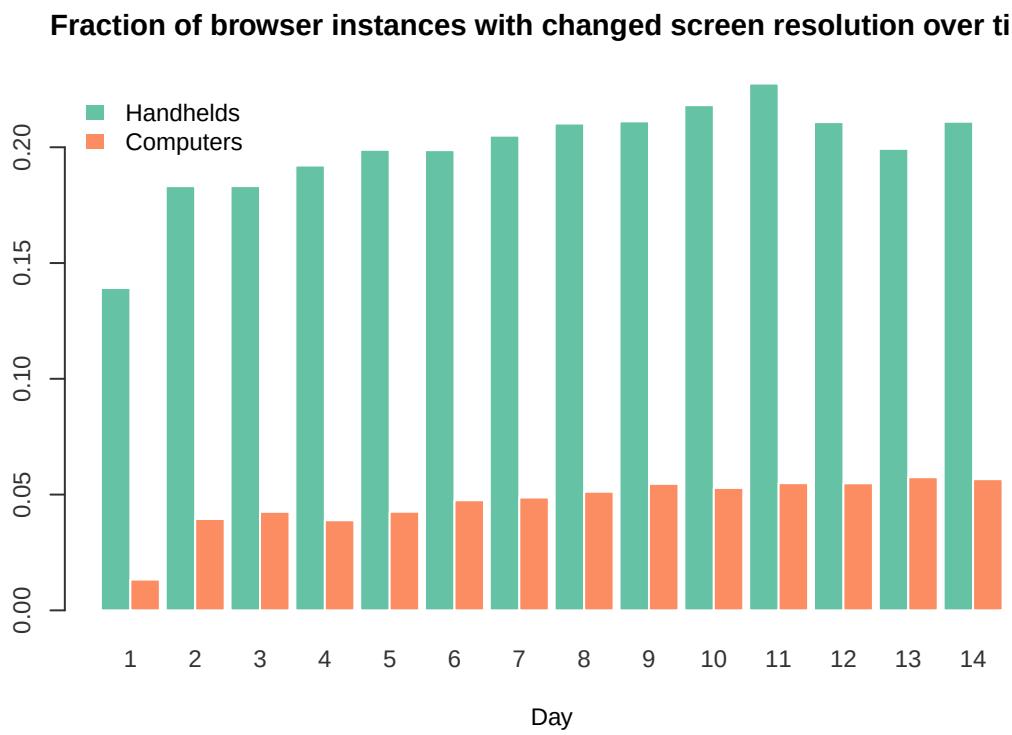

Figure 5.2: The graph shows screen dimension changes per device for campaign data. The bars represent the fraction of recurring browser instances who showed a changed screen dimension in the xth day from their first occurrence.

Browser Plugins and System Fonts

Both browser plugins and system fonts are important features (Eckersley 2010), and the change rate for those are presented in Figure 5.3. The change rate for the browser plugin list seems to be constant, and just over 0.60 of all browser instance displayed changes after a time period of two months. For the system font list (retrieved using Adobe Flash), the behaviour is similar, but with less than 0.50 of all browser instances having changed feature value in the same time period.

The system font list retrieved using CSS exploits displays a much more unstable behaviour; with a much higher rate of change and less consistency.

Accept Headers

The two analysed Accept headers, Accept-Language and Accept-Charset, were quite stable. For Accept-Charset, the fraction of changed feature values in one week was significantly less than 0.05, and the same figure for the Accept-Language feature

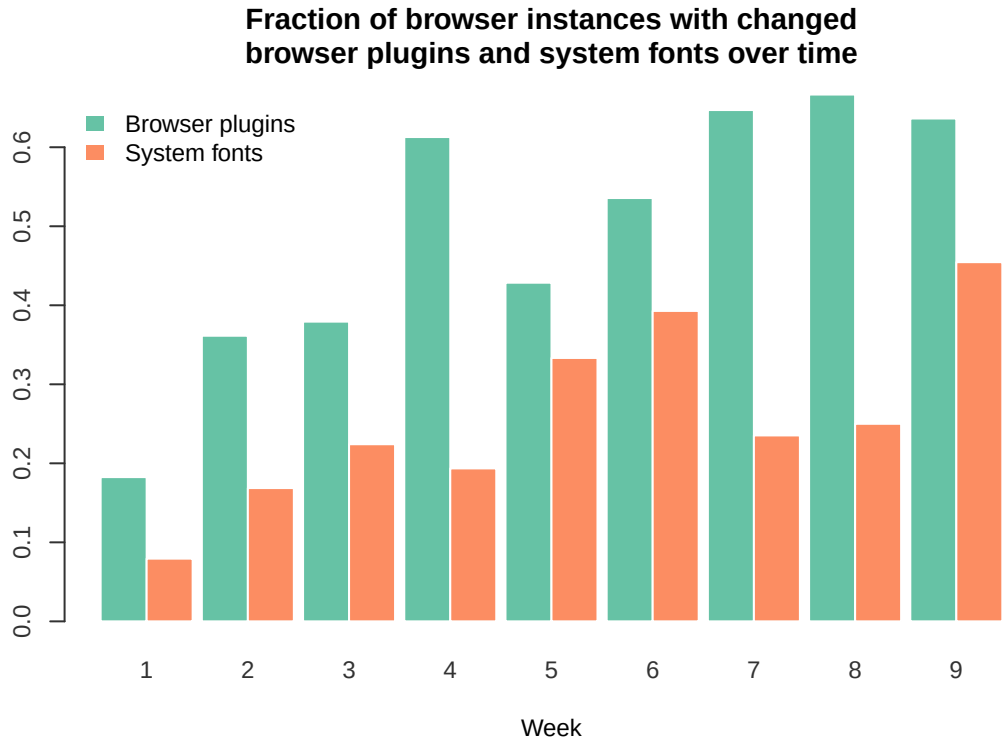

Figure 5.3: The graph shows browser plugin and system font list changes for the research data. The bars represent the fraction of recurring browser instances who showed a changed feature value in the xth week from their first occurrence.

was around 0.01. Moreover, when analysing the data partitioned on web browser, it became clear that the browser most frequently changing these headers was Mozilla Firefox. The other browsers were several times less likely to change these features. This behaviour was identified in the company data as well as in the research data.

The browser language was assumed to behave similarly to the Accept-Language feature, since both probably are the result of some settings made by the user. The actual change rate turned out to be similar, but when partitioning the data on browser again, Internet Explorer was the only browser displaying any changes for the browser language setting.

User-Agent String and Browser Version Number

The UA contains more or less detailed information about a web browser and the OS, depending on which web browser it originates from. In Figure 5.4, the change rate of the complete UA is compared to that of the browser version number, for different browsers. The change rates are nearly identical for the complete UA and the

browser version number for all browsers, only Internet Explorer displays a noticeable difference, albeit small.

However, the change rates between browsers differs much; after three weeks, between 0.7 to 0.9 of browser instances from Google Chrome or Mozilla Firefox have shown changes in their UA, while only 0.1 to 0.2 of the browser instances from Safari and Internet Explorer show any changes.

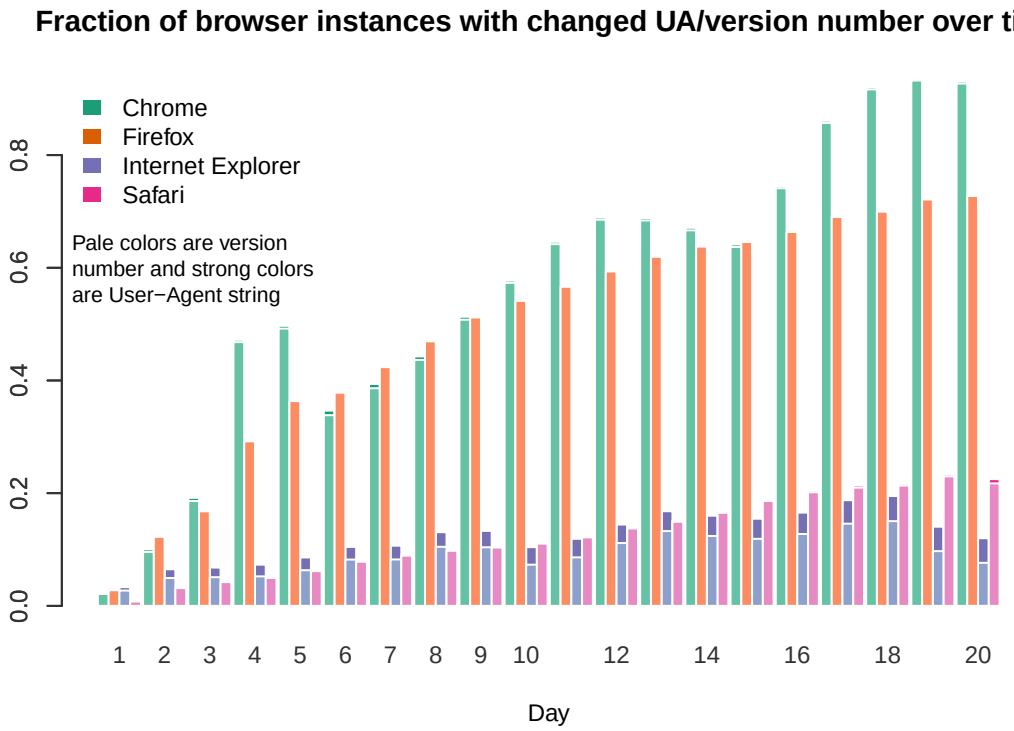

Figure 5.4: The graph shows UA and browser version number changes per browser from campaign data. The bars represent the fraction of recurring browser instances who showed changes in the xth day from their first occurrence. The strong coloured bars represent UA changes and the pale, plotted on top of the strong coloured bars, represent version number changes.

5.2.2 Entropy and Information Gain

The results from the information gain measurements clearly showed that there was a great variation in information gain for different features, as depicted in Figure 5.5. The features are sorted descending on information gain from left to right, and it is evident that IP, browser plugins and system fonts are important features.

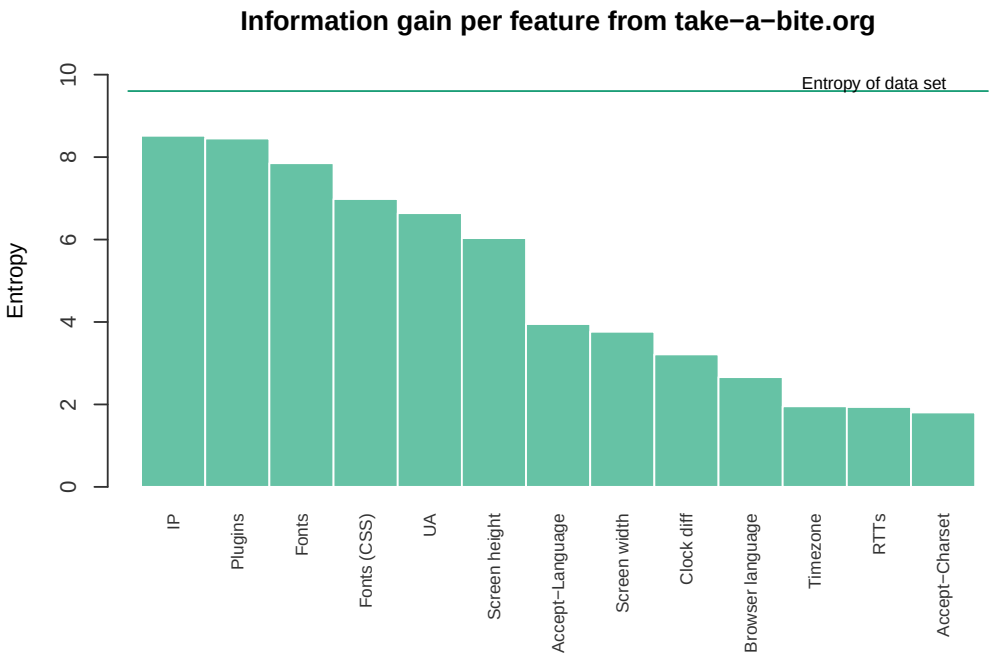

Figure 5.5: The graph shows information gain for a subset of the features collected from `take-a-bite.org`.

Moreover, there were large differences in information gain for individual features for different data sources (see Figure 5.6), but the relative difference between features was similar regardless of the data source, in most cases.

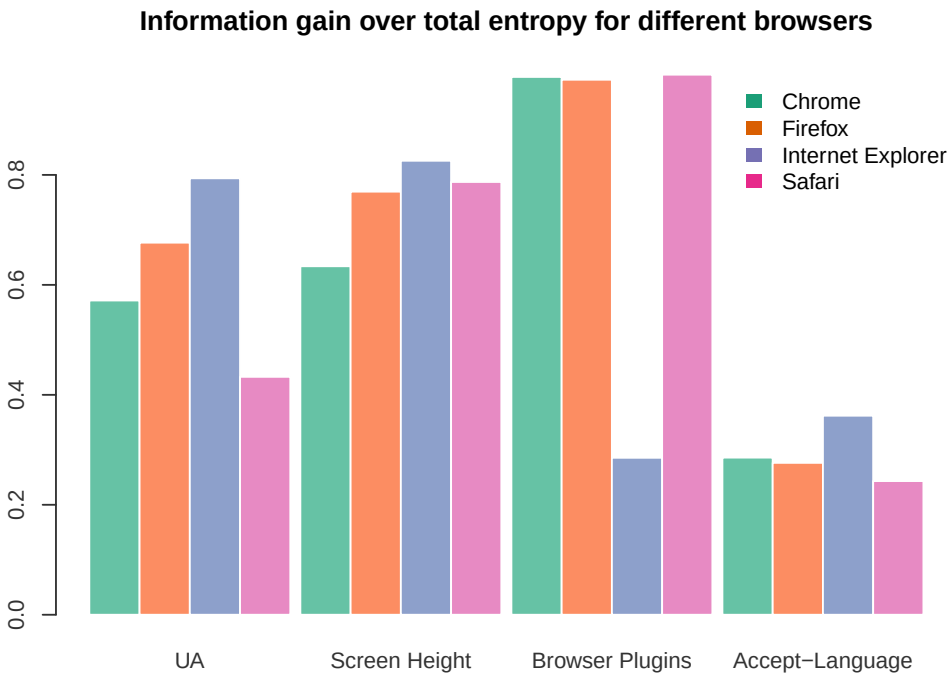

Figure 5.6: The graph shows information gain over total entropy of dataset for different browsers.

5.2.3 Degree of Functionality

The degree of functionality for the feature–label relation is shown is Table 5.1 as the expected number of labels associated with one feature value for the 13 features from the research data.

Generally, the less number of labels associated with a feature value, the more unique the feature value, which in turn indicates that the feature is suitable for using to identify browser instance. However, some of the features, marked with an asterisk, are are seemingly unique per label, but shows a very high rate of change which renders them poorly suited for using to identify browser instances with.

Feature	$E[L_{F_i}]$
IP	1.19
User-Agent	3.43
Browser Plugins	1.33
Browser Language	31.83
Screen Width	13.95
Screen Height	5.55
Timezone	51.59
Minimum Clock Error*	1.40
Fonts (via Adobe Flash)	1.47
Font (via CSS exploit)*	1.60
RTT*	1.04
Accept-Language	10.11
Accept-Charset	72.50

Table 5.1: Expected number of labels associated with a feature value, for the 13 features from the research data. (*) Features that display extremely high rate of change.

Chapter 6

Designing a Heuristic Online Classifier for Internet Tracking Data

There is no way of determining the true labels of the instances in the data sets described in chapter 4, since the labels are set using HTTP cookies and/or Adobe Flash LSO which are both fallible methods. There is thus no way of proving whether a classifier is correct or not, which, in combination with the shear complexity of the problem, led to the decision of seeking a *heuristic* solution.

The browser fingerprints are dynamic and new browser instances are encountered frequently, why an adaptive classifier is required to be able to respond to these continuous changes. Therefor, *online* classifiers are the main focus of this thesis.

This chapter aims to describe the design process and the choices therein, as well as the results obtained when testing the designed algorithms.

6.1 Designing the Classifier

One of the key abilities a solution to this problem must have, aside from identifying previously seen browser instances, is being able to recognise whether the browser instance has been seen before at all or not. Categorical classifiers work with some finite set of labels, among which the true label reside, and returns the most likely or best matching label when classifying an instance. For this problem, it may very well be the case that the true label is not within the set of known labels, why the classifier must be able to not only return a known label but also have mechanisms for generating new labels for previously unseen browser instances. The structure of the model, albeit simplified, is visualised in Figure 6.1. The classifier consists of two

parts; a supervisor (referred to as *internal supervisor*) and a classifier (referred to as *internal classifier*).

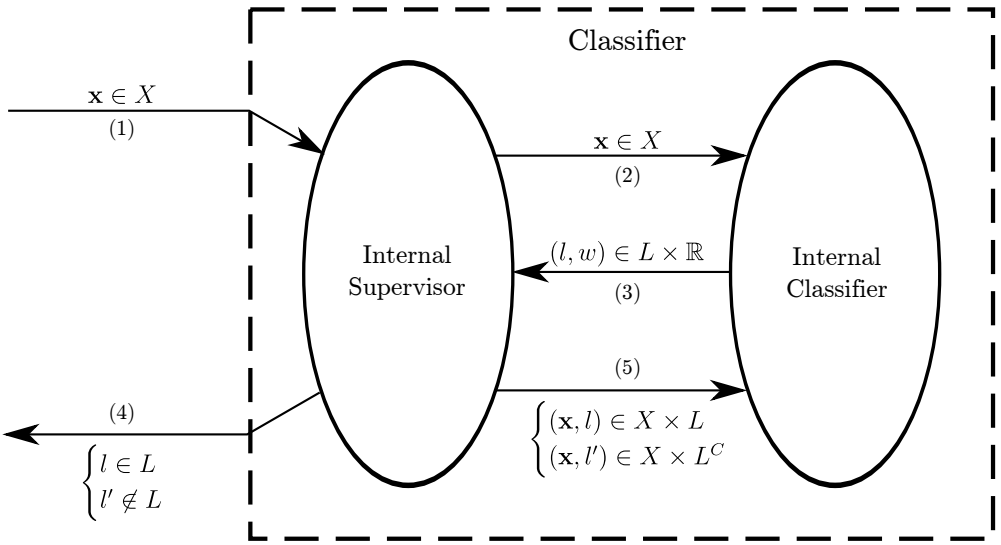

Figure 6.1: This figure represents a structural classifier model description. Given an instance $\mathbf{x}$ (1), the internal supervisor asks the internal classifier (2) to find the most likely label, $l \in L$, and some weight, $w \in \mathbb{R}$, (3). The weight is then compared against some threshold, δ, to determine whether to believe that the instance $\mathbf{x}$ has $c(\mathbf{x}) = l$ or originates from a previously unseen label $l' \notin L$ (l' is a label generated by the internal supervisor), after which it returns l or l' (4) and retrain the internal classifier with either $(\mathbf{x}, l)$ or $(\mathbf{x}, l')$ (5).

For online classifiers, after an instance has been classified the predicted label is, at some later time, compared with the true label and the information is used to refine the classifier. This requires that there is a way to find the true labels of instances as some time after the prediction, which is not the case for this problem; there is no feasible way to learn the true identity of browser instances, with perfect accuracy. This implies that the classifier must be retrained without knowledge of the true label, meaning that it is not truly online. However, the way the classifier is iteratively updated is still the same as that of an online classifier. In the classifier described in Figure 6.1, the internal supervisor's final answer is, internally, assumed to be correct and used for retraining the internal classifier. There is one major drawback with retraining with potentially erroneous labels; the performance of the classifier will degrade for every incorrect answer, since it implies training the internal classifier with false information.

		Actual	
		Positive	*Negative*
Predicted	*Positive*	$TP = TTP + FTP$	FP
	Negative	FN	TN

Table 6.1: A *confusion matrix* including the non-binary answers to the problem.

6.1.1 Measuring Classifier Performance

There are several common measures used when evaluating classifiers (see subsection 2.1.5). However, neither are these intended to be used for online classifiers, nor are they directly applicable to the classification problem in this thesis. The classifier does not return a binary answer; it can either return a known label, which may or may not be correct, or say that the instance belongs to a previously unknown label, and these can clearly not be divided into positive and negative answers.

The classifying process can however be divided into two parts, to facilitate evaluation; by first determining whether the browser instance has been seen before or not, and then, given that it has been, determine which browser instance it is. The first problem has a binary answer, either the browser instance has been seen before or it has not, why the methods described in subsection 2.1.5 are applicable. The second problem only considers those instances who have caused a positive answer to the first problem. The positive answers can be divided into three categories; either the classifier is correct (1), or it mistook a known (2) or unknown (3) browser instance for another known one. (1) is a true positive and (3) is a false positive, but (2) is not as easily definable. Considering only the first problem, (2) is also a true positive since the browser instance had in fact been seen before, even though the classifier failed to identify it correctly. To incorporate this in the measurements, the true positive category has been split into *true true positive* and *false true positive* for (1) and (2), respectively, as visualised in Table 6.1.

Moreover, when the classifier encounters a previously unseen browser instance, and identifies it as such, subsequent occurrences of this browser instance in the data should be identifiable. When testing instances, the classifier cannot learn the true labels of the instances, since those would not be known in a real world scenario, which implies that some additional functionality had to be added to the evaluation process to be able to verify whether subsequent classifications of instances previously thought to be new instances were correct or not. In the scenario that a new browser instance, with true label l, was identified as a new browser instance and given the label l', subsequent classifications of l as l' should be considered correct (TTP). Furthermore, if l is later

falsely identified as another new browser instance, l'', subsequent classifications of l as l'' should be correct and classifications of l as l' incorrect. This somewhat complex behaviour was designed in cooperation with the Company, to fit their potential use of such an algorithm, and is presented in algorithm 6.1.

Algorithm 6.1 Algorithm for evaluating online classifiers.

$\forall l \in L : A_l \leftarrow \{l\}$
$\forall l \in L : a_l \leftarrow l$
for all $\mathbf{x} \in D_{test}$ **do**
  classify $\mathbf{x}$ to obtain $l(\mathbf{x})$ and $w \in \mathbb{R}$
  if $w > \delta$ **then**
    if $c(\mathbf{x})$ has been seen before **then**
      if $l(\mathbf{x}) = a_{c(\mathbf{x})}$ **then**
        $TTP \leftarrow TTP + 1$
      else
        $FTP \leftarrow FTP + 1$
      end if
    else
      $FP \leftarrow FP + 1$
    end if
    if $l(\mathbf{x}) \in A_{c(\mathbf{x})}$ **then**
      $a_{c(\mathbf{x})} \leftarrow l(\mathbf{x})$
    end if
    re-train classifier with label $l(\mathbf{x})$ for instance $\mathbf{x}$
  else
    if $c(\mathbf{x})$ has not been seen before **then**
      $TN \leftarrow TN + 1$
    else
      $FN \leftarrow FN + 1$
    end if
    $l' \leftarrow$ new unique label
    $a_{c(\mathbf{x})} \leftarrow l'$
    $A_{c(\mathbf{x})} \leftarrow A_{c(\mathbf{x})} \cup \{l'\}$
    re-train classifier with label l' for instance $\mathbf{x}$
  end if
end for

A classifier always returns a label, and that label is obviously known from training or re-training, and $w \leq \delta$ means that the instance is to be considered to originate from a previously unseen browser instance.

Redefining Standard Measures

Intuitively, the accuracy is the fraction of correct answers, and for this reason the *online accuracy* for the classifiers treated in this chapter is defined as

$$\text{online accuracy} = \frac{TTP + TN}{TTP + FTP + TN + FP + FN} \,. \tag{6.1}$$

It is important to know that what is referred to as online accuracy throughout this chapter, is *not* the same accuracy as defined in subsection 2.1.5 for two reasons; the online accuracy describes a property of an online classifier rather than one of a static classifier, and secondly, the online classifier has more than two labels.

In a similar manner to the online accuracy, the *online precision* and *online recall* are defined as follows:

$$\text{online precision} = \frac{TTP}{TTP + FTP + FP} \tag{6.2}$$

$$\text{online recall} = \frac{TTP}{TTP + FTP + FN} \tag{6.3}$$

Online precision is thus the fraction of correct classifications of all in which the answer from the internal classifier was used (when $w > \delta$), and online recall the fraction of correct classifications on instances whose true label has been seen before. These can be combined into the *online* F_1*-score*,

$$\text{online } F_1\text{-score} = 2 \cdot \frac{\text{online precision} \cdot \text{online recall}}{\text{online precision} + \text{online recall}} \,. \tag{6.4}$$

Browser Instance Count Error

One application for identifying browser instances is in calculating the number of unique browser instances who has requested some online resource during some time period. This aggregated information was also considered as a performance indicator for a classifier, and the error in browser instance count, ϵ, was calculated as

$$\epsilon = \frac{|L_C| - |L|}{|L|} \,, \tag{6.5}$$

where L is the set of labels in the data set and L_C is the set of labels known to the classifier after training and testing. L_C will contain those labels generated in the classification process, that are not in L, and might hence be larger than L.

6.1.2 Static Fingerprinting Method

As a baseline, a very simple static fingerprinting algorithm was developed, which is described in full in algorithm 6.2.

Algorithm 6.2 Description of a very simple static fingerprinting algorithm.

```
if l(x) is undefined then
  l(x) ← new unique label
end if
return l(x)
```

The static fingerprinting algorithm simply associated a unique label to every unique fingerprint seen, and returned this if it was presented with an identical fingerprint. The method clearly had many flaws, among which the lack of adaptability probably was the most important. In this model, the internal supervisor was algorithm 6.2 and the internal classifier the mapping $l : X \to L$.

6.1.3 Using Naïve Bayes as Internal Classifier

In an attempt to improve the classifier by making it less sensitive to changes, the mapping l from the static fingerprinting method was replaced with a Naïve Bayes classifier as the internal classifier. As discussed in subsection 2.1.3, Naïve Bayes assumes complete independence between features, which seldom holds true in real-world applications, and has a clear distinction between training and testing. This was not however the case when using a system as that described in Figure 6.1, in which the training and testing phases were continuous and interleaved.

Algorithm Design

The goal for Naïve Bayes is to, given an instance $\mathbf{x}$, find the label l, that maximises $P(l|\mathbf{x})$, according to (2.3). The probabilities in this equation are frequencies in the data set, which should be easy to extract during the classification process. For an offline Naïve Bayes these frequencies could be calculated only once after the training phase, but since an online classifier would be updated for every classified instance such preprocessing is inefficient. Moreover, because of the frequent updates, ensuring that updating the classifier was a fast operation was important for the running time of the algorithm.

If all instances in a specific data set have exactly m features, then the frequency matrix F^i for each feature $i = 1, 2, \ldots, m$ can be described as a matrix on the form:

$$F^i = \begin{array}{c} \\ l_1 \\ l_2 \\ \vdots \\ l_k \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & \dots & v_j \end{array} \\ \begin{pmatrix} f^i_{l_1,v_1} & f^i_{l_1,v_2} & \dots & f^i_{l_1,v_j} \\ f^i_{l_2,v_1} & f^i_{l_2,v_2} & \dots & f^i_{l_2,v_j} \\ \vdots & \vdots & \ddots & \vdots \\ f^i_{l_k,v_1} & f^i_{l_k,v_2} & \dots & f^i_{l_k,v_j} \end{pmatrix} \end{array},$$

where j is the number of feature values for feature i and k is the total number of observed labels. As described in Algorithm 6.3, training the internal classifier with $(\mathbf{x}, l)$ implies incrementing f^i_{l,x_i} for all features i, incrementing the total number of observed instances, n, and the number of observed instances with label l, n_l. Moreover, to keep track of all possible values per feature, a vector $\mathbf{V}$, defined such that V_i is the set of all possible values for feature i, is also updated. The feature frequency matrices are initialised with 0 for all values and labels.

Algorithm 6.3 Algorithm for training Naïve Bayes with a training instance $(\mathbf{x}, l)$ with m.

$n \leftarrow n + 1$
$n_l \leftarrow n_l + 1$
for all $i = 1, 2, \dots, m$ **do**
  $V_i \leftarrow V_i \cup \{x_i\}$
  $f^i_{l,x_i} \leftarrow f^i_{l,x_i} + 1$
end for

The algorithm 6.4 describes the classification of an instance. All values are normalised with the total probability of $\mathbf{x}$. For each label, the probability $P(x_i|l)$ is calculated. The algorithm employs Dirichlet smoothing with an assumption that each feature value is uniformly distributed, that is $p_i = \frac{1}{|V_i|}$. The Dirichlet smoothing is parametrised over q and can be fine-tuned for scaling the importance of a priori observations. The classification process must consider all $l \in L$ and then return the most probable browser instance along with the probability for that instance. Since the internal classifier always returns a label, the returned probability can be used to determine if the result should be trusted or not by comparing it to a threshold, δ.

Implementation Process

The classifier was initially implemented in MATLAB, in order to facilitate the performance evaluation using MATLAB's built-in plotting tools and various utilities for statistical analysis. However, evaluating the classifier was very time-consuming and

Algorithm 6.4 Algorithm for classification of an instance $\mathbf{x}$ with m features and Dirichlet smoothing parameter q using Naïve Bayes.

$p_{\mathbf{x}} \leftarrow 0$
$P \leftarrow$ `Array` of size $|L|$
for all $l \in L$ **do**
  $p_l \leftarrow n_l/n$
  for all $i = 1, 2, \ldots, n_F$ **do**
    $p_i \leftarrow 1/|V_i|$
    $p_l \leftarrow p_l \cdot (F_{ki} + q \cdot p_i)/(n_l + q)$
  end for
  $p_{\mathbf{x}} \leftarrow p_{\mathbf{x}} + p_l$
  P.`append`$((p_l, l))$
end for
$p^* \leftarrow 0$
for all $(p_l, l) \in P$ **do**
  $p_l \leftarrow p_l/p_{\mathbf{x}}$
  if $p_l > p^*$ **then**
    $(l^*, p^*) \leftarrow (l, p_l)$
  end if
end for
return (l^*, p^*)

could not be done on the Company's servers, since MATLAB is a proprietary software requiring a license. This drawback lead to the re-implementation of the entire system in Ruby, which was chosen partly for being the primary language used by the Company and partly for being a high-level language. The running time for the classifier was actually worsened after the migration, but it was possible to run the software on the Company's servers. The Ruby version of the algorithm used the `Hash`-object to a large extent, which proved to be a bottleneck; re-allocating new memory for a full hash was time-consuming. To remedy this, the software was rewritten once again, and this time in Python. Python is very similar to Ruby, making the migration process easy. Python seemed to handle the re-allocation of hashes in a better way than Ruby, and the running time was reduced greatly.

6.1.4 Adding Rules to the Internal Supervisor

As the system is described in Figure 6.1, the only information available to the internal supervisor for determining whether a browser instance has been seen before or not is the weight or probability calculated by the internal classifier. It could be possible to increase the probability of making a correct decision if additional information about

the system is incorporated into the model. Such information could be rules defining certain answers that are either directly dismissible or acceptable.

To verify that the classifier performance could be enhanced by adding such rules, a rule stating that any answer including a browser instance seen with a higher browser version number than that of the instance to be classified should be dismissed, and the browser instance should be considered to be new. The performance of the classifier with the additional rule was compared to that of the classifier without the rule.

6.1.5 Investigating the kNN Classifier

To be able to use the kNN algorithm, there has to exist some distance metric for every feature. These metrics are crucial to the performance of kNN, and has to be carefully designed so as not to give to much precedence to less important features.

Moreover, if kNN is to be used as internal classifier, the only way of knowing whether to believe the classifier or not would be to inspect the distance, or some function of it, to the returned label, why the distance metrics must be very reliable. That is, a correct guess should always have a smaller distance than an incorrect guess to enable to use of a threshold, as described previously.

Due to the lack of data with many features, only a brief investigation of kNN was conducted. During this investigation, the following features was considered: browser plugins, screen height, system fonts and Accept-Language. For browser plugins, system fonts and Accept-Language, the distance between feature values was calculated using the Tanimoto metric, see Equation 2.8. The distance between two screen height values, h_1 and h_2, was calculated as

$$d(h_1, h_2) = \frac{|h_1 - h_2|}{h_1 + h_2}\,, \tag{6.6}$$

to be comparable with the Tanimoto metric, which always is in $[0, 1]$. The L_2-norm was used to calculate the length of the resulting vector. Originally, some string valued features were included and the distance between their values was measured using the Levenshtein distance, which proved to be a very time-consuming algorithm, why these were removed.

The classification among the k nearest neighbours was settled using weighted majority vote. No suitable threshold value for δ could be found, why the evaluation was run solely on recurring instances. The research data was used, and five randomised 60-40 percentage splits per measure was done.

6.2 Results

6.2.1 Test Set Accuracy

Data set	Count error	TTP	FTP	TP	TN	FP	FN	Online accuracy
Android	0.414	40,053	624	40,677	0	0	9,323	0.80
Internet Explorer	0.162	42,772	55	42,827	0	0	7,173	0.86
Windows Chrome	0.058	47,403	546	47,949	0	0	2,051	0.96

Table 6.2: The table shows the browser instance count error and online accuracy for an experiment where the training set was used as test set as well. The experiments where conducted on three data sets from the campaign data.

The driver for the test set accuracy test is simple; if the classifier, during the test phase, has observed every instance already, the overall performance is expected to be good.

This experiment was realised by simply training and testing the classifier on the complete data sets, and the results thereof are shown in Table 6.2. The true negative and false positive counts were zero for all data sets, which is to be expected since every instance has been observed during training, according to the definition of test set accuracy.

The lowest online accuracy, 0.80, was observed for the Android instances and the highest, 0.96, for instances from Google Chrome on Microsoft Windows. These are high figures, if compared with the other tests using the Naïve Bayes classifier.

Despite not giving the best results, the Internet Explorer data set gave a significantly lower FTP count, meaning that the instances yielding a high enough probability to exceed the threshold δ were correct to a larger extent than for the other two data sets.

6.2.2 Feature Selection

To identify which set of features produced the best classification results, the static fingerprinting method was applied to all possible combinations of the 13 features from the research data. By comparing the online accuracy and online F_1-score of the results, three features was immediately found to be disinformative, meaning that the online accuracy of classification using combinations of features including any of these was lower than if the feature were removed. These three features were: clock error, system fonts retrieved via CSS exploit and RTT. Moreover during the same analysis,

four features which did not seem to affect the accuracy was also identified; browser language, screen width, timezone and Accept-Charset.

Excluding these seven leaves six features, of which 64 combinations can be made (including the empty set). A vast majority of the best performing best half of these combinations, 30 of 32, included the IP address feature, which indicated that it should be part of a classifier.

By only considering combinations in which IP addresses is present, only 32 combinations remained. Within these 32, every combination including the Accept-Language feature resulted in better classification performance than the same combination without the feature, why this feature also should be included.

The 16 combinations of the remaining four features are presented in Table 6.3, sorted by online accuracy (which coincidentally is the same ordering as for online F_1-score). The table shows that the best performing set of features, according to both online accuracy and online F_1-score, consist of IP, system fonts and Accept-Language. However, the performance differences between this combination and those following is very small, why any conclusions should be drawn with caution.

IP	UA	P	H	F	AL	Online accuracy	Online F_1-score
×				×	×	0.816	0.638
×			×	×	×	0.814	0.623
×	×			×	×	0.805	0.604
×		×			×	0.804	0.602
×		×		×	×	0.801	0.594
×			×		×	0.796	0.586
×	×	×			×	0.795	0.578
×	×	×		×	×	0.792	0.571
×	×		×	×	×	0.791	0.566
×		×	×		×	0.791	0.561
×		×	×	×	×	0.788	0.554
×	×				×	0.783	0.555
×	×		×		×	0.783	0.545
×	×	×	×		×	0.782	0.540
×	×	×	×	×	×	0.780	0.536
×					×	0.757	0.533

Table 6.3: The table shows the performance of different combinations of the six most promising features from the research data; IP, UA, plugins (P), screen height (H), system fonts retrieved via Adobe Flash (F) and Accept-Language (AL).

It is important to note that these results originate from the relatively small research data set and the static fingerprinting method. There is no guarantee that using

these feature sets would give the same results when used any other classifier on any other data set, but the results give an indication on which features are important.

6.2.3 Naïve Bayes Parameter Selection

The internal classifier Naïve Bayes have been implemented using a parametrised smoothing called *Dirichlet smoothing* as described in subsection 6.1.3. The classifier itself is parametrised over δ, a threshold which is used to determine whether the result of the internal classifier should be trusted or not. The internal classifier is parametrised over q, a factor used to scale the importance of a priori probabilities.

A number of experiments have been conducted to find the optimal parameters q and δ for each data set. For each $q \in \{10^{-3}, 10^{-2}, 10^{-1}, 10^{0}, 10^{1}\}$ experiments where conducted with the thresholds $\delta \in \{0.65, 0.70, 0.75, 0.80, 0.85, 0.90, 0.95, 0.99\}$. The internal classifier was trained with 0.50 of the data set and tested with the remaining 0.50. The single-site and the campaign data sets were used in this experiments.

Both Figure 6.2 and Figure 6.3 show similar behaviours, where a high δ threshold seems to maximise the online accuracy for the data sets, whereas the parameter q differs. Classifying the Google Chrome data seems to be more sensitive to increasing the scaling of a priori probabilities and low values for q seems to improve the results. For the Internet Explorer data, increasing q, and thus the importance of a priori probabilities, seems to increase the performance.

There are vast differences in online accuracy behaviour for different parts of the data set partitions, which is evident when comparing Figure 6.4 and Figure 6.2, for instance. Figure 6.4 does not have a clear maximum for a specific combination of δ and q, and many different combinations seem to yield the same result.

It is noteworthy that the browser instance count error in Figure 6.5 does not follow share the same optimal q-δ combination as the online accuracy for the same data set. The aggregated measure browser instance count error behaves differently, and it is clear that the application of the method is important for the choice of parameters.

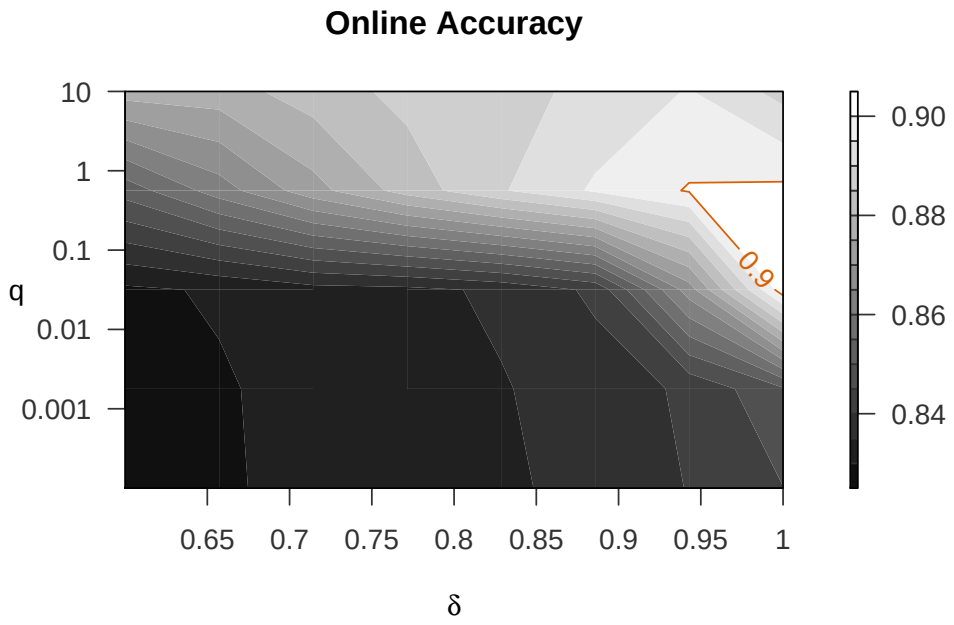

Figure 6.2: The graph shows the online accuracy when using Naïve Bayes as internal classifier, on browser instances using Internet Explorer on Microsoft Windows from the campaign data, for different values of q and δ.

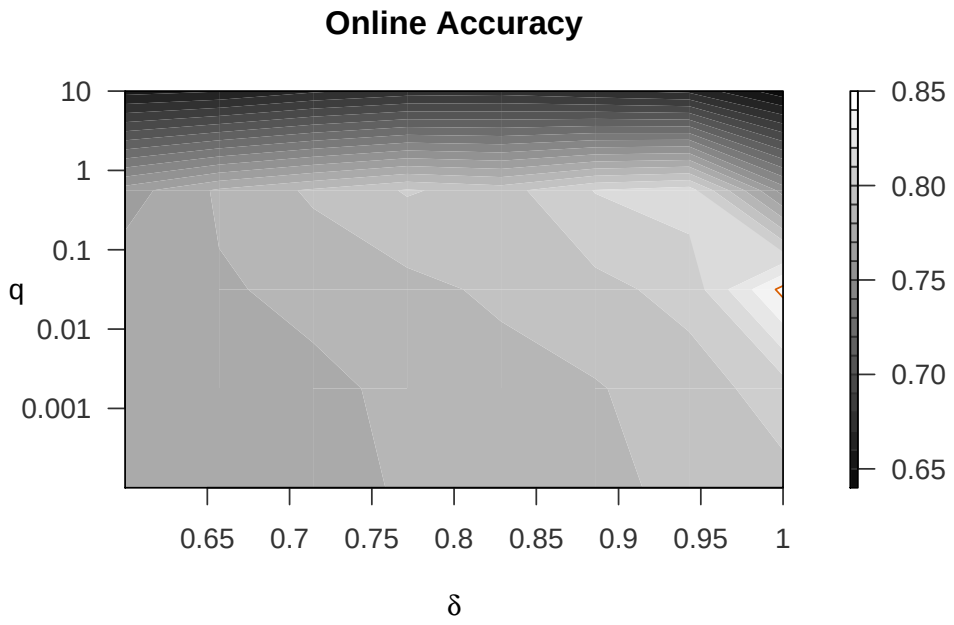

Figure 6.3: The graph shows the online accuracy when using Naïve Bayes as internal classifier, on browser instances using Google Chrome on Microsoft Windows from the single-site data, for different values of q and δ.

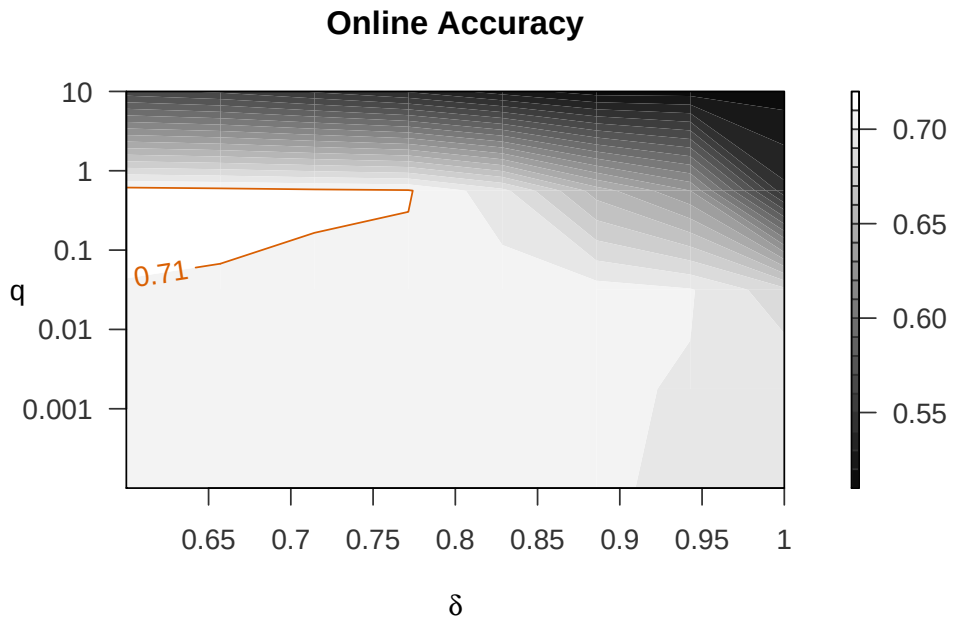

Figure 6.4: The graph shows the online accuracy when using Naïve Bayes as internal classifier, on browser instances using Android handhelds from the campaign data, for different values of q and δ.

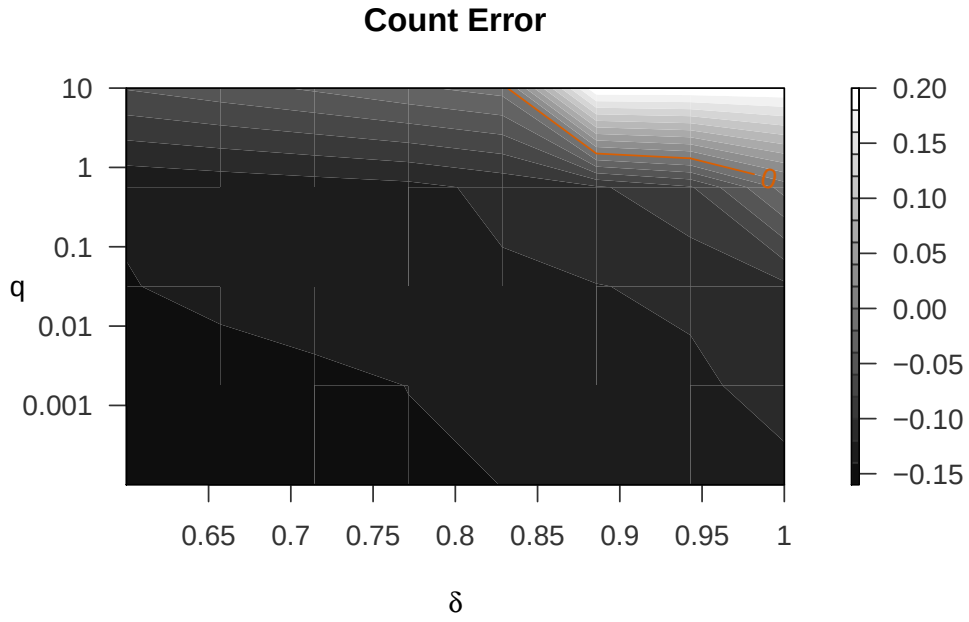

Figure 6.5: The graph shows the browser instance count error when using Naïve Bayes as internal classifier, on browser instances using an Google Chrome on Microsoft Windows from the campaign data, for different values of q and δ.

6.2.4 Classifier Temporal Degrading

To investigate to which extent the classification performance decreases over time as changes occur, the static fingerprinting method was applied to data for several different time periods. For these time periods, online accuracy and browser instance count error was measured and the results are presented in Figure 6.6. The features used to create the fingerprint was IP, UA and screen resolution. The graph comprises three different parts of a partition of the campaign data, each with quite different properties and behaviour; Internet Explorer and Google Chrome on Windows and handheld devices running Android. All three data sets caused similar classification degrading over time with regard to online accuracy, but differences among the three can be seen for the browser instance count error.

In an attempt to motivate the results from the temporal degrading measure, the rate of change for the complete fingerprint was also investigated (see Figure 6.7).

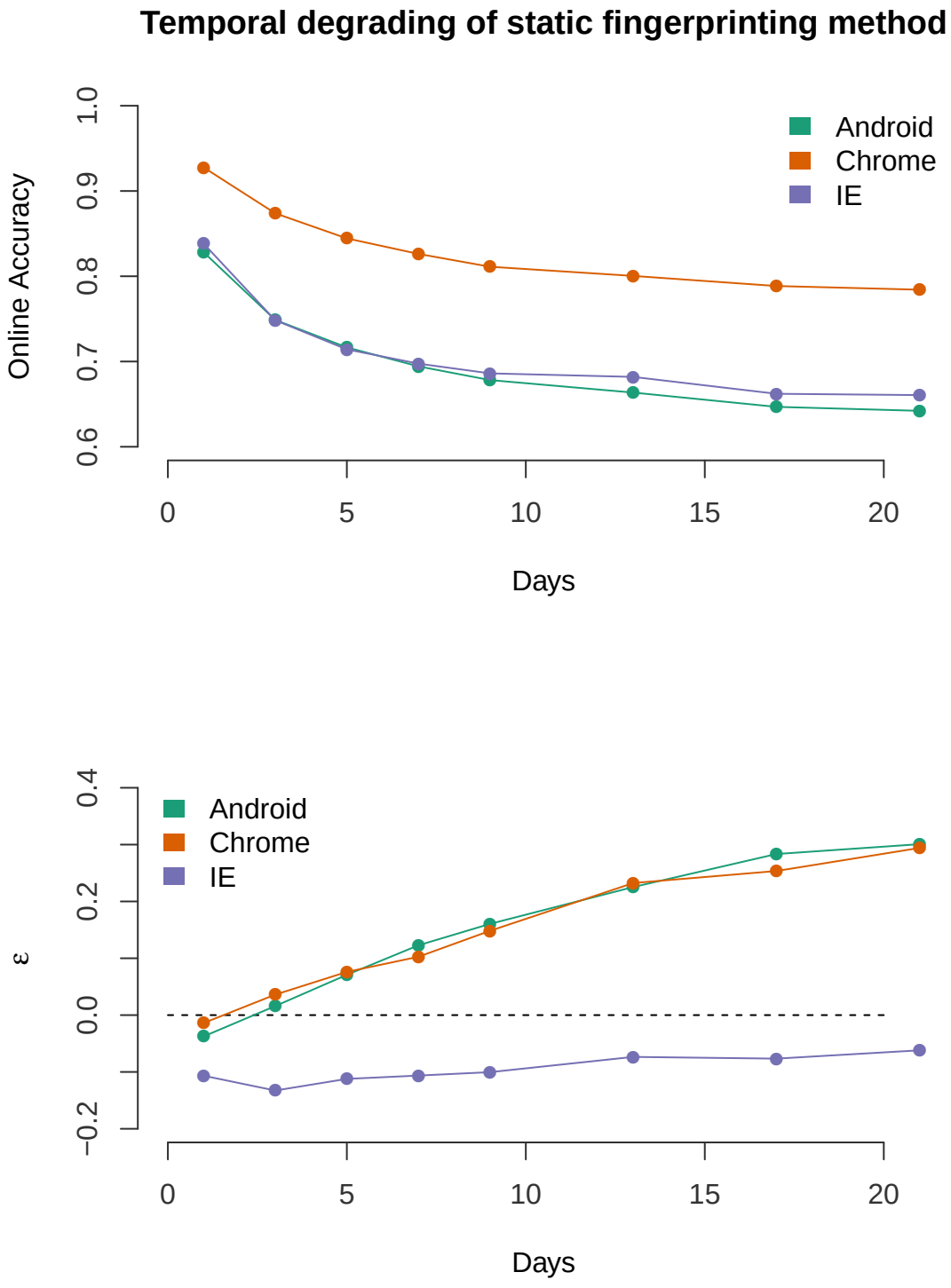

Figure 6.6: The graph shows how the online accuracy and browser instance count error degrades when classifying data over longer time periods.

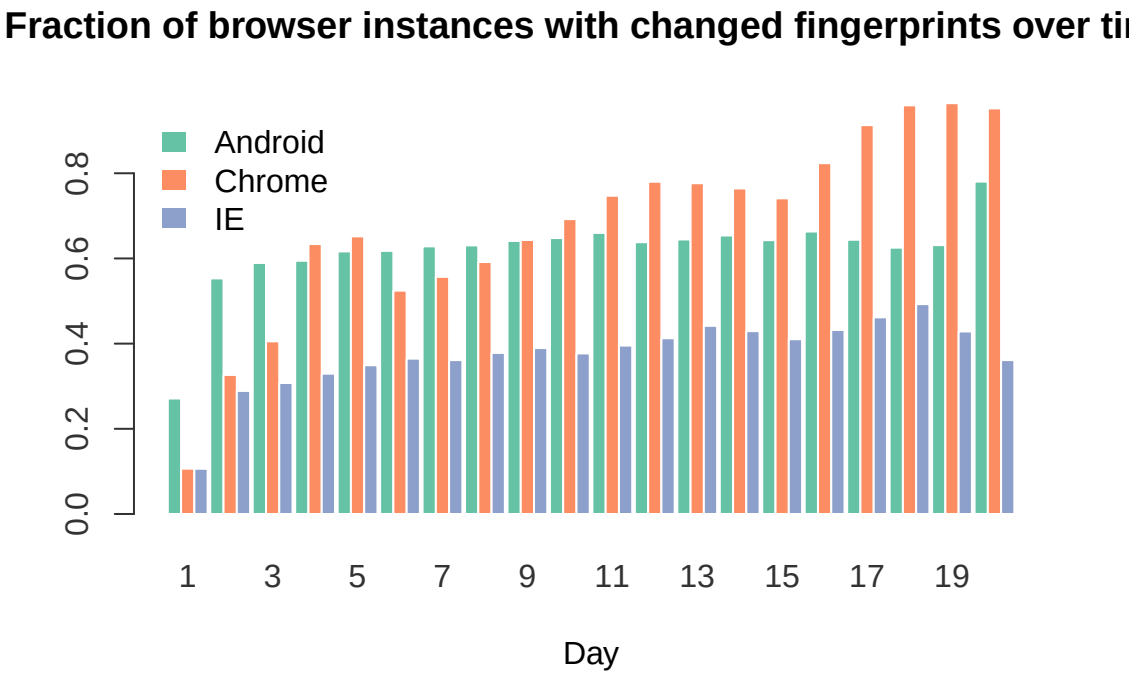

Figure 6.7: The graph shows how fingerprints changes over time.

6.2.5 Classification with Additional Rules

In this project, one rule was devised stating that "an instance seen at time t with browser version v can not have a version number $v' < v$ at a time $t' > t$". There are obviously exceptions to this rule; Internet Explorer for example facilitates the method of spoofing User Agent, which can result in that the actual browser version differs from the reported.

The rule was added when Naïve Bayes was used as an internal classifier, and a 50-50 percentage split was used for all tests. The internal classifier did not suggest a browser instance which violated the above rule in many cases, but when it did, it was incorrect in most cases. When using a sample consisting of 50,000 Internet Explorer instances from the single-site data set, the classifier violated the rule only 32 times, out of which 26 instances actually had never been seen before.

6.2.6 kNN Results

When evaluating the kNN classifier, two measures was investigated; the fraction of correct classifications and the fraction of classification in which the true label was among the k nearest neighbours, regardless of which of these was chosen, and these are denoted *correct* and *among knn* in Figure 6.8. The data used for training and testing was selected randomly, and there was an average of just over 300 instances who had labels observed during the training phase.

Figure 6.8 shows that the highest correctness was achieved for 3NN, at 0.83, and as more neighbours was considered the correctness decreased. The fraction of classifications in which the true label was among the k nearest neighbours increased significantly from 1NN to 3NN, but did not show any considerable changes when adding more neighbours.

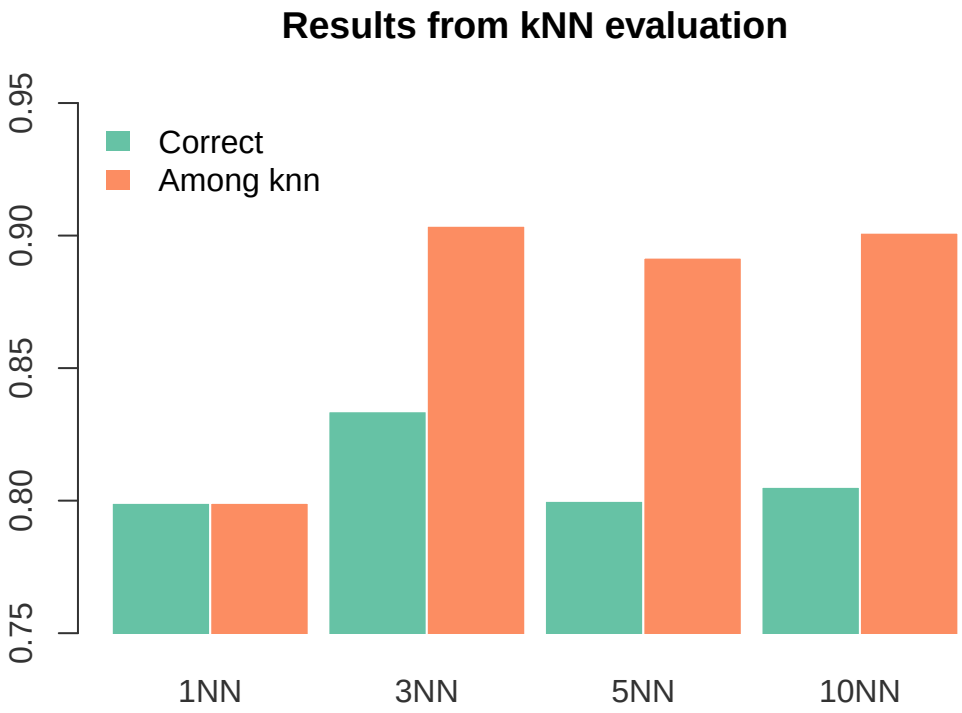

Figure 6.8: The results from evaluation of kNN on an average of just over 300 recurring instances in the research data

Chapter 7

Is Browser Fingerprinting a Viable Alternative to Conventional Methods?

The extensive analysis of Internet data does not only bring clues on how different fingerprint feature compositions affect the results of the classification algorithms, but also how some information differs between different data sources and different devices. We believe that this type of analysis is essential both for improving classification algorithms, but also for better understanding the algorithmic behaviours when partitioning the data or when the data is retrieved from different sources. The results in terms of these aspects are discussed in this chapter.

7.1 Properties and Behaviour of Features

From chapter 5 it is evident that many of the considered features display a complex behaviour; feature value changes are more or less frequent not only depending on the feature, but also on which browser, OS and device type it originates from.

In this section, we will discuss these seemingly complex behaviour and how knowledge about them can be used when designing a classifier.

7.1.1 Different Source – Different Behaviour

One of the main conclusions from the data analysis is that it seems to be profitable to partition the data prior to classification, to enable the use of different classifiers for different types of data. This conclusion originates from the apparent differences in behaviour for features displayed for data with different browser, OS and device type. Examples of this can be found in Figure 5.1, which shows how the IP change

rate differs for computers and handheld devices; in Figure 5.4, which shows how the browser version number change rate differs for different browsers; in Figure 5.6, which shows how the information gain for a set of features differs for different browsers.

If separate classifiers are used for separate parts of a partition, these can be tailored to fit the behaviour of features within each part of the partition. For instance, if the data is partitioned based on device type; the classifier handling instances from handheld devices must rely less on the IP than the classifier handling computers, since the IP changes much more frequently for handheld devices. If the data is instead partitioned based on browser, the results in Figure 5.6 suggests that the browser plugin feature is much less informative on Internet Explorer[1] than on the other browsers.

For the partitions we have used for our analysis, it is impossible for a browser instance to appear in more than one part, by the definition of a browser instance, why classification can be done in parallel without the need to synchronise classifiers. Since this holds for all partitions, it is also true for combinations thereof, which means that data can be partitioned on browser, OS and device type at the same time.

7.1.2 Understanding the Features

This section aims to provide insight into the properties and behaviours of the most interesting features, as concluded from the data analysis.

Your IP Addresses Can Identify You

The IP address seems to be a very useful feature for classification, even though it displays a higher rate of change than many other features. In the research data an IP address was associated with an expected number of 1.19 browser instances, which implies that given an arbitrary IP the correct browser instance could be identified in just over 0.84 cases.

However, due to the change rate, it is dangerous to rely too heavily on the IP address, which could lead to false negative answers when failing to identify that a previously seen browser instance has changed its IP. It could be informative to perform a more extensive study on how the IP changes, in order for a classifier to valuate how probable different types of changes are. Our analysis of changes in the most significant octets of the IP addresses did not lead to any conclusions, but similar techniques might be applicable to identify common behaviours.

[1]The browser plugins feature is collected differently on Internet Explorer, as described in subsection 3.1.2, which is probably the cause for it being less informative for this browser.

The oscillating component in Figure 5.1 indicates that Internet users who browses at regular 24h intervals are less prone to change IP than those use the Internet irregularly or on other intervals. One potential reason for this could be that those who access the Internet on these regular intervals, usually access the Internet from the same location, and probably also through the same network, may it be at work or at home. The IP is much less likely to change when using the same network, than when connecting your device to a new network.

Browser Plugins Are Troublesome

Extracting specific information about a plugin can be difficult, since different plugins tend to display information about themselves differently. For instance, the version number is not always found in the `version` field of the plugin, but sometimes in the `name` or `description`. In addition to not knowing where to find the version number, comes not knowing on which form the version number is stored. For Internet Explorer, the problem is even worse; the list of browser plugins is hidden and plugins can only be retrieved by querying one-by-one, which implies that all possible Internet Explorer plugins has to be known beforehand to get a complete list.

Despite its shortcomings, we believe that if some progress should be made in the browser plugin retrieval issues, the browser plugin list could become an even more important piece of information when trying to identify users, than it already is.

System Fonts Reveals Who You Are

The list of system fonts has been retrieved using two methods (see subsection 3.1.2), and the results from these differ greatly; the font list retrieved via CSS exploits is a bit less unique than the list retrieved via Adobe Flash, and the rate of change is much higher. The reason for this could be a flawed implementation of the font querying algorithm, which leads to non-deterministic results.

The sole reason for using the CSS exploits algorithm would be to get the system fonts on devices without Adobe Flash (such as iPhone and iPad). The method has proven to be both unstable and slow, but could, with some modifications and a shorter and carefully constructed list of fonts to test, provide an important feature for certain devices.

The list of system fonts retrieved using Adobe Flash is shared between an expected number of 1.47 browser instances, which is 0.21 more than for IP. However, less than

0.10 of all browser instances changed their font list within a week, whereas between 0.12 to 0.34[2] of all browser instances changed their IP within the same time period.

The User-Agent String Is More Than a Feature

Between 0.1 to 0.4[3] of all browser instances displayed changes in the User-Agent string within one week, making it a rather volatile feature. It is also about three times less unique as the IP, according to Table 5.1. Despite this, the User-Agent string holds important information; its content has been used to create the partitions described in section 4.3.

When classifying data from a certain part of a partition, a lot of the information in the User-Agent string becomes redundant; if the part consists of data from Google Chrome browsers on Microsoft Windows, for instance, all UAs within the part must contain the substrings `Chrome/` and `Windows NT`. There are however some pieces of information within the UA that can vary for data in the same part of a partition, such as browser version number, processor architecture and, for Internet Explorer, plugin information.

In Figure 5.4, the change rate of the entire UA is compared to that of the browser version number. It is interesting to note that these are almost the same, indicating that a change in any part of the UA also implies a changed browser version number. Internet Explorer browser instances show a slight difference in these change rates, which may be the result of Internet Explorer's UA including information about certain browser plugins, specifically Microsoft developed plugins such as Common Language Runtimes.

Screen Height Beats Screen Width

The screen resolution is made up of two components; the screen height and the screen width, both measured in pixels. Other than it seems to be a decent feature for using in classification, two interesting phenomena has been identified: the screen height is much more informative than screen width and handheld devices are much more prone to changing their screen resolution than computers.

Figure 5.5 indicate that the screen width is a less informative feature than screen height. In the research data, the information gain from the screen height is almost two bits of entropy higher than that of the screen width. Little effort was put into

[2]The IP change rate is strongly dependent on the device type.
[3]The UA change rate is highly dependent on the browser.

investigating what caused this seemingly odd behaviour, and the screen width was simply discarded in the feature selection.

The fraction of browser instances who has changed their screen resolution in one week is about four times higher for handheld devices than for computers, which can be seen in Figure 5.2. After one week, over 0.20 of all handheld devices have changed their screen resolution, while just under 0.05 of all computers have changed theirs. We assumed that this was because of the ability to rotate the screen on handheld devices, which changes the resolution, why an experiment was conducted, in which the total number of pixels was considered. Surprisingly, the results from this experiment was nearly identical to those of the original experiment, implying that the height and width of the screen is not simply swapped when the screen is rotated.

HTTP Accept Headers Can Actually Be Useful

In this thesis, two HTTP Accept headers have been subject to investigation; Accept-Language and Accept-Charset. These have the same structure, and contain the languages and character sets, respectively, a browser will accept. With respect to information gain, Accept-Language is much more informative than Accept-Charset, as depicted in Figure 5.5. This was to be expected since there are significantly more languages than character sets in the world. What is surprising is how informative Accept-Language is, considering that a vast majority of the browser instances in every data set originates from Sweden.

The change rate of both features are low compared to the other features, but it is interesting to note the difference in change rate in the two features for different browsers. Although being stable features, Mozilla Firefox is much more likely to change its accept headers than the other browsers.

7.2 Data Collection

Collecting relevant features is crucial to maximising the performance in the classification process. However, in a real-time environment there are often resource and time constraints which limit the ability to collect some features even if they may add relevant information to the fingerprint. For example, collecting fonts using the CSS exploit is extremely time-consuming, and might hence not be feasible outside of a research environment.

7.2.1 take-a-bite.org

The research website `take-a-bite.org` had few or none constraints and thus enabled us to explore methods for collecting a wide range of features. There are most likely features not considered in this thesis that can be collected from web browsers and used in a classifier, and we believe that it would be beneficial to collect those from a dedicated research environment, to enable the exploration of features which would be deemed infeasible to collect in a commercial environment.

The ability to collect data from a browser is limited by the resources offered by that specific browser. These resources may vary from browser to browser. For example, handhelds such as Apple iPhone and iPad did not support Adobe Flash, at the time of this thesis, meaning that all data collected using Adobe Flash cannot be collected from such devices.

Another example is, that even though all major browsers support JavaScript, there are small deviations from the de facto standard; in particular, the JavaScript engine in Internet Explorer deviates in some aspects from the other major browsers.

Since a collection script must be customised to work on all browsers, these resource differences might put limitations on which types of features can be collected from some devices.

One difficulty with collecting data from consenting users was that it was hard to reach out to a broad public. In Eckersley (2010), the authors goal was to illustrate the privacy concerns surrounding browser uniqueness. This thesis had a diametrically different approach; our goal was to to refine the methods for tracking individuals using browser fingerprints. We were always transparent with our motives, which might have lead to the limited interest in our research website.

We believe that the branding with "Take a bite of the browser cookie" was an important stand from our point of view which illustrated the essence of this project; to replace cookie-based tracking with better, more reliable and less intrusive.

7.2.2 FP.js

The stand-alone collection script `FP.js` was a proof-of-concept of how an actual collection environment could be constructed. It was a successful project in developing a functional collection script, parsing script, compression algorithm and server software. We believe that scripts of this type, in which a subset of the most informative research features is used, can be used in a staging environment between research and

production environment. If more interesting features would have been found in the research environment, they could easily be implemented in this script.

The vast difference between this script and the research website was that this script collected data automatically without affecting the client device. We believe that this is similar to the methods used for collecting data on a real-world website.

The compression algorithm was an essential part of the script, it did increase the size of the script by several hundred percent but it reduced the number of requests needed to submit the fingerprint. The number of requests sent from the client's device was a crucial bottleneck because of the overhead involved in making a request, why this increase in size was well motivated. Decreasing the number of request can either be done by using a compression algorithm like the one we propose or by discarding some information from the fingerprint, by for instance only counting the number of fonts installed on the client's device rather than collecting the complete list.

7.3 Telling Internet Users Apart

The main goal for this thesis has been to develop a method for telling Internet users apart, and we will throughout this section address the most interesting results and findings made in this context. This section will describe why we believe that machine learning is a viable approach to the problem, when and how static fingerprinting is applicable, the benefits from partitioning data, why handheld devices are more troublesome, and a brief note on the ethical issues regarding Internet user tracking.

7.3.1 Machine Learning as a Solution

As concluded through our data analysis; the examined features are highly dynamic. Changes occur continuously and the dependencies between features are often complex. Without the ability to store information on the clients' devices, identifying them will require an understanding of how and when such features are changed. The complexity of the dynamics of this problem, along with the constant need to adapt to environmental changes, have lead us to the conclusion that machine learning is a favourable approach.

Our experiments with different machine learning classifiers have shown promising results, even though there is still plenty of room for improvement. Consider the Naïve Bayes classifier for instance; despite that it only compares feature values on equality, rather than by assuming that similar feature values are more likely, and that it assumes complete independence between features, the results from the experiments

in which it was used as the internal classifier suggests that it was able to learn and adapt to the dynamic input, especially when the parameters were chosen for a specific data set. This was also seen in the experiments with the k Nearest Neighbours classifier, which, despite not considering the IP address, was able to correctly identify almost 0.85 of all recurring browser instances observed over a time period of nearly three months.

There might very well be other feasible approaches to this problem, but we strongly believe that machine learning is a promising candidate.

7.3.2 When is Static Fingerprinting Good Enough

The static fingerprinting method described in subsection 6.1.2 has two main selling features: it is extremely fast and is not *that* bad. Classification consists of a single look-up in a hash or dictionary, which can be done in $\mathcal{O}(1)$ time, and adding instances is consequently a single insertion in the same hash, which can also be done in $\mathcal{O}(1)$ time. The performance of the classifier ranges from good to bad, mainly depending on the length of the time period for classification and the purpose for classification.

As can be seen in Figure 6.6, the online accuracy for the static fingerprinting method is well over 0.9 for one data set when run on data from a single day. This is more than enough for many purposes; consider for instance that you run a targeted online advertisement campaign, in which different advertisements are shown to different users. Being able to accurately target 0.9 of the advertisements means that only one in ten users will be shown an advertisement they are not thought to be interested in, which is probably more than acceptable, especially if comparing to the accuracy of offline advertisements in news papers and on TV.

If, on the other hand, you own a website which offers some service to its users and are interested in the how many times an average visitor visits your site before signing up for your service, an accuracy of 0.9 might not be good enough. Since every tenth guess is incorrect, the static fingerprinting method will only be able to track a small fraction of visitors over more than ten visits, which in turn implies that if the true average number of visits before signing up is higher than ten, the analysis will probably fail to identify this.

Thus is the ability to use the static fingerprinting method highly dependent on the purpose; for aggregated or less sensitive calculations the method might be an adequate choice, but if the ability to follow individual users over time or if other precise abilities are required, more advanced methods should be to prefer.

In the special case of counting the number of unique browser instances, as shown in the lowermost graph in Figure 6.6, the error is negative for shorter time periods and grows with the duration of the time period. This is probably because of browser instances behind proxies, sharing IP and all other features, which will all be believed to be the same browser instance. As time passes, more and more browser instances will change some feature value and consequently be regarded as previously unseen (false negative), leading to the total error increasing past zero. By choosing an adequate set of features, the expected break-even point, that is when the error is zero, can be moved back or forth in time. By choosing less unique features, the break-even time will occur later than if more unique feature were chosen.

7.3.3 Why Partitioning Data is a Good Idea

As mentioned in subsection 7.1.1, we strongly believe that partitioning data is very beneficial. Almost every result from the different parts of the analysed partitions indicates more or less important differences. Consider for instance the parameter selection for the Naïve Bayes classifier, as described in subsection 6.2.3, and how the results clearly indicate that the classifier would benefit from using different parameters for the different data sets.

Our implementation of Naïve Bayes only considers equality when comparing feature values, and the only parameters that are adjustable are the threshold, δ, and the smoothing constant, q. Other classifiers may have more room for tailoring each classifier to the dynamics of their corresponding data set. When experimenting with the kNN algorithm, this became very clear; choosing different distance metrics could give very different results. Consider the browser version number, and how fast it changes depending on the browser. For Mozilla Firefox and Google Chrome, almost every second browser instance changes its version number within the first week, whereas just over a tenth of the Safari and Internet Explorer browser instances changes. Using the same distance metric for these browsers would be far from ideal, since many more browser instances went from Google Chrome version 16 to 17 than from Internet Explorer version 8 to 9, during the collection of the research data.

Classifications of instances in different parts of a partitioning, as proposed by us, can be done in parallel, since no browser instance can belong to more than one partition, according to the definitions of the partitioning and a browser instance. Being able to run classifications in parallel would speed up the classification process, however not by more than by the number of parts in the partition.

We found no way of automating the process of deriving rules for the partitioning of the data, and selected the largest subsets of data by hand. In a real-world scenario, this would probably be enough, even though the set of partitioning rules would have to be maintained as new browsers and OSs are released. The reason for not being able to automate the process was the vast variety of more or less obscure web browsers and OSs along with the lack of a standard format for the User-Agent string.

The definition of a browser instance states that it is a specific installation of a specific web browser on a specific device, why partitioning on any other property than device type, OS or browser may cause browser instances occur in mote than one part, which makes parallelising the classification process difficult.

7.3.4 Problems with Classifying Handheld Devices

Many results in this thesis indicates that handheld devices are harder to classify than computers, see for instance Table 6.2. We believe that this is the result of a combination of several reasons:

- Handheld devices are usually more restricted in what software is installable, which makes them more uniform than computers.
- The IP of handheld devices are changing much more frequently than that of computers, due to the mobility of the devices.
- The screen resolution is also constantly changing, as the device is rotated.
- A large fraction of handheld devices (all iPhones and iPads) lack the support for Adobe Flash, which makes font retrieval much more difficult and imprecise.
- Many handheld devices retrieves updates automatically, which implies that most devices are updated more or less simultaneously.

7.3.5 Ethics Concerning Internet User Tracking

Eckersley (2010) describes the ethic aspects surrounding the collection of information from browsers. We believe that the privacy concerns are important when collecting information about a browser, and even more so when tracking that specific instance over time. However, there are several useful application for browser fingerprinting in which it is required to collect these kind of data, and it is important to realise that not all such application have malicious intents.

One example of an application in which such data could be collected is online fraud prevention; by tracking its users by using their browser fingerprint, a company may be able to detect and prevent fraudulent behaviour when malicious user tries to access their client's services.

Website analytics is another good example; most website owners are interested in what kind of material interests its users and what does not. Observing user behaviours and track users across the website over time may provide useful information when creating an attractive design and for customising the user experience.

Despite having many benevolent applications, tracking Internet users by collecting data from them is a delicate question with obvious privacy concerns. Data collection must never be done without complete transparency towards the user; the user must be aware of being tracked and able to review and delete the collected data.

7.4 Future Research

This thesis has only just begun to scratch the surface of this very complex and intriguing problem. We believe that the solution lies within the domain of machine learning, and that the analysis and results in this thesis give a good insight into the problem and a basic understanding of its challenges and pitfalls.

In this section, we will give recommendations regarding which issues to address when attacking this problem and which phenomena to investigate first.

The additional rule described in subsection 6.1.4 did imply any significant improvements, but we believe that trying to identify similar properties could ease the learning algorithm's learning process. Performing a more thorough investigation on the browser version number to derive a more clever rule could be one potential approach.

For this thesis, the true labels were set using either HTTP cookies or Adobe Flash LSO, which may be a less-than-perfect method due to cookie deletion. Designing a more stable solution for verification is necessary to be able to compare a fingerprinting algorithm against these conventional methods. Any such method could probably not be used in a commercial context, due to privacy issues, but should be possible to apply in an academic context.

The set of features we have analysed within the scope of this thesis only constitute a small fraction of all information that can be collected from a browser instance, and a more in-depth investigation of such information could potentially lead to finding more stable and, at the same time, unique features.

Even though we only conducted a very small experiment with the kNN classifier, it appears to be a promising approach. We draw this conclusion based on the fact that, despite not using the IP, the kNN classifier produced results similar to those of the other classifiers on the research data set. If this method should be investigated further, two key issues need be addressed; firstly, it is not desirable to compute the distances to every other instance ever seen for every classification, why some fast method for filtering out some set of potential candidates among which the k nearest neighbours reside, and secondly, much effort has to be put into the choice of distance metrics, which have to be both fast and informative.

Chapter 8

Conclusions

This chapter tries to illustrate and conclude our most important experiences and results. Our analysis indicates that machine learning algorithms is a promising solution to the problem of how to tell users apart by their browser fingerprint. In some cases, more straight-forward algorithms may be a better choice in terms of computational speed versus algorithmic accuracy. Thorough analysis and processing of the data is a significant part of constructing good algorithms and collecting relevant features.

8.1 Use Machine Learning

The problem dissected in this thesis is both dynamic and complex; fingerprints changes constantly and there are endless possible combinations of feature values. Manually designing an static algorithm sophisticated enough to solve the problem seems close to impossible, why we have concluded that machine learning is to prefer. The algorithm must be able to adapt and respond to the constant changes in the system, and do so continuously. Thus, we strongly recommend that further research of this problem should involve machine learning.

8.2 Static Fingerprinting is Good Enough

For generating aggregated user statistics or for following users over shorter sessions, a static fingerprinting algorithm seems to be the best choice. A such algorithm could be implemented to run in constant time with linear memory usage using a hash function. For shorter time periods, changes in fingerprints are insignificant why more sophisticated methods might be unnecessary. In less sensitive applications, a static fingerprinting method might be a suitable solution.

However, if the ability to follow individual users over time is required, more advanced methods must be used and the results from our experiments indicate that more advanced machine learning techniques are a favourable approach. Since feature values changes over time, a sustainable solution must be dynamic and able to adapt to these changes.

8.3 Most Usable Features

Choosing adequate features is crucial to any classification problem, and much work has been put into it during this thesis. Despite the lack of data, a set of features have been distinguished to be suitable for using in a browser fingerprinting algorithm; IP address, system fonts, browser plugins, User-Agent string, screen height and Accept-Language. In neither of our experiments did a fingerprint containing all of these produce the best results, but combinations thereof did. Depending on the data source, different combinations of these gave the best results, why collecting all of them should be sufficient in most applications. This selection of features is also supported by Eckersley (2010).

8.4 Partition the Data

Major differences in feature properties, such as change rate and uniqueness, have been seen for different web browsers, OSs and device types. Partitioning the data enables customising the classifier for each of the parts in the partition, which should lead to an improved over-all performance. Such differences have also been seen for data from sites of different genres, but partitioning based on site genre does not ensure that one label is not present in more than one part. Moreover, some simple static rules, used as sanity checks for the classifier, have been identified to further increase this performance.

8.5 Concluding Remarks

As the goal for the thesis was to provide a proper analysis of the problem and investigate whether a machine learning approach could be sensible, we believe that the thesis was successful. The thesis gives a strong foundation for future research and should be useful when designing a classifier and the collection process.

References

P. Cunningham & S. J. Delany (2007). k-Nearest Neighbour Classifiers. *Multiple Classifier Systems* pp. 1–17.

Directive (2009/136/EC). Directive 2009/136/EC of the European Parliament and of the Council of 25 November 2009 amending Directive 2002/22/EC on universal service and users rights relating to electronic communications networks and services, Directive 2002/58/EC concerning the processing of personal data and the protection of privacy in the electronic communications sector and Regulation (EC) No 2006/2004 on cooperation between national authorities responsible for the enforcement of consumer protection laws.

R. O. Duda, et al. (2001). *Pattern Classification*. Wiley, 2 edn.

P. Eckersley (2010). How Unique Is Your Web Browser? In M. Atallah & N. Hopper (eds.), *Privacy Enhancing Technologies*, vol. 6205 of *Lecture Notes in Computer Science*, pp. 1–18. Springer Berlin / Heidelberg.

Internet Advertising Bureau (2012). IAB Internet Advertising Revenue Report: 2010 Full Year Results.

T. M. Mitchell (1997). *Machine Learning*. McGraw-Hill Science/Engineering/Math, 2 edn.

M. D. Smucker & J. Allan (2005). An investigation of dirichlet prior smoothings performance advantage. Tech. rep., Department of Computer Science, University of Massachusetts Amherst.

O. Tene & J. Polonetsky (2012). To Track or 'Do Not Track': Advancing Transparency and Individual Control in Online Behavioral Advertising. *Minnesota Journal of Law, Science & Technology* **13**.

Appendix A

Complete list of features collected via `take-a-bite.org`

IP (string)

UA From the UA, the following parameters were extracted:

- `agent_type` (string)
- `agent_name` (string)
- `agent_version` (string)
- `os_type` (string)
- `os_name` (string)
- `os_versionName` (string)
- `os_versionNumber` (string)
- `os_producer` (string)
- `os_producerURL` (string)
- `linux_distibution` (string)
- `agent_language` (string)
- `agent_languageTag` (string)
- `description` (string)

`navigator` From the `navigator` object in the DOM tree, the following parameters were extracted:

- `appCodeName` (string)

- `appName` (string)
- `appVersion` (string)
- `cookieEnabled` (boolean)
- `language` (string)
- `plugins` (array)
- `platform` (string)
- `product` (string)
- `productSub` (string)
- `userAgent` (string)
- `vendor` (string)
- `vendorSub` (string)

`screen` From the `screen` object in the DOM tree, the following parameters were extracted:

- `width` (number)
- `height` (number)
- `availWidth` (number)
- `availHeight` (number)
- `availLeft` (number)
- `availTop` (number)
- `colorDepth` (number)
- `pixelDepth` (number)

`window` From the `window` object in the DOM tree, the following parameters were extracted:

- `innerWidth` (number)
- `innerHeight` (number)
- `outerWidth` (number)
- `outerHeight` (number)

Timezone (number)

Clock diff The difference between the server's and client's internal clocks were estimated through a series of back-and-forth requests to determine upper and lower bounds.

RTTs The RTT between client and server were determined through a series of back-and-forth requests.

Fonts The fonts were collected using both Adobe Flash and a CSS exploit. If present on the client's device, Adobe Flash returns a complete list of fonts whereas the CSS method tests for the existance of known fonts.

HTTP-Accept From the HTTP-Accept headers, the following parameters were collected:

- Accept-Language (array)
- Accept-Charset (array)
- Accept-Encoding (array)
- Accept (array)

Timestamp (number)

Appendix B

Compelete list of features collected via `FP.js`

IP (string)

UA (string)

`navigator` From the `navigator` object in the DOM tree, the following parameter were extracted:

- `plugins` (array)

`screen` From the `screen` object in the DOM tree, the following parameter were extracted:

- `availHeight` (number)

`window` From the `window` object in the DOM tree, the following parameters were extracted:

- `innerWidth` (number)
- `innerHeight` (number)
- `outerWidth` (number)
- `outerHeight` (number)

Fonts The fonts were collected using Adobe Flash. If present on the client's device, Adobe Flash returns a complete list of fonts.

HTTP-Accept From the HTTP-Accept headers, the following parameters were collected:

- Accept-Language (array)
- Accept-Charset (array)

Timestamp (number)