On the Implications of Program Analysis Abstractions to Concept Drift in Malware Classification

University College London
MEng Computer Science
Submission Date: 12th May 2022
Supervisor: Lorenzo Cavallaro

Mark Anson

Abstract

To many, Android malware classification is considered a solved problem. Modern classification approaches can now produce results at near-perfect accuracy, precision, and recall. However, these malware classifiers are still subject to the problem of concept drift, in particular the decline of the quality of a trained classifier over time. Many approaches, such as statistical assessment and incremental retraining, have been put forward to solve this. Instead of proposing a solution to detect or fix concept drift once it arises, this paper instead focuses on the classification pipeline itself, the process of converting an initial set of APKs to a working classifier. Through experimentation, we can draw insights into how varying different elements of the classifier, including the feature space, representation, classification method, and dataset size, can influence the resulting concept drift.

By creating four different experiments on three different feature sets using the TESSERACT framework, we can vary the aforementioned aspects of the pipeline to produce a set of results demonstrating the effect in terms of concept drift over time. Our results demonstrate that these variations to the classification pipeline can produce interesting effects on the resulting concept drift. Firstly we find that for features in a set, the feature that reduced concept drift the most is most likely the one with the largest number of elements rather than any other variable. Secondly, we find classification approaches do not necessarily behave as expected under concept drift, with more complex classification methods often performing worse than more simple ones. Thirdly, we observe that generally, vectorization approaches have a much less significant effect than classification approaches. Finally, we see that a smaller feature set size is heavily correlated with a much more variable output.

Contents

Chapter 1

Introduction

As of 2021, there were 3 billion active Android devices [1] in use around the world, giving Android a very dominant stake in the world of mobile operating systems. Unfortunately due to the huge popularity of the Android operating system, and its relative vulnerability in comparison to IOS [2], the platform is very often a target for malicious actors [62]. As such, high-quality Android malware detection is vital.

There is a huge amount of research has been carried out over the years into the issue of Android malware classification, that is deciding whether an application is most likely malware or not. Huge numbers of classifiers exist, such as *Drebin* [3],*MaMaDroid* [4], *Blade* [5], and DroidSIFT [6] among many others, many with high-performance figures to the point where further work to increase the raw figures could be considered to be of diminishing returns.

Instead of focusing on raw performance figures, this project furthers the work of Pendlebury et al [7]. As written in the paper, "Malware classifiers operate in dynamic contexts. As malware evolves and new variants and families appear over time, prediction quality decays. Therefore, temporal consistency matters for evaluating the effectiveness of a classifier." This paper focuses on this temporal inconsistency, otherwise called "concept drift", "time decay" or simply "drift". A machine learning classifier can only learn to solve for the data it has available in its dataset. Even a huge dataset is unlikely to produce a model that can accurately predict malware into the future, as methods and approaches slowly change.

The TESSERACT paper demonstrates various effects and mitigation strategies of drift throughout their report on three classification algorithms, Drebin, MaMaDroid, and a Deep Learning example. To extend this work, this report uses Drebin, MaMaDroid, and BLADE to draw novel comparisons between classification methods in the context of concept drift. Understanding the relationship between different analysis methods, their accuracy and precision through the use of F-measures, and complexity, while accounting for the effect of concept drift. This is vital in gaining a better understanding of the problem so that future researchers can be better able to pick classification approaches that are as resistant to concept drift as possible while still accounting for other important factors important in their specific context.

1.1 Approach

This project was carried out with an iterative approach. The first iteration involved the exploration and development of a pipeline that would take in relevant feature spaces (sets of features from Android APKs), represent them in vector space, perform learning-based detection to build a malware classifier, and then run within the TESSERACT testing environment to perform time-sensitive analysis on the classifiers we create. From then on iterations added new features and explorations to this pipeline, allowing me to produce more data to analyse. The final runs of the four experiments seen in this experiment were conducted towards the end of the project, with the data from those experiments being the basis for our conclusions in this paper.

1.1.1 Aims and Objectives

1.1.2 Aims

In this paper, we aim to explore different program analysis techniques, representations and other variables across the Android malware classification pipeline to determine the relative advantages and disadvantages of variations in approach, with a particular focus on concept drift. By building pipelines with different approaches in program analysis, data representations, and classification models, we can compare how different techniques affect key variables such as accuracy, cost (in terms of time/resource requirements), performance decay over time (concept drift), and understandability, among others. Using this understanding we can make suggestions as to the best approach to take depending on the use case and contrast with current existing classification approaches in the literature.

To achieve this aim, we intend to answer four distinct research questions, each focusing on a different aspect of Android malware classification and its effect on concept drift.

1. RQ1 - Are metadata-based features, such as permissions, more resilient to concept drift than those assembled from the bytecode?
2. RQ2 - What is the effect of machine learning classification approach on concept drift in the context of different program analysis techniques?
3. RQ3 - What is the effect of feature representation on concept drift in the context of different program analysis techniques?
4. RQ4 - What is the effect of dataset size on concept drift in the context of different program analysis techniques?

1.1.3 Objectives

1. Review current literature surrounding both Android malware classification and program analysis as a whole with the aim of finding current approaches
2. Review feature spaces and program analysis strategy that could produce interesting results, along with some potential candidates for machine learning techniques, and decide on what to pursue within the project.
3. Develop a pipeline using the TESSERACT [7] codebase that allows variation in approach for variables such as feature set and program analysis strategy, machine learning techniques, and data representation.

4. Evaluate the effectiveness of these techniques, against both themselves and existing techniques in the literature. In particular, we are looking at performance decay alongside accuracy, precision, and complexity.

Chapter 2

Background and Literature Review

2.1 The Classification Pipeline

The available research for Android malware is extensive, with a seemingly endless number of approaches and optimisations attempting to solve the issue of Android malware classification. In many of the most prominent classifiers, such as Drebin [3] and MaMaDroid [4], the general pipeline from data to working classifier is usually the same, as detailed in figure 2.1.

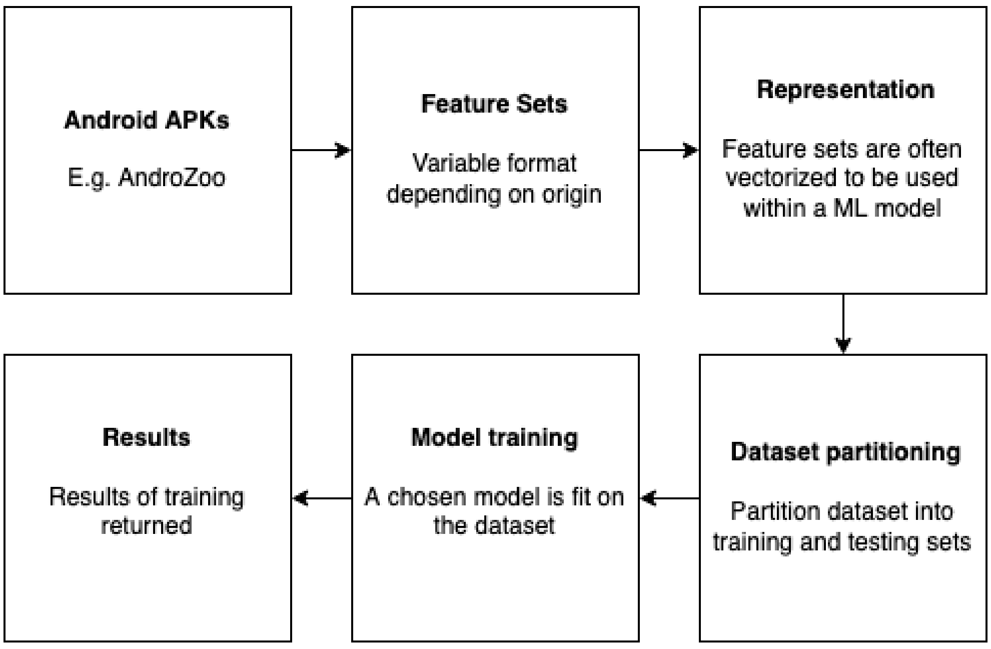

Android APKs
E.g. AndroZoo
Feature Sets
Variable format depending on origin
Representation
Feature sets are often vectorized to be used within a ML model
Dataset partitioning
Partition dataset into training and testing sets
Model training
A chosen model is fit on the dataset
Results
Results of training returned

Figure 2.1: Graphical depiction of the pipeline

2.1.1 Detailed Explanation of Pipeline Elements

Android Application Packages (APKs) The standard format of Android applications is found in the Google play store as well as other third-party stores. For Android malware classification, several large datasets, most notably AndroZoo [8], have hundreds of thousands to millions of individual APK files, along with appropriate metadata. For this paper, the most important pieces of metadata associated with each APK is the timestamp for when the APK was created and a malware/benign classification, usually denoted with malware as the positive case and benign and the negative. In the case of AndroZoo, each APK is analysed by multiple different antivirus products to provide a more accurate dataset.

Feature Sets Also called datasets throughout this paper, these are sets of extracted features used to train the machine learning classifier. Such feature sets can vary quite significantly in layout. For example, the Drebin feature set consists solely of a set of extracted features, with an APK either containing or not containing a feature, whereas BLADE datasets consist of a set of extracted opcodes with an associated frequency.

Representation The raw feature set must be represented in a way that can be used by a machine learning model. This representation is often achieved with feature vectors [9].

Dataset partitioning - The feature set is partitioned into training and testing sets that can be used to train the model. Approaches such as *train-test-split* [10] or *Stratified K-folding* [11] can be used to split the dataset into training and test data on which the machine learning model can be trained. Inside the TESSERACT testing environment, we use a *time-aware-train-test-split* to split the dataset into an initial training set of a predetermined size on the earliest features in the dataset, and a testing set of all future features. This method allows us to determine the effect of concept drift over time during testing.

Model training The chosen model, for example, a LinearSVC [45] or RandomForestClassifier [48], is then trained on the dataset splits by fitting the data to the model. Inside the TESSERACT testing environment, this is achieved with the *fit_predict_update* function, which incrementally trains the classifier on the splits and returns results for us to quantify.

Results The *fit_predict_update* function from the TESSERACT codebase returns a dataset of results, most notably recording F1 scores for each month of training.

2.2 Concept Drift

Concept drift is a form of temporal bias [7] and generally refers to the idea that the patterns and relations between data will evolve and change over time [15]. “Hidden contexts” and relationships found by a machine learning model may not remain consistent as a dataset changes [16]. This relates specifically to Android malware as the malware ecosystem is constantly changing, with new Malware families and verities being implemented by malicious actors [7]. In 2019, Pendlebury et al [7] created and reviewed a framework for analysing the effect of concept drift on Android malware classification, as well as the effect of “Spatial bias”. This framework is called TESSERACT and acts as the basis and backbone of the research carried out in this paper. The paper’s outcomes, along with what was left unexplored, heavily influence the direction of this paper.

The paper introduces both temporal and spatial bias. Temporal bias includes concept drift but the paper also looked at the detection of “past objects” (malware that is older than the dataset a classifier has trained on) and differences in the time period between goodware and malware. Spatial bias includes “assumptions on percentages of malware” in both training and testing datasets.

Of particular interest to this report are the outcomes of time decay analysis. The paper demonstrates a clear decline in the quality of the evaluation of new malware over a 24 month period for both Drebin and MaMaDroid, as seen in Figure 2.2. This analysis in its most straightforward form is described as baseline AUT analysis”.

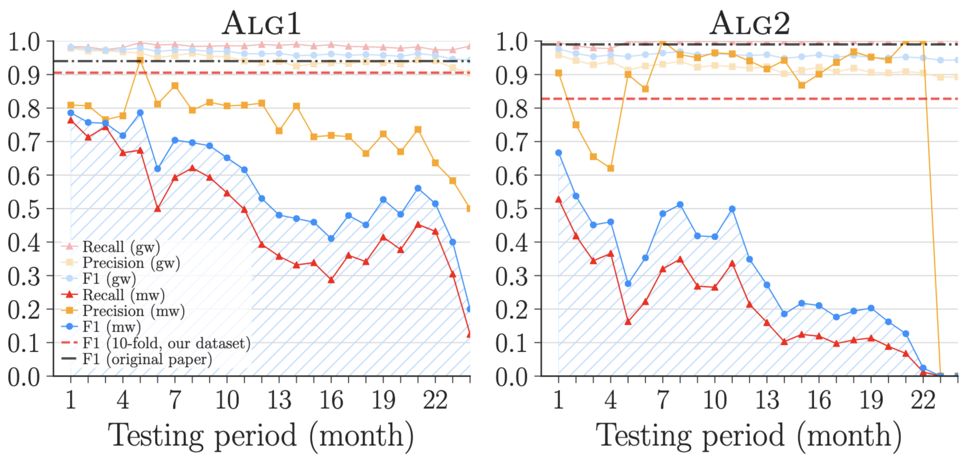


Figure 2.2: Time evaluation of Drebin (Alg1) and MaMaDroid (Alg2) [7]

TESSERACT also provides a comparison to a deep-learning based approach outlined in a 2018 paper from Suciu et al [12]. Although the details of such an approach are out of the scope of this paper, TESSERACT can demonstrate the resilience of deep learning approaches in comparison to the linear classification methods used in Drebin and MaMaDroid.

TESSERACT also spends significant resources analysing effective ways to minimise or delay time decay. Strategies such as incremental retraining, active learning, and classification with rejection are demonstrated to be effective methods at reducing the effect of time decay.

This report has a few key limitations, which have become the basis of some of the experiments detailed in this report. Firstly, this paper does not take into account the wide variety of classification algorithms available. For example, the report analyses MaMaDroid with a Random Forest classifier [26], however in the MaMaDroid report, Mariconti et al review their classifier with Random Forest, 1-Nearest Neighbor, 3-Nearest Neighbor, and Support Vector Machines. This limitation is the motivation for our RQ2.

Secondly, this report does not take into account the effect of variable dataset sizes on the accuracy and usability of time decay analysis. Mariconti et al based their analysis on a dataset of 129K samples (8.5K begin and 35.5K malware). The dataset accessible through the TESSERACT source code [13] contains over 120K samples. However, the number of available APKs used to train classifiers can change considerably depending on implementation. Using larger or smaller datasets in your training sample could impact time decay considerably. This limitation became apparent during this project when attempting to produce time decay analysis on classifiers using much smaller datasets and discovering the output produced was completely unusable and acts as justification for RQ4.

This paper also left room for further analysis of time decay on different classifiers as the report only focuses on Support Vector Machines [21], which acts as part of the justification for RQ2.

Papers such as a 2017 paper from Jordaney et al [18] have proposed mechanisms by which concept drift can be identified. The aforementioned paper details a statistical comparison approach for identifying the presence of concept drift in Windows and Android malware classification models. This approach from Jordaney et al focused on two specific dimensions, the “desired performance level” and the “proportion of samples in an epoch that [a] malware analysis team is willing to manually investigate”. This kind of approach is worth noting due to the similarity of the topic, however, is tangential to the work detailed in this paper. The TESSERACT approach, and by extension this paper, uses the decline in quality of a classifier over time in a “walled garden” testing environment, whereas Trancend sets out to quantify concept drift in deployed environments.

2.3 Android malware classifiers

This paper has a particular focus on feature extraction through static program analysis, following on from the TESSERACT [7] paper. Dynamic program analysis and feature extraction is considered out of scope for this project. Well-known and effective Android malware classification technology, such as Drebin and MaMaDroid, which underpin this paper, engages entirely in static analysis. Static analysis approaches are sufficient to produce effective and high-quality classification approaches.

As explained in the classification pipeline overview, concept drift is only a measure of the performance of a classifier. As such, it is important to understand the state of Android malware classification to understand current approaches to program analysis abstraction and representation. A particularly notable approach is *DREBIN* [3], a “lightweight method for detection of Android malware”. As previously mentioned, Drebin is used as a benchmark in several papers discussing concept drift and will be a key benchmark used in this paper as well.

Drebin uses “broad static analysis” to create a feature set of 8 different features. Several of these features come directly from app metadata, the ”Manifest”. Of particular interest to this project is *Requested Permissions* and *Used Permissions*. The difference in effectiveness between metadata features such as permissions and features extracted from the binary itself is not something that is fully explored in the paper, and certainly not something explored in the terms of concept drift.

Malware family	**Top 5 features**		
	Feature s	Feature set	Weight w_s
FakeInstaller	`sendSMS`	S_7 Suspicious API Call	1.12
	`SEND_SMS`	S_2 Requested permissions	0.84
	`android.hardware.telephony`	S_1 Hardware components	0.57
	`sendTextMessage`	S_5 Restricted API calls	0.52
	`READ_PHONE_STATE`	S_2 Requested permissions	0.50
DroidKungFu	`SIG_STR`	S_4 Filtered intents	2.02
	`system/bin/su`	S_7 Suspicious API calls	1.30
	`BATTERY_CHANGED_ACTION`	S_4 Filtered intents	1.26
	`READ_PHONE_STATE`	S_2 Requested permissions	0.54
	`getSubscriberId`	S_7 Suspicious API calls	0.49
GoldDream	`sendSMS`	S_7 Suspicious API calls	1.07
	`lebar.gicp.net`	S_8 Network addresses	0.93
	`DELETE_PACKAGES`	S_2 Requested permission	0.58
	`android.provider.Telephony.SMS_RECEIVED`	S_4 Filtered intents	0.56
	`getSubscriberId`	S_7 Suspicious API calls	0.53
GingerMaster	`USER_PRESENT`	S_4 Filtered intents	0.67
	`getSubscriberId`	S_7 Suspicious API calls	0.64
	`READ_PHONE_STATE`	S_2 Requested permissions	0.55
	`system/bin/su`	S_7 Suspicious API calls	0.44
	`HttpPost`	S_7 Suspicious API calls	0.38

Figure 2.3: Top 5 features from FakeInstaller, DroidKungFu, Goldmaster, and GingerMaster families [3]

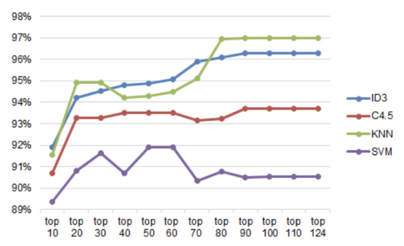
Feature sets	**Malware families**																			
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
S_1 Hardware components	✓			✓				✓					✓							
S_2 Requested permissions	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
S_3 App components			✓	✓	✓					✓		✓		✓			✓			
S_4 Filtered intents		✓				✓	✓						✓			✓				
S_5 Restricted API calls	✓				✓															
S_6 Used permissions																				✓
S_7 Suspicious API calls	✓	✓		✓		✓	✓		✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	
S_8 Network addresses			✓					✓	✓	✓	✓				✓	✓		✓	✓	✓

Figure 2.4: Contribution of feature sets to the detection of malware families [3]

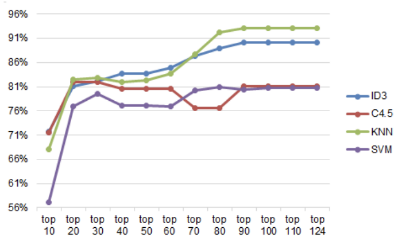
The Drebin analysed top features used by the Drebin classifier, as seen in Figure 2.3. Although *Suspicious API Call* appears the most often, *Requested Permissions* is also a key element, suggesting its usefulness in classification. In Figure 2.4, *Requested Permissions* is the only feature utilised across all malware families analysed, reinforcing the need for further study.

In this paper, we wanted to explore this further in the context of concept drift, as stated in RQ1, could "fixed" elements such as permissions prove more resilient to concept drift than more ambiguous concepts? Unlike other elements, permissions could remain a more consistently employed feature in malware; as noted in Figure 2.3, spammers will always need to access *SEND_SMS* to achieve their aims, regardless of how other elements of the application function.

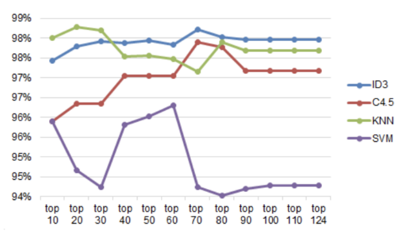
The Drebin classifier is somewhat of an outlier in the way it utilises a range of completely separate features to classify samples. Another key paper, *DroidAPIMiner: Mining API-Level features for Robust Malware Detection in Android* [19], takes a different approach, focusing on the extraction of Android APIs and package level information from the bytecode. Their approach produced accuracy scores “as high as 99%”. Using frequency analysis, the researchers were able to refine a list of “distinct” (Class Name, Method Name, and Descriptor) APIs associated with malware. These APIs can be split into categories including Application-specific resource APIs, Android Framework and System Resource APIs among others. The researchers are also able to collect information about third-party packages and API Parameters for their data set. The classifier used four different classification models including K-nearest neighbour [20], Linear Support Vector Machines [21], ID3 Decision Trees [22], and C4.5 Decision Trees [23] and found significant variations in performance between the four classification methods, as demonstrated in Figures 2.5 and 2.6. This paper helped reinforce the justification for RQ2, as it demonstrated quite a significant variation in quality between the four different classification models, further suggesting the need for review in the context of concept drift. Originally, DroidAPIMinier was going to feature as a comparison in this report, however, due to some accessibility limitations detailed later in this we were unable to include it. We do however include in our experiments a classification approach from Mariconti et al known as MaMaDroid. In the paper, *MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models* [4], their approach is directly compared with DroidAPIMiner and found to significantly outperform it, potentially limiting the usefulness of including DroidAPIMiner in this study.



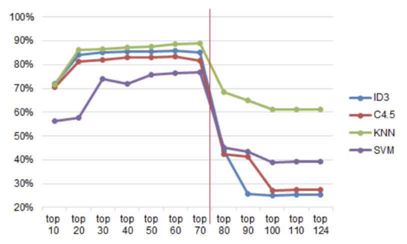
(a) Accuracy



(b) TPR

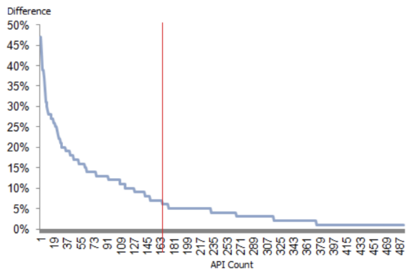


(c) TNR

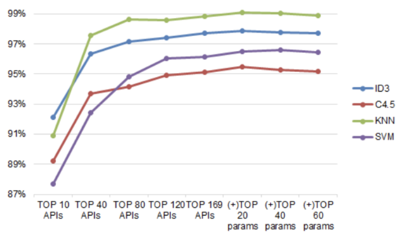


(d) Classification Rate of Modified Malware Set

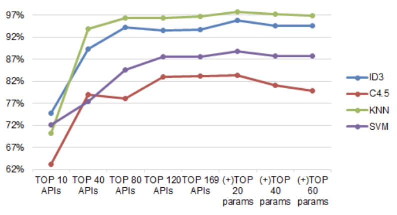
Figure 2.5: Performance of Permission-based Models [19]



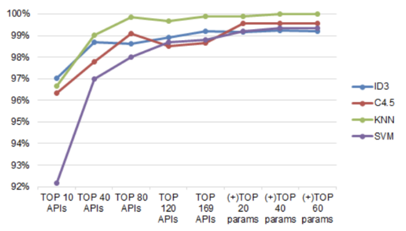
(a) API Features Distribution



(b) Accuracy



(c) TPR



(d) TNR

Figure 2.6: Performance of API-based Models [19]

The classification approach from Mariconti et al [4] forms Markov chains “from the sequence of abstracted API calls performed in the app, and uses it to extract features and perform classification.” This novel use of Markov chains to create a feature set yielded excellent results, with F1 scores of up to 99%. However, this approach was not as effective at resisting concept drift as compared to Drebin [3], as noted in Figure 2.2. Mariconti et al suggest that DroidAPIMiner relies “on most common calls observed during training”, which “prompts the need for constant retraining, due to the evolution of malware and the Android API alike”, however, their approach of using sequences of abstracted API calls allows them to account for changes in the Android API.

Another notable approach to Android malware classification was proposed by Kang et al [24] in 2016, which used a feature set of opcodes extracted from application bytecodes and then classified with Support Vector Machines, with a final F1 score on their dataset of 2520 samples of 98%. The key argument in the paper is that many approaches rely on “expert analysis to design or determine the discriminate features that are passed to the machine learning system used to make the final classification system”, such as requested permissions or API calls, “excluding potentially useful learning information”. This approach instead collects all n-opcodes in an application and allows the classifier to find the most useful features. The evaluation of this approach in the report is somewhat limited, with a relatively small dataset of samples used for evaluation when compared with other mainstream papers such as Drebin. Larger datasets may have demonstrated different and more reliable results. This significant variation in dataset size between papers such as this and others such as Drebin [3] or MaMaDroid [4] also serves to justify RQ4, as dataset size variance is something innate within this research space, but is not something that has been explored within the context of concept drift.

Another issue with Kang et al’s [24] work was a complete lack of available data or source code, something that seems to plague this sector of research and something detailed later as a limitation of this project. As such this work is completely unreproducible. However, a much more recent paper from 2021 by Sihang et al [5], with available data, had a similar approach. Named BLADE, Sihang et al's approach to Android malware classification had a particular focus on beating obfuscation of Android bytecode by attackers. Their approach "represents each malware sample with [an] Opcode Segment Document (OSD) generated from its DEX code", where the DEX code (Dalvik Executable format) is the bytecode executed at runtime [25], establishing symbols for all 224 Dvalik instructions. Classification on the extracted feature set is then implemented with C4.5 [23], Random Forest [26] and Sequential Minimal Optimisation [27]. Experiments were then performed on multiple different large datasets, including Drebin, giving consistently high F1 and accuracy scores. This project, therefore, presented a very promising candidate for concept drift experimentation and is a key point of comparison in this paper.

2.4 Data sets and codebases

As part of research into into related work, this project also reviewed both available datasets and codebases associated with Android malware classification and concept drift. We were very kindly provided both the Drebin implementation used by Mariconti et al [7] in their paper and the full TESSERACT evaluation framework. We were also able to directly access the Drebin and MaMaDroid feature set files used in the TESSERACT paper, which saved a lot of time and effort required to reproduce them.

In 2014, Drebin was re-implemented in 2017 in a public Github repository owned by user *MLDroid*. The exact purpose of this re-implementation is not clear but it could be associated with an Android malware detection framework also named MLDroid [29]. This implementation features both feature generation and classification. In testing, we were able to produce features from input APK files which seemed to be reasonable although no in-depth comparison was done to compare the results with the feature set from the TESSERACT [7] paper. This re-implementation is notable because of the various methods by which vectorization of the produced feature set is performed. In earlier iterations of the codebase (as found in the Github repository), both the *DictVectorizer* [30] and *TfidfVectorizer* [31] from scikit-learn's [32] python library were considered, with *TfidfVectorizer* the one in use at the last commit. In contrast, the TESSERACT [7] implementation of Drebin feature set classification uses the *DictVectorizer*. The Drebin paper [3] itself proposes a basic binary vector embedding for set S, where $S := S_1 \cup S_2 \cup ... \cup S_8$ for each of the 8 feature sets. They create an S-dimensional vector $\phi(x)$ defined for a set of applications X as follows:

$\phi : X \rightarrow 0, 1^{|S|}, \phi(x) \mapsto (I(x, s))_{s \in S}$ where indicator function I(x, S) is defined as:

$$I(x,s) = \begin{cases} 1 & \text{if the application } x \text{ contains feature } s \\ 0 & \text{otherwise.} \end{cases}$$

This method of vectorization is represented as the *DictVectorizer* [30] in the scikit-learn [32] library, which creates a "one boolean-valued feature ... for each of the possible string values that the feature can take on." The *TfidfVectorizer* in contrast is generally more relevant to analysing long-form text in documents for Natural Language Processing tasks [33] than classification feature sets and considers the "term frequency of a word in a document" [33]. This variance in vectorization approach is what formed the motivation for RQ3, as it is likely that these different approaches in data representation would affect resultant concept drift.

During the research phase of this project, we also reviewed an implementation of DroidAPIMiner [34] created by Chen Jun Hero in 2018. This particular implementation ran well (albeit very slowly) and produced very reasonable datasets with the APIs we provided, however, we suffered from limitations with the size of the feature set we could build with available APK files. We were provided with a subset of 1517 APK files (split between benign and malicious) from the original set used in the TESSERACT [7] paper, however using this small dataset size inside the TESSERACT testing environment as detailed in section 3.1 produce completely unusable output, bringing us to the conclusion that the dataset available to us was simply too small.

Accessible high-quality datasets that can be used for the analysis of concept drift in the context of Android malware were difficult to find. For many datasets, providing timestamps associated with each APK (necessary to perform analysis of concept drift over time) is not considered a high priority. For example, AndroZoo [36], a collection of Android APKs analysed by “tens of different AntiVirus products” has 18,799,179 APKs available through their API, of which 8,411,143 (44.7%) do not have an associated release date attached to them.

This problem is only compounded by obsolescence. Many large Android Malware dataset projects have been abandoned in previous years with final datasets left unreleased. These include the *Android Malware Genome Project* [37], the *AMD Project* (website no longer accessible) [38], and the *Android PRAGuard Dataset* [39] which are all now discontinued. This makes a comprehensive analysis of the Android malware landscape difficult not just for this project but for the research sector as a whole.

Chapter 3

Experimental approach

3.1 The Experimental Pipeline

In order to perform our four experiments to answer our four research questions this project relies on an experimental pipeline using the TESSERACT codebase. This section gives details of each element that makes up this pipeline and provides a pseudocode outline. The TESSERACT codebase, and therefore by extension the work carried out by this report, is written in Python 3.7 and uses a number of libraries including scikit-learn [32] to perform classification and subsequent analysis.

3.1.1 Feature Sets

Our experiments utilise a total of three different feature sets from three different classifiers in order to give insights into the effect of various feature abstractions.

Our Drebin [3] feature set is the set used as part of the TESSERACT [7] paper. It features 129728 elements made up of 7 features, with more detail outlined in section 4.1.1.

Our MaMaDroid feature set was also used as part of the TESSERACT [7] paper. It features a full set of extracted features from the same 129728 original applications as the Drebin feature set. These features are the extracted Markov chains detailed in the MaMaDroid [4] paper.

The BLADE [5] feature set was assembled from two different sources. The BLADE dataset - featuring a set of OSD documents (as explained in section 2.3, was released on Kaggle [42], a "machine learning and data science community" featuring ML related public datasets. Sadly the source code for the dataset is unavailable so it is not possible to directly verify the data, but the results seen below provide solid evidence that the data provided is of good quality.

The dataset provided on Kaggle consists of feature sets created from collections including AndroAutopsy [43], AndroTracker, Drebin, and PRAGuard [39]. However, this data was difficult to work with. For the Drebin dataset, the collection did not feature benign samples, only malware, making it redundant. There were also no timestamps within the dataset, which are required for analysis of concept drift through time. In order to solve this problem, we combined the AndroAutopsy dataset of benign and malware samples with the textual description of the dataset that contained the relevant timestamps to produce the dataset of 8948 items. This difference in the size of the dataset between BLADE and the other two datasets has a substantial impact on the results seen later in this report.

3.1.2 Representation & Classification

The method of representation and classification will vary depending on the experiment. In these experiments we define the *default* method to vectorize the features using the DictVectorizer [30] and then classify with scikit-learn's LinearSVC (support vector classifier) [45]. As we vary different parts of the experimental pipeline in our four experiments, it is important to define a baseline on which other results are compared. Choosing this combination is firstly due to its use as the vectorizer and classifier of choice for the TESSERACT [7] project. Using these, therefore, gives us grounding in previously completed work and makes it easier to make direct comparisons, Secondly, these methods already give good results, and so any improvements we can make will represent novel contributions to this area of research. In the MaMaDroid [4] paper, the LinearSVC classifier is regarded as less useful than the other ones tested, however, given its usefulness for Drebin and fast training time, it is still a very reasonable pick as the default classifier for these experiments. It is important to note that although the RandomForestClassifier [48] is the classification method of choice for MaMaDroid and BLADE [5], training on this classifier on available hardware took about half an hour for MaMaDroid and about five hours for Drebin. Using this method as a default in other experiments would have been extremely impractical.

3.1.3 Result plotting

TESSERACT produces a set of results to demonstrate concept drift. After training the dataset over a specified number of months at the beginning of the time period, where the time period is the period of time for which we have data to analyse in the dataset, TESSERACT then returns the F1 scores obtained by testing the trained classifier on each subsequent month in the dataset. For Drebin and MaMaDroid we train on 12 months and test on 24, and for Blade we train on 10 months and test on 12 due to the reduced dataset size. Graphs can then be plotted demonstrating the negative and positive class F1 scores for each month. In this case, the negative class represents goodware and the positive class malware. The discussion in the subsequent experiments focuses on the positive class (malware) as the negative class (goodware) results are usually close to perfect. Goodware (gw) results are always shown with the dotted lines, with malware (mw) shown with the solid lines.

3.1.4 Pipeline pseudo-code

The general framework in which this project's experiments are run is referred to as the TESSERACT [7] testing environment. Algorithm 1 gives a high-level overview.

Algorithm 1 TESSERACT testing environment pseudo-code

```
Load dataset into X (features), Y (classification), and t (timestamp)
Perform a fit_transform [63] using a vectorizer
Perform a "time aware" partition of the dataset
Perform a sliding window classification on the timestamp partitioned dataset with a particular classifier type
Plot F1 results and calculate AUT scores
```

The full implementation of this algorithm is available in Appendix A, figure A.2

3.1.5 Classifier baselines

We can run our three feature sets with *default* parameters inside our TESSERACT testing environment set up for this project to produce some initial time-aware analysis that can act as both a useful explainer for the kind of output the TESSERACT testing environment can create and a baseline for future results. Figures 3.1, 3.2, and 3.3 give the analysis for Drebin [3], MaMaDroid [4], and BLADE [5] respectively.

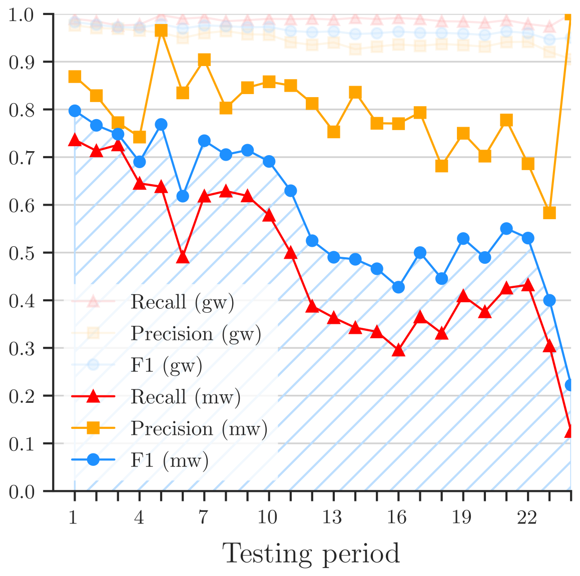


Figure 3.1: Drebin [3] default output in TESSERACT testing environment (DictVectorizer & LinearSVC)

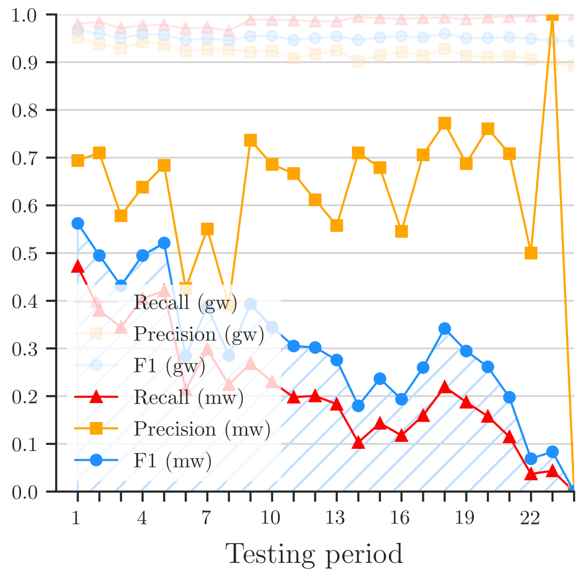


Figure 3.2: MaMaDroid [4] default output in TESSERACT testing environment (DictVectorizer & LinearSVC)

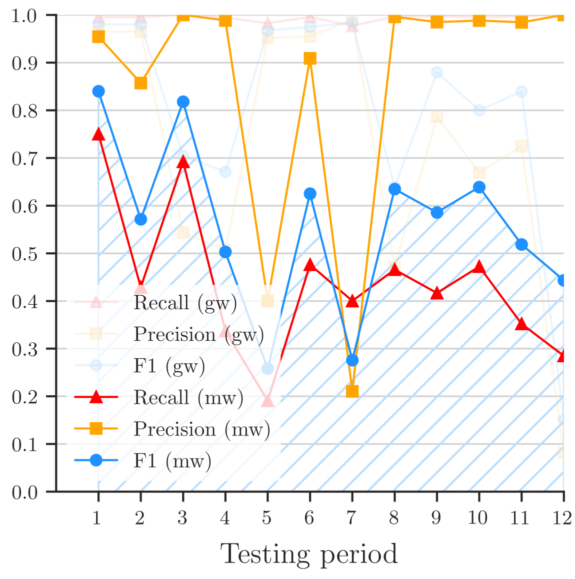


Figure 3.3: BLADE [5] default output in TESSERACT testing environment (DictVectorizer & LinearSVC)

The results for Drebin and MaMaDroid we produce in our environment can be compared to the results achieved in Pendlebury et al's [7] paper detailing the TESSERACT environment. We have the figures from the report available in figure 2.2. We can observe the results we achieve for Drebin are nearly identical to the results achieved in the TESSERACT paper. This is very promising and suggests we have set up our environment correctly, however this is not the case for MaMaDroid, where although a similar trend is produced, the F1 scores are simply not as high. In the original TESSERACT paper, MaMaDroid was represented with the RandomForestClassifier rather than the LinearSVC, which would explain the difference between the scores in the TESSERACT paper and the scores we produce in our testing environment. Later on, in the second experiment, detailed in section 4.2, we do run MaMaDroid with a RandomForestClassifier and although produced are similar, the two graphs still do not match. This is most likely down to slightly different parameters set in the RandomForestClassifier between this report and the TESSERACT paper. The specific parameters set out in the MaMaDroid paper and then used in the TESSERACT paper, namely using 101 trees with a max depth of 64, are different from the default settings for a RandomForestClassifier in sklearn [48]. For the sake of reproducibility, straightforwardness, bias minimisation, and in an attempt to generalise our results, in this paper, we opt to pick to keep sklearn objects to their default state as much as possible, as such we choose to use the RandomForestClassifier in its default settings rather than specify the specific parameters used in the MaMaDroid paper.

This difference most likely has to do with some variance between the particular environment set up for the TESSERACT paper this one. It could also be that the final MaMaDroid feature set used to produce the results seen in the TESSERACT paper is different from the set used in this project. This does not invalidate our results however as the plots are still of sufficiently good quality, and the results in this report are compared against each other rather than the results from the TESSERACT paper.

BLADE [4], seen in figure 3.3 has not been tested in this environment before, so there is nothing outside this paper to compare it to. It is obvious when comparing the output for Drebin and MaMaDroid that the Dataset is much smaller and more restricted, manifesting in a lot more volatility. This becomes increasingly notable during the following experiments.

3.1.6 Metrics

Our results use two key metrics. These are F1 scores and AUT scores. F1 scores are defined as the harmonic mean between precision and recall [60][61]. F1 is a very useful metric, employed heavily by Pendlebury et al in their TESSERACT paper [7]. It allows us to account for both recall and precision in one metric, which in turn allows us to account for the true positive, false positive, and false-negative rate of our classifiers. As can be seen in our results, the true negative rate, that is benign applications correctly identified, is almost perfect in most representations, and as such is not important to consider.

$$F_1 = \frac{2}{\frac{1}{recall} \times \frac{1}{precision}} = 2 \times \frac{precision \times recall}{precision + recall}$$

Where precision and recall are defined as:

$precision = \frac{tp}{tp+fp}$

$recall = \frac{tp}{tp+fn}$

The other key metric used in our analysis is AUT. Area Under Time was coined by Pendlebury et al [7] as a way to "evaluate performance of a malware classifier against time decay over N time units in realistic experimental settings." It does this by calculating the area under the performance curve over time and is defined as follows:

$$AUT(f, N) = \frac{1}{N-1} \sum_{k=1}^{N-1} \frac{[f(x_{k+1}) + f(x_k)]}{2}$$

Figure 3.4: AUT definition from Pendlebury et al [7].

AUT is a "simple yet effective metric that captures the performance of a classifier with respect to time decay, de-facto promoting a fair comparison across different approaches." [7]

Chapter 4

Experiments

Note about graph formats In this chapter, we display our concept drift results in graphs that appear differently from ones seen previously including figures 3.1 and 3.2. The previous graphs are generated from pre-existing visualisation code within the Tesseract framework, hence their similarity to the ones found in the TESSERACT paper [7]. Instead of using this pre-existing implementation, we thought it would be more straightforward to re-develop a visualisation method that best fits the results of this project. Although the graphs look different, the results come from the same place.

4.1 Drebin [3] feature comparison

The aim of the Drebin [3] feature comparison experiment is to answer RQ1 - *Are metadata-based features, such as permissions, more resilient to concept drift than those assembled from the bytecode?* This is done by directly comparing features in the context of concept drift.

4.1.1 Method

In figure 4.1 and 4.2, we split the features into two distinct groups, features derived from the *manifest* (application metadata), and features derived from the bytecode itself. It is also notable that the number of elements for each feature class set is not consistent. Network addresses for example have over five times the number of features as many other feature sets considered. This will skew the results but it was left in intentionally, as the number of elements available is an important factor to consider. It may be for example that although manifest derived features could be more resilient to concept drift, it may be that this simply is not relevant if their inclusion does not provide enough value to the classifier outside the context of concept drift.

Drebin manifest feature set	**AUT(F1, 24)**
app_permissions	0.545
intents	0.582
activities	0.593

Figure 4.1: Drebin [3] manifest feature sets and associated dataset information

Drebin bytecode feature set	AUT(F1, 24)
api_calls	0.576
api_permissions	0.581
interesting_calls	0.592
urls	0.455
full_dataset	0.583

Figure 4.2: Drebin [3] bytecode feature sets and associated dataset information

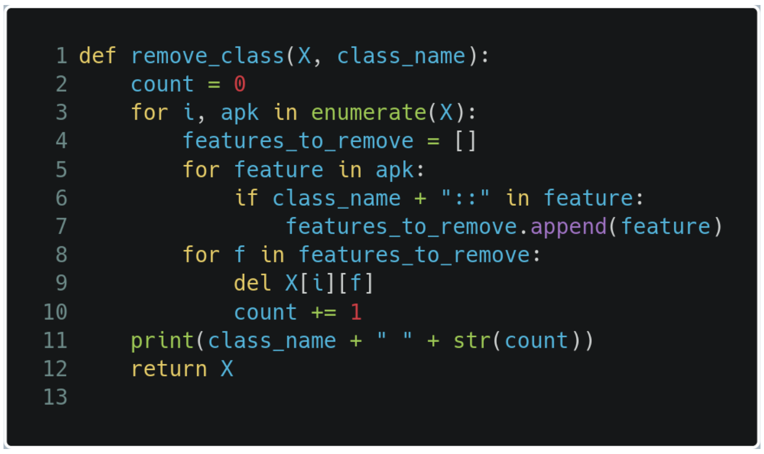
In order to effectively compare each element, it was decided that running each feature class through the TESSERACT test environment would not produce useful results as the feature sets would be too small and variable in size. Instead, this experiment does the opposite, removing one feature at a time from the Drebin [3] dataset and comparing the effect on the rest of the dataset. This approach means that when analysing the results the worse score indicates the feature in question has a higher degree of importance in the context of concept drift. We can iterate through all 7 feature classes within the Drebin [3] dataset, removing one feature at a time, and then run the remaining dataset through the TESSERACT testing environment with default parameters. We can then draw our conclusions by comparing each feature individually and as part of our two subsets.

4.1.2 Experiment

For this experiment, we iteratively remove each of the seven feature sets within the Drebin dataset. In the X (feature) set, of the Drebin dataset, each specific feature is prepended with the feature name. Some examples are included in figure 4.3. This allows us to search through the X set and remove each element with the relevant feature name. The code to do so is shown in figure 4.4.

```
Example features:
    "app_permissions::name='Android_permission_WRITE_MEDIA_STORAGE'
    "urls::http:\/\/vpclub_octech_com_cn\/download_app_html"
    "activities::_activity_QueryCheckGoodsActivity"
```

Figure 4.3: Drebin X feature format examples



```python
def remove_class(X, class_name):
    count = 0
    for i, apk in enumerate(X):
        features_to_remove = []
        for feature in apk:
            if class_name + "::" in feature:
                features_to_remove.append(feature)
        for f in features_to_remove:
            del X[i][f]
            count += 1
    print(class_name + " " + str(count))
    return X

```

Figure 4.4: Code to remove feature class from Drebin dataset

This manipulated dataset is then run through the TESSERACT testing environment with default parameters.

4.1.3 Results

Figure 4.5 gives demonstrates the effect of concept drift over the 24 month testing period. It is notable that the variance between each feature set is not particularly large with the exception of *urls*, which is notably the worst performer. As seen in figure 4.2, the *urls* feature set has by far the largest number of items, which most likely explains it as an outlier. Tables 4.6 and 4.7 give AUT scores for manifest and bytecode feature sets respectively. Our results do not convincingly prove that metadata features are significantly more or less resilient to concept drift, but there are some interesting takeaways. Running the full Drebin feature set gives us an AUT score of 0.583. This means that some of the features, specifically *activities* and *interesting_calls* make the Drebin dataset less resilient to concept drift than it would be without them. It is likely that these two feature sets have significant variance over time and therefore act to decrease the quality of the classifier over time. The set *actvities* represents a core part of the metadata set and is shown to be the largest detriment to concept drift resilience.

Feature sets *intents* and *api_permissions* barely affect the AUT figure at all however most likely for different reasons. *api_permissions* very closely mirrors the full dataset results in figure 4.5, which implies it is the least useful feature in the set and is generally ignored by the classifier. *intents* however does differ from the full dataset line, but seems to average out to the same AUT score as it under-performs the full dataset in the later months of the testing period. This would imply that the *intents* feature set skews from being more important to the classifier in the earlier testing months and then less important in the later months, averaging out to the same AUT score.

The average AUT score for the bytecode feature sets, seen in figure 4.7, is affected heavily by *urls*, which gives by far the lowest score and can is consistently the worst performer in figure 4.5. This demonstrates that *urls* is by far the most significant feature when it comes to concept drift. This weighting is most likely down to the feature set size, which is roughly five times larger than many of the other sets. Discounting this feature set gives an average for the bytecode gives an average of 0.583, which would make the manifest feature set slightly more concept drift resilient.

The lowest AUT score not including *urls* is *app_permissions*, and is demonstrated to give consistently lower scores in figure 4.5. This however does not suggest that *app_permissions* provides more resilience to concept drift than other methods, as we do not see the gap between *app_permissions* and the full dataset widen over time. Our data suggests that *app_permissions* is simply a consistently important metric in terms of accuracy and precision of the classifier. The only feature set here that is seen to provide resilience to concept drift is *urls*, as it is the only one that is seen to decline at a greater rate than the full dataset.

To conclude this section, this evidence does not support the claim that metadata-based features provide greater concept drift resilience than features from the bytecode, but instead suggests that many features have little effect on concept drift at all. From this data, we can suggest that the greatest indicator of concept-drift resilience is simply the size of the feature set. However, it is important to note that this dataset is limited, and further analysis of other feature sets may reveal new trends.

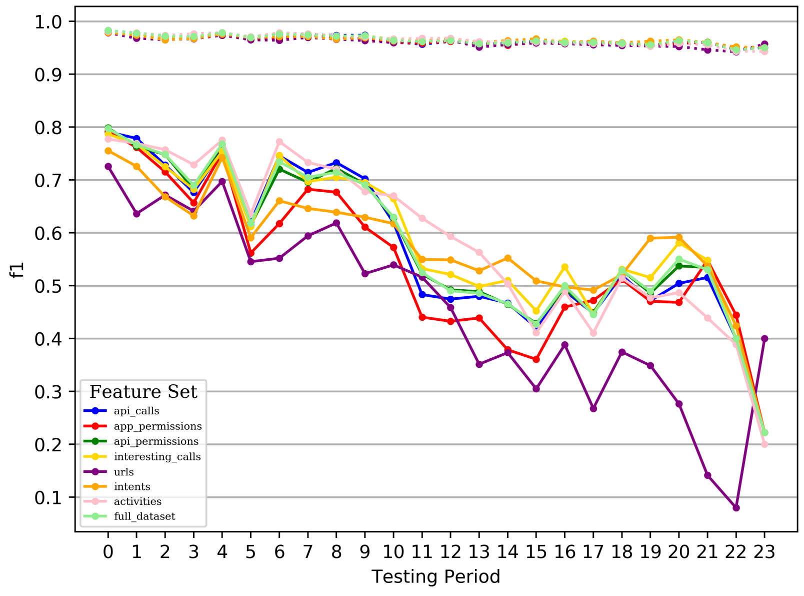


Figure 4.5: Time-aware plot of Drebin [3] features. (mw = solid line, gw = dotted line)

Drebin manifest feature set	**AUT(F1, 24)**
app_permissions	0.545
intents	0.582
activities	0.593
Average	0.573

Figure 4.6: Drebin [3] manifest feature set AUT results

Drebin bytecode feature set	**AUT(F1, 24)**
api_calls	0.576
api_permissions	0.581
interesting_calls	0.592
urls	0.455
Average	0.43

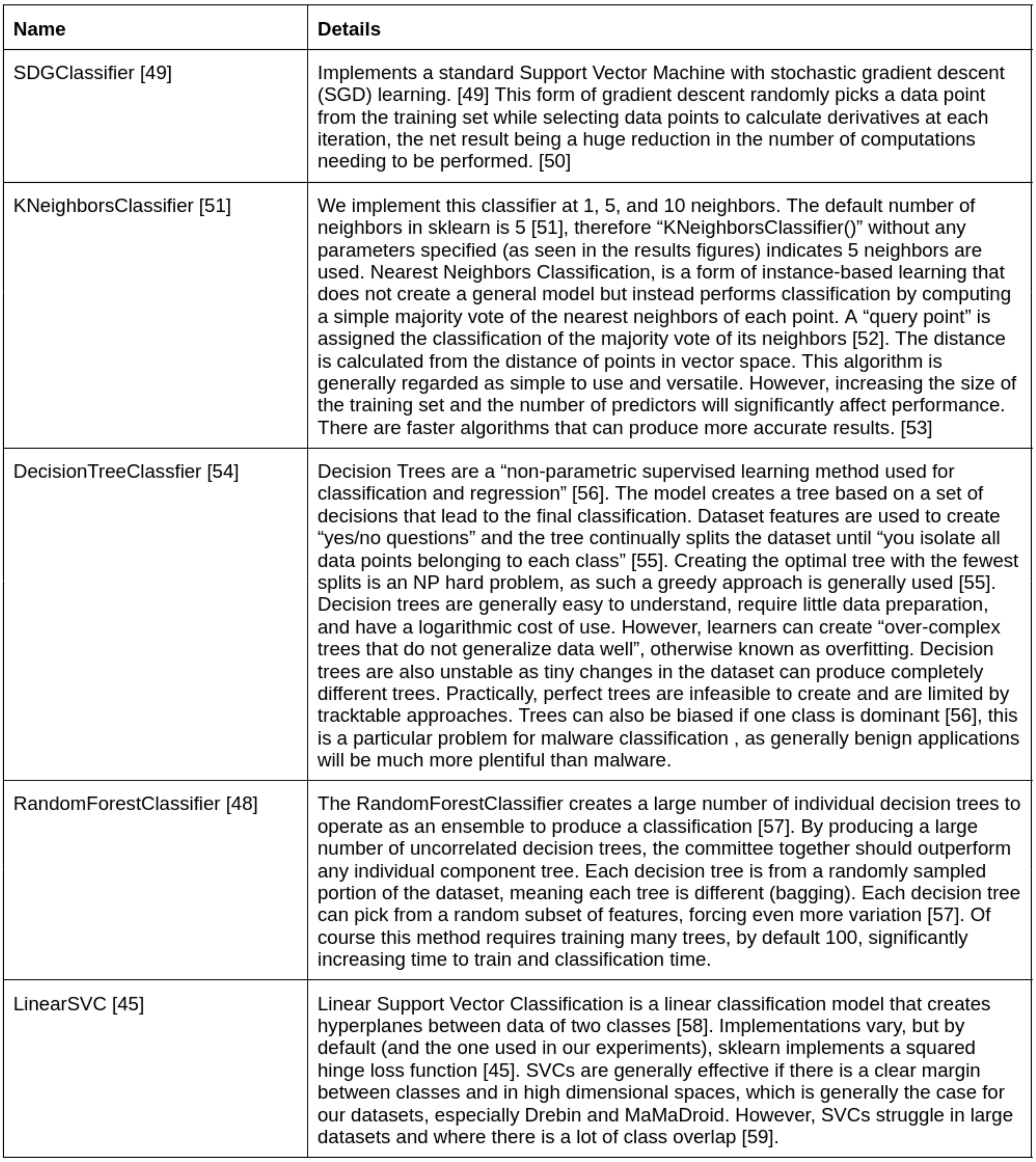
Figure 4.7: Drebin [3] bytecode feature set AUT results

4.2 Classification

The classification comparison experiment aims to answer RQ2 - *What is the effect of machine learning classification approach on concept drift in the context of different program analysis techniques?* This is achieved by varying the classifier used and comparing the results.

4.2.1 Method

The methodology behind this experiment is straightforward. To eliminate other variables the method of feature representation is fixed to our default *DictVectorizer* [30] and varying the classification approach. This experiment was then carried out on all three of our feature sets to give enough data to draw useful conclusions. This project is particularly interested in classifiers that appear repeatedly in the literature. An overview of the classifiers we use in this experiment is given in figure 4.8.

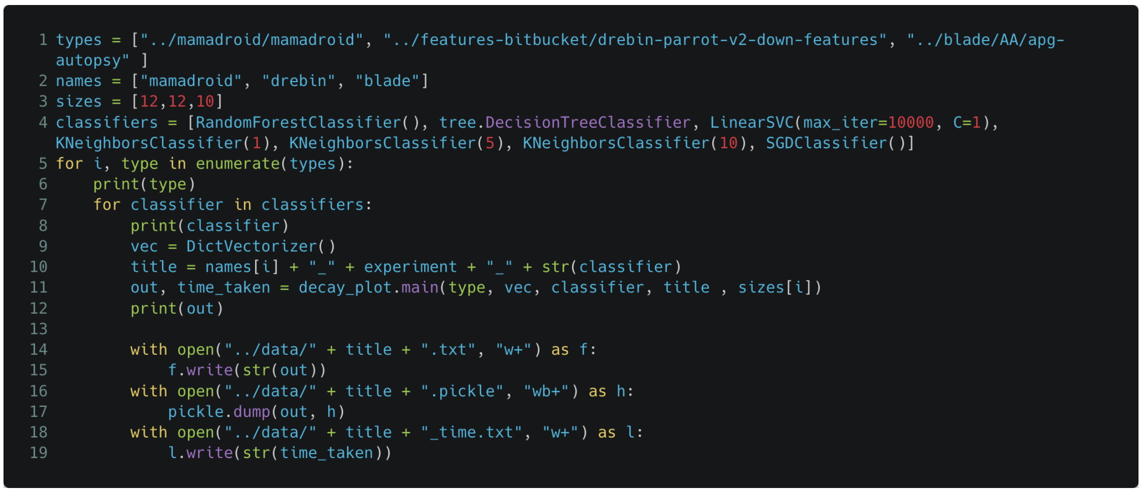


Name	Details
SDGClassifier [49]	Implements a standard Support Vector Machine with stochastic gradient descent (SGD) learning. [49] This form of gradient descent randomly picks a data point from the training set while selecting data points to calculate derivatives at each iteration, the net result being a huge reduction in the number of computations needing to be performed. [50]
KNeighborsClassifier [51]	We implement this classifier at 1, 5, and 10 neighbors. The default number of neighbors in sklearn is 5 [51], therefore "KNeighborsClassifier()" without any parameters specified (as seen in the results figures) indicates 5 neighbors are used. Nearest Neighbors Classification, is a form of instance-based learning that does not create a general model but instead performs classification by computing a simple majority vote of the nearest neighbors of each point. A "query point" is assigned the classification of the majority vote of its neighbors [52]. The distance is calculated from the distance of points in vector space. This algorithm is generally regarded as simple to use and versatile. However, increasing the size of the training set and the number of predictors will significantly affect performance. There are faster algorithms that can produce more accurate results. [53]
DecisionTreeClassfier [54]	Decision Trees are a "non-parametric supervised learning method used for classification and regression" [56]. The model creates a tree based on a set of decisions that lead to the final classification. Dataset features are used to create "yes/no questions" and the tree continually splits the dataset until "you isolate all data points belonging to each class" [55]. Creating the optimal tree with the fewest splits is an NP hard problem, as such a greedy approach is generally used [55]. Decision trees are generally easy to understand, require little data preparation, and have a logarithmic cost of use. However, learners can create "over-complex trees that do not generalize data well", otherwise known as overfitting. Decision trees are also unstable as tiny changes in the dataset can produce completely different trees. Practically, perfect trees are infeasible to create and are limited by tracktable approaches. Trees can also be biased if one class is dominant [56], this is a particular problem for malware classification , as generally benign applications will be much more plentiful than malware.
RandomForestClassifier [48]	The RandomForestClassifier creates a large number of individual decision trees to operate as an ensemble to produce a classification [57]. By producing a large number of uncorrelated decision trees, the committee together should outperform any individual component tree. Each decision tree is from a randomly sampled portion of the dataset, meaning each tree is different (bagging). Each decision tree can pick from a random subset of features, forcing even more variation [57]. Of course this method requires training many trees, by default 100, significantly increasing time to train and classification time.
LinearSVC [45]	Linear Support Vector Classification is a linear classification model that creates hyperplanes between data of two classes [58]. Implementations vary, but by default (and the one used in our experiments), sklearn implements a squared hinge loss function [45]. SVCs are generally effective if there is a clear margin between classes and in high dimensional spaces, which is generally the case for our datasets, especially Drebin and MaMaDroid. However, SVCs struggle in large datasets and where there is a lot of class overlap [59].

Figure 4.8: An overview of classifier methods used in this experiment

4.2.2 Experiment

To carry out this experiment. We build a test framework that iteratively runs our TESSERACT testing environment with each of the classifiers mentioned in table 4.8. We use our default vectorizer, the DictVectorizer [30], to represent the data. We then save the results and the time taken to run the experiment, allowing us to build graphs and tables later on. This code can be seen in figure 4.9.



```python
types = ["../mamadroid/mamadroid", "../features-bitbucket/drebin-parrot-v2-down-features", "../blade/AA/apg-autopsy" ]
names = ["mamadroid", "drebin", "blade"]
sizes = [12,12,10]
classifiers = [RandomForestClassifier(), tree.DecisionTreeClassifier, LinearSVC(max_iter=10000, C=1), KNeighborsClassifier(1), KNeighborsClassifier(5), KNeighborsClassifier(10), SGDClassifier()]
for i, type in enumerate(types):
    print(type)
    for classifier in classifiers:
        print(classifier)
        vec = DictVectorizer()
        title = names[i] + "_" + experiment + "_" + str(classifier)
        out, time_taken = decay_plot.main(type, vec, classifier, title , sizes[i])
        print(out)

        with open("../data/" + title + ".txt", "w+") as f:
            f.write(str(out))
        with open("../data/" + title + ".pickle", "wb+") as h:
            pickle.dump(out, h)
        with open("../data/" + title + "_time.txt", "w+") as l:
            l.write(str(time_taken))
```

Figure 4.9: An overview of classifier methods used in this experiment

4.2.3 Results

Figure 4.10 gives the F1 results for our three classifiers. AUT results are given in table 4.11. Training times, another useful metric to consider, is given in table 4.12. Although the graphs are busy, we are able to draw some useful conclusions. Most notably our BLADE results are incredibly variable. This is down to the much smaller dataset for BLADE as compared to Drebin and MaMaDroid.

For Drebin, we find that none of the various methods we test can overwhelmingly outperform the default LinearSVC. The SGDClassifier does have the highest AUT score, however, we find the line drawn in figure 4.10 to be much more variable in the later months. It is difficult to say where this variance comes from, but regardless it is consistently higher than LinearSVC. This is also very surprising given the significantly shorter training time for the SGDClassifier over LinearSVC. This makes this approach a “win-win”, especially in environments where computational power is constrained. In general, for Drebin we see a trend where a few classifiers, SGDClassifier, LinearSVC, and DecisionTreeClassifier, can outperform the others at the beginning of the testing period but are not able to hold onto this advantage, and even performed worse than other methods (see LinearSVC) in the later periods. This result is particularly interesting as it demonstrates that the classifier of choice will vary depending on the length of time a classification approach is required to withstand concept drift. If someone is building a model to only last about a year, one of the three aforementioned classifiers would make the best choices, but there may be instances where picking a classifier with less variation is more desirable. The three variations of the KNeighborsClassifier give particularly uniform results, with less variation throughout the testing period. In terms of time, for Drebin, most of the classifiers are within 200 to 300 seconds with DecisionTreeClassifier and RandomForestClassifier taking significantly longer, and SDGClassifier taking significantly less time (as previously mentioned). For Drebin, this makes the SGDClassfier the obvious best choice in resource-constrained environments.

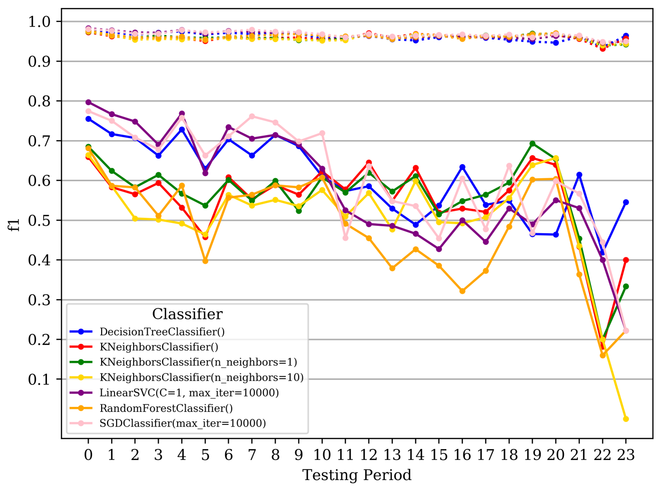
MaMaDroid's results consistently under-performed in comparison to Drebin. This is not particularly surprising, as MaMaDroid was also shown to under-perform Drebin in the TESSERACT paper. Although there was variance in AUT between the classifiers for MaMaDroid, the general trend observed in figure 4.10b is similar for all classifiers. Put another way, most classifiers tested in this experiment decay at a similar rate. Some notable exceptions to this rule are the SGDClassifier, which produces extremely variable results and generally decays very rapidly, and the RandomForestClassifier, which towards the end of the testing period decays quite rapidly. Surprisingly, the RandomForestClassifier performed as badly as is shown in our experiment considering it is the classifier of choice in the MaMaDroid paper. This suggests that the best classifier tested statically on a dataset may not necessarily line up with the best classifier at combating concept drift. In this example, it is the KNeighborsClassifier with five neighbours that produces the best AUT score overall. It is also notable that the SGDClassifier, which was the best out of the classifiers for the Drebin feature set, was the worst performer in this case. From Drebin and MaMaDroid there is no obvious best classifier to choose, which presents an interesting result in itself.

Finally, for BLADE, we see a graph in figure 4.10c that is incredibly inconsistent. This is almost certainly down to the significantly reduced dataset size for BLADE as compared with Drebin and MaMaDroid. This result can immediately be applied to RQ4, suggesting that a decreased dataset size will lead to much more wild and unpredictable results. This result is something that we will see reiterated in the next two experiments. Due to this generally wild output, it is difficult to concretely draw conclusions from the data presented. Notably, BLADE gives the largest variation in result, with the three KNeighborsClassifier variations being a good microcosm of this effect. We see the single neighbour KNeighborsClassifier perform by far the best out of any of the classifiers, whereas the five neighbours classifier performs poorly and the ten neighbours classifier is by far the worst, with the worst plot seen so far. The most significant takeaway from our BLADE results is we see a different classifier from Drebin and MaMaDroid, KNeighborsClassifier with one neighbour, perform the best. Choosing a classifier for a particular feature set must take into account how that particular feature set will behave both statically and under the effect of concept drift. Finally, it is notable for BLADE that we see the F1 scores for goodware, shown in the dotted lines, not be near-perfect for the first time. This result acts again as a good indicator that the dataset size is too small.

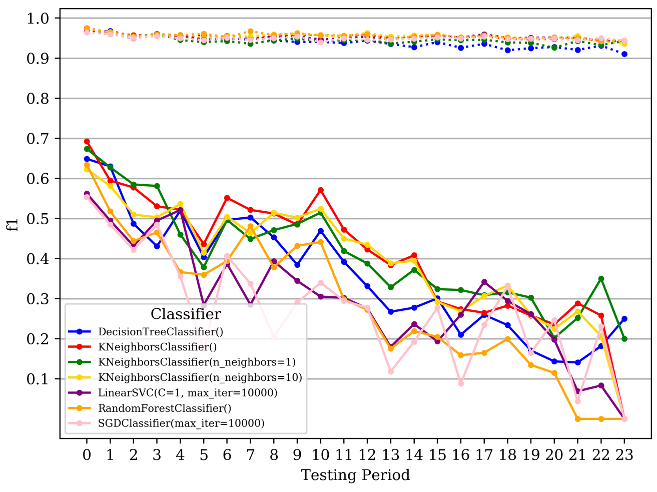
In both Drebin and BLADE we see the KNeighborsClassifier performs best with only one neighbour. We know that using two many neighbours can create errors [53], and having that become apparent so quickly could indicate that our datasets are not best suited to this form of classification. It may be that the delineation between our two classes, benign and malware, is not sufficient enough for a nearest neighbour approach to work effectively, thus limiting the effectiveness of multiple neighbours.

A particularly surprising result for all three feature sets is RandomForestClassifier [48], Which took by far the longest to train in Drebin and Blade and the second-longest in MaMaDroid. The classifier gave particularly poor results. In all three cases, it even gave worse results than the DecisionTreeClassifier. This was not expected. The RandomForestClassifier is effectively a set of DecisionTreeClassifiers, so we would expect the RandomForestClassifier to generally outperform the DecisionTreeclassifier. It may be that our particular setup resulted in overfitting when using the RandomForestClassifier on the initial training set, leading to particularly poor results. It may be that rerunning the results on a smaller set of DecisionTrees in our forest would produce significantly better results.

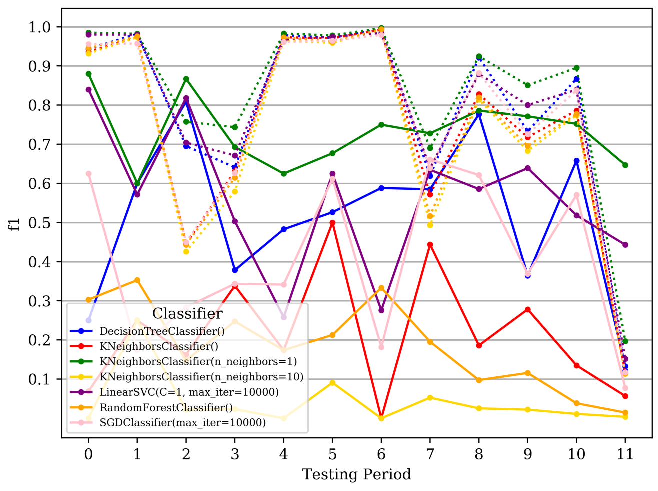
This set of results draws out some key conclusions. The best classifier, both in terms of maximising the quality of the results and minimising concept drift, varies depending on implementation. Drebin demonstrated some classifiers on some feature sets can be biased towards earlier testing periods before potentially declining more significantly in later periods. This demonstrates that the period a classification approach is required to be resilient to concept drift could potentially impact the best classifier for a particular feature set. Other feature sets, such as MaMaDroid, produce less variation, and the best approach is much more obvious. BLADE demonstrates the effect of having a particularly small dataset, with a much greater variation in results seen.



(a) Drebin



(b) MaMaDroid

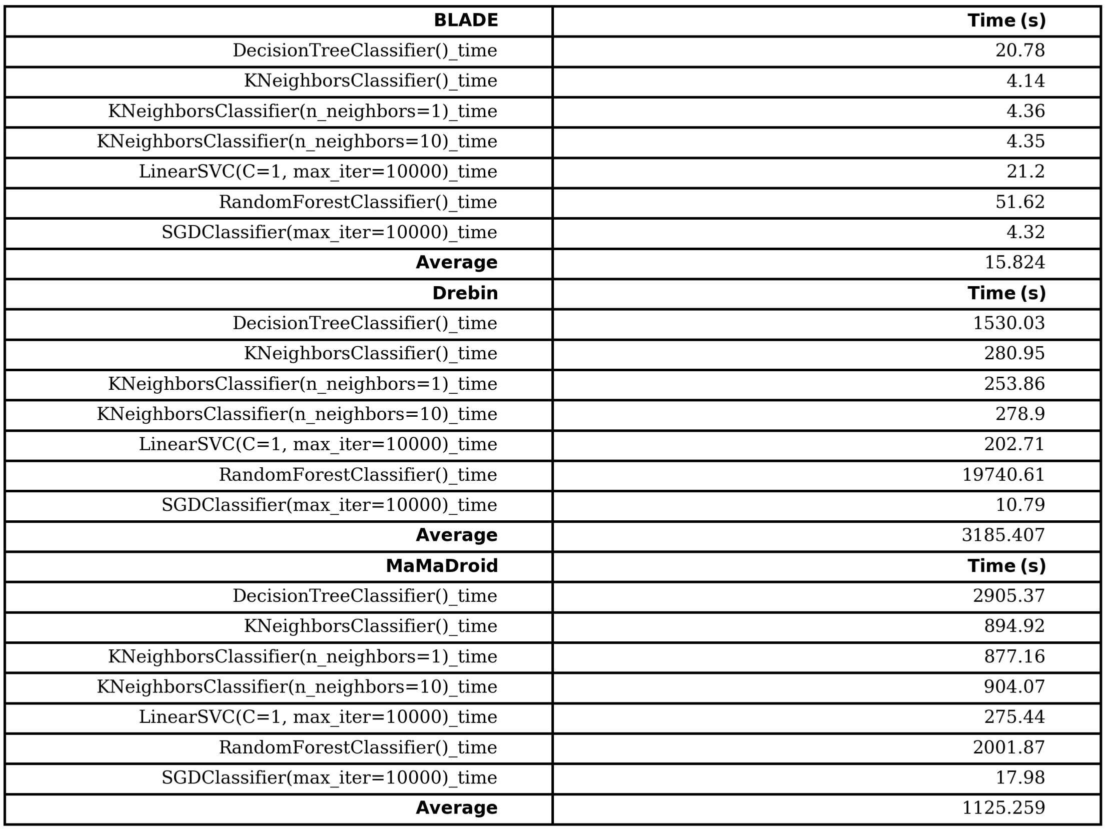


(c) BLADE

Figure 4.10: F1 results for experiment 2. (mw = solid line, gw = dotted line)

BLADE	**AUT(F1, 24)**
DecisionTreeClassifier()	0.542
KNeighborsClassifier()	0.23
KNeighborsClassifier(n_neighbors=1)	0.728
KNeighborsClassifier(n_neighbors=10)	0.045
LinearSVC(C=1, max_iter=10000)	0.552
RandomForestClassifier()	0.189
SGDClassifier(max_iter=10000)	0.41
Drebin	**AUT(F1, 24)**
DecisionTreeClassifier()	0.603
KNeighborsClassifier()	0.55
KNeighborsClassifier(n_neighbors=1)	0.561
KNeighborsClassifier(n_neighbors=10)	0.512
LinearSVC(C=1, max_iter=10000)	0.583
RandomForestClassifier()	0.481
SGDClassifier(max_iter=10000)	0.614
MaMaDroid	**AUT(F1, 24)**
DecisionTreeClassifier()	0.354
KNeighborsClassifier()	0.413
KNeighborsClassifier(n_neighbors=1)	0.407
KNeighborsClassifier(n_neighbors=10)	0.4
LinearSVC(C=1, max_iter=10000)	0.301
RandomForestClassifier()	0.284
SGDClassifier(max_iter=10000)	0.273

Figure 4.11: Experiment 2 AUT results



BLADE	**Time (s)**
DecisionTreeClassifier()_time	20.78
KNeighborsClassifier()_time	4.14
KNeighborsClassifier(n_neighbors=1)_time	4.36
KNeighborsClassifier(n_neighbors=10)_time	4.35
LinearSVC(C=1, max_iter=10000)_time	21.2
RandomForestClassifier()_time	51.62
SGDClassifier(max_iter=10000)_time	4.32
Average	15.824
Drebin	**Time (s)**
DecisionTreeClassifier()_time	1530.03
KNeighborsClassifier()_time	280.95
KNeighborsClassifier(n_neighbors=1)_time	253.86
KNeighborsClassifier(n_neighbors=10)_time	278.9
LinearSVC(C=1, max_iter=10000)_time	202.71
RandomForestClassifier()_time	19740.61
SGDClassifier(max_iter=10000)_time	10.79
Average	3185.407
MaMaDroid	**Time (s)**
DecisionTreeClassifier()_time	2905.37
KNeighborsClassifier()_time	894.92
KNeighborsClassifier(n_neighbors=1)_time	877.16
KNeighborsClassifier(n_neighbors=10)_time	904.07
LinearSVC(C=1, max_iter=10000)_time	275.44
RandomForestClassifier()_time	2001.87
SGDClassifier(max_iter=10000)_time	17.98
Average	1125.259

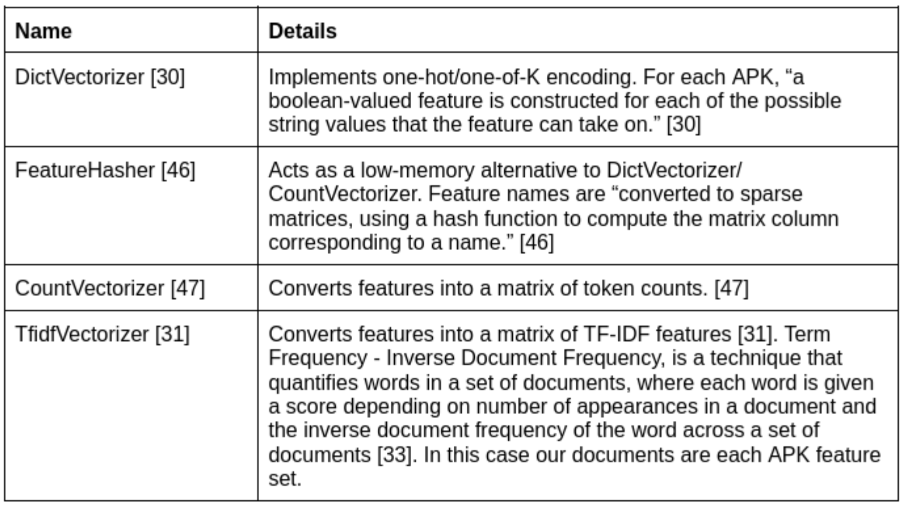
Figure 4.12: Experiment 2 AUT results

4.3 Feature representation

The aim of the feature representation experiment is to answer RQ3 - *Q3 - What is the effect of feature representation on concept drift in the context of different program analysis techniques?* We intend to analyse a variety of vectorization methods and compare their effectiveness, as well as other benefits and drawbacks, on our three classification datasets.

4.3.1 Method

We chose a variety of approaches to feature vectorization, detailed in the table in figure 4.13. The aim was to choose a variety of techniques that are well known and used, accessible, and provide interesting trade-offs to analyse.



Name	Details
DictVectorizer [30]	Implements one-hot/one-of-K encoding. For each APK, “a boolean-valued feature is constructed for each of the possible string values that the feature can take on.” [30]
FeatureHasher [46]	Acts as a low-memory alternative to DictVectorizer/ CountVectorizer. Feature names are “converted to sparse matrices, using a hash function to compute the matrix column corresponding to a name.” [46]
CountVectorizer [47]	Converts features into a matrix of token counts. [47]
TfidfVectorizer [31]	Converts features into a matrix of TF-IDF features [31]. Term Frequency - Inverse Document Frequency, is a technique that quantifies words in a set of documents, where each word is given a score depending on number of appearances in a document and the inverse document frequency of the word across a set of documents [33]. In this case our documents are each APK feature set.

Figure 4.13: Feature representation descriptions

Like Experiment 2, seen in section 4.2, the methodology is straightforward. By fixing the method of classification to our default, the LinearSVC [45], we can vary our method of vectorization and compare the results in order to get an understanding of how the method of representation affects our classifiers in the context of concept drift.

4.3.2 Experiment

Each of the three feature sets are run iteratively within the TESSERACT testing environment using each of the four vectorizers listed in figure 4.13. The implementation is similar to experiment 2, as detailed in figure 4.9, except we vary the vectorizer and keep the classifier constant.

4.3.3 Results

Figure 4.14 gives the F1 results of the time-aware analysis for Drebin, MaMaDroid, and BLADE for each of the four vectorizers. Table 4.15 gives the AUT scores and table 4.19 gives time taken for each experiment. It is notable that in figure 4.14c The DictVectorizer line is hidden because it perfectly mirrors the FeatureHasher line. In all three results, we see that the TfidfVectorizer is the worst performer. This can seem reasonable considering the vectorization method is not particularly designed for use in this kind of machine learning task and is generally more useful when using working with text documents, however, the extreme difference between the TfidfVectorizer and the other three approaches could potentially suggest some methodological error.

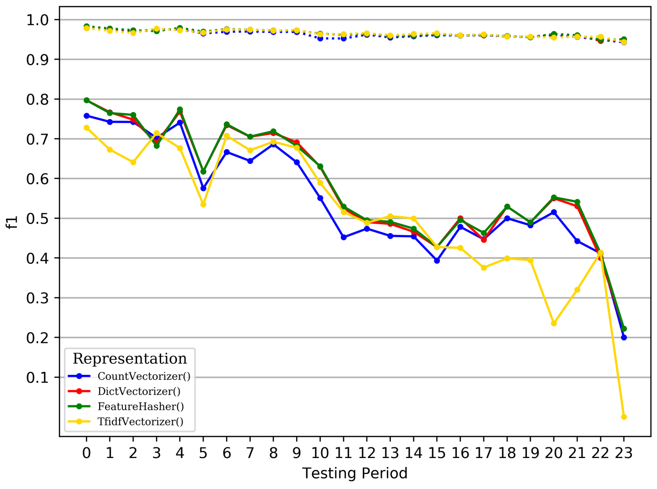
Of the remaining three we found that both FeatureHasher and DictVectorizer generally give very consistent or even identical outcomes, with the two lines in all three graphs overlapping almost perfectly and AUT scores almost identical. Although not something we reviewed in our experiment, documentation [46] suggests that FeatureHasher is a lower memory alternative to DictVectorizer, potentially making it a better choice for training in resource-constrained environments given that the quality of the classification is almost identical between the two.

CountVectorizer was able to outperform the DictVectorizer in both BLADE and MaMaDroid datasets, but not Drebin. This may be to do with the type of dataset BLADE and MaMaDroid are, as compared with Drebin. The BLADE dataset in particular is made up of a defined set of tokens, potentially explaining why a vectorization method that counts occurrences of tokens would be particularly effective as there are only a set number of constants that can ever appear. Drebin is made up of a much larger number of potential tokens, so the counts made by the vectorizer will generally be lower and potentially less useful.

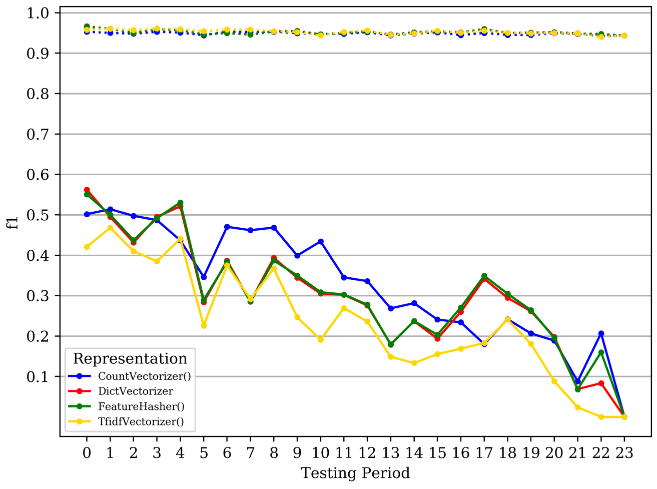
The BLADE dataset has the largest amount of variability, something already seen in the previous experiment, as can be seen in both the AUT scores and in figure 4.14c. This is most likely down again to a reduced dataset size. For Drebin and MaMaDroid, there is not a huge amount of variation in the lines shown in figures 4.14a and 4.14b. This would suggest that our choice of classifier does not have a particularly large effect on concept drift over time. Each vectorizer creates a decline in quality that is very similar. This was a rather surprising result, as we were expecting to see greater variability in results with the idea that the way in which data was represented would make some impact on time decay by changing the way the classifier fit the vectorized feature sets. In reality what this experiment has proved is that vectorization does not play a part in time decay, but instead can be optimised to maximise output F1 scores. As noted previously, the inconsistent goodware (dotted) lines in figure 4.14c are due to the particularly small dataset BLADE provides.

In all three datasets, the CountVectorizer increases the training time, as seen in figure 4.16, considerably. FeatureHasher in both Drebin and MaMaDroid is slower than DictVectorizer, but this result is reversed in BLADE, although in general, the training times are similar. TfidfVectorizer trains incredibly quickly compared to the others, again suggesting this method's lack of suitability for the task. Although variable, this suggests that for a resource-constrained environment, the DictVectorizer may be ideal for both Drebin and MaMaDroid, giving reasonable F1 scores and concept drift resilience while training much faster. Given BLADE's reduced dataset size is it difficult to draw any conclusions from these time results.

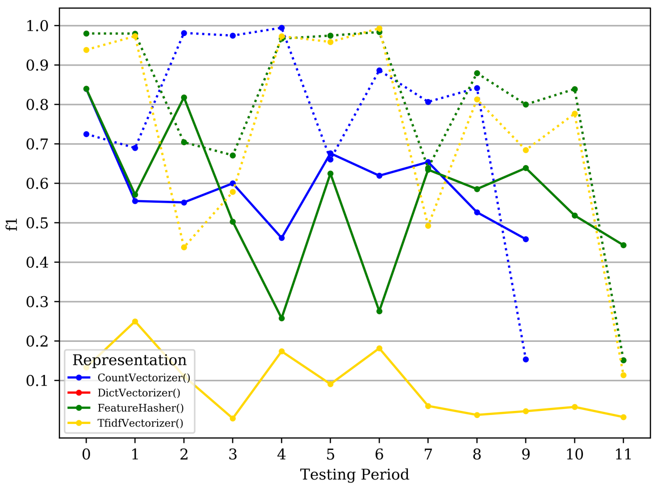
To conclude this section, with the exception of BLADE, the four tested classifiers give reasonably consistent output with the exception of the TfidfVectorizer. The TfidfVectorizer may simply be unsuited to this kind of classification task with the datasets we provide. Further review may be required to rule out any kind of methodological error, but the results are not extreme enough for Drebin and MaMaDroid to indicate something disastrously wrong. The choice of vectorizer does have some impact on the effect of concept drift but from this data, it is difficult to build any clear patterns of behaviour. These results would suggest that a representation approach could be picked in many cases by just considering resultant F1 scores.



(a) Drebin



(b) MaMaDroid



(c) BLADE

Figure 4.14: F1 results for experiment 3, using LinearSVC as the classifier. (mw = solid line, gw = dotted line)

BLADE	**AUT(F1, 24)**
CountVectorizer()	0.588
DictVectorizer()	0.552
FeatureHasher()	0.552
TfidfVectorizer()	0.09
Drebin	**AUT(F1, 24)**
CountVectorizer()	0.551
DictVectorizer()	0.583
FeatureHasher()	0.586
TfidfVectorizer()	0.519
MaMaDroid	**AUT(F1, 24)**
CountVectorizer()	0.33
DictVectorizer	0.301
FeatureHasher()	0.307
TfidfVectorizer()	0.236

Figure 4.15: AUT results for experiment 3

BLADE	**Time (s)**
CountVectorizer()	74.95
DictVectorizer()	21.2
FeatureHasher()	17.6
TfidfVectorizer()	2.87
Average	29.155
Drebin	**Time (s)**
CountVectorizer()	2725.28
DictVectorizer()	202.71
FeatureHasher()	245.1
TfidfVectorizer()	26.63
Average	799.93
MaMaDroid	**Time (s)**
CountVectorizer()	5449.49
DictVectorizer()	275.44
FeatureHasher()	648.34
TfidfVectorizer()	45.71
Average	1604.745

Figure 4.16: Train times for experiment 3

4.4 Dataset size

This is somewhat of a meta experiment, analysing how best to analyse concept drift. As mentioned in section 2.4, one recurring issue we had during this project was working with projects with datasets of insufficient size or quality. Effectively analysing concept drift requires a large number of timestamped samples spread over a sufficient period of time to produce useful results. Our datasets for Drebin [3] and MaMaDroid [4] are very large, with 129728 features in each set, whereas our BLADE [5] set was only 8948 elements and our DroidAPIMiner [19] set, that was discarded from our experiments for being too small, had 1517 elements. All this leads to the question of what the limits are for acceptable dataset size when testing time decay. The scope of this experiment is limited to just the size of the set, with no focus on the way dates are spread out within each dataset. We are making a reasonable assumption that our Drebin and MaMaDroid datasets are of sufficiently high quality that we can solely focus on the size of the set. Future experiments could review how the spread of dates within the dataset affects performance.

4.4.1 Method

In order to focus on just dataset size, it was important to keep the ratios of malware to goodware and the numbers of APKs in each month period consistent. As such, we must perform pre-processing on our datasets to eliminate elements consistently. We chose to take percentage slices of the dataset starting from 5% and working up to 100%. This reduces both the size of the training and testing set uniformly. We can then run the remaining dataset in our TESSERACT testing environment with default parameters to compare our results.

4.4.2 Experiment

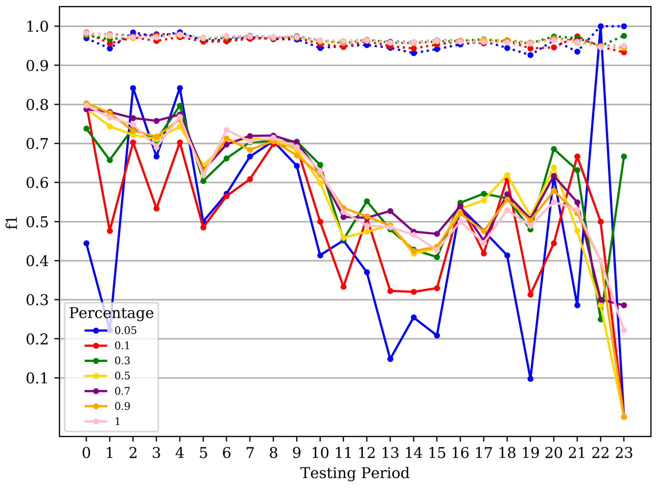
Pre-processing was implemented using python dictionaries to split the dataset up by malware/goodware, and then by month. So we would have a separate list of APK feature sets for each combination of month and good/malware. E.g. 2013/06-goodware. We can then iterate through each of these lists and append the correct percentage of items back into the main list of APK feature sets. To find the correct number of items we multiply the length of each list by the percentage (x) and then take the first x items from each list. The code used to achieve this is available in Appendix A, figure A.1. Using these partial datasets, we can iteratively run our TESSSERACT testing environment in a similar way to experiments 2 and 3, altering the dataset used in each iteration.

4.4.3 Results

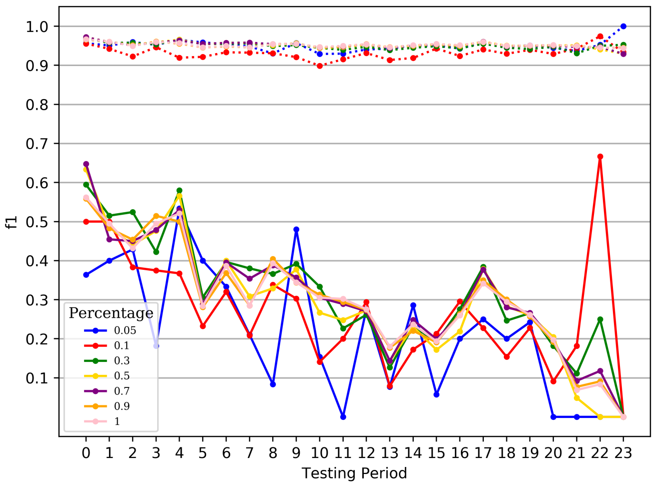
Figures 4.17a, 4.17b, and 4.17c give the results of time-aware analysis of Drebin, MaMaDroid, and BLADE respectively at dataset percentage levels ranging from 100% down to 5%. Figure 4.18 gives the AUT results and figure 4.19 gives the number of APKs represented in each dataset split. What is notable from these graphs, in particular figures 4.17a and 4.17b is that we see less of an obvious decline in the quality of the classifier and more of an increase in variance. The results we see at the lower percentage levels, for instance at 30 & 50 percent, we see larger deviations from the results from the full dataset. This effect only becomes more dramatic at the lower percentages. Besides this, the trend is generally still continued and although we see a general decline in AUT score, it does not collapse entirely. By restricting the dataset uniformly, it is understandable why we see this increase in variance. Smaller test set sizes, coupled with a smaller train set size, will produce testing periods that by chance score very high or very low depending on what samples happen to be left in each testing set for each testing period. The smaller the set in each testing period, the greater impact random chance will have on the final F1 score. These results do not just reflect a failure on the part of the classifier to train on a reduced dataset size, but also a failure on the test sets to give useful results in each testing period. It is only in BLADE where the dataset gets extremely small do we a total collapse in classifier quality.

As BLADE starts with a much smaller dataset size, it gives us an insight into the effect of decreasing the dataset size to an even greater extent. Below a few thousand elements we finally see time decay take a much more extreme effect. In particular, the 30% lines show the most significant drop off in score of any plot seen so far. Of course, once we get much lower we see the complete collapse of F1 throughout the entire time period.

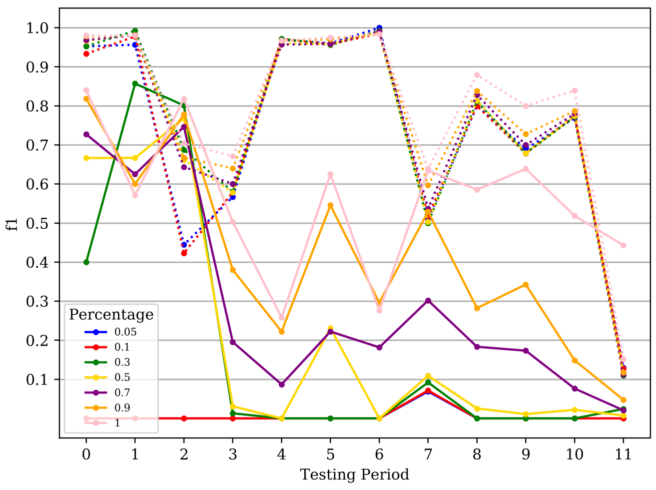
These results give us some clear conclusions. Dataset size, disregarding the distribution of malware and benign applications in the dataset, is correlated with increased volatility in results, even if general trends are still somewhat followed. It is difficult to give a concrete figure of a “cut-off” point for dataset size, as tolerance to this variability will change from implementation to implementation. In general, both Drebin and MaMaDroid saw a significant spike in variability at 10% of the dataset, so around 13000 items. We could therefore suggest that future implementations should aim to have at least double that number to have results with somewhat reasonable variance.



(a) Drebin

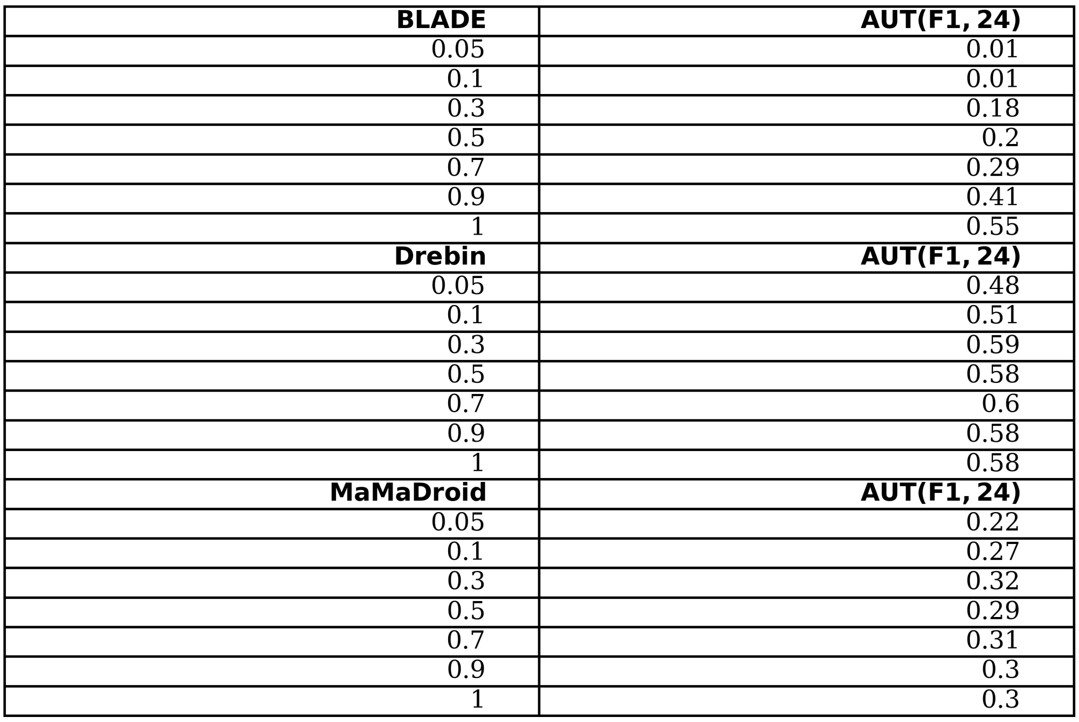


(b) MaMaDroid



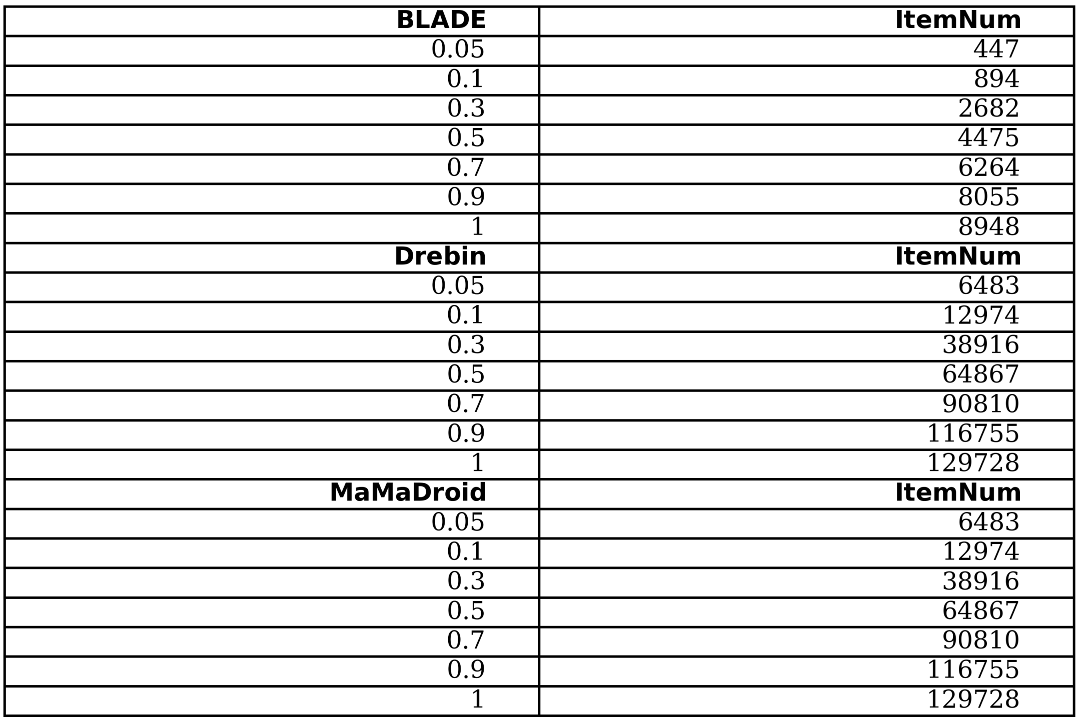
(c) BLADE

Figure 4.17: F1 results for experiment 4



BLADE	**AUT(F1, 24)**
0.05	0.01
0.1	0.01
0.3	0.18
0.5	0.2
0.7	0.29
0.9	0.41
1	0.55
Drebin	**AUT(F1, 24)**
0.05	0.48
0.1	0.51
0.3	0.59
0.5	0.58
0.7	0.6
0.9	0.58
1	0.58
MaMaDroid	**AUT(F1, 24)**
0.05	0.22
0.1	0.27
0.3	0.32
0.5	0.29
0.7	0.31
0.9	0.3
1	0.3

Figure 4.18: AUT results for experiment 4



BLADE	ItemNum
0.05	447
0.1	894
0.3	2682
0.5	4475
0.7	6264
0.9	8055
1	8948
Drebin	**ItemNum**
0.05	6483
0.1	12974
0.3	38916
0.5	64867
0.7	90810
0.9	116755
1	129728
MaMaDroid	**ItemNum**
0.05	6483
0.1	12974
0.3	38916
0.5	64867
0.7	90810
0.9	116755
1	129728

Figure 4.19: Number of APKs in each dataset

Chapter 5

Conclusion

5.1 Summary of Achievements

This project aimed to explore the Android malware classification pipeline in the context of concept drift. By exploring different program analysis techniques, representations, and dataset sizes we were able to produce novel insights into how elements of the classification pipeline and concept drift are related. In experiment 1, we compared and contrasted elements of the Drebin [3] dataset to consider if metadata-based features were more resilient to concept drift than ones derived from the bytecode. We discovered that these two sets were not dissimilar enough to make such a claim, and instead discovered that the size of the feature within the dataset seems to have a much more significant effect on concept drift resilience.

In experiment 2, we compared and contrasted various machine learning based classifiers and discovered a significant variance in the results. We found that the effect of the classification method on concept drift is variable depending on the feature set used. On some feature sets, for instance, for Drebin, we see classifiers that perform well in the earlier testing periods before declining more rapidly later on and other classifiers that remain much more consistent throughout. We also found that there is no clear best classifier for any of the three feature sets tested. Our classification methods were also very variable in time to train, which could impact the decision of which classifier to choose in very resource-constrained environments. In contrast to expectation, we found that the RandomForestClassifier was one of the worst methods of classification. We had previously expected the RandomForestClassifier to produce the best results given its complexity compared to other methods.

In experiment 3, we observe that there is some variation in results returned by our four tested vectorization approaches, however, there is generally not a huge disparity between the three approaches, especially in the context of concept drift. The vectorization approach chosen for any particular pipeline will most likely just come down to maximising performance rather than being concerned about the implications of concept drift.

Finally, experiment 4 did produce some interesting insights, drawing a correlation between dataset size and variance of classifier quality over time, with larger dataset sizes producing more stable results. What constitutes a dataset that is considered too small will depend on the particular use case.

The kind of experimentation carried out in this report can act as a bedrock for future work. We have succeeded in giving insight into the implications of program analysis abstractions to concept drift by varying parts of the classification pipeline in experiments that produce findings that improve the base of knowledge available in the malware classification research space.

5.2 Critical Analysis and Future Work

The experiments carried out in this paper are by no means definitive, and there is plenty of room for further exploration. Different combinations of representation, classification, dataset size, and feature set may produce different and interesting results. In particular, it would be interesting to rerun some of these experiments with MaMaDroid using a well-tuned RandomForestClassifier. Such an approach would require a sufficiently powerful machine, but it would be interesting to see if any of the results for MaMaDroid seen in Experiment 3 or 4 are impacted. It would also be interesting to expand upon the reduced dataset size experiment, looking at how manipulating the ratios between goodware and malware, as well as restricting just the size of the training set rather than the whole dataset affects the results. One of the key limitations of this project was the lack of datasets of significantly high quality and size to work with, and future work with access to such datasets may be able to yield different results. Initially, this project was going to look deeper into the effect of how different features abstracted from source APKs can affect concept drift, but this was difficult due to the aforementioned issues with dataset availability. Future work with the time and resources to rebuild unavailable datasets from cutting edge classification approaches could aim to fill this gap.

Of particular interest at the beginning of this project was a comparison between feature sets with specifically selected features, such as Drebin, versus an approach that takes a less focused approach such as BLADE where all occurrences of a more general feature (such as opcodes or API-flow patterns) are taken as part of the feature set. The comparison between “expert-selected” and “dumb” feature sets was not something that could be fully explored in this paper as the data was simply not available. A larger scale project able to re-create some classifiers from each of these two categories and build sufficiently large feature sets could produce some interesting insights into the effect of concept drift. Our hypothesis remains that “dumb” feature sets would be less resilient to concept drift as they could be more prone to over-training by the classifier. A carefully selected set of features that remain similar over time could produce results, that while may not produce the best accuracy and precision scores, could be the most resilient to concept drift.

Other avenues of exploration could include reviewing classification over longer time periods than 12 or 24 months. Combining various approaches to the classification pipeline with known concept drift mitigation strategies such as Trancend [18] could also yield novel results.

Chapter 6

Bibliography

1. A. Cranz, “There are over 3 billion active Android devices,” The Verge, 18-May-2021. [Online]. Available: https://www.theverge.com/2021/5/18/22440813/android-devices-active-number-smartphones-google-2021. [Accessed: 26-Apr-2022].
2. B. Toulas, “2021 mobile security: Android more vulnerabilities, IOS more Zero-days,” BleepingComputer, 14-Mar-2022. [Online]. Available: https://www.bleepingcomputer.com/news/security/2021-mobile-security-android-more-vulnerabilities-ios-more-zero-days/. [Accessed: 26-Apr-2022].
3. D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, “Drebin: Effective and explainable detection of Android malware in your pocket,” Proceedings 2014 Network and Distributed System Security Symposium, Feb. 2014.
4. E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini, “MaMaDroid: Detecting Android malware by building Markov Chains of behavioral models,” Proceedings 2017 Network and Distributed System Security Symposium, 2017.
5. V. Sihag, M. Vardhan, and P. Singh, “BLADE: Robust malware detection against obfuscation in Android,” Forensic Science International: Digital Investigation, Sep. 2021.
6. M. Zhang, Y. Duan, H. Yin, and Z. Zhao, “Semantics-aware Android malware classification using weighted contextual API dependency graphs,” Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, 2014.
7. F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, “TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time,” 28th USENIX Security Symposium, 2019.
8. K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, “AndroZoo: Collecting Millions of Android Apps for the Research Community,” Proceedings of the 13th International Conference on Mining Software Repositories, 2016, pp. 468–471. doi: 10.1145/2901739.2903508.
9. H. Pang, K. Moore, and A. Padmanbha, “Feature vector,” Brillaint.org. [Online]. Available: https://brilliant.org/wiki/feature-vector/. [Accessed: 26-Apr-2022].
10. “Sklearn.model_selection.train_test_split,” scikit-learn. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html. [Accessed:

26-Apr-2022].

11. *Citation number unused*
12. O. Suciu, S. E. Coull, and J. Johns, "Exploring adversarial examples in malware detection," 2019 IEEE Security and Privacy Workshops (SPW), 2019.
13. "TESSERACT," S2Lab. [Online]. Available: https://s2lab.cs.ucl.ac.uk/projects/tesseract/. [Accessed: 26-Apr-2022].
14. D. Hu, Z. Ma, X. Zhang, P. Li, D. Ye, and B. Ling, "The concept drift problem in Android malware detection and its solution," Security and Communication Networks, Sep. 2017.
15. I. Žliobaitė, M. Pechenizkiy, and J. Gama, "An overview of concept drift applications," Studies in Big Data, pp. 91–114, 2015.
16. J. Brownlee, "A gentle introduction to concept drift in machine learning," Machine Learning Mastery, 10-Dec-2020. [Online]. Available: https://machinelearningmastery.com/gentle-introduction-concept-drift-machine-learning/. [Accessed: 26-Apr-2022].
17. A. Tsymbal, "The Problem of Concept Drift: Definitions and Related Work," May 2004.
18. R. Jordaney et al., "Transcend: Detecting Concept Drift in Malware Classification Models," 26th USENIX Security Symposium, 2017.
19. Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in Android," International Conference on Security and Privacy in Communication Systems, pp. 86–103, Sep. 2013.
20. O. Harrison, "Machine learning basics with the K-nearest neighbors algorithm," Towards Data Science, 10-Sep-2018. [Online]. Available: https://towardsdatascience.com/machine-learning-basics-with-the-k-nearest-neighbors-algorithm-6a6e71d01761. [Accessed: 26-Apr-2022].
21. R. Gandhi, "Support Vector Machine - introduction to machine learning algorithms," Towards Data Science, 07-Jun-2018. [Online]. Available: https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47. [Accessed: 26-Apr-2022].
22. Y. Sakkaf, "Decision trees for classification: Id3 Algorithm explained," Towards Data Science, 31-Mar-2020. [Online]. Available: https://towardsdatascience.com/decision-trees-for-classification-id3-algorithm-explained-89df76e72df1. [Accessed: 26-Apr-2022].
23. S. Saha, "What is the C4.5 algorithm and how does it work?," Towards Data Science, 20-Aug-2018. [Online]. Available: https://towardsdatascience.com/what-is-the-c4-5-algorithm-and-how-does-it-work-2b971a9e7db0. [Accessed: 26-Apr-2022].
24. B. Kang, S. Y. Yerima, K. Mclaughlin, and S. Sezer, "N-opcode analysis for Android malware classification and Categorization," 2016 International Conference On Cyber Security And Protection Of Digital Services (Cyber Security), Jun. 2016.
25. J. Lynch, "The dex file format," Bugsnag blog, 04-Jan-2018. [Online]. Available: https://www.bugsnag.com/blog/dex-and-d8. [Accessed: 26-Apr-2022].
26. T. Yiu, "Understanding random forest," Towards Data Science, 12-Jun-2019. [Online]. Available: https://towardsdatascience.com/understanding-random-forest-58381e0602d2. [Accessed: 26-Apr-2022].

27. J. Platt, “Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines,” Advances in Kernel Methods-Support Vector Learning, vol. 208, Jul. 1998.

28. MLDroid, “MLDroid/Drebin: Drebin - NDSS 2014 re-implementation,” GitHub. [Online]. Available: https://github.com/MLDroid/drebin. [Accessed: 26-Apr-2022].

29. A. Mahindru and A. L. Sangal, “MLDroid—framework for Android malware detection using Machine Learning Techniques,” Neural Computing and Applications, vol. 33, no. 10, pp. 5183–5240, Sep. 2020.

30. “Sklearn.feature_extraction.DictVectorizer,” scikit-learn. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.DictVectorizer.html. [Accessed: 26-Apr-2022].

31. “Sklearn.feature_extraction.text.TfidfVectorizer,” scikit-learn. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html. [Accessed: 26-Apr-2022].

32. F. Pedregosa et al., “Scikit-learn: Machine Learning in Python,” Journal of Machine Learning Research, vol. 12, pp. 2825–2830, 2011.

33. B. Stecanella, “Understanding TF-ID: A simple introduction,” MonkeyLearn Blog, 10-May-2019. [Online]. Available: https://monkeylearn.com/blog/what-is-tf-idf/. [Accessed: 26-Apr-2022].

34. ChenJunHero, “Chenjunhero/Droidapiminer: Mining api-level features for robust malware detection in Android,” GitHub. [Online]. Available: https://github.com/ChenJunHero/DroidAPIMiner. [Accessed: 26-Apr-2022].

35. *Citation number unused*

36. *Citation number unused*

37. Y. Zhou and X. Jiang, “Dissecting android malware: Characterization and evolution,” 33rd IEEE Symposium on Security and Privacy, May 2012.

38. Tba, “Android Malware Dataset,” Website unavailable . [Online]. Available: http://amd.arguslab.org/. [Accessed: 26-Apr-2022].

39. “Android Praguard Dataset,” Android PRAGuard Dataset — PRA Lab. [Online]. Available: http://pralab.diee.unica.it/en/AndroidPRAGuardDataset. [Accessed: 26-Apr-2022].

40. “Sklearn.model_selection.Stratifiedkfold,” scikit-learn. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html. [Accessed: 26-Apr-2022].

41. “Stratified K fold cross validation,” GeeksforGeeks, 27-Apr-2022. [Online]. Available: https://www.geeksforgeeks.org/stratified-k-fold-cross-validation/. [Accessed: 26-Apr-2022].

42. V. Sihag, “Blade android malware dataset,” Kaggle, 2021. [Online]. Available: https://www.kaggle.com/datasets/vikassihag/blade-dataset. [Accessed: 26-Apr-2022].

43. “Andro-AutoPsy : Anti-malware system based on similarity matching of malware and malware creator-centric information,” HCRL. [Online]. Available: https://ocslab.hksecurity.net/andro-autopsy. [Accessed: 26-Apr-2022].

44. *Citation number unused*

45. “Sklearn.svm.LinearSVC,” scikit-learn. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html. [Accessed: 26-Apr-2022].

46. “Sklearn.feature_extraction.Featurehasher,” scikit-learn. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.FeatureHasher.html. [Accessed: 26-Apr-2022].

47. “Sklearn.feature_extraction.text.CountVectorizer,” scikit-learn. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html. [Accessed: 26-Apr-2022].

48. “Sklearn.ensemble.RandomForestClassifier,” scikit-learn. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html. [Accessed: 26-Apr-2022].

49. “Sklearn.linear_model.SGDClassifier,” scikit-learn. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html. [Accessed: 26-Apr-2022].

50. A. Srinivasan, “Stochastic gradient descent-clearly explained!!,” Towards Data Science, 07-Sep-2019. [Online]. Available: https://towardsdatascience.com/stochastic-gradient-descent-clearly-explained-53d239905d31. [Accessed: 26-Apr-2022].

51. “Sklearn.neighbors.KNeighborsClassifier,” scikit. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html. [Accessed: 26-Apr-2022].

52. “1.6. Nearest Neighbors,” scikit-learn. [Online]. Available: https://scikit-learn.org/stable/modules/neighbors.html#classification. [Accessed: 26-Apr-2022].

53. O. Harrison, “Machine learning basics with the K-nearest neighbors algorithm,” Towards Data Science, 10-Sep-2018. [Online]. Available: https://towardsdatascience.com/machine-learning-basics-with-the-k-nearest-neighbors-algorithm-6a6e71d01761. [Accessed: 26-Apr-2022].

54. “Sklearn.tree.DecisionTreeClassifier,” scikit-learn. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html. [Accessed: 26-Apr-2022].

55. C. Bento, “Decision tree classifier explained in real-life: Picking a vacation destination,” Towards Data Science, 28-Jun-2021. [Online]. Available: https://towardsdatascience.com/decision-tree-classifier-explained-in-real-life-picking-a-vacation-destination-6226b2b60575. [Accessed: 26-Apr-2022].

56. “1.10. Decision Trees,” scikit-learn. [Online]. Available: https://scikit-learn.org/stable/modules/tree.html#tree. [Accessed: 26-Apr-2022].

57. T. Yiu, “Understanding Random Forest,” Towards Data Science, 12-Jun-2019. [Online]. Available: https://towardsdatascience.com/understanding-random-forest-58381e0602d2. [Accessed: 26-Apr-2022].

58. R. Pupale, “Support Vector Machines(SVM) - An Overview,” Towards Data Science, 16-Jun-2018. [Online]. Available: https://towardsdatascience.com/https-medium-com-pupalerushikesh-svm-f4b42800e989. [Accessed: 26-Apr-2022].

59. “Support Vector Machine in Machine Learning,” GeeksforGeeks, 22-Dec-2020. [Online]. Available: https://www.geeksforgeeks.org/support-vector-machine-in-machine-learning/. [Accessed: 26-Apr-2022].

60. “Sklearn.metrics.f1_score,” scikit-learn. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html. [Accessed: 26-Apr-2022].

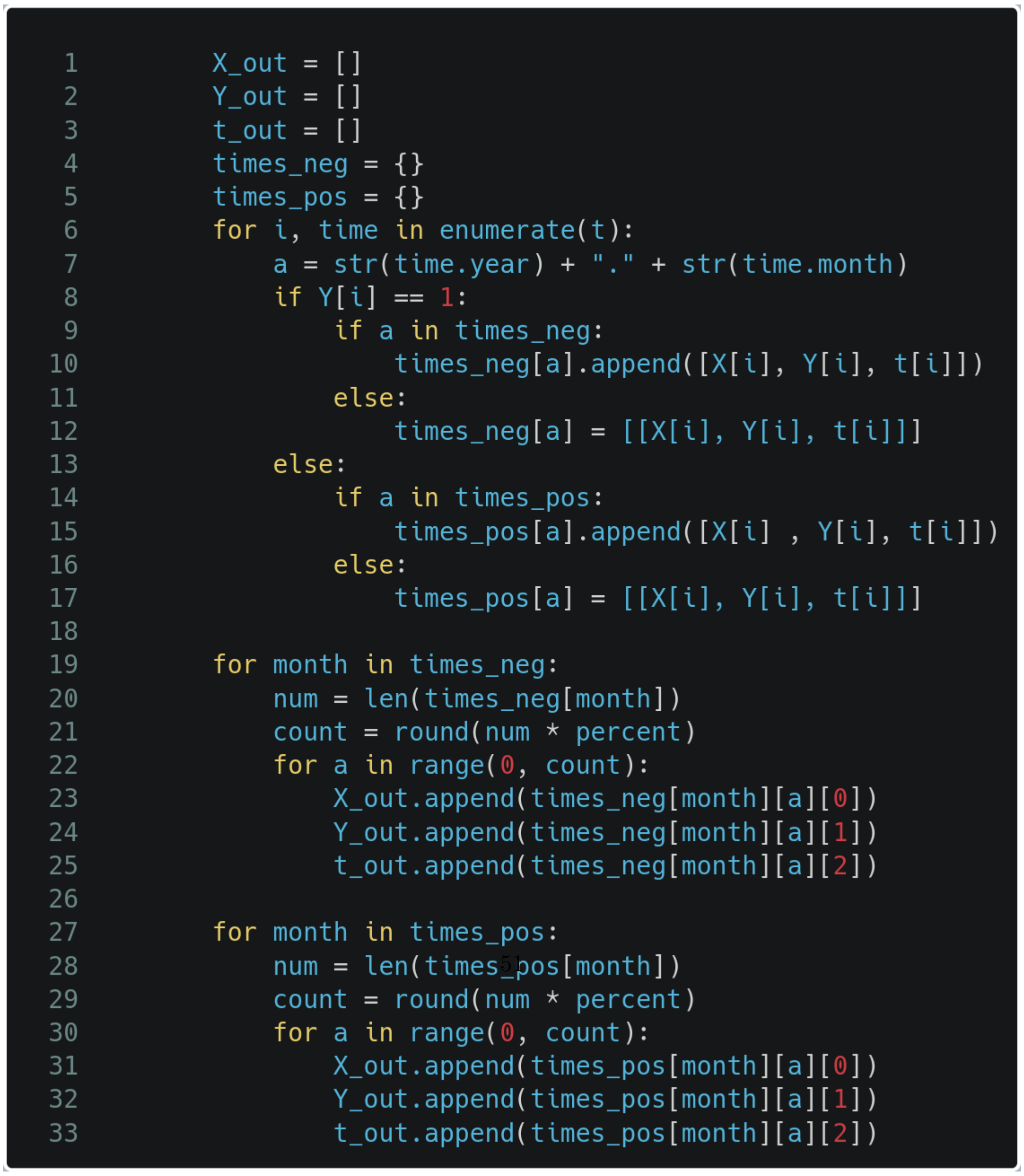
61. J. Korstanje, “The F1 score,” Towards Data Science, 31-Aug-2021. [Online]. Available: https://towardsdatascience.com/the-f1-score-bec2bbc38aa6. [Accessed: 26-Apr-2022].

62. D. Rafter, “Android vs. IOS: Which is better for security?,” Norton, 22-Mar-2022. [Online]. Available: https://us.norton.com/internetsecurity-mobile-android-vs-ios-which-is-more-secure.html. [Accessed: 26-Apr-2022].

63. C. Khanna, “What and why behind fit_transform() vs transform() in scikit-learn!,” Towards Data Science, 26-Aug-2020. [Online]. Available: https://towardsdatascience.com/what-and-why-behind-fit-transform-vs-transform-in-scikit-learn-78f915cf96fe. [Accessed: 26-Apr-2022].

Appendix A

Code snippets



```
        X_out = []
        Y_out = []
        t_out = []
        times_neg = {}
        times_pos = {}
        for i, time in enumerate(t):
            a = str(time.year) + "." + str(time.month)
            if Y[i] == 1:
                if a in times_neg:
                    times_neg[a].append([X[i], Y[i], t[i]])
                else:
                    times_neg[a] = [[X[i], Y[i], t[i]]]
            else:
                if a in times_pos:
                    times_pos[a].append([X[i] , Y[i], t[i]])
                else:
                    times_pos[a] = [[X[i], Y[i], t[i]]]

        for month in times_neg:
            num = len(times_neg[month])
            count = round(num * percent)
            for a in range(0, count):
                X_out.append(times_neg[month][a][0])
                Y_out.append(times_neg[month][a][1])
                t_out.append(times_neg[month][a][2])

        for month in times_pos:
            num = len(times_pos[month])
            count = round(num * percent)
            for a in range(0, count):
                X_out.append(times_pos[month][a][0])
                Y_out.append(times_pos[month][a][1])
                t_out.append(times_pos[month][a][2])
```

Figure A.1: Code to split dataset in experiment 4

```python
import numpy as np
from masters.load_features import *
from tesseract import evaluation, temporal, metrics, viz
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.feature_extraction.text import CountVectorizer
import time


def parse_blade(X, Y, t):
    times_dict = {'2011.1': {0}, '2011.6': {0}, '2011.11': {0}, '2012.4': {0}, '2011.12': {0}, '2011.7': {0},
                  '2013.5': {0, 1}, '2012.10': {0, 1}, '2011.5': {0}, '2011.2': {0}, '2011.4': {0}, '2012.6': {0},
                  '2013.9': {0, 1}, '2011.9': {0}, '2012.7': {0, 1}, '2012.9': {0, 1}, '2011.3': {0}, '2013.4': {0, 1},
                  '2013.10': {0, 1}, '2012.2': {0}, '2012.5': {0}, '2011.8': {0}, '2013.7': {0, 1}, '2012.11': {0, 1},
                  '2012.3': {0, 1}, '2013.3': {0, 1}, '2013.2': {0, 1}, '2012.12': {0, 1}, '2011.10': {0},
                  '2013.11': {0, 1}, '2013.8': {0, 1}, '2013.6': {0, 1}, '2012.1': {0}, '2012.8': {0, 1},
                  '2013.1': {0, 1}, '2013.12': {0, 1}, '2014.2': {1}, '2014.5': {1}, '2014.6': {1}, '2014.3': {1},
                  '2014.4': {1}, '2014.1': {1}}

    X1 = X
    Y1 = Y
    t1 = t
    X = []
    Y = []
    t = []
    for i, item in enumerate(X1):  # 2010
        if len(item) > 0 and t1[i].year > 2010 and len(times_dict[str(t1[i].year) + "." + str(t1[i].month)]) == 2:
            X.append(item)
            Y.append(Y1[i])
            t.append(t1[i])

    print(len(X))
    times = {}
    for i, timey in enumerate(t):
        a = str(timey.year) + "." + str(timey.month)
        if a in times:
            times[a].add(Y[i])
        else:
            times[a] = set()
            times[a].add(Y[i])

    return X, Y, t


def main(load_src, vec, clf, pltname, train_size):
    X, Y, t = load_features(load_src)

    # Handle blade special case formatting
    if load_src == "../blade/AA/apg-autopsy":
        X, Y, t = parse_blade(X, Y, t)

    # Convert datasets to strings for TfidfVectorizer and CountVetorizer
    if str(vec) == str(TfidfVectorizer()) or str(vec) == str(CountVectorizer()):
        X_all = []
        for item in X:
            X_all.append(str(item))
        X = X_all

    x = vec.fit_transform(X)
    y = np.asarray(Y)
    tv = np.asarray(t)

    # Partition dataset
    splits = temporal.time_aware_train_test_split(
        x, y, tv, train_size=train_size, test_size=1, granularity='month')


    start = time.time()

    # Train set and produce results
    results = evaluation.fit_predict_update(clf, *splits)
    time_taken = time.time() - start

    # View results
    metrics.print_metrics(results)

    #Save decay graph
    plt = viz.plot_decay(results)
    plt.savefig("../figs/" + pltname)

    #Return data and time taken
    return results, time_taken
```

Figure A.2: Implementation of TESSERACT testing environment