LEARNING API MAPPINGS FOR PROGRAMMING PLATFORMS

By

YOGESH PADMANABAN

A thesis submitted to the

Graduate School-New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Computer Science

written under the direction of

Dr. Vinod Ganapathy

and approved by

____

____

____

New Brunswick, New Jersey

January, 2013

ABSTRACT OF THE THESIS

Learning API mappings for programming platforms.

by YOGESH PADMANABAN

Thesis Director: Dr. Vinod Ganapathy

Software developers often need to port applications written for a source platform to a target platform. One of the key tasks here is to find matching API calls in the target platform for the given API calls in the source platform. This task involves exhaustive reading in target platform API (Application Programming Interface) documentation to identify API methods corresponding to the given API methods of the source platform. We introduce an approach to the problem of inferring mapping between the APIs of a source and target platforms. It is constructed based on independently developed applications on source and target platforms performing similar functionality. We observe that in building these applications, developers exercised knowledge of the corresponding APIs. We develop two dynamic analysis techniques to systematically harvest this knowledge and infer likely mappings between the Graphical APIs of JavaME and Android Graphical platform.

Rosetta Mapper: A tool which provides a ranked list of target API methods or method sequences that likely map to each source API method or method sequences.

Rosetta Classifier: A supervised learning tool which classifies whether the given mapping between source API and target API is true or false using support vector machines.

Acknowledgements

I gratefully acknowledge the guidance and generous support of my supervisor, Professor Vinod Ganapathy, throughout my masters.

I would like to thank my colleague Mrs. Amruta Gokhale for her consistent support and motivation. I would like to thank my fellow researchers at DISCOLAB. Discussions and meetings we used to have proved really helpful for my research and thesis.

I would like to thank Prof. Badri Nath and Prof. Alex Borgida for supervising my thesis defense.

Dedication

I would like to dedicate my work to my parents Dr.R.Padmanaban (Vet) and Dr.P.Kousalya (MBBS) for their love and support.

I also would like to dedicate my work to my brother Mr. Raja Ramanan for his motivation and support.

Table of Contents

List of Tables

Terms and Acronyms

API: Application Programming Interface is a protocol intended to be used as an interface by software components to communicate with each other.

API Mapping: It is a notion of expressing functional equivalence of API methods from the source platform to the target platform.

Tracing: The process of recording API methods executed by an application is called Tracing.

Trace files: The logs of API methods obtained by tracing the functionally similar applications run on the source and the target platforms.

SVM: Support Vector Machines.

List of Figures

Chapter 1

Introduction

1.1 Objective

Software developers often wish to make their applications available on a wide variety of programming platforms or would want to migrate their application to a new programming platform. The motives could be performance, more/richer features or business requirements. Hence, given a code base in the source platform, the programmer needs to port them to a target platform. The process of porting programming code from one system to another is called as Code Migration. Code migration is a tedious, error prone and highly labor intensive process. Consider an example: suppose that we wish to port a Java2 Platform Mobile Edition (JavaME)-based game to an Android-powered device. Among other tasks, we must modify the game to use Android's API [14] (the target platform) instead of JavaME's API [25] (the source platform). Unfortunately, the process of identifying the API methods in the target platform that implements the same functionality corresponding to that of a source platform API method is cumbersome. We must manually examine the SDKs of the source and target APIs to determine the right method (or sequence of methods) to use. One way to address this problem is to populate a database of mappings between the APIs of the source and target platforms. In this database, each source API method (or method sequence) is mapped to a target API method (or method sequence) that implements its functionality. Such a database would make the code migration simpler.

1.2 Background

Code Migration:

It is the porting of programming code from one system to another. The requirement might be portability, performance, and/or more features. Source code conversion is inherently a tedious, error-prone and labor intensive process. There are three distinct levels of code migration with increasing complexity, cost and risk. Simple migration involves the movement of one programming language to a newer version (version migration). A second, more complicated level of migration involves moving to a different programming language (language migration). Migrating to an entirely new platform or operating system is the most complex type of migration.

Version Migration:

Simple movement from one version of a language to a newer version for more efficiency or features. Here the basic structure of the programming constructs usually do not change. In many cases, the old code would actually work, but new routines or modularization can be improved by retooling the code to fit the nature of the new language. Example Lucene search engine 2.9 to 3.0, Java 1.5 to Java 5 Transition. Different JRE major versions may implement different versions of Unicode, which will change the way some parts of Lucene treat the text. Java 1.4 uses Unicode 3.0, while Java 5 uses Unicode 4.0. Thus the packages *SimpleAnalyzer, StopAnalyzer, LetterTokenizer, LowerCaseFilter,* may return different results, while package *StandardAnalyzer* will return the same results under Java5 as it did under Java 1.4.[36]

Language Migration:

This involves moving to completely new programming language. This type of code migration often requires that programmers learn an entirely new language, or new programmers be brought in to assist with the migration. At times a transcompiler or a source to source compiler can be used but still the automated translation has to go through manual verification. For example, earlier C++ was transcompiled to C with the *cfront* transcompiler by Bjarne Stroutstrup himself [38]. Another example is PHP transcompiled to C++ using HipHop at Facebook. [37]

Platform Migration:

The most complex code migration is migrating to an entirely new platform and/or operating system. This not only changes the programming language, but also the machine code behind the language. While most modern programming languages shield the programmer from lower level code, knowledge of the OS and how it operates is essential to producing code that is efficient and executes as expected. Example Building ORACLE database for the windows platform from the Linux platform involves almost creating entire new database from scratch. Porting applications from windows Lumia phone to IPhone of IOS platform would involve translation of functionalities from visual C++ to Objective C and the matching the capabilities of Windows phone with an IPhone.

Regardless of the type of code migration, the approach should be the same for code migration. The migration team or programmer should break the source code into each module, function and sub-routine into its purpose and construct the design for the source program. We map the functionalities and classes to the corresponding functions and APIs in the target platform. We modify the design for the new code. Generate the new code structure. We translate the Functional Logic using the mappings between the functionalities produced. We rewrite the architectural, paradigm, programming language differences that exists between the platforms. We then evaluate the conversion exhaustively using testing tools, manual verification and repeat the process to ensure quality.

1.3 J2ObjC

Consider the transcompiler tool J2ObjC (migration from Java to Objective C of IOS platform transcompiler developed by Google [39]. The conversion typically involves the following steps.

1. Rewriter: rewrites the Java code that doesn't have an Objective-C equivalent, such as static variables.

2. Autoboxer: adds code to box numeric primitive values and unbox numeric Wrapper classes. (int to Integer class)

3. IOS Type Converter: Converts types that are directly mapped from Java to Foundation Classes (Microsoft Foundation Class library [40]).

4. IOS Method Converter: maps method declarations and invocations to equivalent Foundation class methods.

5. Initialization Normalizer: moves initializer statements into constructors and class initialization methods.

6. Anonymous Class Converter: modifies anonymous classes to be inner classes, which includes fields for the final variables referenced by that class.

7. Inner Class converter: pulls inner classes out to be top-level classes in the same compilation unit.

8. Destructor generator: adds dealloc or finalize methods, depending on Garbage Collection options.

This thesis is all about step 4 above. Namely mapping API methods between two programming platforms. We consider the JavaME and Android platforms and focus on Graphics API to perform our experiments.

1.4 Problem in Hand

A developer of a gaming app, for instance, may wish to make his app available on smart phones and tablets manufactured by various vendors, on desktops. The key hurdle that he faces in doing so is to port his app to these software and hardware platforms. Why is porting software a difficult problem? Consider an example: suppose that we wish to port a Java2 Platform Mobile Edition (JavaME)-based game to an Android-powered device. Among other tasks, we must modify the game to use Android's API [14] (the target platform) instead of JavaME's API [25] (the source platform). Unfortunately, the process of identifying the API methods in the target platform that implement the same functionality corresponding to that of a source platform API method is cumbersome. We must manually examine the SDKs of the source and target APIs to determine the right method (or sequence of methods) to use. To add complexity, there could be multiple ways in which a source API method can be implemented using the target's API methods. For example, the $fillRect()$ method in JavaME's graphics API, which fills a specified rectangle with color, can be implemented using either one of these two sequences of methods in Android's graphics API: $setStyle(); drawRect()$ or as $moveTo(); lineTo();\ lineTo(); lineTo(); lineTo(); drawPath()$ (we have omitted class names and the parameters to these method calls). One way to address this problem is to populate a database of mappings between the APIs of the source and target platforms. In this database, each source API method (or method sequence) is mapped to a target API method (or method sequence) that implements its functionality. The database could contain multiple mappings (possibly ranked) for each source API method in cases where its functionality can be implemented in different ways by the target API. The mapping database significantly eases our task. We need only consider the mappings in this database to find suitable target API methods to replace a source API method, instead of painstakingly poring over the SDKs and their documentation. Such mapping databases do exist, but only for a few source/target API pairs (e.g., Android, iOS and

Symbian Qt to Windows 7 [5] and iOS to Qt [3]), and they are populated by domain experts well-versed in the source and target APIs.

1.5 Contribution

We present an approach to automate the creation of mapping databases for any given source or target APIs. To bootstrap our approach, we rely on the availability of a few similar application pairs on the source and target platform. A source platform application S and a target platform application$T$, possibly developed independently by different sets of programmers, constitute a similar application pair if they implement the same high-level functionality.

Our approach builds upon the observation that in implementing *S and T, their developers exercised knowledge about the APIs of the corresponding platforms*. We provide a systematic way to harvest this knowledge into a mapping database, which can then benefit other developers porting applications from the source to the target platform. Our approach works by recording traces of S and T executing similar tasks, structurally analyzing these traces, and extracting likely mappings using probabilistic inference.

In one method, each mapping output by our approach is associated with a probability, which indicates the likelihood of the mapping being true. The intuition is that the more evidence we see of a mapping, e.g., the same pair of API methods being used across many traces to implement the same functionality, the higher the likelihood of the mapping. The output of our approach is a ranked list of mappings inferred for each source API method.

In the second method, the idea is to train a classifier to classify whether a mapping of API method A of source platform is possible or not to an API method B of the target platform. This classifier is built upon heuristics collected from the trace files, (different from before). Classifier can also act like a validation method to verify whether the given mapping between source API A and target API B is true or false. As with any machine learning statistic tools the more traces and better heuristics, the better the classifier. We demonstrate our first approach by building a prototype tool called Rosetta Mapper to infer likely mappings between the JavaME graphics API [25] and the Android graphics API [14]. We chose JavaME and Android because both platforms use the same language (Java) for application development; however, it may also be possible to apply Rosetta Mapper to the case where the source and target APIs use different languages for application development. We evaluated Rosetta with a set of twenty-one independently-developed JavaME/ Android application pairs. Rosetta mapper was able to find at least one valid mapping within the top ten ranked results for 71% of the JavaME API methods observed in our traces. Further, for 42% of JavaME API methods, the top-ranked result was a valid mapping. We evaluated the Rosetta classifier with ten independently-developed JavaME/ Android application pairs. We obtained training data for 7000 mappings with manual verification. We fed the training data to the classifier and obtained the classification with 86% recall and 54% accuracy.

Chapter 2

Approach Overview

We present the workflow of our approach (Figure. 1), tailored to JavaME and Android as the source and target platforms, respectively. We only provide informal intuitions here, and defer the details to chapter IV. The workflow has four steps.

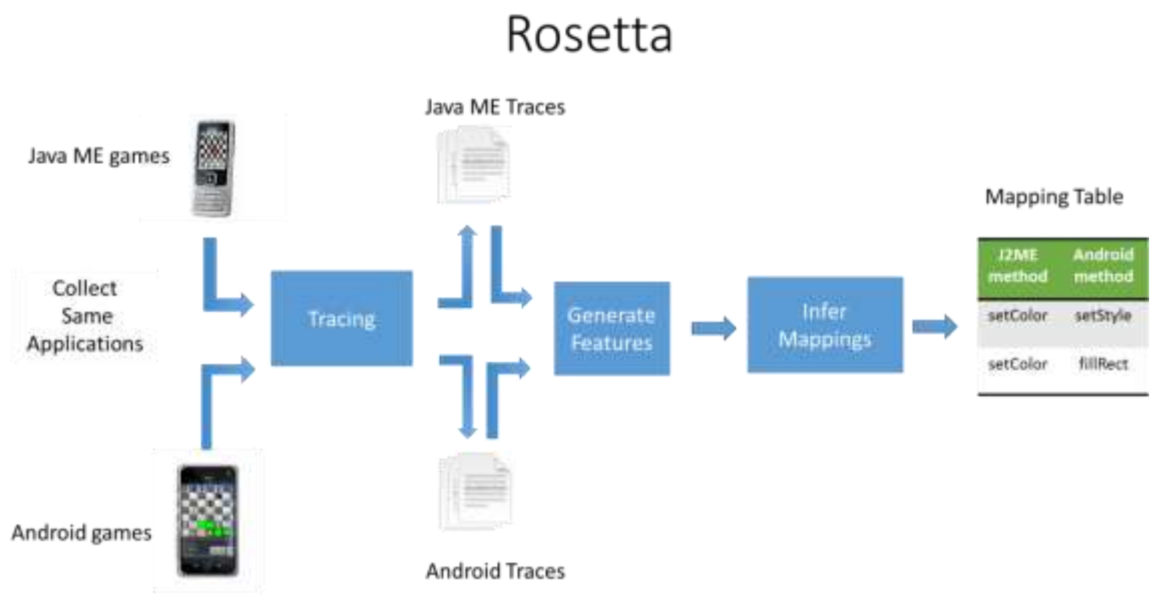


Figure 1. Rosetta

2.1 The Collection of *Similar* Application pairs

The first step is to gather a database of applications in both the source and target platform. For each source application in the database, we require a target application that implements the same high level functionality. We have focused on graphical APIs and hence the applications are simply games on these platforms. For example, if we have a TicTacToe game for JavaME, we should locate a TicTacToe game for Android that is as functionally and visually (GUI-wise) similar to the JavaME game as possible. Given the popularity of modern mobile platforms, and the desire of end-users to use similar applications across platforms, such games of same names are easy to come by. But to get visually similar games from android and j2me platform, downloading the proper apk/jar files were quite a challenge. Given that the games were independently developed for these two platforms, there were differences in functionality. For example, the Android game may offer more menu options or has ads or smoother play that are different from those of the JavaME game which are not sophisticated enough. We shall discuss how we overcome some of these irregularities in coming sections.

2.2 Execution and Collection of traces

In this step, we take each application pair, and execute them in similar ways, i.e., we provide inputs to exercise similar functionality and ultimately same output in these applications. Or more precisely, we give inputs to each of the application run so as to have the same output on the screen. This is the place where we ensure the equality of the execution of the application pairs. We have several executions per application pair. We choose the ones, where the output matched the most *visually*.

As we do so, we also log a trace of API calls invoked by the applications. This gives a trace pair, consisting of one trace each for the source and target applications. Fig. 3 presents a snippet from a trace pair that we gathered for TicTacToe games on the JavaME and Android platforms. They were collected by starting the game, waiting for the screen to display a grid of squares, and exiting. Since the traces in each pair were obtained by exercising similar functionality in the source and target applications, these traces must contain some API calls that can be mapped to each other. This is the key intuition underlying our approach. Of course, an application can be invoked in many ways, and in this step, we collect up to 5 visually similar trace pairs for each application pair. The output of this step is a database of functionally equivalent trace pairs (Trace_S , Trace_T) across all of the application pairs collected in Step 1. In addition to inherent differences in the platforms, these applications are created by independent developers and we had to struggle through several randomness of these application pairs, and their executions in order to obtain traces pairs of reasonable *visual* equivalence.

2.3 Trace Analysis and inference

In this step we analyze each trace pair to infer likely API mappings implied by the pair. The main idea behind the inference algorithm is that given two traces which perform a common task, the components of the traces namely the source and the target API calls can be mapped to each other if they exhibit similar values over a number of attributes. Our attributes are determined by the structure of the traces.

Mapping: A pair of source API and a target API, which would have potentially equal functionality with the source API. A mapping is true if their functionality is found to be similar. Otherwise it is a false mapping.

For every such mapping, we take a vector of values for these attributes and feed them to the inference engine. We cast the attribute vectors as inputs to a probabilistic inference algorithm (§ IV). The output of this step is a probability for each pair of source and target API calls $A()/a()$ that indicates the likelihood of $A()$ mapping to $a()$ (denoted here as A$()->$ $a()$). In addition to "one on one" source to target API mapping, we also try to derive "many to one" mappings. Here we had tried to accomplish 2 to 1 and 1 to 2 mappings. Our algorithm can also infer mappings between method sequences (e.g.,$A() \rightarrow a(); b()\ \ or\ A(); B() \rightarrow a()$).

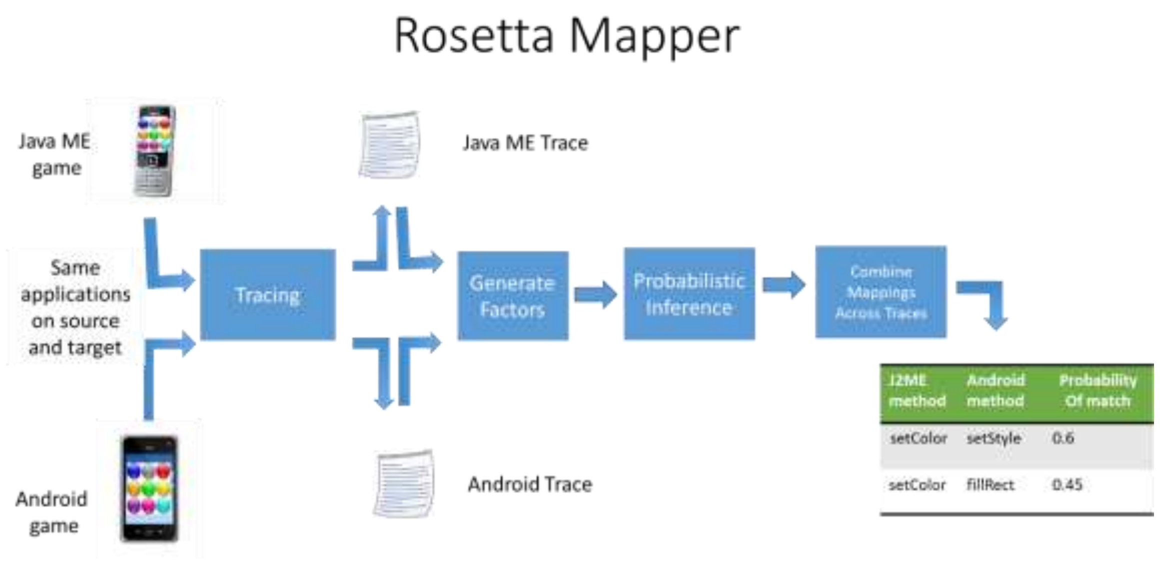


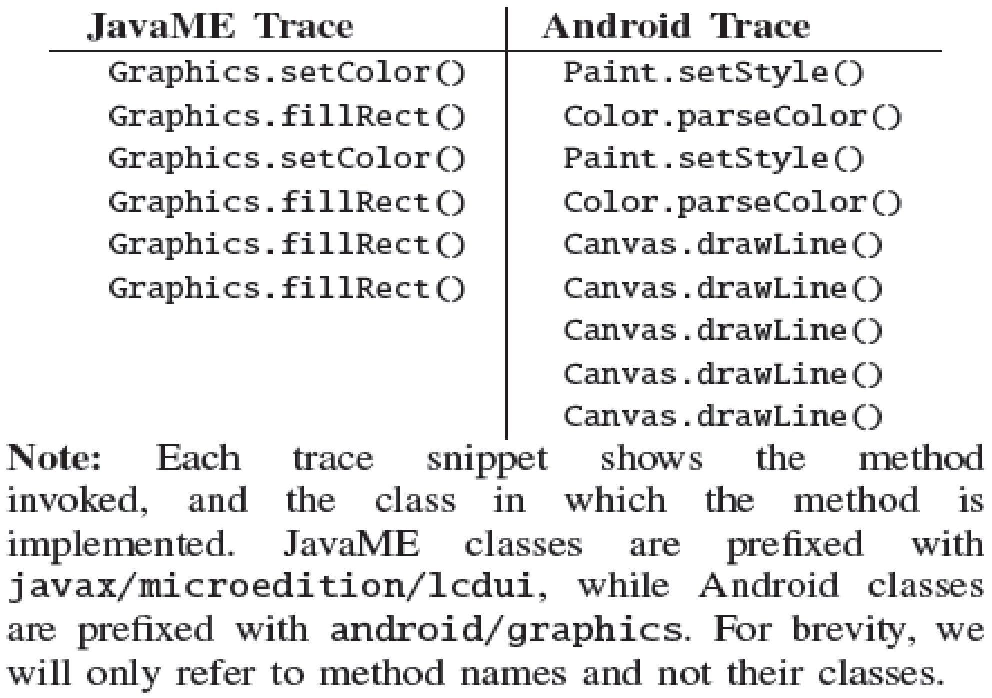
Figure 2. Rosetta Mapper

2.3.1 Rosetta Mapper

We now discuss the attributes used by our approach using our running example. Our aim is to find likely mappings for the JavaME calls $setColor()$ and $fillRect()$. For simplicity, we restrict our discussion to the snippets of the traces shown in Fig. 3. In reality, our analysis considers the entire trace.

Call frequency:

Our algorithm works like a reverse engineering process. If $A()$ in the source API maps to the $a()$ in the target API, then the frequency with which these method calls occur in functionally-similar trace pairs might also match. The trace pairs may differ in the absolute number of method calls that they contain, so we focus on the relative count of each method call, which is the raw count of the number of times that a method is called, normalized by trace length. Using this attribute, the following API mappings appear likely, $setColor() \rightarrow parseColor()$, $setColor() \rightarrow setStyle()$, $fillRect() \rightarrow drawLine()$, while the others appear unlikely.



JavaME Trace	Android Trace
Graphics.setColor()	Paint.setStyle()
Graphics.fillRect()	Color.parseColor()
Graphics.setColor()	Paint.setStyle()
Graphics.fillRect()	Color.parseColor()
Graphics.fillRect()	Canvas.drawLine()
Graphics.fillRect()	Canvas.drawLine()
	Canvas.drawLine()
	Canvas.drawLine()
	Canvas.drawLine()

Note: Each trace snippet shows the method invoked, and the class in which the method is implemented. JavaME classes are prefixed with `javax/microedition/lcdui`, while Android classes are prefixed with `android/graphics`. For brevity, we will only refer to method names and not their classes.

Figure 3. Snippets from traces of similar executions of tic-tac-toe

We can extend this notion for mapping of one to two api calls. Thus our approach works on method sequences as well, and infers that $setColor() \rightarrow setStyle(); parseColor()$ is a mapping.

Call position:

The location of method calls in a trace pair provides further information to determine likely mappings. Why? The reason is simple. If the API calls tend to do the same task, the introductory calls such as an *init()* call would likely be found at the top 10% of the trace and the conclusive calls such as exit API calls would occupy the last 10% of the trace. The sequence of the output is ensured to be the same by means of visual verification and consequently the string of API calls of both platforms which performs similar output should also have positional similarity. Since the traces exercise the same application functionality, method calls that map to each other must appear at “similar” positions in the trace pair. This is a sample to explain the notion of positional similarity actual positional information is of the dimension of 1% to 100%. Meaning the traces is divided into 100 parts from 1-100 denoting the actual portions of the trace files.

Call context

This factor would measure how the API calls interacts with other API calls in a trace file. The context in which a method call $A()$ appears is defined to be the set of API methods that appears in its vicinity in the execution trace. (e.g., within a pre-set threshold distance, preceding or following $A()$ in the trace). This measures how the interaction is between the any other call $B()$ with the given API call $A()$ of the source platform. We measure similar metric for the target platform. We see how the method $A()$ shows similarity to the corresponding $a()$ of the target platform in interaction with other calls of the trace. Why does this attribute make sense? If the API call $A()$ is predominantly paired up with many sibling API calls of the source trace, then it is natural to expect the target API call $a()$ to exhibit similar patterns of pairing up with corresponding siblings of the target API calls. We can capture this property by means of obtaining the relative frequencies of every other call in the trace in the vicinity. For example, $setColor()$ calls appear in the preceding context of $fillRect()$ calls in the JavaME trace (using a threshold distance of 2 calls). Likewise, $parseColor()$ and $setStyle()$ appear in the preceding context of $drawLine()$ in the Android trace. In fact, the sequence $setStyle();\ parseColor()$ appears in the preceding context of the first $drawLine()$. From this, we can infer that if $fillRect() \rightarrow drawLine()$ holds, then the following mapping $setColor() \rightarrow\ setStyle(); parseColor()$ is likely to hold.

Method names

Obviously, the similar names implies the API calls are related to each other. While trace structure, as captured by the attributes above, contributes to inference, method (and class) names also contain useful information about likely mappings. We use Levenshtein edit distance to compute the similarity of method names. The Levenshtein distance between two words is equal to the number of single-character edits required to change one word into the other. This metric matches how close the given two API calls are in terms of names. For example, the Levenshtein distance between "kitten" and "sitting" is 3, since the following three edits change one into the other, and there is no way to do it with fewer than three edits:

1. **k**itten → **s**itten (substitution of 's' for 'k')
2. sitt**e**n → sitt**i**n (substitution of 'i' for 'e')
3. sittin → sittin**g** (insertion of 'g' at the end).

Using this attribute, for instance, we can lend credence to the belief that $setColor()$ maps to $parseColor()$. It is also important to remember that the method names are not sufficient enough to find the mapping themselves as it might be tempting to believe so. For example $\boldsymbol{g}etColor()$ is the closest match to $\boldsymbol{s}etColor()$ in terms of their names, yet they perform completely different functionality.

The above metrics are obtained for every trace pair and we compute the corresponding mappings between the possible source and the target APIs present in the trace pair and rank them according their probabilities of mapping. We use user defined thresholds to limit the number of results that are output and verify the obtained results manually.

Combining inferences across traces:

Note that this inference in Step 3 was based upon analyzing a single trace pair. In the final step, we combine inferences across multiple traces. During combination, we weight inferences obtained from individual trace pair analysis using the confidence of that inference. Intuitively, more data we have about an inferred mapping from a trace pair, the stronger our confidence in that inference. Thus, our confidence in mapping $A() \rightarrow a()$ obtained from a trace pair where these calls occur several times is stronger than the same mapping obtained from a trace pair where these calls occur infrequently. We use this heuristic to combine inferences across trace pairs.

2.3.2 Rosetta Classifier

We want to see if it is possible to automate the classification of a mapping between source API and the target API method to be true or false. While the ground truth is still manual verification (involves going through the API documentation of the source and target platform), we build a classifier which classifies the given mapping as true or not by using supervised learning technique. The classifier is based on set of values which describes the nature of the mapping called feature vectors, which in turn is based on the structural information of the trace files. Every element (mapping) in the dataset is defined by a feature vector. Training data is the set of mappings between the source and the target API methods with corresponding feature vectors which are known to be either true or false. We shall now discuss the feature vector for an element. We have constructed the features which are similar to the factors in factor graph approach. While the factor graph obtains the computes factors; obtains mappings from every pair of traces individually and combines the results across traces, the Rosetta classifier would compute features across traces pair and obtain mappings from all of the trace pairs collectively. Thus whenever we have a new pair of traces, we need run the generate newer set of the features and train the classifier with better set of training data, to obtain better set of mappings.

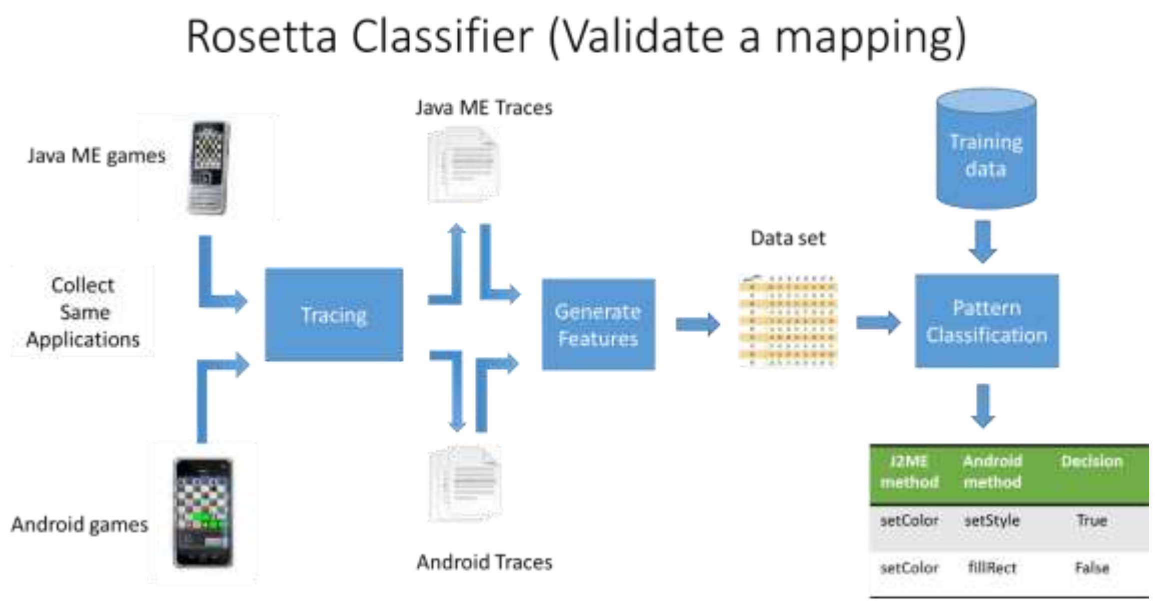


Figure 4. Rosetta Classifier

Features:

In machine learning and pattern recognition, a **feature** is an individual measurable heuristic property of a phenomenon being observed. Here the phenomenon is the mapping between the source API method and the target API method. Choosing discriminating and independent features is key to any pattern recognition algorithm being successful in classification. The set of features of a given data instance is often grouped into a feature vector. The reason for doing this is that the vector can be treated mathematically. For example, many algorithms compute a score for classifying an instance into a particular category by linearly combining a feature vector with a vector of weights, using a linear predictor function. In character recognition, features may include horizontal and vertical profiles, number of internal holes, stroke detection and many others. In speech recognition, features for recognizing phonemes can include noise ratios, length of sounds, relative power, filter matches and many others. In spam detection algorithms, features may include whether certain email headers are present or absent, whether they are well formed, what language the email appears to be, the grammatical correctness of the text, Markovian frequency analysis and many others.

Mutual Information

In probability theory and information theory, the **mutual information** of two random variables X and Y is a quantity that measures the mutual dependence of the two random variables. Mutual information measures the information that X and Y share: it measures how much knowing one of these variables reduces uncertainty about the other. For example, if $X\ and\ Y$ are independent, then knowing X does not give any information about Y and vice versa, so their mutual information is zero. At the other extreme, if X and Y are identical then all information conveyed by X is shared with Y: knowing X determines the value of Y and vice versa. As a result, in the case of identity the mutual information is the same as the uncertainty contained in $Y\ (or\ X)$ alone, namely the entropy of $Y\ (or\ X$: clearly if X and Y are identical they have equal entropy). Formally, the mutual information of two discrete random variables X and Y can be defined as:

$$I(X;Y) = \sum_{y\,\in\,\mathrm{Y}} \sum_{x\,\in\,\mathrm{X}} p(X = x, Y = y) \log\left(\frac{p(X = x, Y = y)}{p(X = x)p(Y = y)}\right)$$

Where, $p(x, y)$ is the joint probability distribution function of X and Y, and $p(x)$ and $p(y)$ are the marginal probability distribution functions of X and Y respectively.

1. Mutual information on frequency

In our approach we consider the random variables X and Y to be relative frequency of the J2ME and Android API methods in their corresponding traces. Thus the random variable relative frequency takes the possible values of {0.00 to 0.99} in each of the traces. We are measuring the amount of information shared by the relative frequency of the API calls of the J2ME and Android, how much information do they share. In other words, how does knowing the relative frequency of the j2me method can affect the uncertainty of the relative occurrence of the android API method. The more they share the information amongst each other on the relative frequency, it is possible that they are good candidates for a true mapping.

2. Mutual information on position

Similar to the information sharing on relative frequency on the source and the target API calls, their sharing of information in their position in the trace files also could possibly mean that they are potentially true mapping. Thus we compute the mutual information on the position of the API calls. We obtain the mutual information of the J2ME and Android calls for 100 different positions across the available traces. These are similar to the position and the frequency factors we had earlier.

3. String edit distance

While trace structure, as captured by the attributes above, contributes to inference, method (and class) names also contain useful information about likely mappings. We use Levenshtein edit distance to compute the similarity of method names. The Levenshtein distance between two words is equal to the number of single-character edits required to change one word into the other. This metric matches how close the given two API calls are in terms of names. We simply use the name factor once again to generate features for the classifier.

4. Digrams

This feature would gauge how the API calls interacts with other API calls in the trace file. This is similar to the context factor of the factor graph approach except that instead of the vicinity parameter we look for $B()$ at the entire trace file for the context of a given API method $A()$. We measure the relative frequency of every other API call $B()$ in trace file along with the given API call $A()$. We also calculate similarly for the target platform. Now for each mapping between the source and the target platform we have the above features defining the feature vector. These feature vectors act as the training data for a classifier built using supervised learning technique namely support vector machines.

Chapter 3

Framework to Represent and Infer Mappings

We now describe the framework used to represent and infer likely mappings between APIs. We restrict ourselves to identifying likely mappings between methods of the source and target APIs.

3.1 API Mapping

It is a notion of expressing functional equivalence of API methods from the source platform to the target platform. We do not claim the equivalence of the API calls. But rather we suggest that "for an API method of the source platform, we could try to use this API method of the target platform in order to perform the target program functionality". It is a subjective and subtle notion rather than an objective or a declarative way of saying that "method A of source platform maps to method α of the target platform" plainly. Basically, it is all in the programmer's hand and his choice of implementing the functional calls. We are simply here to suggest a possible replacement API methods for given set of source API methods from the program of the source platform, which would ease the creation of program performing similar functionality in the target platform.

We do not currently consider the problem of determining mappings between arguments to these methods; that is a related problem that can be addressed using parameter recommendation systems (e.g., [33]). we also do not take into account of the return value or its data type of the API methods. We do not account for the transfer of values between the API calls. There are taint based technologies, techniques to obtain the flow of data [41].

The following is the Framework for Inferring likely mappings between source and the target APIs. Let $S = \{A , B , C , ...\} \ and \ T = \{a , b , c , ...\}$ denote the sets of methods in the source and target APIs, respectively. Our goal is to determine which methods in T implement the same functionality on the target platform as each method in S on the source platform. To denote mappings, our framework considers a set of Boolean variables, X_t^s where $s \epsilon S \, and \, t \epsilon T$. The Boolean variable X_a^A is set to true if the method call $A()$ maps to the method call () , and false otherwise. This framework extends naturally to the case where a method (or sequence of methods) in S can be implemented with a method sequence in T. For example, suppose that the method $A()$ is implemented using the sequence $a(); b()$ in the target. We can denote this by assigning true to the Boolean variable X_{ab}^A . Likewise, we can also denote cases where the functionality of a sequence of methods $A(); B()$ in the source platform is implemented using a method $a()$ on the target platform by setting the Boolean variable X_a^{AB} to true . Although our framework theoretically supports inference of mappings between arbitrary length method sequences (e.g., $X_{abc\ldots}^{ABC\ldots}$ = true), for performance reasons we configured Rosetta to only infer mappings between method sequences of length two. We approach the problem of inferring API mappings by studying the structure of execution traces of similar applications on the source and target platforms. We use the trace attributes informally discussed before to deduce API mappings. Our approach is inherently probabilistic. It cannot conclude whether a call $A()$ definitively maps to a call () ; rather it determines the likelihood of the mapping. Thus, it infers

$Pr[X_a^A = true]$for each Boolean variable X_a^A. As it observes more evidence of the mapping X_a^A being true, it assigns a higher value to this probability. Therefore, our framework treats each Boolean variable X_a^A as a random variable, and our probabilistic inference algorithm determines the probability distributions of these random variables. Each application trace has several method calls, and the inference algorithm must leverage the structure of these traces to determine likely mappings. The inference algorithm draws conclusions not just about individual random variables, but also about how they are related.

For example, consider a trace snippet Trace

$S = (\ldots, A(); A(); B(); B(); , \ldots)$ of an application from the source API, and a snippet Trace

$T = (\ldots ; a(); a(); b(); b(); \ldots)$ from the corresponding execution of a similar application on the target. Suppose also that these are the only occurrences of $A()$, $B()$, $a()$ and $b()$ in Trace S and Trace T , respectively, and that these execution traces have approximately the same number of method calls in total. By just observing these snippets, and relying on the frequency of 3 method calls, each of the following cases is a possibility: $X_a^A = true, X_b^A = true, X_a^B = true$ and $X_b^B = true$. However, if $X_b^A = true$, then because of the relative placement of method calls in these traces (i.e., call context), it is unlikely that $X_a^B = true$. Now suppose that by observing more execution traces, we are able to obtain more evidence that $X_b^A = true$, then we can leverage the structure of this pair of traces to deduce that X_a^B is unlikely to be a mapping. Intuitively, the structure of the trace allows us to propagate the belief that if X_b^A is true, then X_a^B is false. Our probabilistic inference algorithm uses factor graphs [18], [31] for belief propagation.

3.2 Factor Graphs

In probability theory and its applications, a **factor graph** is a particular type of graphical model, with applications in Bayesian inference, which enables efficient computation of marginal distributions through the sum-product algorithm. **Belief propagation**, also known as **sum-product** is a message passing algorithm for performing inference on graphical models, such as Bayesian networks and Markov random fields. It calculates the marginal distribution for each unobserved node, conditional on any observed nodes. In our problem the marginal distribution represents the probability of a mapping being true, while the factors represents the attributes we calculated from the trace files. Now given these attributes values to the factor graph, we get the probability of a mapping being true as the output through sum-product algorithm, which basically computes the marginal distribution of the for each observed nodes (the mapping represented by means of Squares in the diagram) conditioned on any observed nodes (the attributes represented by means of circles in the diagram). Let X = $\{x_1, x_2, \ldots, x_n\}$ be a set of Boolean random variables and J$(x_1, x_2, \ldots, x_n)$ be a joint probability distribution over these random variables. J assigns a probability to each assignment of truth values to the random variables in X. Given such a joint probability distribution, it is natural to ask what the values of the *marginal probabilities* $M_k(x_k)$ of each random variable x_k in X are.

Marginal probabilities are defined as

$$M_k(x_k) = \sum_{i=1, i \neq k,\, x_i \in [\,True, False]}^{n} \mathrm{J}(x_1, x_2, \ldots, x_n).$$

That is, they calculate the probability distribution of x_k alone by summing up J(: : :) over all possible assignments to the other random variables. $M_k(x_k)$ allows us to compute

$Pr[x_k \;=\; true]$. Factor graphs allow efficient computation of marginal probabilities from joint probability distributions when the joint distribution can be expressed as a product of factors, i.e.,

$$\mathrm{J}(x_1, x_2, \ldots, x_n) = \prod_i f_i(X_i)$$

Each fi is a factor, and is a function of some subset of variables $x_i \subseteq X$. For example, consider a probability distribution J(x, y, z). Let this distribution depend on three factors $f(x, y)$, g(y, z), and h(z), i.e., J(x, y, z) = f(x, y).g(y, z). h(z),

Each of the factors are defined as follows:

$f(x, y) = 0.9 \;\; if\; x \;\vee y \;=\; False$, 0.1 otherwise.

$g(y, z) \;=\; 0.9\, if\; y \;\vee\; z \;=\; True,$ 0.1 otherwise.

$h(z) \;=\; 0.9\, if\; z \;=\; True,$ 0.1 otherwise.

Because J is a probability distribution, the product J$(x, y, z)$is in fact Z: $f(x;\ y)$: $g(y;\ z)$: $h(z)$, where Z is a normalization constant introduced to ensure that the probabilities sum up to 1. In the rest of thesis, the normalization constant is implied, and will not be shown explicitly.

A factor graph denotes such joint probabilities pictorially as a bipartite graph with two kinds of nodes, function nodes and variable nodes. Each function node corresponds to a factor, while each variable node corresponds to a random variable. A function node has outgoing edges to each of the variable nodes over which it operates. The figure alongside depicts the factor graph of J. The AI community has devised efficient solvers [18], [31] that operate over such graphical models to determine marginal probabilities of individual random variables. We do not discuss the details of these solvers, since we just use them in a black box fashion. Factor graphs cleanly represent how the random variables are related to each other, and can influence the overall probability distribution. One of the key characteristics of factor graphs, which led us to use them in our work, is that they were designed for belief propagation, i.e., in transmitting beliefs about the probability distribution of one random variable to determine the distribution of another. To illustrate this, consider the probability distribution J(x, y, z)discussed above. J can be interpreted as denoting the probabilities of the outcomes of an underlying Boolean formula for various assignments to $x, y, and\ z$. Under this interpretation, we could say that the Boolean formula evaluates to true for those assignments to $x, y\ and\ z$ for which the value of J(x, y, z) is above a certain threshold, e.g., if the threshold is 0.6, and $J(true;\ true;\ false)$ is 0.7, we say that the formula is true under this assignment. Now suppose that y is likely to be false, i.e., $Pr[y\ =\ false]$ is above a threshold. We are asked to find under what conditions on x and z the Boolean formula still evaluates to true, i.e., $J(x;\ y;\ z)$ is above the threshold. From the definitions of the factors $f, g, and\ h$, we know that J obtains values that are likely to exceed the threshold $if\ x \vee y\ =\ false, y \vee z\ =\ true$ and $z\ is\ true$. Given that y is likely to be $false$, these factors lead us to deduce that x is also likely to be $false$ (giving $f(x, y)$ a high value) and z is likely to be true (giving $g(y;\ z)\ and\ h(z)$ high values), thereby pushing the value of J above the threshold. Intuitively, the factor graph allows us to propagate the belief about the value of y into beliefs about the value of $x\ and\ z$. §IV shows how we cast the problem of inferring likely API mappings using factor graphs, thereby allowing us to transmit beliefs about one mapping into beliefs about others.

3.2 Supervised Learning

We now discuss the machine learning technique which used by the Rosetta Classifier to classify the mapping into true or false mapping based on the feature vector obtained for each mapping.

Supervised learning is the machine learning task of inferring a function from labeled training data. The training data consist of a set of *training examples*. In supervised learning, each example is a *pair* consisting of an input object (typically a vector) and a desired output value (also called the *supervisory signal*). A supervised learning algorithm analyzes the training data and produces an inferred function, which is called a *classifier* (if the output is discrete, see classification) or a *regression function* (if the output is continuous, see regression). The inferred function should predict the correct output value for any valid input object. This requires the learning algorithm to generalize from the training data to the testing data and to the unseen in a "reasonable" way.

In order to solve a given problem of supervised learning, one has to perform the following steps:

1. Determine the type of training examples. Before doing anything else, the user should decide what kind of data is to be used as a training set. In the case of handwriting analysis, for example, this might be a single handwritten character, an entire handwritten word, or an entire line of handwriting. In our approach, we have the mapping between the source and the target api methods and the corresponding features.

2. Gather a training set. A set of input objects is gathered and corresponding outputs are also gathered, either from human experts or from measurements.

3. Determine the input feature representation of the learned function. The accuracy of the learned function depends strongly on how the input object is represented. Typically, the input object is transformed into a feature vector, which contains a number of features that are descriptive of the object. Determine the structure of the learned function and corresponding learning algorithm. For example, the engineer may choose to use support vector machines or decision trees.

4. Complete the design. Run the learning algorithm on the gathered training set. Some supervised learning algorithms require the user to determine certain control parameters. These parameters may be adjusted by optimizing performance on a subset (called a *validation* set) of the training set, or via cross-validation.

5. Evaluate the accuracy of the learned function. After parameter adjustment and learning, the performance of the resulting function should be measured on a test set that is separate from the training set.

In our approach, the input is the feature vector for the mapping. The output is the decision whether the mapping is true or false. Training data consists of these feature vectors and known decision (true or false) which are obtained manually by searching the API documentation of the source and the target platform.

We have training data consisting of the feature vector and the corresponding known decision. We look at the technique which we use to build a classifier which trains on this training data and predict the mapping to be true or false for a new feature vector.

3.2.1 Support Vector Machines

In machine learning, **support vector machines** (**SVMs**, also **support vector networks**) are supervised learning models with associated learning algorithms that analyze data and recognize patterns, used for classification and regression analysis. The basic SVM takes a set of input data and predicts, for each given input, which of two possible classes forms the output, making it a non-probabilistic binary linear classifier. Given a set of training examples, each marked as belonging to one of two categories, an SVM training algorithm builds a model that assigns new examples into one category or the other. An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall on. In addition to performing linear classification, SVMs can efficiently perform non-linear classification using what is called the kernel trick, implicitly mapping their inputs into high-dimensional feature spaces. The idea behind SVMs is to make use of a (nonlinear) mapping function φ that transforms data in input space to data in feature space in such a way as to render a problem linearly separable. The SVM then automatically discovers the optimal separating hyperplane (which, when mapped back into input space via φ^{-1}, can be a complex decision surface). A support vector machine constructs a hyperplane or set of hyperplanes in a high- or infinite-dimensional space, which can be used for classification, regression, or other tasks. Intuitively, a good separation is achieved by the hyperplane that has the largest distance to the nearest training data point of any class (so-called functional margin), since in general the larger the margin the lower the generalization error of the classifier.

Given some training data D, a set of n points of the form:

$$\mathrm{D} = \{(\mathrm{x_i}, \mathrm{y_i}) | \, \mathrm{x_i} \in \mathrm{R^p}, \mathrm{y_i} \in \{-1,1\}\}, \mathrm{i} = 1, \ldots, \mathrm{n}$$

Where y_i is either 1 or −1, indicating the class to which the point $x_i$belongs. Each x_i is a p-dimensional real vector. We want to find the maximum-margin hyperplane that divides the points having $y_i = 1$ from those having $y_i = -1$. Any hyperplane can be written as the set of points x satisfying

$$w.x - b = 0,$$

Where "." denotes the dot product and w the normal vector to the hyperplane. The parameter $b/\|w\|$ determines the offset of the hyperplane from the origin along the normal vector w. If the training data are linearly separable, we can select two hyperplanes in a way that they separate the data and there are no points between them, and then try to maximize their distance. The region bounded by them is called "the margin". These hyperplanes can be described by the equations

$$w.x - b = 1, \& \quad w.x - b = -1,$$

By using geometry, we find the distance between these two hyperplanes is $2/\|w\|$, so we want to minimize$\|w\|$. As we also have to prevent data points from falling into the margin, we add the following constraint: for each i either

$$w.x_i - b \geq 1, \qquad for\ x_i\ of\ the\ first\ class$$

Or

$$w.x_i - b \leq 1, \qquad for\ x_i\ of\ the\ second\ class$$

The optimization problem presented in the preceding section is difficult to solve because it depends on $\|w\|$, the norm of w, which involves a square root.

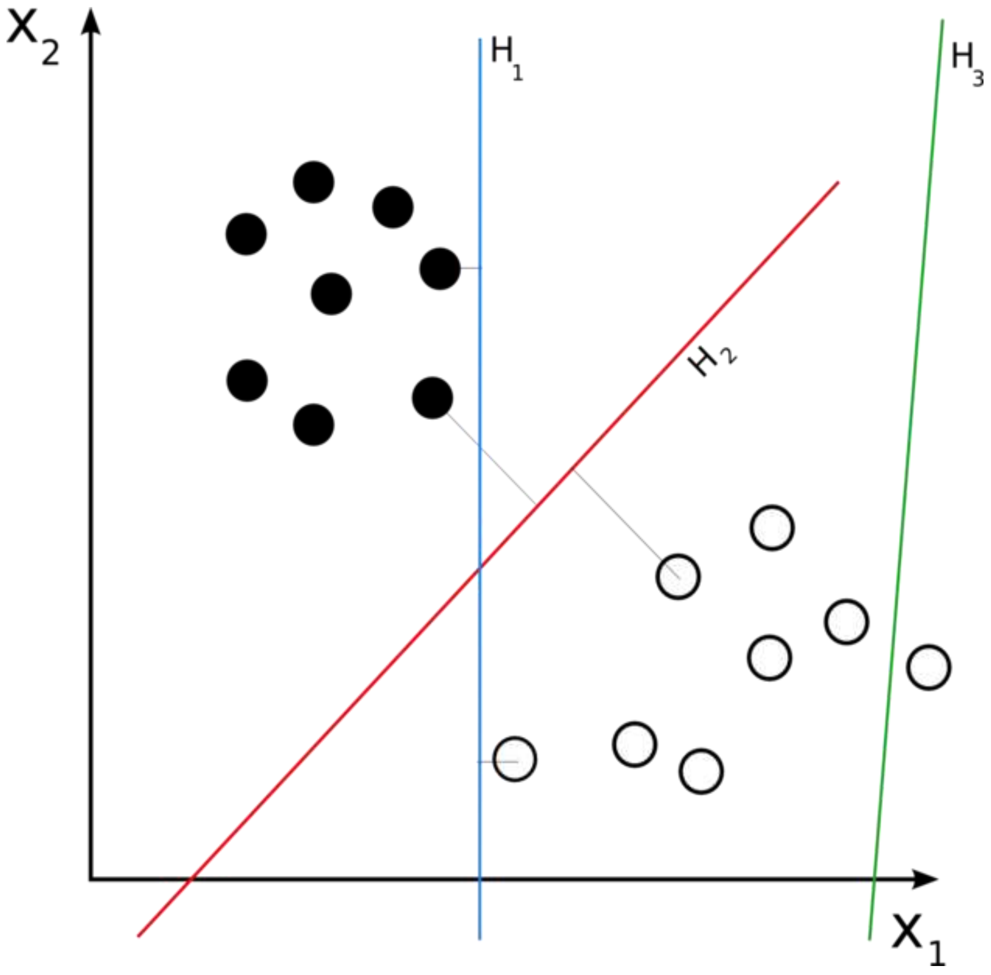


Figure 5. Classifiers of different margins

H3 (green) doesn't separate the two classes. H1 (blue) does, with a small margin and H2 (red) with the maximum margin. Filled circles are positive and hollow circles are negative.

Fortunately it is possible to alter the equation by substituting $\|w\|$ with $\frac{1}{2}\|w\|^2$ (the factor of 1/2 being used for mathematical convenience) without changing the solution (the minimum of the original and the modified equation have the same w and b). This is a quadratic programming optimization problem.

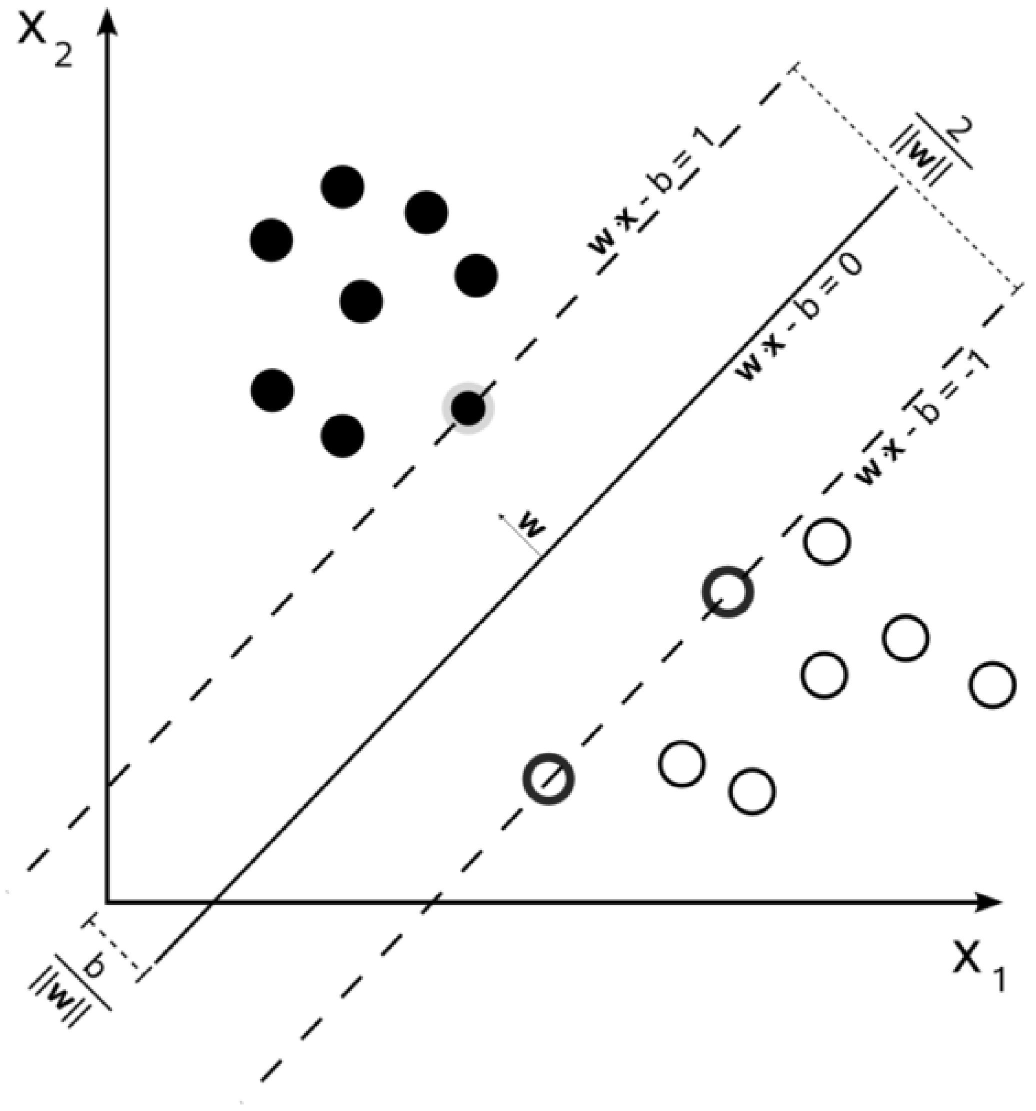


Figure 6. Classifiers of maximum margins

Samples on the margin are called the support vectors

Minimize (in w, b) $\frac{1}{2}\|w\|^2$ subject to (for any $i = 1, \ldots, n$) $y_i(w.x_i - b) \geq 1$

By introducing Lagrange multipliers α (a technique to solve constrained optimization problem), the previous constrained problem can be expressed as

$$\min_{w,b} \max_{\alpha \geq 0} \left\{ \frac{1}{2}\|w\|^2 - \sum_{i=1}^{n} \alpha_i [\, y_i(\, w.x_i - b\,) - 1\,] \right\}$$

This problem can now be solved by standard quadratic programming techniques and programs. The "stationary" Karush–Kuhn–Tucker condition implies that the solution can be expressed as a linear combination of the training vectors.

$$w = \sum_{i=1}^{n} \alpha_i y_i x_i.$$

Only a few α_i will be greater than zero. The corresponding x_i are exactly the *support vectors*, which lie on the margin and satisfy $y_i(w.x_i - b) = 1$. From this one can derive that the support vectors also satisfy

$$w.x_i - b_i = 1/y_i = y_i \qquad \leftrightarrow \qquad b = w.x_i - y_i$$

Writing the classification rule in its unconstrained dual form reveals that the maximum margin hyperplane and therefore the classification task is only a function of the *support vectors*, the training data that lie on the margin.

Using the fact that $\|w\|^2 = w.w$ and substituting $w = \sum_{i=1}^{n} \alpha_i y_i x_i$. one can show that the dual of the SVM reduces to the following optimization problem:

Maximize (in α_i) the Lagrange formula would look like:

$$\tilde{L}(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j x_i^T x_j = \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j k(x_i, x_j)$$

Subject to (for any $i = 1, \dots, n$)

$\alpha_i \geq 0,$

and to the constraint from the minimization in b

$$\sum_{i=1}^{n} \alpha_i y_i = 0.$$

Here the ***kernel*** is defined by. $k(x_i, x_j) = x_i . x_j$

w can be computed thanks to the α_i terms: $w = \sum_{i=1}^{n} \alpha_i y_i x_i$.

w is the normal vector of the hyperplane which divides the given training set of data into two classes. α_i are the Lagrange multipliers used to minimize the distance between the hyperplane and the data points. y_i are the values of +1 or -1 or the binary classification output values. x_i are the input training data. This plane will tell whether the future points lie on the category 1 or -1. Basic idea of support vector machines is that it constructs an optimal hyperplane for linearly separable patterns of data. For nonlinear separation it transforms the coordinates into a new space by means a *kernel* function.

The feature vector consists of the following array of values

1. Mutual information on frequency,

2. Mutual information on position.

3. String edit distance

4. Digram approach.

5. Correlation coefficient among the frequency.

The feature vector generated for the mapping in the dataset is plotted in a k dimensional Euclidean space, where k is the number of features considered. The mappings are divided into true or false based on the hyperplane created by the SVM. The hyperplane defines the boundary beyond which the mappings are true thus corresponding to y=1 and subsequently false to y=-1 for the mapping before the boundary. The boundary is matured by training data so as make sure the margin (or the distance between the boundary and the nearest data point) is maximized. Now that the equation of the decision surface is found we can compute the nature of the future data points as positive or negative. Correspondingly the decision surface (hyperplane) can now classify new mapping with feature vector as true or false mapping.

Kernel and SVM

For non-linearly separable data points, they are translated into a new set of coordinates of higher dimensions. The translation from the coordinates of the feature space to this higher dimensional space is such that in this higher dimensional space the data points are linearly separable by means of a hyperplane. The translation is by means of a ***kernel*** function. We won't go in detail of the various types of kernels and their way of solving nonlinearly separable data classification. Consider the following set of data points in the 2D plot figure 7. It has linearly separable data {the data points which could be separated by means of a line}. Figure 8 shows how SM would classify it using a hyperplane and Support vectors. Figure 9 shows a nonlinearly separable data. We extrapolate the points to a 3D using the kernel function:

$\varphi(x1, x2) = (x1, x2, [x1^2 + x2^2 - 5]/3)$. We see a bunch of black dashes representing the hyperplane which separates the data points with max margin.

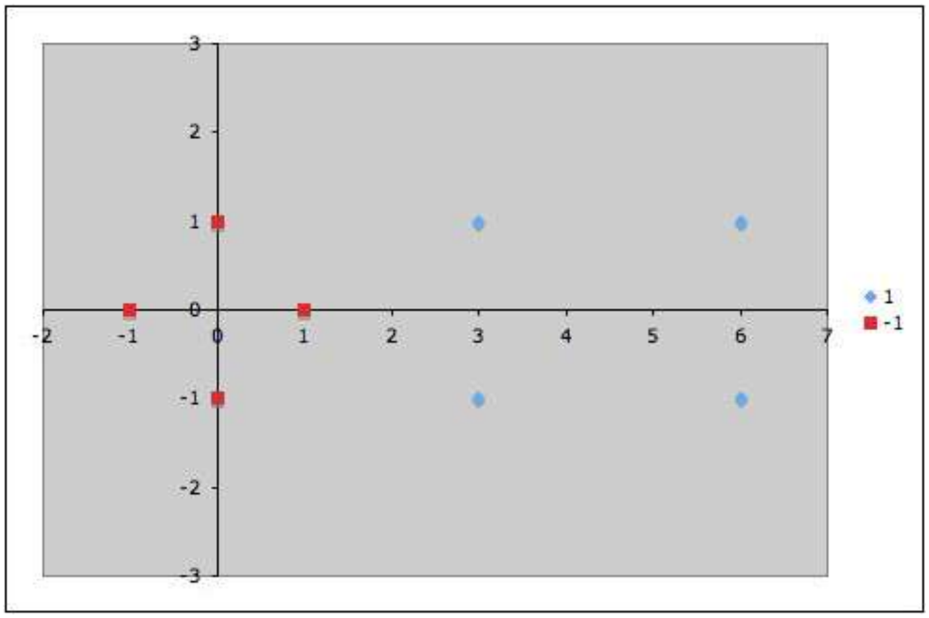


Figure 7. Linearly separable data points in 2D

Sample linearly separable data points in 2D. Blue diamonds are positive examples and red squares are negative examples.

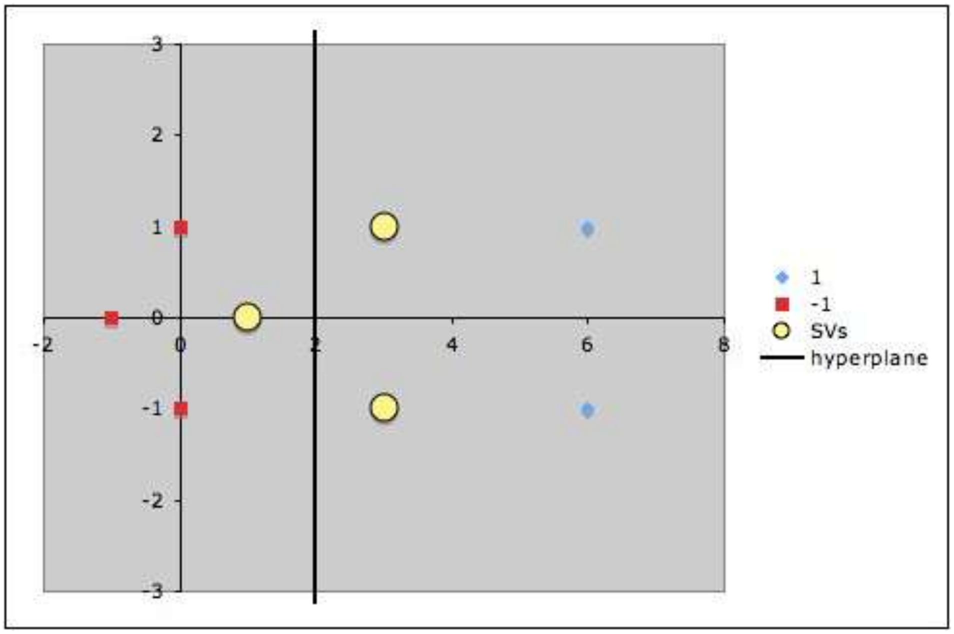


Figure 8. SVM with linearly separable data

Using Support vectors (yellow) and the hyperplane decision surface.

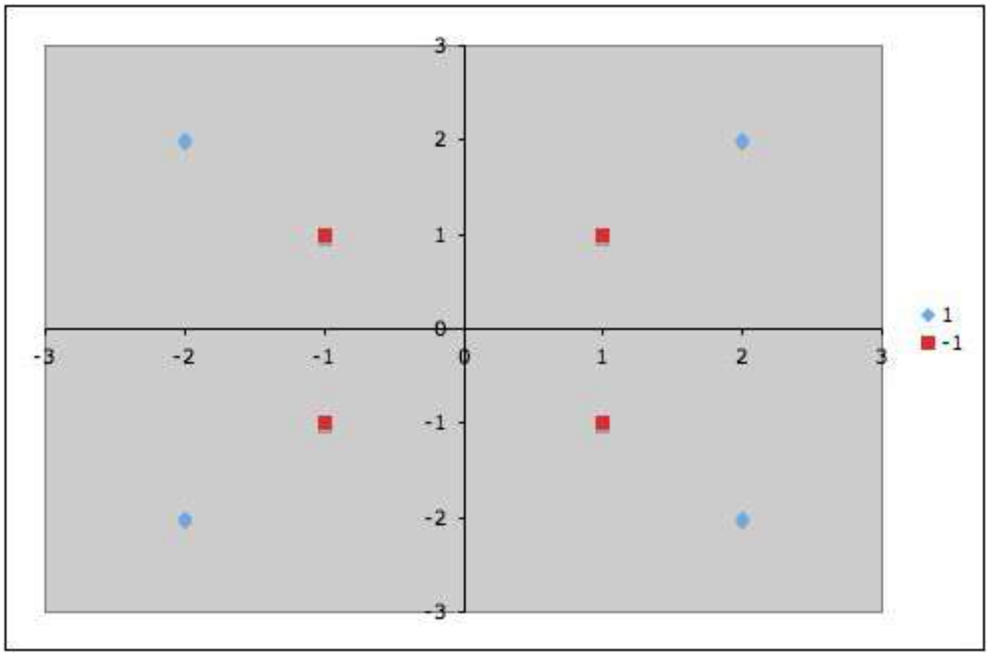


Figure 9. Nonlinearly separable data

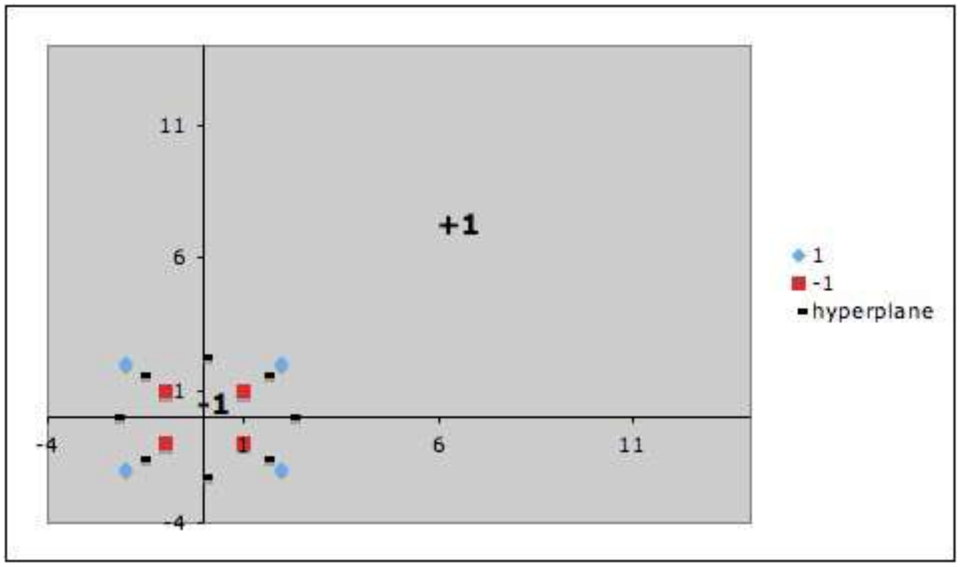


Figure 10. SVM projects the data from 2D to 3D using kernel.

$\varphi(x1, x2) = (x1, x2, [x1^2 + x2^2 - 5]/3)$. The "+1" and the "-1" represent the Z dimension coordinates. Look for the bunch of 8 black dashes. They represent the hyper plane which linearly separates the data points into positive and negative examples. Thus increasing the dimension of the data using the kernel improves the ability of the classifier to come up with a linearly separating hyper plane. We use such kernels in our problem in order to classify our data points corresponding to the mappings described by the feature vectors into positive and negative examples or true and false mappings.

Chapter 4

Experimental Setup for Rosetta

We now describe in detail the Rosetta prototype, which currently infers mappings between the JavaME and Android graphics API. Specifically, we focus on the machinery that enables Steps 2-4 discussed in chapter II. We choose graphical APIs for the Java and the Android Platform since:

1. The visual verification of equal functionality of the given two application is relatively easy rather doing it computationally.

2. We choose JavaME and Android platforms because Games of same names are easier to find in both the mobile platforms.

4.1 Support Vector Machines

To record execution traces, we instrumented JavaME programs via byte code rewriting. We used the ASM toolkit [9] to insert logging functionality that records the name of each method call (and class name), prior to invocation. During runtime, this result in a trace of all methods invoked. We then filter out just those methods that derive from the class javax/microedition/lcdui—the JavaME graphics API. In all, this API has 281 distinct methods. Rosetta infers mappings for those methods that appear in application traces. We did not employ bytecode rewriting for Android applications because of the lack of publicly-available tools to rewrite Android's dex bytecode. Instead, we leveraged the Dalvik virtual machine (v2.1r1) to record the names of all methods invoked by an application. We record all method and class names, and then filter methods in the following classes : $android/graphics, android/text, android/view, android/$ $widget, android/content/DialogInterface, android/app/Dialog, android/app/$

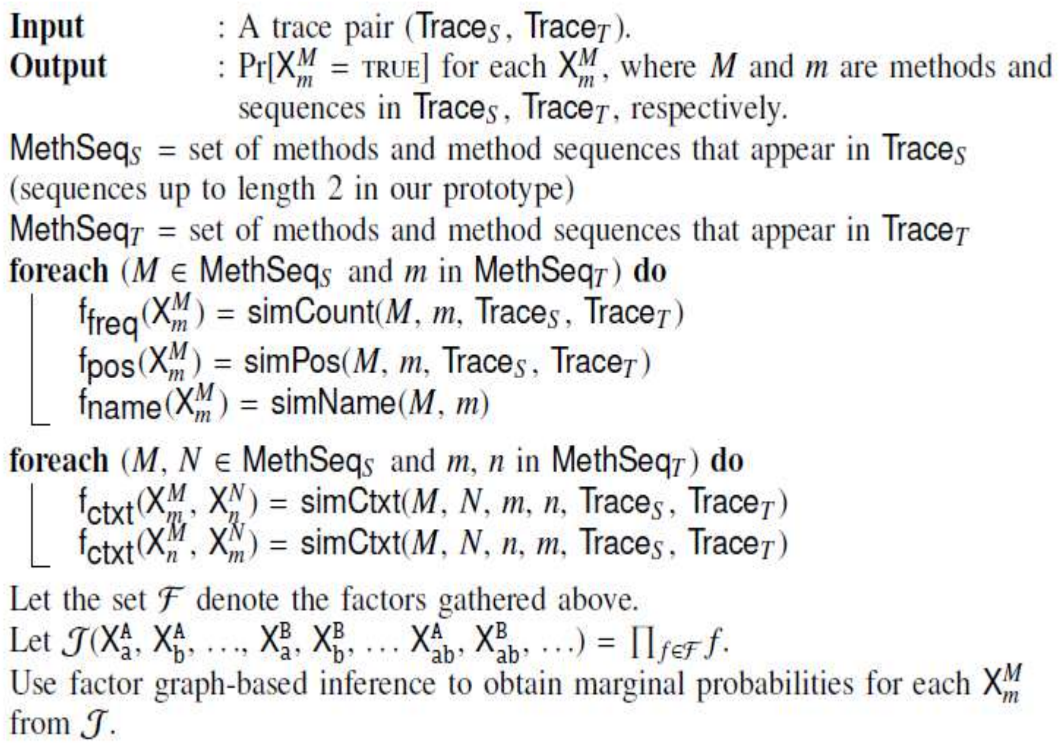
$AlertDialog, android/\ app/ActionBar$. This API has 3,837 distinct methods. The difference in the sizes of these APIs illustrates in part the difficulties that a programmer manually porting

an application would face. With this infrastructure, a Rosetta user can collect traces for a pair of similar applications on both platforms. As discussed in chapter II, the user must exercise similar functionality in both applications, thereby collecting a pair of traces that record this functionality. This process can be repeated multiple times for the same application, exercising different functionalities, thereby resulting in a database of trace pairs. Given similar games, they will likely have the same logic and similar GUIs on both applications. We simply performed the same moves on games in both platforms, avoiding situations that involve randomness where possible (e.g., choosing the two-user mode to avoid the computer picking moves at random). However, randomness is not always avoidable, e.g., some games only support user versus computer modes, and this randomness may manifest in the corresponding portions of the traces as well. Despite this, Rosetta infers high-quality mappings because its inference algorithm prioritizes method mappings that persist both across the entirety of each trace pair, and across multiple trace pairs. The games we had selected were simple board games with least amount of dynamic content possible. This way the moves and the actions are predictable and the corresponding moves can be performed on the other platform to obtain the trace files.

4.2 Trace Analysis and Inference

4.2.1 Rosetta Mapper

In this step, Rosetta analyzes each trace pair collected in the previous step and draws inferences about likely API mappings implied by that trace pair. Recall from chapter III that we use a Boolean random variable X_a^A to denote a mapping between $A()\ and\ a()$ (likewise X_{ab}^A etc., for method sequences). In this step, Rosetta uses factor graphs to compute $Pr[X_m^M = true]$, for each such random variable, where M and m denote individual methods or method sequences from the source and target APIs, respectively. The value of this probability determines the likelihood that the corresponding mapping holds.



```
Input        : A trace pair ($\mathsf{Trace}_S$, $\mathsf{Trace}_T$).
Output       : $\Pr[\mathsf{X}_m^M = \textsc{true}]$ for each $\mathsf{X}_m^M$, where $M$ and $m$ are methods and
               sequences in $\mathsf{Trace}_S$, $\mathsf{Trace}_T$, respectively.
$\mathsf{MethSeq}_S$ = set of methods and method sequences that appear in $\mathsf{Trace}_S$
(sequences up to length 2 in our prototype)
$\mathsf{MethSeq}_T$ = set of methods and method sequences that appear in $\mathsf{Trace}_T$
foreach ($M \in \mathsf{MethSeq}_S$ and $m$ in $\mathsf{MethSeq}_T$) do
    $\mathsf{f_{freq}}(\mathsf{X}_m^M) = \mathsf{simCount}(M, m, \mathsf{Trace}_S, \mathsf{Trace}_T)$
    $\mathsf{f_{pos}}(\mathsf{X}_m^M) = \mathsf{simPos}(M, m, \mathsf{Trace}_S, \mathsf{Trace}_T)$
    $\mathsf{f_{name}}(\mathsf{X}_m^M) = \mathsf{simName}(M, m)$
foreach ($M, N \in \mathsf{MethSeq}_S$ and $m, n$ in $\mathsf{MethSeq}_T$) do
    $\mathsf{f_{ctxt}}(\mathsf{X}_m^M, \mathsf{X}_n^N) = \mathsf{simCtxt}(M, N, m, n, \mathsf{Trace}_S, \mathsf{Trace}_T)$
    $\mathsf{f_{ctxt}}(\mathsf{X}_n^M, \mathsf{X}_m^N) = \mathsf{simCtxt}(M, N, n, m, \mathsf{Trace}_S, \mathsf{Trace}_T)$
Let the set $\mathcal{F}$ denote the factors gathered above.
Let $\mathcal{J}(\mathsf{X}_a^A, \mathsf{X}_b^A, \ldots, \mathsf{X}_a^B, \mathsf{X}_b^B, \ldots \mathsf{X}_{ab}^A, \mathsf{X}_{ab}^B, \ldots) = \prod_{f \in \mathcal{F}} f$.
Use factor graph-based inference to obtain marginal probabilities for each $\mathsf{X}_m^M$
from $\mathcal{J}$.
```

Figure 11. Algorithm 1

The set of random variables X_m^M implicitly defines a joint probability distribution J over these random variables: $J(X_a^A\ \ , X_b^A, \ldots, X_a^B\ \ , X_b^B, \ldots X_{ab}^A, X_{ab}^B, \ldots)$. As in chapter III, we can assign J a Boolean interpretation. That is, we treat J as a probability distribution that estimates the likelihood of an underlying Boolean formula being true, under various truth assignments to the random variables X_a^A, X_b^A, etc. From this joint distribution, our goal is to find the probability distributions of the individual random variables. Under this Boolean interpretation, if $Pr[X_a^A = true]$ acquires a high value, it means that the Boolean formula underlying J is likely to be true if X_a^A is true, thereby leading us to conclude that $A()$ is likely to map to $a()$. Likewise, if $Pr[X_a^A = true]$ acquires a low value, $A()$ is unlikely to map to $a()$.

```
simCount($M$, $m$, $\text{Trace}_S$, $\text{Trace}_T$)
begin
    relCount($M$) = $\frac{\#\text{Occurences of } M \text{ in Trace}_S}{\text{Length}(\text{Trace}_S)}$
    relCount($m$) = $\frac{\#\text{Occurences of } m \text{ in Trace}_T}{\text{Length}(\text{Trace}_T)}$
    Return min[$\frac{\text{relCount}(M)}{\text{relCount}(m)}$, $\frac{\text{relCount}(m)}{\text{relCount}(M)}$]
end
simPos($M$, $m$, $\text{Trace}_S$, $\text{Trace}_T$)
begin
    relPos($M$)[$i$] = relative positions of $M$ in $\text{Trace}_S$
    relPos($m$)[$i$] = relative positions of $m$ in $\text{Trace}_T$
    avgRP($M$) = average of values in relPos($M$)[$i$]
    avgRP($m$) = average of values in relPos($m$)[$i$]
    if (the values of relPos($M$)[$i$] are within one standard deviation of
    avgRP($M$) and likewise for relPos($m$)[$i$]) then Return min[$\frac{\text{avgRP}(M)}{\text{avgRP}(m)}$,
    $\frac{\text{avgRP}(m)}{\text{avgRP}(M)}$]
    else Return UNDECIDED
end
simName($M$, $m$)
begin
    if ($M$ and $m$ are individual methods) then Return LEVENSHTEIN.RATIO($M$, $m$)
    else Return UNDECIDED
end
simCtxt($M$, $N$, $m$, $n$, $\text{Trace}_S$, $\text{Trace}_T$)
begin
    relCount($M$, $N$) = $\frac{\#\text{Occurences of } M \text{ in preceding context of } N \text{ in Trace}_S}{\text{Length}(\text{Trace}_S)}$
    relCount($m$, $n$) = $\frac{\#\text{Occurences of } m \text{ in preceding context of } n \text{ in Trace}_T}{\text{Length}(\text{Trace}_T)}$
    relCount($N$, $M$) = $\frac{\#\text{Occurences of } N \text{ in preceding context of } M \text{ in Trace}_S}{\text{Length}(\text{Trace}_S)}$
    relCount($n$, $m$) = $\frac{\#\text{Occurences of } n \text{ in preceding context of } m \text{ in Trace}_T}{\text{Length}(\text{Trace}_T)}$
    if  ApproxMatch(relCount($M$, $N$), relCount($m$, $n$)) and
    ApproxMatch(relCount($N$, $M$), relCount($n$, $m$)) then Return HIGH, else
    Return (1 - HIGH).
end
```

Figure 12. Algorithm 2, subroutines needed by Algorithm 1

The main challenge in directly realizing this intuition within an inference tool is that the Boolean formula underlying J is unknown (for if it were known, then any satisfying assignment to it would directly yield an API mapping!). As a result, the joint probability distribution J cannot be explicitly computed. However, the attributes described in §II determine the conditions that are likely to influence J.Thus, we formalize each of these attributes as factors, and estimate the joint probability distribution as the product of these factors. Of course, these factors are not comprehensive, i.e., there may be other factors that influence the value of J. Rosetta can naturally accommodate any new factors; they are simply treated as additional factors in the product, and passed to the factor graph solver. Rosetta's trace analysis computes four families of factors, one each for the four attributes. It combines them and uses them for probabilistic inference of likely mappings as shown in Algorithm 1.

1. Call frequency (ffreq)

The intuition underlying this factor is that if M maps to m (where M and m are individual methods or method sequences), then the frequency with which they appear in functionally similar traces might match. Thus, we compute the relative count of M and m as the number of times that they appear, normalized by the corresponding trace length. We then use the ratio of relative counts of M and m to compute ffreq. This is described in the subroutine simCount, shown in Algorithm 2.

2. *Call position (fpos)*

We observed in our experiments that certain API methods and sequences appear only at specific positions in the trace. For example, API methods that initialize the screen or game state appear only at the beginning of the trace. To identify such methods and sequences, we use a similarity metric that determines the relative position of the appearance of the method call or call sequence in the trace, i.e., its offset from the beginning of the trace, normalized by the trace length. Of course, there may be multiple appearances of the method call or sequence in the trace, so we average their relative positions. In this factor, we restrict ourselves only to calls and sequences that are localized in a certain portion of the trace, i.e., if the relative positions are not within one standard deviation of the average, this factor does not contribute positively or negatively to the likelihood of the mapping (undecided is a probability of 0.5). This is described in the subroutine simPos in Algorithm 2.

3 *Method names (fname)*

We use the names of methods in the source and target APIs to determine likely mappings. Unlike the other factors, which are determined by trace structure (i.e., program behavior), this is factor relies on a syntactic feature. The simName subroutine in Algorithm 2 uses a ratio based upon the Levenshtein edit distance, computed using a standard Python library [2]. This ratio ranges from 1 for identical strings, to 0 for strings that do not have a common substring. simName only returns a valid ratio for individual methods; for sequences, it returns undecided.

4 *Call context (fctxt)*

We define the context of a method call $A()$ in a trace as the set of method calls that appear in the vicinity of $A()$. Likewise, the context of a sequence $A();B()$ is the set of method calls that appear in the vicinity of this sequence (if it exists in the trace). Considering context allows us to propagate beliefs about likely mappings. Recall the example presented in chapter III, where considering the frequency of the method calls $A(), B(), a(),$ and $b()$ alone does not allow precise inference of mappings. In that example, the context of the calls allows us to infer that if X_b^A is true, and then X_a^B is unlikely to be true. Of the four factors that we consider, fctxt is the only one that relates pairs of random variables; the others assign probabilities to individual random variables. We define the context of a method call or sequence M in the trace as the set of method calls and sequences that appear within a fixed distance k of M in the trace; in our prototype,

$k = 4$. When computing the context of M, we also consider whether the entities its context precede M or follow M. To compute context as a factor, we use the function simCtxt, which considers all pairs of methods and method sequences (M, N) that appear in the source trace, and all pairs (m, n) that appear in the target trace. We then count the number of times M appears in the preceding context of N in the source trace (i.e., within $k = 4$ calls preceding each occurrence of N) and normalize this using the trace length $\big(relCount(M, N)\big)$; likewise we compute $relCount(N, M)$, and the corresponding metrics for the target trace. We then check whether $relCount(M, N)$ "matches" $relCount(m, n)$, and $relCount(N, M)$ "matches" $relCount(n, m)$ We do not require the relative counts to match exactly; rather their difference should be below a certain threshold (10% in our prototype); $ApproxMatch$ encodes this matching function. If both the counts match, then the factor $fctxt(X_m^M , X_n^N)$ positively influences the inference that if X_m^M is $true$, then X_n^N is also $true$, and vice-versa. The simCtxt function ensures this by returning a high probability value. Likewise, if the counts do not match, $fctxt(X_m^M , X_n^N)$ would indicate that X_m^M and X_n^N are unlikely to be true simultaneously. Note that this does not preclude X_m^M or X_n^N from being $true$ individually. Intuitively, the Boolean interpretation of $fctxt(X_m^M , X_n^N)$ is $(X_m^M \wedge X_n^N)$. In our prototype, we set the value of high as 0.7. We conducted a sensitivity study by varying the value of high between 0.6 and 0.8, and observed that it did not significantly change the set of likely mappings output by Rosetta Mapper. To illustrate the context factor, consider again the example from §III. There, $simCtxt(A, B, a, b)$ would be high, while $simCtxt(A, B, b, a)$ would be 1-high (using exact matches for $relCount$ instead of $ApproxMatch$, to ease illustration). Therefore, we can infer that X_a^A and X_b^B could both be $true$, but that X_b^A and X_a^B are unlikely to be true simultaneously. We implemented Rosetta Mapper's trace analysis and factor generation algorithms in about 1300 lines of Python code. We used the implementation of factor graphs in the Bayes Net Toolbox (BNT) [23] for probabilistic inference.

Rosetta generates one factor for each Boolean X_m^M for each of the three factor families ffreq, fpos, fname. Letting S and T denote the number of unique source and target API calls observed in the trace, there are $O(S^2T^2)$ such Boolean variables (because M and m include individual methods and method sequences of length two). Likewise, Rosetta Mapper generates two fctxt factors for each pair (M, N) and (m, n), resulting in a total of $O(S^4T^4)$ factors. We restricted Rosetta to work with method sequences of lengths one and two because of the rapid growth in the number of factors. Future work could consider optimizations to prune the number of factors, thereby allowing inference of mappings for longer length method sequences.

Combining Inferences across Traces:

As discussed so far, we apply probabilistic inference to each trace pair, which results in different values of $Pr[X_m^M = true]$, for each Boolean variable X_m^M . In this step, we combine these inferences across the entire database of trace pairs. One way to combine these probabilities is to simply average them. However, if we do so, we ignore the confidence that we have in our inferences from each trace pair. Intuitively, the more occurrences we see of a method call or sequence M in a source trace, the more confidence we have in the values of $Pr[X_m^M = true]$. Therefore, we compute a weighted average of these probabilities, with the relative count of each source call as the weight.

$$Pr[X_m^M = true] = \frac{\sum_{Traces} relCount(M) \times \ Pr[X_m^M = true]}{\sum_{Traces} relCount(M)}$$

We chose this approach because of its modularity. As we collect more trace pairs and inferences from them, we can combine them with mappings inferred from other traces in a straightforward way using the weighted average approach. Alternatively, we could have chosen to concatenate individual traces (in the same order for both components of each trace pair) to produce a "super-trace," and perform probabilistic inference over this super-trace. However, if we do so, then we would have to reproduce the super-trace after each new trace pair that is collected, and execute the inference algorithm over the ever-growing super-trace. This approach is neither memory-efficient nor time-efficient. In contrast, our weighted average approach provides more modular support to add inferences from new trace pairs as they become available. This weighted average is presented to the user as the output from Rosetta. We present the output of Rosetta to the user in terms of the inferred mappings for each source API call. For each source API method or method sequence, we present a list of mappings inferred for it, ranked in decreasing order of the likelihood of the mapping.

4.2.2 Rosetta Classifier

In order to build a classifier which would differentiate between a true mappings and false mappings, we obtain the relevant features as discussed before. Each of the features describes the mapping. The features are based on structural properties of the traces obtained for the source and the target platform. In Rosetta mapper we calculate the factors for every pair of traces, obtain the mapping per trace pair using factor graph approach. Finally we combine the mappings obtained from all the pairs based on weight average method. In contrast the features of the Rosetta classifier are computed for all the trace pairs at once. In Rosetta mapper the mappings are generated for each pair of traces containing J2ME and android API method calls only from that pair of trace files. But in Rosetta classifier the mappings are generated from all the available pairs of trace files and it contains the all possible J2ME and Android calls from all possible trace files. For example Let $S_i = \{A_i , B_i , C_i , \dots\}$ $and\ T_i = \{a_i , b_i , c_i , \dots\}$ denote the sets of methods in the source and target APIs, respectively for trace pair $i.$ $Let\ S_j = \{A_j , B_j , C_j , \dots\}$ $and\ T_j = \{a_j , b_j , c_j , \dots\}$ denote the sets of methods in the source and target APIs, respectively for trace pair j. While the Rosetta mapper computes mapping of the form $X_{a_i}^{A_i}$ and $X_{a_j}^{A_j}$, the Rosetta classifier looks across the trace files and computes mapping of the form $X_{a_j}^{A_i}$ thus the mappings obtained from the classifier would consists of all the possible J2me method calls mapped to all possible Android API method calls. Thus in order to compute the features value for the Rosetta classifier, we would take all the possible unique J2me and Android calls present in all the trace files. Then for each J2me method, we map it to every possible Android method present in the android trace files. For each such mappings obtained, we calculate feature values which are computed across all the possible pairs of trace files. Thus the feature values obtained for every mapping is a summary of values obtained across trace files. The feature values tend to be indirect description of the mapping in hand rather than a direct metric such as frequency factor obtained for a single pair of traces as followed by the Rosetta Mapper. We shall now discuss in detail the calculation of the features vectors for each of the mapping.

Feature Extraction:

Mutual Information on Frequency:

Mutual information of two random variables X and Y is a quantity that measures the mutual dependence between them. Mutual information measures the information that X and Y share. We compute the relative frequency of the API method in every pair of trace files for both platform and for every possible unique API methods. Thus each method will have array of relative frequency values which corresponds to each of the available trace pairs. Now the random variable X and Y considered to find the mutual information are the relative frequencies of the API methods in J2ME and Android platform in the trace files.

```
Input : set of trace pairs $\{(Trace_{S^i}, Trace_{T^i})\ \ldots\}$
Output : Mutual information on frequency $MI\_freq(X_m^M)$,
$M$ and $m$ are methods in $Trace_{S^i}$ and $Trace_{T^i}$ respectively.
$U_S$ and $U_T$ represent the set of all unique methods appearing in
traces $Trace_{S^i}$ and $Trace_{T^i}$ respectively.
Random variable $J^M$ denote the relative frequency of a $M$ in $Trace_S$.
Random variable $A^m$ denote the relative frequency of a $m$ in $Trace_T$.
$x$ and $y$ denotes all possible relative frequency values.
$MutualInfo\_freq$
Begin:
    $f(M)$=$Findfreq\_source()$;
    $f(m)$=$Findfreq\_target()$;
    Prob($J^M$)=$Probfreq\_source()$;
    Prob($A^m$)=$Probfreq\_target()$;
  foreach ( $M \in U_S$ ) do
    foreach ( $m \in U_T$ ) do
        sum = 0;
        foreach possible $x$ (0.00 to 1) do
            foreach possible $y$ (0.00 to 1) do
                Prob($J^M$)($A^m$)[$x$][$y$] = $\frac{count\ of\ (J^M = x)\ and\ (A^m = y)\ in\ trace\ pairs}{Total\ number\ of\ trace\ pairs}$

                sum=sum + Prob($J^M$)($A^m$)[$x$][$y$] $\times$ log$\left\{\frac{\text{Prob}(J^M)(A^m)[x][y]}{\text{Prob}(J^M)[x]\times\text{Prob}(A^m)[y]}\right\}$
        $MI\_freq(X_m^M)$=sum;
  return $MI\_freq(X_m^M)$
End.
```

Figure 13. Mutual Information on frequency from Trace Files

```
$Findfreq\_source$
Begin:
  foreach ( $M \in U_S$ ) do
    foreach trace ($Trace_{S^i}$) do
      f($M$)[$i$] = $\frac{\#Occurence\ of\ M\ in\ Trace_{S^i}}{Length(Trace_{S^i})}$
  return f($M$);
End.

$Probfreq\_source$
Begin:
   $f(M)$=$Findfreq\_source$();
  foreach ( $M \in U_S$ ) do
    foreach $x$ (0.00 to 1)
      Prob($J^M$)[$x$] = $\frac{count\ of\ (J^M = x)}{Total\ number\ of\ trace\ files}$
  return Prob($J^M$);
End.

$Findfreq\_target$
Begin:
  foreach ( $m \in U_T$ ) do
    foreach trace ($Trace_{T^i}$) do
      $f(m)[i]$ = $\frac{\#Occurence\ of\ m\ in\ Trace_{T^i}}{Length(Trace_{T^i})}$
  return $f(m)$;
End.

$Probfreq\_target$
Begin:
   $f(m)$=$Findfreq\_target$();
  foreach ( $m \in U_T$ ) do
    foreach $y$ (0.00 to 1)
      Prob($A^m$)[$y$] = $\frac{count\ of\ (A^m = y)}{Total\ number\ of\ trace\ files}$
  return Prob($A^m$);
End.
```

Figure 14. Mutual information routines

$P(X = x)$ is the probability that the API method of J2ME exhibits a relative frequency of x among the given N pairs of traces. $P(Y = y)$ is the probability that the API method of Android exhibits a relative frequency of y among the given N pairs of traces.

$P(X = x, Y = y)$ is the joint probability that J2ME API exhibits a relative frequency of x and the Android API exhibits a relative frequency of y divided by the total number of trace files. Thus we compute for each possible value of the relative frequency namely from 0.00 to 0.99 we compute the mutual information which is given by the following formula.

$$I(X;Y) = \sum_{y \in Y} \sum_{x \in X} p(X = x, Y = y) \log\left(\frac{p(X = x, Y = y)}{p(X = x)p(Y = y)} \right)$$

Example:

Let us consider two pairs of trace files

$S_1 = \{A, B, A, C\}, T_1 = \{b, a, b, c\}$ &

$S_2 = \{A, B, B, C\},\ T_2 = \{b, a, a, c\}$

Random variable J^M denote the relative frequency of a J2ME method M in trace. Similarly, Random variable A^m denote the relative frequency of a Android method m in trace. For the given example S1 has following relative frequency distribution,

S1: J^A =0.5, J^B =0.25, J^C =0.25

S2: J^A =0.25, J^B =0.5, J^C =0.25

Similarly for the android file we have,

T1: A^a =0.25, A^b =0.5, A^c =0.25

T2: A^a =0.5, A^b =0.25, A^c =0.25

The possible mappings are

$$A \leftrightarrow a, A \leftrightarrow b, A \leftrightarrow c, B \leftrightarrow a, B \leftrightarrow b, C \leftrightarrow a, C \leftrightarrow b \ and \ C \leftrightarrow c.$$

C_p^M denote the *count* of relative frequency of api method M takes a probability value p among the available traces.

For J2ME methods:

$C_{0.25}^A$= 1; $C_{0.5}^A$= 1 , $C_{0.25}^B$= 1 ; $C_{0.5}^B$= 1 & $C_{0.25}^C$= 2

For Android methods

$C_{0.25}^a$= 1; $C_{0.5}^a$= 1 , $C_{0.25}^b$= 1 ; $C_{0.5}^b$= 1 & $C_{0.25}^c$= 2

Now dividing these count values by the total number of traces, we get the Probability of the random variable (Relative frequency of the API method) taking a particular value.

For J2ME API method the probability of the random variable J^M taking a value p is given by

$$P(J^M = p) = C_p^M / N$$

Where N is the total number of traces.

For android method m the probability of the random variable A^m

taking a value p is given by

$$P(A^m = p) = C_p^m / N$$

For each possible mapping J2ME method M and Android method m, $M \leftrightarrow m$:

For each p_j value 0.0 to 0.99 taken by the random variable J^M

For each p_a value 0.0 to 0.99 taken by the random variable A^m

We count the number of times J^M $takes\ value\ p_j\ and\ A^m\ takes\ value\ p_a$.

$$C_{A^m = p_a}^{J^M = p_j}$$

The above denotes the number of times the random variables J^M $takes\ p_j\ and\ A^m\ takes\ p_a$. Each of the random variables J^M and A^m take one value per trace pair $\{S_i , T_i\}$.

Example For the pair $\{S_1 , T_1\}$ in the given example, J^A =0.5, A^a =0.25

We divide this number by the total number of traces. We get the joint probability distribution given by:

$$P(J^M = p_j , A^m = p_a) = C_{A^m = p_a}^{J^M = p_j} / N$$

Now the Mutual Information for the mapping J2ME method M, Android method $m\ is\ denoted\ by\ M \leftrightarrow m$ is given by:

$$\mathrm{I}(J^M : A^m) = \sum_{p_j = 0.0}^{0.99} \sum_{p_a = 0.0}^{0.99} P(J^M = p_j , A^m = p_a) \log \left(\frac{P(J^M = p_j , A^m = p_a)}{P(J^M = p_j) P(A^m = p_a)} \right)$$

Mutual Information on Position:

The given trace files are divided into 100 positions. We find the relative frequency of each API method for each trace pair in every position for both the platforms. Thus we get 100 mutual information on frequency values for each position. We find the average of the mutual information over 100 values algorithm. We follow a same algorithm to the mutual information calculation for each possible position value. For each position $k = 0.00\ to\ 0.99$ Corresponding positional mutual information is given by

$$I_k(J^M{:}\,A^m) = \sum_{p_j=0.0}^{0.99} \sum_{p_a=0.0}^{0.99} P(J^M = p_j\,, A^m = p_a\,) \log\left(\frac{P(J^M=p_j\,, A^m=p_a\,)}{P(J^M=p_j\,)P(A^m=p_a\,)} \right)$$

We then average the mutual information values for all position values to find the Mutual information on position feature.

$$I_{pos}(J^M{:}\,A^m) = \sum_{k=0}^{99} I_k(J^M{:}\,A^m)/100$$

Thus the mutual Information obtained is average of all the position wise mutual information. We compute mutual information for every possible mapping across all of the given traces at once. Thus whenever there is a new trace added to the collection of traces, the corresponding features mutual information has to be recomputed and it takes time.

Digrams:

This feature would gauge how good the API calls interacts with other API calls in the trace file. This is similar to the context factor of the factor graph approach except that instead of the vicinity parameter we look for $B()$ at the entire trace file for the context of a given API method $A()$. For every trace file $S_i = \{A_i, B_i, C_i, \dots\}$ $and\ T_i = \{a_i, b_i, c_i, \dots\}$, where $i = \{1, \dots, N\}$ where N represents the number of trace pairs available . We collect the all possible unique J2ME API call $\{A(), B(), \dots\}$ and all possible Android calls $\{a(), b(), \dots,\}$.We obtain all possible mappings X_m^M where M represents the J2ME call while m represents the Android call. Consider every possible mapping pairs X_m^M $and\ Y_n^N$ where N and M represent J2ME calls while $m\ and\ n$ represents android calls given $N \neq M\ and\ n \neq m.$ Now for each tracing pair S_i and T_i. We convert the trace file into digram file where the API calls other than $N\ and\ M$ are replaced with 'e' from

J2ME file and API calls other than $n\ and\ m$ are replaced with 'e'.

```
Input : set of trace pairs $\{(Trace_S^i, Trace_T^i)\ ...\}$
Output : Digram scores of all the mappings pairs.
$M1$ and $M2$ are two distinct methods in appearing in
traces $Trace_S$ and $Trace_T$ respectively.
$U_S$ and $U_T$ represent the set of all unique methods appearing in
traces $Trace_S$ and $Trace_T$ respectively.
$Digram$
Begin:
  foreach ( $M1 \in U_S$ ) do
    foreach ( $m1 \in U_T$ ) do
      foreach ( $M2 \in U_S$ ) do
        foreach ( $m2 \in U_T$ ) do
          sum = 0;
          foreach $trace\ pair\ i$ do
            sum=sum + $Dscore(M1, M2, m1, m2, i, Trace_S^i, Trace_T^i)$;
          $digram(X_{m1}^{M1}, X_{m2}^{M2}) = \frac{sum}{Total\ number\ of\ trace\ pairs}$

  return $digram$
End.
```

Figure 15. Digram feature for all pair of mappings

Thus we have the digram files $\hat{S} = \{N, e, e, M, e, e, N, N, \dots,\}\ and\ \hat{T} = \{n, e, e, m, e, e, n, n, \dots,\}$ respectively. We count the number of $\{N, N\}, \{N, M\}, \{M, M\}, \{M, N\}, \{N, e\}, \{M, e\}, \{e, N\}, \{e, M\}$ for the J2ME trace files. Now for the Android Trace file, we compute the number of $\{n, n\}, \{n, m\}, \{m, m\}, \{m, n\}, \{n, e\}, \{m, e\}, \{e, n\}, \{e, m\}$ for the android file.

```
Input : $mappings$ pair $M1 \leftrightarrow m1$ $and$ $M2 \leftrightarrow m2$ , $trace$
 $files$ $Trace_S^i$ , $Trace_T^i$
Output : Digram score the mappings pair.
$Digram$
Begin:
Score=0;
$Dtrace_S^i$= replace methods other than $M1$ $and$ $M2$ in $Trace_S^i$ with $e$
$Dtrace_T^i$= replace methods other than $m1$ $and$ $m2$ in $Trace_T^i$ with $e$
$CM1\_M2$=count of "$M1M2$" in $\mathrm{Trace}_S^i$;
$CM2\_M1$=count of "$M2M1$" in $\mathrm{Trace}_S^i$;
$CM1\_e$=count of "$M1e$" in $\mathrm{Trace}_S^i$;
$Ce\_M1$=count of "$eM1$" in $\mathrm{Trace}_S^i$;
$CM2\_e$=count of "$M2e$" in $\mathrm{Trace}_S^i$;
$Ce\_M2$=count of "$eM2$" in $\mathrm{Trace}_S^i$;
$Cm1\_m2$=count of "$m1m2$" in $\mathrm{Trace}_T^i$;
$Cm2\_m1$=count of "$m2m1$" in $\mathrm{Trace}_T^i$;
$Cm1\_e$=count of "$m1e$" in $\mathrm{Trace}_T^i$;
$Ce\_m1$=count of "$em1$" in $\mathrm{Trace}_T^i$;
$Cm2\_e$=count of "$m2e$" in $\mathrm{Trace}_T^i$;
$Ce\_m2$=count of "$em2$" in $\mathrm{Trace}_T^i$;
If ($CM1\_M2 == Cm1\_m2$) Score++;
If ($CM2\_M1 == Cm2\_m1$) Score++;
If ($CM1\_e == Cm1\_e$) Score++;
If ($Ce\_M2 == Ce\_m2$) Score++;
If ($CM2\_e == Cm2\_e$) Score++;
If ($Ce\_M1 == Ce\_m1$) Score++;
 return $Score$;
End.
```

Figure 16. Digram feature for all pair of mappings

For mappings $X_m^M \quad and\ Y_n^N$ we compare the matching number of

$\{N, N\}\ with\ \{n, n\}, \{N, M\}\ with\ \{n, m\}, \{M, N\}\ with\ \{m, n\}$,

$\{M, M\}\ with\ \{m, m\}, \{N, e\}\ with\ \{n, e\}, \{M, e\}\ with\ \{m, e\},$

$\{e, N\}\ with\ \{e, n\}\ and\ \{e, M\}\ with\ \{e, m\}.$ For each match we give a score of 1 and thus the max score is 8. We average the scores across the number of trace pairs across every mapping pairs Y_n^N and store it for each $mapping\ X_m^M$ for the array of all possible mappings obtained from before. Thus we compute the digram feature which measures how the interaction is among the j2me and the android API methods across trace files.

Training Data:

After obtaining the feature values for each of the above mentioned features for every mapping, we collect training data required for the classifier to perform classification. This is basically deciding whether the given mapping is true or false. We go through the documentation for the J2ME and Android namely java docs and android docs available online.

Classification:

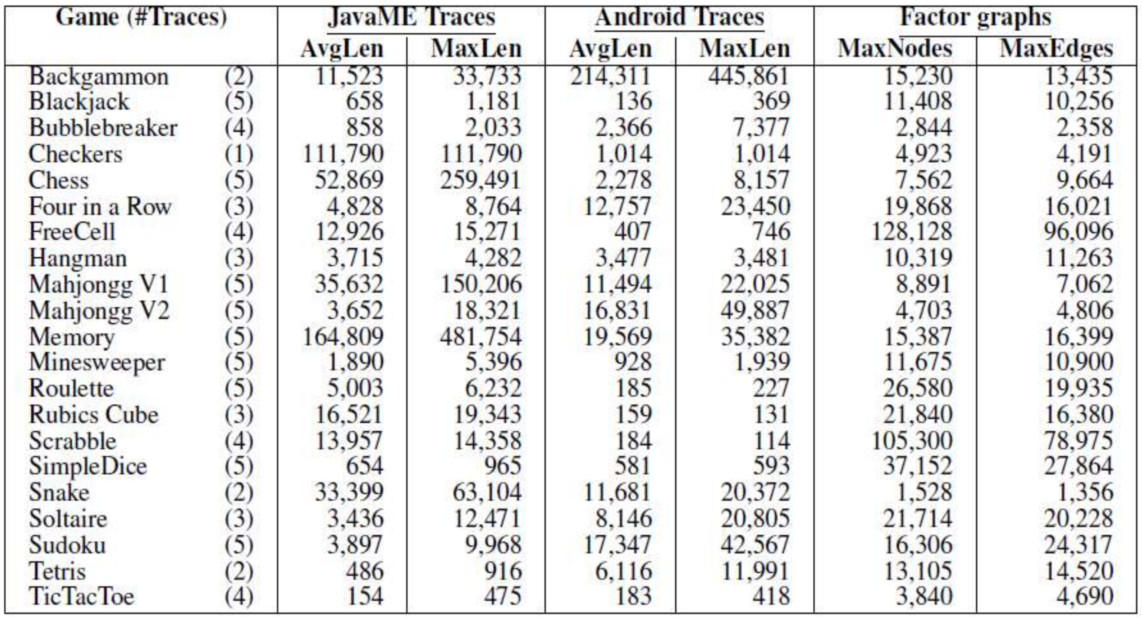
We implemented Rosetta Classifier's trace analysis and feature generation algorithms in about 2000 lines of Perl code. We use Support Vector machines implemented in R, a statistical programming language with data analytics. R provides a wide variety of statistical and graphical techniques, including linear and nonlinear modeling, classical statistical tests, time-series analysis, classification, clustering, and others. Once we obtain the training data, we divide them into training (two-thirds) and testing data (one-third). We run SVM routines implemented in $klaR, kernlab, e1071$ modules of R package. We supply the data to SVM and get the parameters defining the hyper plane separating the true and false mappings. We vary the different available set of kernels and various parameters of each kernel type to find the best match for the classifier. We compute the confusion matrix of {True Positives, True negatives, false positives, false negatives} to decide the best possible SVM machine to classify the given data points. We also tried Probability min max machine in MATLAB besides the regular linear and logistic regression routines of R.

Chapter 5

Evaluation

5.1 Methodology

To evaluate Rosetta, we collected a set of 21 JavaME applications for which we could find functionally-equivalent counterparts in the Android market. In particular, we chose board games, for two reasons. First, many popular board games are available for both the JavaME and Android platforms. Checking the functional equivalence of two games is as simple as playing the games and ensuring that the moves of the game are implemented in the same way on both versions. Second, the use of board games also eases the task of collecting trace pairs. Moves in board games are easy to remember and can be repeated on both game versions to produce functionally-equivalent traces on both platforms.

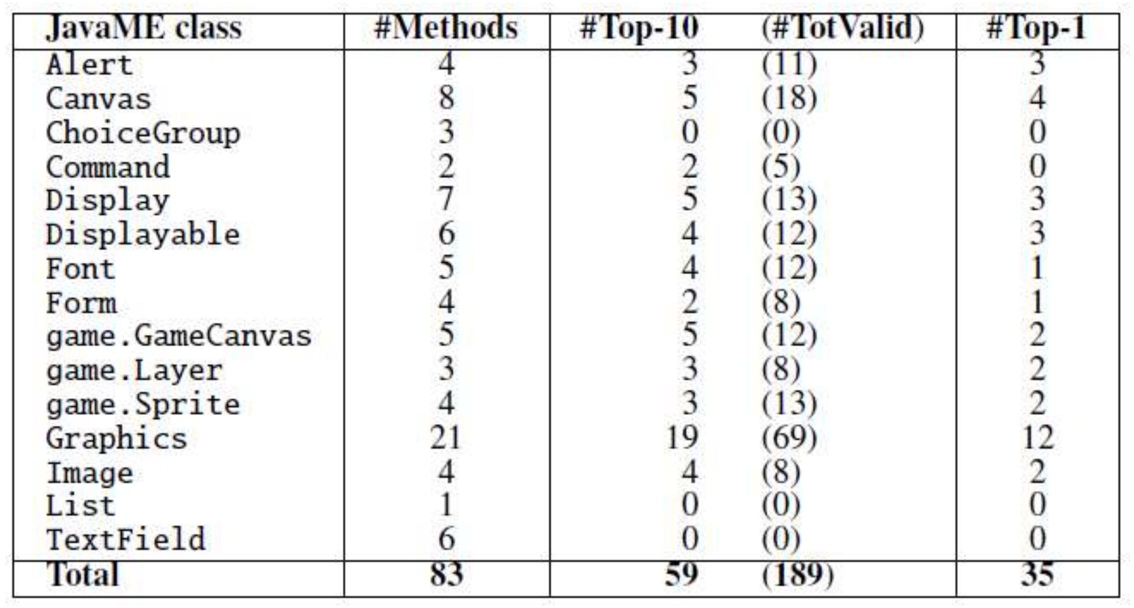


Game (#Traces)		JavaME Traces		Android Traces		Factor graphs	
		AvgLen	MaxLen	AvgLen	MaxLen	MaxNodes	MaxEdges
Backgammon	(2)	11,523	33,733	214,311	445,861	15,230	13,435
Blackjack	(5)	658	1,181	136	369	11,408	10,256
Bubblebreaker	(4)	858	2,033	2,366	7,377	2,844	2,358
Checkers	(1)	111,790	111,790	1,014	1,014	4,923	4,191
Chess	(5)	52,869	259,491	2,278	8,157	7,562	9,664
Four in a Row	(3)	4,828	8,764	12,757	23,450	19,868	16,021
FreeCell	(4)	12,926	15,271	407	746	128,128	96,096
Hangman	(3)	3,715	4,282	3,477	3,481	10,319	11,263
Mahjongg V1	(5)	35,632	150,206	11,494	22,025	8,891	7,062
Mahjongg V2	(5)	3,652	18,321	16,831	49,887	4,703	4,806
Memory	(5)	164,809	481,754	19,569	35,382	15,387	16,399
Minesweeper	(5)	1,890	5,396	928	1,939	11,675	10,900
Roulette	(5)	5,003	6,232	185	227	26,580	19,935
Rubics Cube	(3)	16,521	19,343	159	131	21,840	16,380
Scrabble	(4)	13,957	14,358	184	114	105,300	78,975
SimpleDice	(5)	654	965	581	593	37,152	27,864
Snake	(2)	33,399	63,104	11,681	20,372	1,528	1,356
Soltaire	(3)	3,436	12,471	8,146	20,805	21,714	20,228
Sudoku	(5)	3,897	9,968	17,347	42,567	16,306	24,317
Tetris	(2)	486	916	6,116	11,991	13,105	14,520
TicTacToe	(4)	154	475	183	418	3,840	4,690

Figure 17. Trace Stats

To collect traces, we ran JavaME games using the emulator distributed with the Sun Java Wireless Toolkit, version 2.5.1 [26].

For Android games, we used the emulator that is distributed with the Android 2.1 SDK. Fig. 16 shows the games that we used, the number of trace pairs that we collected for each game, and the sizes of the traces for the JavaME and Android versions. We ran Rosetta on these traces to obtain a set of likely mappings. This set is presented to the user as a ranked list of mappings inferred for each JavaME method (or method sequence). To evaluate the list associated with each JavaME method, we consulted the documentation of the JavaME and Android graphics APIs to determine which members of the list are valid mappings.



JavaME class	#Methods	#Top-10	(#TotValid)	#Top-1
Alert	4	3	(11)	3
Canvas	8	5	(18)	4
ChoiceGroup	3	0	(0)	0
Command	2	2	(5)	0
Display	7	5	(13)	3
Displayable	6	4	(12)	3
Font	5	4	(12)	1
Form	4	2	(8)	1
game.GameCanvas	5	5	(12)	2
game.Layer	3	3	(8)	2
game.Sprite	4	3	(13)	2
Graphics	21	19	(69)	12
Image	4	4	(8)	2
List	1	0	(0)	0
TextField	6	0	(0)	0
Total	**83**	**59**	**(189)**	**35**

Figure 18. Results of applying Rosetta Mapper

Results of applying Rosetta to the traces obtained from the games shown in Fig. 17. We have shown the number of unique JavaME methods (categorized by class) for which Rosetta inferred at least one valid mapping in the top ten (#Top-10), and the total number of valid mappings found in the top ten (#TotValid). Also shown is the number of JavaME calls for which Rosetta's top ranked inference was a valid mapping (#Top-1).

5.1.1 Rosetta Mapper

Fig. 18 presents the results of running Rosetta mapper on the traces that we collected. This figure shows the number of distinct JavaME methods that we observed in the trace, grouped by the parent JavaME class to which they belong. Thus, for example, there were four unique JavaME methods belonging to the Alert class in our traces, namely $Alert.setCommandListener(), Alert.setString(), Alert.setType(), and\ \ Alert.setTimeout()$. In all, there were 83 unique JavaME methods in our traces. For each of these 83 JavaME methods, we determined whether the top ten ranked mappings reported for that method contained a method (or method sequence) from the Android API that would implement its functionality. As reported in Fig. 18, we found such valid mappings for 59 of these JavaME methods (71%). Fig. 19 depicts in more detail the rank distribution of the first valid mapping found for each of these 59 JavaME methods. As this figure shows, the top-ranked mapping for 35 of these methods was a valid one (42% of the observed JavaME methods). Recall that a JavaME method can possibly implemented in multiple ways using the Android API. Thus, the ranked list associated with that method could possibly contain multiple valid mappings. Rosetta's output contained a total of 189 valid mappings within the top ten results of the 59 JavaME methods. Fig. 20 depicts the rank distribution of all the valid mappings found by Rosetta Mapper. Below, we illustrate a few examples of mappings inferred by Rosetta Mapper:

(1) $The\ Graphics.clipRect()$ method in JavaME intersects the current clip with a specified rectangle. Rosetta correctly inferred the Android method Canvas.clipRect() as its top ranked mapping.

(2) In $JavaME\ Graphics.drawChar()$ draws a specified character using the current font and color. In Rosetta's output, the sequence $Paint.setColor(); Canvas.drawText()$, which first sets the color and then draws text, was the second-ranked mapping for $Graphics.drawChar()$.

(3) $The\ Graphics.drawRect()$ JavaME method was mapped to the Android methods $Canvas.drawRect()$ (rank 1) and $Canvas.drawLines()$ (rank 7), all of which can draw rectangles.

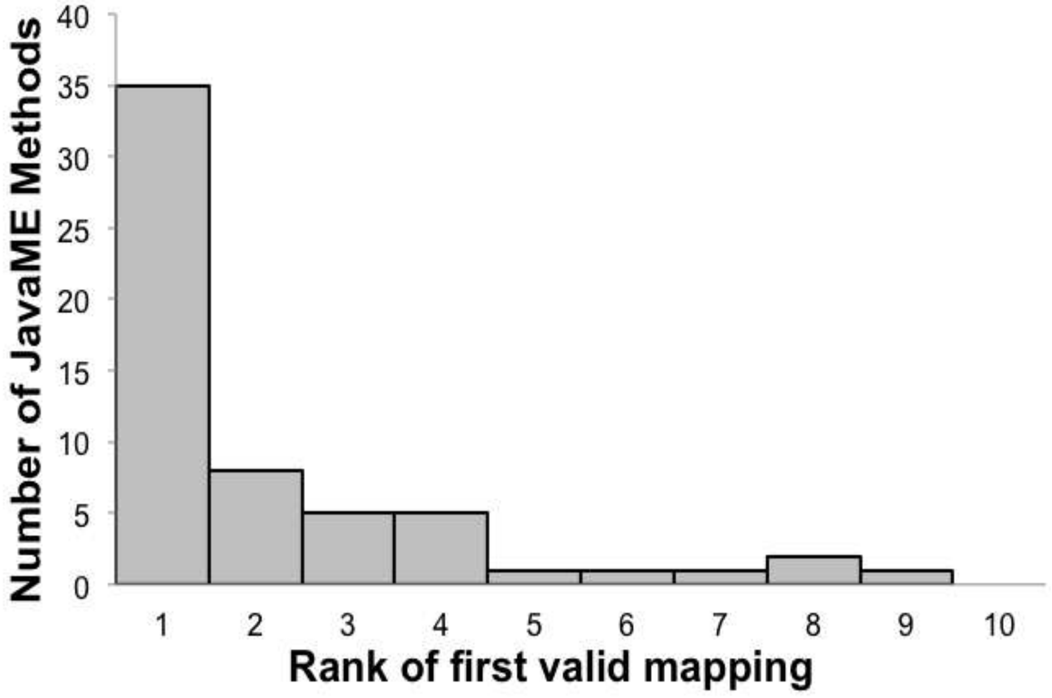


Figure 19. Rank distribution of first valid mappings

This figure shows the rank distribution of the first valid mapping found for each JavaME method. In all, for each of 59 JavaME methods, a valid Android mapping appeared within the top ten results reported for that method. For 35 JavaME methods, the top-ranked mapping was a valid one.

Impact of Individual Factors

Rosetta's output is the result of combining all the four factors discussed in IV. We also evaluated the extent to which each of these factors contributes to the output. To do so, we constructed factor graphs (using the traces in Fig.17) individually with ffreq, fpos, and fname. Because fctxt correlates two mappings, we did not consider it in isolation.

We also considered a few combinations of factors, to study how the mappings change as factors are added. We inferred mappings with these factor graphs, and compared the resulting mappings with those obtained by Rosetta. For each factor graph variant in Fig. 21, we report the fraction of Rosetta's 189 valid mappings that correspond to a valid mapping inferred by the Variant. As we did with Rosetta, we only report valid mappings that rank in the top ten in the output of these factor graph variants.

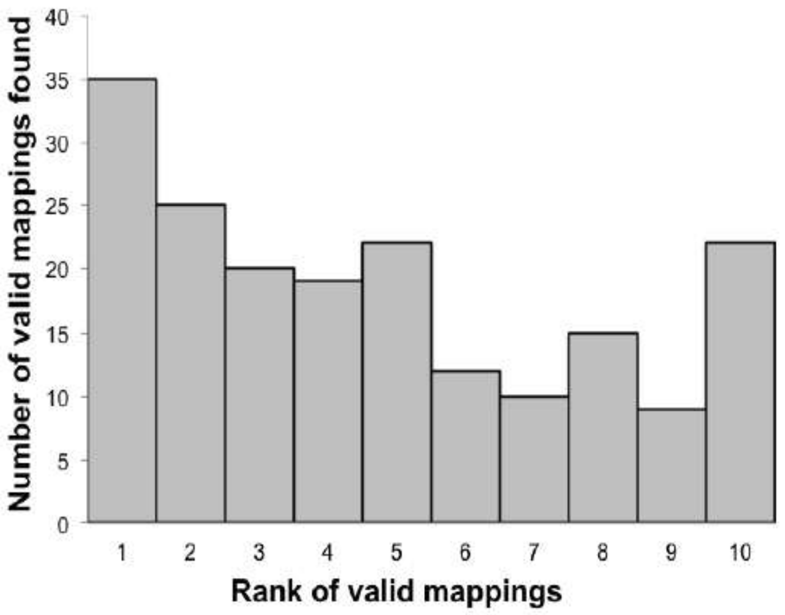
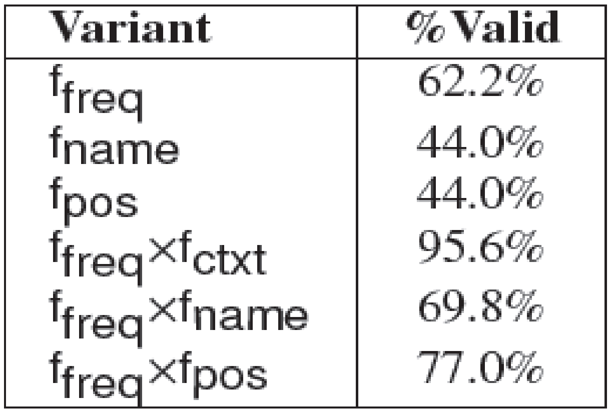


Figure 20. Rank distribution of all valid mappings

This figure shows the rank distribution of all valid mappings found by Rosetta. For each rank, it shows the number of valid mappings that appear at that rank across all JavaME API methods observed in our traces. In all, we found 189 valid mappings ranking in the top ten.

As Fig. 21 shows, the addition of more factors generally improves the accuracy of the reported mappings. This is because the addition of more Factors incorporates more trace features into the probabilistic inference algorithm, thereby removing spurious mappings from the top ten.



Variant	%Valid
f_{freq}	62.2%
f_{name}	44.0%
f_{pos}	44.0%
$f_{freq} \times f_{ctxt}$	95.6%
$f_{freq} \times f_{name}$	69.8%
$f_{freq} \times f_{pos}$	77.0%

Figure 21. Impact of factor graph variants

5.1.2 Rosetta Classifier

The nature of the features used by the Rosetta classifier is exhaustive. Each of the features is obtained for all possible unique combination of source and destination API calls. There is combinatorial explosion while generating the features for the Rosetta Classifier. Thus we started with just 10 games out of the available 21 games traced before. Each of the ten games had around 5 traces per platform. The total number of possible mappings obtained was around 86000. Out of which 7000 were chosen to form the classifier. These 7000 mappings were manually verified to be true or false after reading the documentation available online. We d ivided this dataset into training and testing data. The training data consisted of 259 true mappings and 2679 false mappings. The testing data consists of 134 true mappings and 1335 false mappings. The SVM built in R using the *Bessel* Kernel function was able to correctly classify 116 out 134 true mappings and 1240 out of 1335 false mappings. Recall *R* is the number of correctly classified positive examples divided by the total *number of actual positive examples* in the test set. Recall rate is 86% for our Rosetta Classifier. Precision *P* is the number of correctly classified positive examples divided by the total *number of examples that are classified as positive*. The Precision rate is 54%.

5.2 Runtime Performance

5.2.1 Rosetta Mapper

We ran Rosetta Mapper on a PC with an Intel Core2 Duo CPU, running at 2.80 GHz and 4GB memory, running Ubuntu 10.04. For each of the traces reported in Fig. 17, Rosetta took between 2–55 minutes to analyze traces and output mappings. On average, approximately 80% of this time was consumed by the algorithms in IV which produce factor graphs, and 20% was consumed by the factor graph solver. The solver consumed 2GB memory for the largest of our factor graphs. As discussed, Rosetta currently supports inference on method sequences of length up to two. We also configured Rosetta to work with longer method sequences, which would produce larger factor graphs. However, the factor graph solver ran out of memory in these cases.

5.2.2 Rosetta Classifier

We ran Rosetta Classifier on Amazon C1 High-CPU Extra Large EC2 instances. It had 20 ECUs, 8 cores, 7GB memory, Ubuntu Server 11.10 Operating system. Obtaining the features was the most time consuming part. Since the nature of the experiments on the Rosetta classifier takes every possible API method into consideration for calculating every possible mapping, the size of the data grew huge. Digram are calculated for every possible mapping with every other possible mapping and thus it is extremely high intense computation. As the number of traces increases, the possible number of J2ME methods and Android API methods increases. Thus the number of mappings vastly and consequently the digrams calculation is growing huge. But eventually the total number of mappings possible is only 281 * 3837. Hence after a significant number of traces, the increase in the number of traces doesn't affect the computation complexity. But still to compute Digrams for 281*3837 mappings with every other 281*3837-1 over the all the available traces is indeed computation intensive. The irony is that for data mining algorithms to work we need really huge amount of data and more the data, better the classifier. The problem is that even though we have big datasets from the experiments. The training data is obtained manually. Thus for the 10 games (and corresponding 50 trace files per platform) we have chosen for Rosetta Classifier, we have around 86000 mappings. In that huge set of possible mapping, We eventually obtained around 7000 training data which has decision value "true or false". There is no escape from manual efforts here as there is no ground truth which could be automated. But it does help to train classifier to choose better. The digram feature took around 10-12 days in these instances. At times process would crash for lack of memory.

The mutual information over 10 games of 50 traces for each of source and target platforms took around 3-4 days to generate the values for each combination. Obtaining the SVM did not take much time as we have narrowed down the dataset to 7000 mappings.

5.3 Experiments with Micro Emulator

There have been some recent efforts [1], [6], [10], [4] to allow the execution of legacy JavaME applications on the Android platform. MicroEmulator [1] is one such open-source tool that works on JavaME applications. It rewrites JavaME API calls in the input application with the corresponding Android API calls. The implementation of MicroEmulator therefore implicitly defines a mapping between the JavaME and Android graphics APIs. However, MicroEmulator does not translate the entire JavaME graphics API, and only allows JavaME applications with rudimentary GUIs to execute on Android devices. Nevertheless, the mappings that it does implement provide a basis to evaluate Rosetta. We considered a subset of five JavaME games from Fig. 17 that MicroEmulator could successfully translate (Chess, Minesweeper, Snake, Sudoku, and Tic-tac-toe), executed them, and collected the corresponding traces of JavaME API calls. We then repeated the same set of experiments on the MicroEmulator-converted versions of these applications and collected the corresponding traces of Android API calls, and fed these trace pairs to Rosetta. In our evaluation, we determined whether the API mappings inferred by Rosetta contained the mappings implemented in MicroEmulator. 8 Across the JavaME traces for these five games, We observed 18 distinct JavaME graphics API methods translated by MicroEmulator. In Rosetta Mapper’s output, we found at least one valid mapping for 17 of these JavaME methods within the top ten ranked results for the corresponding JavaME method. Rosetta only failed to discover an API mapping for the JavaME method Alert.setString().

Out of 17 JavaME methods with valid mappings, 8 appeared as top-ranked mappings for the corresponding JavaME methods. MicroEmulator also contains multiple mappings for several of the translated JavaME methods. For the 18 distinct methods in our traces, MicroEmulator contains a total of 26 mappings. Of these, Rosetta's output contained 20 valid mappings within the top ten results for the corresponding JavaME methods.

Chapter 6

Conclusion

6.1 Related Work

The work most directly related to ours is the MAM (mining API mappings) project [34].

MAM’s goal is the same as Rosetta’s, i.e., to mine software repositories to infer how a source API maps to a target API. The MAM prototype was targeted towards Java as the source API and C# as the target API. Despite sharing the same goal, MAM and Rosetta differ significantly in the approaches that they use, each with its advantages and drawbacks. To mine API mappings between a source and a target API, MAM relies on the existence of software packages that have been ported manually from the source to the target platform. For each such software package, MAM then uses static analysis and name similarity to “align” methods and classes in the source platform implementation with those of the target platform implementation. Aligned methods are assumed to implement the same functionality. MAM’s use of static analysis allows it to infer a large number of API mappings (about 25,800 mappings between Java and C#). It also allows MAM to infer likely mappings between arguments to methods in the source and target APIs, which Rosetta does not currently do. However, unlike Rosetta, MAM requires that the same source application be available on the source and target platforms. MAM’s approach of aligning classes and methods across two implementations of a software package does not allow the inference of likely API mappings if there are similar, but independently-developed applications for the source and target platforms. MAM’s approach is also limited in that it uses name similarity as the only heuristic to bootstrap its API mapping algorithm. In contrast, Rosetta uses a number of attributes combined together as factors, and can easily be extended to accommodate new similarity attributes as they are designed. Most importantly, while MAM uses a purely syntactic approach to discover likely API mappings, Rosetta's approach uses similarities in application behavior. Nita and Notkin [24] develop techniques to allow developers to adapt their applications to alternative APIs. They provide a way for developers to specify a mapping between a source and a target API, following which a source to source transformation automatically completes the transformations necessary to adapt the application to the target API. Rosetta can potentially complement this work by inferring likely mappings.

Androider [29] is a tool to reverse-engineer application GUIs. Androider uses aspect-oriented programming techniques to extract a platform-independent GUI model by inspecting in-memory representations of GUIs. This model can then be used to port GUIs across different platforms. While Androider provides a GUI for a target platform by analyzing GUIs of a source platform at runtime, Rosetta instead infers API mappings, and is not restricted to GUI-related APIs. Bartolomei et al. [7] analyzed wrappers between two Java GUI APIs and extracted common design patterns used by wrapper developers. They focused on mapping object types between the two APIs and identified the challenges faced by wrapper developers. Method mappings given by Rosetta can possibly be used along with their design patterns to ease the job of writing wrappers. A number of prior projects provide tool support to assist programmers working with large APIs (e.g., [22], [28], [32], [16], [35], [11], [33]). The programmer's time is often spent in determining which API method to use to accomplish a particular task. These projects use myriad techniques to develop programming assistants that ease the task of working with complex, evolving APIs. Rosetta is complementary to these efforts, in that it works with cross-platform APIs. Some of these prior efforts can potentially be used in conjunction with Rosetta, e.g., to determine suitable arguments to pass to target API methods [33]. A related line of research is on resources for API learning (e.g., [12], [8], [15], [30], [27]). These projects attempt to ease the task of a programmer by synthesizing API usage examples, evolution of API usage, and extracting knowledge from API documentation. Again, most of these techniques work on APIs on a single platform. Rosetta cross-platform approach and can possibly be used in conjunction with these techniques to facilitate cross-platform API learning. Finally, Rosetta mapper's probabilistic inference approach was inspired by other uses of factor graphs in the software engineering literature. Merlin [20] uses factor graphs to classify methods in Web applications as sources, sinks and sanitizers. Such specifications are useful for verification tools, which attempt to determine whether there is a path from a source to a sink that does not traverse through a sanitizer. To infer such specifications, Merlin casts a number of heuristics (e.g., the likelihood of a sanitizer not being placed between a source and a sink is low, it is unlikely for multiple sanitizers to be placed between a source and a sink) as factors, and uses belief propagation. Kremenek et al. [17] also made similar use of factor graphs to infer specifications of allocator and deallocator methods in systems code. Factor graphs are just one approach to mining specifications; a number of prior projects have considered other techniques for data mining and inferring belief for specification inference and bug detection (e.g., [13], [19], [21]). Future work could consider the use of these inference techniques in Rosetta as well.

6.2 Issues

6.2.1 Threats to Validity

There are a number of threats to the validity of our results, which we discuss now. First, although we attempted to find JavaME and Android games that are functionally equivalent, Differences do exist in their implementations. This is because JavaME is an older mobile platform that does not support as rich an API as Android; many Android calls do not even have equivalents in JavaME. Together with randomness that is inherent in certain board games (IV), this could result in traces which contain Android calls implementing functionality unseen in the source application. They may mislead the attributes used by our inference algorithm (e.g., frequency of calls), leading to both invalid mappings as well as valid mappings being suppressed in the output. Second, there is no “ground truth” of JavaME to Android mappings available to evaluate Rosetta’s output (the mappings in MicroEmulator aside), as a result of which we are unable to report standard metrics, such as precision and recall. We interpreted Rosetta’s results by consulting API documentation. Such natural language documentation is inherently ambiguous and prone to misinterpretation. We mitigated this threat by having two authors independently cross-validate the results.

Finally, a threat to external validity, i.e., the extent to which our results can be generalized, comes from the fact that we only inferred mappings using board games. It is unclear how many mappings we would have inferred using other graphical applications (e.g., office applications).

6.2.2 Randomness

Right from collecting the games to tracing them to produces “equivalent traces”, there is randomness in the process. Getting two same games of same look and feel for both J2me and android it is quite challenging. We downloaded 7 to 9 apks/jars for each game on an average. Then we match the closest possible games. For example, the game chess is available in both the J2me and Android platform. But the implementations of chess in both the platform vary vastly. While J2ME games are simple, primitive, blurred and has no sophistication, the android games were advanced, highly vibrant and produced clearer versions of the games with numerous features. The main difference between the games were that the J2Me games were predominantly keyboard oriented and android games were mostly mouse oriented but still some supported keys. Thus the number of steps required to reproduce the output state were different in the games. The following is the summary of the randomness we encountered and the possible solution we attempted in order to reduce the dynamic content in the trace file, while the tracing the games.

Table 1. Randomness in application pairs of JavaME and Android games

Si .No	GAMES	Randomness	Solution
1	Checkers, chess, Othello, Gomuku, Scrabble, Tic-tac-Toe, Analog stopwatch	There is not much of randomness as the games are always run as orchestrated 2 player games.	Nothing much.
2	Tetris	Due to the incoming brick shapes between two versions of games.	Traces are taken with same number line of bricks cleared in either version of the games.
3	Sudoku, kakuro	Due to the permutations of digits in Sudoku/kakuro grid.	Traces are taken with a same number of 3*3 grids / number of lines {vertical or horizontal} of digits filled.
4	Slots, bingo, roulette, wish wheel, poker, Blackjack	Due to the very nature of the game.	Games were played so as to equate the results of both the implementations. E.g. traces represents the trials till a jackpot is won, bingo is obtained or a blackjack is won in both the implementations. But still not every time it was possible. So we had to go through a run which ends in no profit, no loss, some profit, and some loss.
5	Solitaire, FreeCell, spider Cell	Due to the arrangement, permutation of cards.	games were played so as to equate the number of Aces open(free cell), number of matches possible(solitaire, FreeCell)
6	Slide	Due to arrangement of the images in various permutations of boxes.	The traces were taken as a horizontal or a vertical line of boxes are matched.
7	Sokoban (a game where a teddy bear puts boxes into bins amidst pathways)	Due to the variation in the pathways.	The traces were taken to make sure that the number/direction of boxes dropped in bins were same.

8	Pipes (a game where the broken pipes and joints are to be turned to be connected into a circuit to make a bulb glow)	The number, orientation of the joints and pipes.	The traces are taken to ensure that the number of joints connected is the same.
9	Billiards	Due to the very nature of the game.	Tracing were done to ensure that the number of balls pocketed were same.
10	Simple dice	They are supposed to be random.	games were played till the face up is the same

Besides these, there are randomness due to

1. The games graphics,
2. Game input methods (touch, keypad, and mouse)
3. Ads in between.
4. Different sounds/motion sequence for different actions.
5. The highlighting of positions/movements by some of the games.
6. Videos/animations of the emblems of the manufacturer in between and during the start of games.
7. Sophistication of games usually in android with more choices/ options/ introduction and copyright claims.
8. Some of them had more screens.

6.2.3 Questions

1. *What about other APIs and programming languages?*

We had chosen graphical APIs in our experiments so as to ease the process of generating traces. Visual verification simply solves the problem of making sure that the applications executes similarly. The approach could as well be used for other APIs of Java and android as long as we have applications of equal functionalities and a mechanism to validate it. Object oriented programming languages where the APIs are intensely utilized can use our approach. C# and java; Objective C and Java; Visual C# to Visual C++ can be mapped with our approaches. We tried the idea with programming contest entries, where the programming task for a contest can be implemented in several programming languages such as C#, java by the developers. We asked the popular programming contests such as {interviewstreet.com, topcoder.com, El Judge,} for datasets with source code/applications which are of different platforms and produce the same output. Here there is no need for a mechanism to verify that the applications has equal functionalities. However we didn't get any useful responses from the programming contest websites.

2. *What about other programming platforms?*

In other programming platforms such as windows platform the Rosetta tool can used to learn mappings VC++ and VC# or from C# to Java since they use API methods for most of the functionalities. The factors and features produced in the Rosetta tool are independent of the nature of the API calls. Hence our technique transcends the border of APIs and platforms.

3. *What about other programming paradigms?*

Our tool is based on APIs of object oriented nature. It cannot help with migration across programming paradigms. For example, the tool might not be of help if we want migrate from C to Java. C is a structural language and Java is an object oriented language.

4. *Where our tool might not work?*

In cases where the APIs of the programming languages are not intensely utilized, the tool is not of much help. In scripting languages such as Perl, Python, the utilization of API methods is fairly low as the programming tools such as regular expressions, dynamic arrays, hashes, are not API oriented. If we want to migrate from Perl to Java, The number of API methods calls in perl will be negligible when compared to that of the Java.

5. *Why Visual verification?*

The basic assumption we make in our thesis is that we believe in visual verification of output of two games to be same. What if this assumption is incorrect? What if the rectangle produced by the game is not due to drawRect or drawline but it is performed by drawImage, where the image is simply an empty rectangle. Then the possible mapping would be drawRect maps to drawImage. Is this correct? Well it drills down to the definition of functional equivalence between API methods. If we define the equivalence of two functionalities as “the result of the performing these two functionalities will same have the same effect”. Here the effect of drawImage (using an image of a rectangle) and drawRect are simply a rectangle drawn on the canvas. Now let us focus on the aim of the thesis the aim of the thesis is to produce a recommendation to the developer, a set of possible target API methods for a given source API method. Now the question is “can we recommend drawImage as one of alternative to drawRect of the source API method?” The answer is yes we can. It is just fine to draw a rectangle using an image when u have image of the rectangle. The rectangle in the image might be more beautiful than the rectangle that can be drawn by the target platform. But the reverse is not true. The drawImage cannot be mapped to drawRect ().why? There are numerous possibilities for an image rather than drawRect hence those mappings would be false. Mappings are usually commutative. But these are one sided mappings. This issue is because we have *VISUAL* verification, which states that “All that appears similar to eyes are similar” and the APIs we have chosen is Graphical APIs. For other APIs, if we have corresponding applications using those APIs and valid verification methods, then the technique will work fine.

6. *What about scalability?*

The issue of Scalability can be looked upon in two ways. Scalability with the present experimental setup and scalability of the techniques we had discussed.

For the present experimental setup we had taken graphical APIs of J2ME and Android while games are the application for these APIs. Right from the initial steps of collection of games till the verification of the results of API mappings to be true or false, we are limited by manual efforts.

Collecting games: We could not get more than 40 unique, downloadable, traceable (reasonably equal sized traces), simple, reasonably matching, board games available in both platforms. We got around 21 such games. We downloaded around 400 games to get them.

Tracing games: The tracing of the games is manual. We cannot automate the process of playing the games so as to produce same output screens.

Verifying the results: The concept of equal functionality is more subjective rather than an objective, concrete and tangible notion. Here the end customers could be developers. We are trying to give a notion of equality of two API methods calls to them. There is flexibility here so as to identify if two functionalities are similar or if two functionalities are related to each other and this involves human time. Once the traces are inferred using the techniques mentioned before, they needed to be verified manually as there is no ground truth with us. We had to go through the API documentation and pronounce that the API mapping is true or false. For the training data of the SVM classifier, we had around 86000 possible mappings to be decided as true or false. We did for around 7000 mappings.

7. *Scalability in terms of the idea of Rosetta*:

The idea of Rosetta if applied for applications for which we know that they are similar, the approach is quite scalable except for the verification part. Suppose a software Company has created software/applications/android for Java Platform and IOS Platform *earlier*. This collection of applications can now be utilized to create traces of equal functionalities. Visual verification is needed only for games from eclectic sources to confirm the equality of their functionalities. For applications for which we know that they perform the same task(techniques created by the software company), we don't have to verify them again. The collection of traces can now be automated. The more the traces, the better are the metrics. The analytics using factor graph approach and the support vector machines are already automated. Thus we can produce the mappings from these experiments. Now the verification part is still the bottleneck and has to be done manually. It simply cannot be avoided.

8. *Do we really need an API mapping tool?*

I mean if the results of the experiments are like "drawRect and fillRect are a mapping", anybody can identify them to be one. Meaning, it doesn't take time for a programmer to come up with FillRect in Android for a drawRect of J2ME or consider an obvious mapping "setColor matches to setColor". Most of the migration is performed like the notion of this simple logic (the names are similar). The functionalities developed in the programming languages tend to be similar. But the fact is there is a vast set of APIs for which finding the corresponding matching API in the target platform is not straightforward. Example, the method "$addCommand\ of\ javax/microedition/lcdui/Displayable\ class$" maps to the method "$writeToParcel\ android/widget/RemoteViews$. Adds a command to the displayable object.

9. *Why SVM*?

SVMs are the latest generation of learning algorithms for classification.

Pre 1980: Almost all learning methods learned linear decision surfaces. They had good theoretical properties.

1980's: Decision trees and neural networks allowed efficient learning of nonlinear decision surfaces. Little theoretical basis and they suffer from local minima.

1990's: Efficient learning algorithms for non-linear functions based on computing learning theory developed. Led to SVM

Last decade: New efficient separability of non-linear functions that use "kernel functions"

Unique features of SVM and Kernel Methods:

• Are explicitly based on a theoretical model of learning.

• Come with theoretical guarantees about their performance as well as state-of-the-art success in real-world applications.

• Have a modular design that allows one to separately implement and design their components

• Are not affected by local minima

• Do not suffer from the curse of dimensionality. Curse of Dimensionality is the problem of higher dimension which leads to more complex computations and lesser useful results.

10. *Why the name ROSETTA?*

The name Rosetta is came from Rosetta stone. The Rosetta stone is a stone with writing on it in two languages (Egyptian and Greek), using three scripts (hieroglyphic, demotic and Greek).

This was used to translate the languages. We are trying similar idea for programming languages and hence the name.

11. *What is the reason for going all possible mapping generation:*

In the Rosetta approach, the mappings are derived only from the traces pairs individually and the combinations of J2ME api calls across traces is not taken care of. Rosetta Classifier takes care of mappings across trace pair and it pays the price for it. Huge number of possible mappings. But the benefits overweigh the efforts. The mappings obtained from the Rosetta classifier are exhaustive! Meaning even there is a slight possibility of the mapping between two unheard API calls, the Rosetta Classifier can attempt to classify it. The initial training data is tiresome to obtain. But the training session is only once. Once the classifier is trained well enough it can keep on updating the parameters with every new value of the mapping and it grows faster.

12. *How do we incorporate less frequent API calls*?

The less frequent calls confuses the metrics and causes a lot of exception problems.

The attributes we measure for factors or frequencies are all about the structure of the traces predominantly. When the number of calls appearing the traces is negligible compared to the size of the traces ~0.1% then the probability metrics such as mutual information gives erroneous output as getting negative values for the mutual information. Apparently, values of mutual information should be in bits and always greater than zero. But the point wise mutual information namely the $\log(\frac{p(x,y)}{p(x)p(y)})$ $can\ get\ negative.$ Now because of their lower frequencies the total probability calculation for the mapping may not sum up to one and subsequently the mutual information, the expected value of poit-wise information is reduced to negative value. The lower frequency API calls also creates problem with the Rosetta Mapper, in a conclusion cannot be arrived about their mapping. Suppose if a mapping is made of two less frequent API methods which are rarely found in the trace file, the attributes would be ineffective as the evidence to declare them as mapping or not is insufficient. The solution would be to more number of traces, but even that has similar problems.

13. What about Android's own internal calls?

Many a times while tracing the Android applications, Android comes up with variety of API methods which are simply aliases/references/internal pointers to other methods of the inner class and when these aliases are traced, we have no clue what to do with them. We simply ignore them. Example, access$000 Landroid/view/SurfaceView; here,"access$000" is a pointer to function present at location 000 of the android inner class member from an outer non static member function.

14. Is many to many mapping possible?? Yes. Why haven't we uncovered it??

Many-to-many mappings might be present in the traces just like any $1-on-n\ or\ n-on-1\ or\ 1-on-1$ mapping sequences. In Rosetta Classifier, it was computation constrained as the approach is encompassing all the possible API calls and all possible mappings. In Rosetta mapper, the mapping sequences of size 2 $(1-on-2)$ was computation intensive in itself. It takes weeks to run the factor generations for method sequences of size 2. For many to many mappings (at least of size$2-on-2$) , it could take weeks even with our high machines(64GB RAM , 32 cores).For Rosetta Classifier especially the digram feature

this could span for a month.

6.3 References

[1] Microemulator. http://www.microemu.org/.

[2] python-levenshtein 0.10.2. http://pypi.python.org/pypi/python-Levenshtein.

[3] Qt API mapping for iOS developers. http://www.developer.nokia.com/Develop/Porting/API Mapping.

[4] Upontek. http://www.upontek.com/index.php.

[5] Windows phone interoperability. http://windowsphone.interoperabilitybridges.com/porting.

[6] Assembla. J2ME Android bridge. http://www.assembla.com/spaces/j2ab/wiki.

[7] T.T. Bartolomei, K. Czarnecki, and R. Laandmmel. Swing to SWT and back: Patterns for API migration by wrapping. In ICSM, 2010.

[8] R. Buse and W. Weimer. Synthesizing API usage examples. In ICSE, 2012.

[9] OW2 Consortium. Asm toolkit. http://asm.ow2.org.

[10] Netmite Corporation. App runner. http://www.netmite.com/android/.

[11] B. Dagenais and M. P. Robillard. Semdi_: Analysis and recommendation support for API evolution. In ICSE, 2009.

[12] B. Dagenais and M. P. Robillard. Recovering traceability links between an API and its learning resources. In ICSE, 2012.

[13] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In SOSP, 2001.

[14] Google. Android API reference. http://developer.android.com/reference/packages.html

[15] S. Hens, M. Monperrus, and M. Mezini. Semi-automatically extracting FAQs to improve accessibility of software development knowledge. In ICSE, 2012.

[16] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In ICSE, 2005.

[17] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: Inferring the specification within. In OSDI, 2006.

[18] F.R. Kschischang, B.J. Frey, and H.-A. Loeliger. Factor graphs and the sum-product algorithm. IEEE Trans. Inf. Theory, Feb 2001.

[19] Z. Li and Y. Zhou. Pr-miner: Automatically extracting implicit programming rules and detecting violations in large software code. In FSE, 2005.

[20] B. Livshits, A. Nori, S. Rajamani, and A. Banerjee. Merlin: Inferring specifications for explicit information flow problems. In PLDI, 2009.

[21] B. Livshits and T. Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In FSE, 2005.

[22] D. Mandelin, L. Xu, R. Bodik, and D. Kimelman. Jungloid mining: Helping to navigate the API jungle. In PLDI, 2005.

[23] Kevin Murphy. Bayes Net toolbox for Matlab, October 2007. http://code.google.com/p/bnt/.

[24] M. Nita and D. Notkin. Using twinning to adapt programs to alternative APIs. In ICSE, May 2010.

[25] Oracle. Java ME API reference. http://docs.oracle.com/javame/config/cldc/ref-impl/midp2.0/jsr118/index.html

[26] Oracle. Sun Java wireless toolkit for CLDC, 2.5.1. http://java.sun.com/products/sjwtoolkit/download-2 5 1.html

[27] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. M. Paradkar. Inferring method specifications from natural language API descriptions. In ICSE, 2012.

[28] D. Perelman, S. Gulwani, T. Ball, and D. Grossman. Type-directed completion of partial expressions. In PLDI, 2012.

[29] E. Shah and E. Tilevich. Reverse-engineering user interfaces to facilitate porting to and across mobile devices and platforms. In Workshop on NExt-generation Applications of smarTphones, 2011.

[30] G. Uddin, B. Dagenais, and M. P. Robillard. Temporal analysis of API usage concepts. In ICSE, 2012.

[31] J. S. Yedidia, W. T. Freeman, and Y. Weiss. Exploring artificial intelligence in the new millennium. chapter Understanding belief propagation and its generalizations. 2003.

[32] K. Yessenov, Z. Xu, and A. Solar-Lezama. Data-driven synthesis for object-oriented frameworks. In OOPSLA, 2011.

[33] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, and J. Zhao. Automatic parameter recommendation for practical API usage. In ICSE, 2012.

[34] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang. Mining API mapping for language migration. In ICSE, 2010.

[35] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending API usage patterns. In ECOOP, 2009.

[36] Lucene project : lucene.apache.org/core/4_0_0/jRE_VERSION_MIGRATION.html

[37] hiphop for Php: https://developers.facebook.com/blog/post/2010/02/02/hiphop-for-php--move-fast/

[38] Cfront: http://www.softwarepreservation.org/projects/c_plus_plus/index.html#cfront

[39] Java to ObjC: http://google-opensource.blogspot.com/2012/09/j2objc-java-to-ios-objective-c.html

[40] Microsoft Foundation Classes: http://msdn.microsoft.com/enus/library/d06h2x6e%28VS.90%29.aspx

[41] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for Real time Privacy Monitoring on Smartphones, Proceedings of the 9th USENIX Symposium on Operating Systems design and Implementation (OSDI), October, 2010. Vancouver, BC. (Presented by Liu Yang)

[42] Davenport, Mark A., Richard G. Baraniuk, and Clayton D. Scott. "Minimax support vector machines." *Statistical Signal Processing, 2007. SSP'07. IEEE/SP 14th Workshop on*. IEEE, 2007.

[43] Lanckriet, G. R., Ghaoui, L. E., Bhattacharyya, C., & Jordan, M. I. (2003). A robust minimax approach to classification. *The Journal of Machine Learning Research*, *3*, 555-582.

6.2 Appendix

Sample Code:

The following code is used to obtain the mutual information on frequency and position for a collection of trace files.

```
#!/usr/bin/perl -w
use Math::BigFloat;
Math::BigFloat->precision(-4);
opendir(IMD, ".") || die("Cannot open directory");
@thefiles= readdir(IMD);
open(FILE2,">formfile.txt") or die"cant open the file 3 $!";
open(FILE3,">resultfile1.txt") or die"cant open the file 3 $!";
open(FILE4,">debug.txt") or die"cant open the file 4 $!";
print @thefiles;
my %hash1=();
my %hash2=();
my %hash3=();
my $indexA=0;
my $indexJ=0;
my $prob;
my $i=0;
my $j=0;
my $sum=0;
my $su=0;
my $count2=0;
my $word=0;
my $index=0;
my $flag=0;
my $size=0;
my $count=0;
my $prob_index=0;
my $position=0;
my $sizeJ=0;
my $sizeA=0;
my @mutual_info=();
my $mut=0;
my $px=0;
my $py=0;
my $probx=0;
my $proby=0;

sub hashposAscNum
{
  $hash{$funct}[2]{$b} <=> $hash{$funct}[2]{$a};
```

```
}

sub hash2posAscNum
{
  $hash2{$funct}[2]{$b} <=> $hash2{$funct}[2]{$a};
}

foreach $file1(@thefiles) #for All files
{
  if( $file1!~/^\.+$/ && $file1=~/J.*.txt/)    #count the files like J*
    {
      $sizeJ++;
    }

 if( $file1!~/^\.+$/ && $file1=~/A.*.txt/) #count the files like A*
    {
      $sizeA++;
    }
}

foreach $file1(@thefiles) #for All files
{
   if( $file1!~/^\.+$/ && $file1=~/J.*.txt/) #select the files like J*
    {
          print FILE3 "index: $indexJ\t filename:$file1\n";
          open(FILE, "$file1") or die"cant open the file $file1 $!";
          my @words=<FILE>; #collect the function calls in an array
          $count=0;
           $size=scalar(@words);
           foreach $word (@words) #foreach function call
              {
                 @ka=split(/\n/,$word); #remove \n or \* in them
                  if($ka[0]=~/\*/)
                   {
                        @ta=split(/\*/,$ka[0]);
                          $word=$ta[1];
                   }

                  else
                       {
                          $word=$ka[0];
                        }
              $count++;

# $hash{word}[0][indexJ][0]
# hash-> This is the hash variable which collects all the data
# we have hash(J2ME) and hash2(Android)
# {word}->function call strings are the keys for the hash.
# $hash{word}[0]-> this is 2D array with 2 columns for every row.
# $hash{word}[0][$indexJ]-> indexJ/A marks the index for every file(J2ME or Android)
# $hash{word}[0][$indexJ][0]->denotes the frequency of the word in the file. (an element)
# $hash{word}[0][$indexJ][1]->denotes the probability of the word in the file.(an element)
# $hash{word}[0][$indexJ][2][0]->denotes the positions of the word in the file in an array.
# $hash{word}[0][$indexJ][2][1]->denotes the specific positional indices from 0.1 to 1.0 (hash)
# $hash{word}[0][$indexJ][2][1]{$indexP}->denotes the count of specific positional indexP
# $hash{word}[1]-> stores the probability index(0.01 to 0.99) and (a hash)
# $hash{word}[1]{prob_index}[0]->the count of the prob index found in files.
# $hash{word}[2]{pos_index}-> stores the possible probability values existing in every positions
# $hash{word}[2]{pos_index}{prob_index}[0]-> stores the count of prob index found at various positions in various files.

                if (exists $hash{$word})
                {
                   $hash{$word}[0][$indexJ][0]=$hash{$word}[0][$indexJ][0]+1;
                   $position=sprintf "%.2f",$count/$size;
                   push @{$hash{$word}[0][$indexJ][2][0]},$position;
                    if(exists $hash{$word}[0][$indexJ][2][1]{$position})
                      {
                         $hash{$word}[0][$indexJ][2][1]{$position}=$hash{$word}[0][$indexJ][2][1]{$position}+1;
                      }
                    else
                     {
                         $hash{$word}[0][$indexJ][2][1]{$position}=1;
                     }
                }
                else
                {
                    $hash{$word}[0][$indexJ][0]=1;
                    $position=sprintf "%.2f",$count/$size;
                    push @{$hash{$word}[0][$indexJ][2][0]},$position;



                  if(exists $hash{$word}[0][$indexJ][2][1]{$position})
                     {
                        $hash{$word}[0][$indexJ][2][1]{$position}=$hash{$word}[0][$indexJ][2][1]{$position}+1;
                     }
                else
                     {
                         $hash{$word}[0][$indexJ][2][1]{$position}=1;
```

```
                                    }
                              }
                }
           print FILE3 "count : $count\n";

      foreach $word (@words)
      {

           $hash{$word}[0][$indexJ][1]=sprintf "%.2f",$hash{$word}[0][$indexJ][0]/$size;
      }

      foreach $word (keys %hash)
     {
           $prob_index=$hash{$word}[0][$indexJ][1];

            if(exists $hash{$word}[1]{$prob_index})
          {
               $hash{$word}[1]{$prob_index}[0]=$hash{$word}[1]{$prob_index}[0]+1;
          }
          else
        {
              $hash{$word}[1]{$prob_index}[0]=1;
        }
     }

     $indexJ++;
}

if( $file1!~/^\.+$/ && $file1=~/A.*.txt/)
     {
             #print FILE2 "index: $indexA\t filename:$file1\n";
             open(FILE, "$file1") or die"cant open the file $file1 $!";
             my @words2=<FILE>;
             $count2=0;
             $size=scalar(@words2);

          foreach $word (@words2)
         {
                @ka=split(/\n/,$word);
                if($ka[0]=~/\*/)
                {
                     @ta=split(/\*/,$ka[0]);
                      $word=$ta[1];
                }
                else
        {
            $word=$ka[0];
        }
        $count2++;
        if (exists $hash2{$word})
        {
             $hash2{$word}[0][$indexA][0]=$hash2{$word}[0][$indexA][0]+1;
             $position=sprintf "%.2f",$count2/$size;
             push @{$hash2{$word}[0][$indexA][2][0]},$position;

                  if(exists $hash2{$word}[0][$indexA][2][1]{$position})
                  {
                     $hash2{$word}[0][$indexA][2][1]{$position}=$hash2{$word}[0][$indexA][2][1]{$position}+1;
                  }
                  else
                  {
                     $hash2{$word}[0][$indexA][2][1]{$position}=1;
                  }
        }
        else
        {
               $hash2{$word}[0][$indexA][0]=1;
               $position=sprintf "%.2f",$count2/$size;
               push @{$hash2{$word}[0][$indexA][2][0]},$position;

               if(exists $hash2{$word}[0][$indexA][2][1]{$position})
               {
                    $hash2{$word}[0][$indexA][2][1]{$position}=$hash2{$word}[0][$indexA][2][1]{$position}+1;
               }
               else
               {
                    $hash2{$word}[0][$indexA][2][1]{$position}=1;
               }
          }          #print "keys at hash2 :".(keys %{$hash2{$word}[0][$indexA][2][1]});
 }
 #print FILE2 "count2 : $count2\n";
 foreach $word (@words2)
        {
             $hash2{$word}[0][$indexA][1]=sprintf "%.2f",$hash2{$word}[0][$indexA][0]/$size;
        }
foreach $word (keys %hash2)
        {
             $prob_index=$hash2{$word}[0][$indexA][1];
             if(exists $hash2{$word}[1]{$prob_index})
             {
                    $hash2{$word}[1]{$prob_index}[0]=$hash2{$word}[1]{$prob_index}[0]+1;
                }
              else
                {
                    $hash2{$word}[1]{$prob_index}[0]=1;
                }
           }
    #print FILE3 "sum:$sum\nsu:$su\n";
    $indexA++;
     }
}
# compute the positional probability values in each of the $hash{word}[2]{pos_index}-> stores the possible probability
values existing # in every positions
# $hash{word}[2]{pos_index}{prob_index}[0]-> stores the count of prob index found at various positions in various files.
$indexJ=0;
$indexA=0;
foreach $file1(@thefiles) #for All files
{
     if( $file1!~/^\.+$/ && $file1=~/J.*.txt/) #select the files like J*
    {
          #print FILE2 "index: $indexJ\t filename:$file1\n";
          open(FILE, "$file1") or die"cant open the file $file1 $!";
          my @words=<FILE>;               #collect the function calls in an array
          $count=0;
          $size=scalar(@words);
           foreach $word (@words)          #foreach function call
             {
                  @ka=split(/\n/,$word);   #remove \n or \* in them
                  if($ka[0]=~/\*/)
                 {
                       @ta=split(/\*/,$ka[0]);
                       $word=$ta[1];
                 }
                 else
                {
                    $word=$ka[0];
                }
# $hash{word}[0][indexJ][0]
# hash-> This is the hash variable which collects all the data
#        we have hash(J2ME) and hash2(Android)
# {word}->function call strings are the keys for the hash.
# $hash{word}[0]-> this is 2D array with 2 columns for every row.
# $hash{word}[0][$indexJ]-> indexJ/A marks the index for every file(J2ME or Android)
# $hash{word}[0][$indexJ][0]->denotes the frequency of the word in the file. (an element)
# $hash{word}[0][$indexJ][1]->denotes the probability of the word in the file.(an element)
# $hash{word}[0][$indexJ][2][0]->denotes the positions of the word in the file in an array.
# $hash{word}[0][$indexJ][2][1]->denotes the specific positional indices from 0.1 to 1.0 (hash)
# $hash{word}[0][$indexJ][2][1]{$indexP}->denotes the count of specific positional indexP
# $hash{word}[0][$indexJ][2][2]{$indexP}-> prob of occurence at position w.r.t the entire file with a two digit precision.
# $hash{word}[1]-> stores the probability index(0.01 to 0.99) and (a hash)
# $hash{word}[1]{prob_index}[0]->the count of the prob index found in files.
# $hash{word}[2]{pos_index}-> stores the possible probability values existing in every positions
# $hash{word}[2]{pos_index}{prob_index}[0]-> stores the count of prob index found at various positions in various files.

  if (exists $hash{$word})
  {
      foreach $position (keys %{$hash{$word}[0][$indexJ][2][1]})
      {
            #print FILE3 " $position:";
            #calculating the prob of occurence at position "$position" w.r.t the entire file to a two digit precision.

           $hash{$word}[0][$indexJ][2][2]{$position}=
           sprint"%.2f",$hash{$word}[0][$indexJ][2][1]{$position}/$hash{$word}[0][$indexJ][0];

            #print FILE3 "$hash{$word}[0][$indexJ][2][2]{$position},";
      }
   }
}
 foreach $word (keys %hash)
  {
     foreach $position (keys %{$hash{$word}[0][$indexJ][2][1]})
     {
           $prob_index=sprintf "%.2f",$hash{$word}[0][$indexJ][2][1]{$position}/$hash{$word}[0][$indexJ][0];

           #print FILE3 ", position: $position: \t ind $prob_index\n";

          if (exists $hash{$word}[2]{$position}{$prob_index})
            {
                 $hash{$word}[2]{$position}{$prob_index}[0]=$hash{$word}[2]{$position}{$prob_index}[0]+1;
            }
         else
           {
                 $hash{$word}[2]{$position}{$prob_index}[0]=1;
           }
      }
   }
   $indexJ++;
}

 if( $file1!~/^\.+$/ && $file1=~/A.*.txt/)
```

```
{
   #print FILE2 "index: $indexA\t filename:$file1\n";
   open(FILE, "$file1") or die"cant open the file $file1 $!";
   my @words2=<FILE>;
   $count2=0;
   $size=scalar(@words2);
   foreach $word (@words2)
   {
         @ka=split(/\n/,$word);
         if($ka[0]=~/\*/)
         {
            @ta=split(/\*/,$ka[0]);
             $word=$ta[1];
         }
         else
         {
               $word=$ka[0];
         }
         if (exists $hash2{$word})
         {
               foreach $position (keys %{$hash2{$word}[0][$indexA][2][1]})
               {
                   #print FILE2 " $position:";
                    #calculating the prob of occurence at position "$position" w.r.t the entire file to a two digit precision.
                   $hash2{$word}[0][$indexA][2][2]{$position}=
                   sprintf "%.2f",$hash2{$word}[0][$indexA][2][1]{$position}/$hash2{$word}[0][$indexA][0];
                   #print FILE2 "$hash2{$word}[0][$indexA][2][2]{$position},";
               }
         }
    }

   foreach $word (keys %hash2)
   {
        foreach $position (keys %{$hash2{$word}[0][$indexA][2][1]})
        {
          $prob_index=sprintf "%.2f",$hash2{$word}[0][$indexA][2][1]{$position}/$hash2{$word}[0][$indexA][0];
            if(exists $hash2{$word}[2]{$position}{$prob_index})
            {
                  $hash2{$word}[2]{$position}{$prob_index}[0]=$hash2{$word}[2]{$position}{$prob_index}[0]+1;
             }
         else
            {
                   $hash2{$word}[2]{$position}{$prob_index}[0]=1;
            }
      }
```

```
          }
         $indexA++;
        }
    }
    print FILE2 "\n\n HASH\n\n";
         foreach $funct (keys %hash)
        {
              print FILE2 "$funct:\t\t\n";
              foreach $j (0 ..$#{ $hash{$funct}[$i]} )
              {
                     print FILE2 "\n\tfrequency = ";
                    if($hash{$funct}[0][$j][0] eq "")
                    {
                          print FILE2 "0\t";
                    }
                    else
                    {
                           print FILE2 "$hash{$funct}[0][$j][0]\t";
                    }
                    print FILE2 "probability = ";

                    if($hash{$funct}[0][$j][1] eq "")
                    {
                           print FILE2 "0\t";
                    }
                    else
                    {
                                   print FILE2 "$hash{$funct}[0][$j][1]";
                    }
                    print FILE2 "\tpos={";
                   foreach $i (@{$hash{$funct}[0][$j][2][0]})
                    {
                         print FILE2 " $i ";
                    }
                    print FILE2 "}\n";
               }
               print FILE2 "frequency distribution \n";
               foreach $prob_key (keys %{$hash{$funct}[1]})
               {
                    if($prob_key eq "")
                    {
                           print FILE2 "0\t";
                    }
                    else
                    {
```

```
                    print FILE2 "$prob_key:";
               }
               print FILE2 "$hash{$funct}[1]{$prob_key}[0]\n";
          }
           print FILE2 "\n";
           print FILE2 "Positional distribution \n";
          foreach $pos_key (sort {$a <=> $b} (keys %{$hash{$funct}[2]}))
          {
               if($pos_key eq "")
               {
                     print FILE2 "0=>{";
               }
               else
               {
                    print FILE2 "position:$pos_key=>{";
               }
               foreach $prob_key (keys %{$hash{$funct}[2]{$pos_key}})
               {
                     if($prob_key eq "")
                      {
                           print FILE2 "0\t";
                      }
                    else
                     {
                          print FILE2 "$prob_key:";
                    }
                     print FILE2 "$hash{$funct}[2]{$pos_key}{$prob_key}[0],";
               }
               print FILE2 "}\n";
          }
         print FILE2 "\n";
     }
     print FILE2 "\n\n HASH2\n\n";
      foreach $funct (keys %hash2)
      {
            print FILE2 "$funct:\t\t\n";
           foreach $j (0 ..$#{ $hash2{$funct}[$i]} )
           {
                print FILE2 "\n\tfrequency = ";
                if($hash2{$funct}[0][$j][0] eq "")
                {
                       print FILE2 "0\t";
               }
               else
               {
```

```
            print FILE2 "$hash2{$funct}[0][$j][0]\t";
        }
        print FILE2 "probability = ";
        if($hash2{$funct}[0][$j][1] eq "")
        {
             print FILE2 "0\t";
        }
        else
        {
            print FILE2 "$hash2{$funct}[0][$j][1]";
        }
        print FILE2 "\tpos={";
        foreach $i (@{$hash2{$funct}[0][$j][2][0]})
        {
           print FILE2 " $i ";
        }
        print FILE2 "}\n";
    }
   foreach $prob_key (keys %{$hash2{$funct}[1]})
   {
         if($prob_key eq "")
         {
               print FILE2 "0\t";
         }
         else
         {
              print FILE2 "$prob_key:";
         }
         print FILE2 "$hash2{$funct}[1]{$prob_key}[0]\n";
   }
        print FILE2 "\n";
        print FILE2 "Positional distribution \n";
        print FILE2 "K:".keys %{$hash2{$funct}[2]};
        foreach $pos_key (sort {$a <=> $b} (keys %{$hash2{$funct}[2]}))
        {
            if($pos_key eq "")
            {
                 print FILE2 "0=>{";
            }
            else
            {
                 print FILE2 "position:$pos_key:=>{";
             }
            foreach $prob_key (keys %{$hash2{$funct}[2]{$pos_key}})
            {
                    if($prob_key eq "")
                    {
                        print FILE2 "0:";
                    }
                    else
                    {
                        print FILE2 "$prob_key:";
                    }
                    print FILE2 "$hash2{$funct}[2]{$pos_key}{$prob_key}[0],";
                }
                print FILE2 "}\t";
            }
            print FILE2 "\n";
        }

#Calculation of mutual information between the inidividual function calls.
#the Random variables are the presence of the function calls in the trace files.
#obtain the individual and the combined probabilities of occurence of the function calls at discrete intervals
#P(X=0.00) to P(X=0.99) and P(x=0.0,Y=0.0 to X=0.99,Y=0.99)
my $probX=0;
my $probY=0;
my $probXY=0;
my $XYcount=0;
my $mut_sum=0;
my %MUTUAL=();
my $mut_string=0;

foreach $j2me (keys %hash)
{
   #$j2me="setColor         javax/microedition/lcdui/Graphics";
   #$j2me="setTimeout       javax/microedition/lcdui/Alert";
  #   $j2me="drawRect        javax/microedition/lcdui/Graphics";
  #calculate px
  foreach $andr (keys %hash2)
  {
      #calculate py
      #$andr="drawRect    Landroid/graphics/Canvas;";
      # $andr="getChildCount";
      #$andr="getHeight    Landroid/view/View;";
      $mut_string=$j2me."::::::".$andr;
      $mut_sum=0;
      my $err_cnt=0;
      my @mut_add=();
      my $l=0;
      my $k=0;
```

```
for( $i=0;$i<=100;$i++)
  {
      $px=sprintf "%.2f",$i/100;
       if(exists $hash2{$andr}[1]{$px})
        {
              $probX=$hash2{$andr}[1]{$px}[0]/($sizeA);
              if($probX>0)
              {
                   #print FILE3 "px:$px\t";
                    for( $j=0;$j<=100;$j++)
                   {
                         $py=sprintf "%.2f",$j/100;
                         if(exists $hash{$j2me}[1]{$py})
                         {
                              $probY=$hash{$j2me}[1]{$py}[0]/($sizeJ);
                               if($probY >0)
                               {
                                   $XYcount=0;
                                   for($l=0;$l<$sizeA;$l++)
                                   {
                                         $PX=sprintf "%.2f",$hash2{$andr}[0][$l][1];
                                        $PY=sprintf "%.2f",$hash{$j2me}[0][$l][1];

                                       for( $k=0;$k<$sizeJ;$k++)
                                       {

                                          if($hash2{$andr}[0][$l][0]>0 && $hash{$j2me}[0][$l][0]>0)
                                          {
                                             if($PX==$px && $PY==$py)
                                             {
                                                print FILE3 "pX:$px pY:$py\tPX: $PX\tPY :$PY\n";
                                                $XYcount++;
                                             }
                                          }
                                       }
                                   }
                              my $size=$sizeJ;
                              $probXY=$XYcount/($sizeJ);

                         if($probXY > 0)
                         {
                               my $sum=($probXY*log($probXY/($probX*$probY)))/log 2;
                              $mut_sum+=($probXY*log($probXY/($probX*$probY)))/log 2;
                              print FILE3 "PAIR:$mut_string\tprobxy:$probXY\t";
                               print FILE3 "px:$probX\tpy:$probY\tMI_bit:$sum\t aggr_MI:$mut_sum\n";
```

```
                                                push @mut_add,$probX;
                                                push @mut_add,$hash2{$andr}[1]{$px}[0];
                                                push @mut_add,$px;
                                                push @mut_add,$probY;
                                                push @mut_add,$hash{$j2me}[1]{$py}[0];
                                                push @mut_add,$py;
                                                push @mut_add,$probXY;
                                                push @mut_add,$XYcount;
                                                push @mut_add,$probXY*log($probXY/($probX*$probY));
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                    $MUTUAL{$mut_string}[0]=$mut_sum;

                    if($MUTUAL{$mut_string} <0 || $MUTUAL{$mut_string} >=1)
                    {
                        print FILE4 "\n\n$mut_string\n$MUTUAL{$mut_string}\n\n";
                        print "\nmutual additions\n";
                        foreach $i(@mut_add)
                        {
                            print FILE4 $i."\n";
                        }
                    }
                }
}
sub hashValueAscendingNum
 {
    $MUTUAL{$b}[2] <=> $MUTUAL{$a}[2];
 }
 $probX=0;
 $probY=0;
 $probXY=0;
 $XYcount=0;
 $mut_sum=0;
 my %MUTUALP=();
 my %MUTUALS=();
 $mut_string=0;
foreach $j2me (keys %hash)
{
    foreach $andr (keys %hash2)
    {
```

```
$mut_string=$j2me."::::::".$andr;
my $err_cnt=0;
my @mut_add=();
my $l=0;
my $k=0;
$probX=0;
my $mut_cnt=0;
for($k=0;$k<=100;$k++)
{
        $mut_sum=0;
        $pos=sprintf "%.2f",$k/100;
       if((exists $hash2{$andr}[2]{$pos}) and (exists $hash{$j2me}[2]{$pos}))
      {
            $mut_cnt++;
            for( $i=0;$i<=100;$i++)
            {
                  $px=sprintf "%.2f",$i/100;
                   if(exists $hash2{$andr}[2]{$pos}{$px})
                  {
                        $probX=$hash2{$andr}[2]{$pos}{$px}[0]/($sizeA);
                        if($probX>0)
                        {
                             #print FILE3 "\tpx:$px\n";
                              for( $j=0;$j<=100;$j++)
                             {
                                     $py=sprintf "%.2f",$j/100;
                                     if(exists $hash{$j2me}[2]{$pos}{$py})
                                     {
                                          $probY=$hash{$j2me}[2]{$pos}{$py}[0]/($sizeJ);
                                           if($probY >0)
                                          {
                                               $XYcount=0;

                                                 for($l=0;$l<$sizeA;$l++)
                                                  {
                                                       $PX=sprintf "%.2f",$hash2{$andr}[0][$l][2][2]{$pos};
                                                       $PY=sprintf "%.2f",$hash{$j2me}[0][$l][2][2]{$pos};
                                                       if($hash2{$andr}[0][$l][2][1]{$pos}>0 )
                                                       {
                                                          if($hash{$j2me}[0][$l][2][1]{$pos}>0)
                                                          {
                                                            if($PX==$px && $PY==$py)
                                                            {
                                                                $XYcount++;
                                                            }
                                                  }
                                                }
                                              }
                                              my $size=$sizeJ;
                                              $probXY=$XYcount/($sizeJ);

                                              #print FILE3 "\txycount:$XYcount $size \nprobXY:$probXY\n";
                                        if($probXY > 0)
                                        {
                                          my $sum=($probXY*log($probXY/($probX*$probY)))/log 2;
                                          $mut_sum+=($probXY*log($probXY/($probX*$probY)))/log 2;
                                          #print FILE3 "\tPAIR:$mut_string\tprobxy:$probXY\tpx:$probX";
                                          # print FILE3 "\tpy:$probY\tMI_bit:$sum\taggr_MI:$mut_sum\n";
                                        }
                                      }
                                    }
                                  }
                                }
                              }
                            }
              }
              $MUTUALP{$mut_string}{$pos}=$mut_sum;
            }

             foreach $position (keys %{$MUTUALP{$mut_string}})
             {
                  $MUTUALS{$mut_string}+=$MUTUALP{$mut_string}{$position};
             }

             if($mut_cnt>0)
             {
                $MUTUALS{$mut_string}=$MUTUALS{$mut_string}/$mut_cnt;
             }
             #my $getch=<STDIN>;

           $MUTUAL{$mut_string}[1]=$MUTUALS{$mut_string};
           $MUTUAL{$mut_string}[2]=$MUTUAL{$mut_string}[0]+$MUTUAL{$mut_string}[1];
           $MUTUAL{$mut_string}[3]=$MUTUAL{$mut_string}[0]*$MUTUAL{$mut_string}[1];
  }
}

 sub hasValueAscendingNum
{
  $MUTUAL{$a} <=> $MUTUAL{$b};
}
```

```
                #print the output to the file.
print FILE3 "\n\n\nMUTUAL INFORMATION\n\n\n";
foreach $mut_key (sort hashValueAscendingNum (keys %MUTUAL))
{
    print FILE3 "\n$mut_key,$MUTUAL{$mut_key}[0],$MUTUAL{$mut_key}[1]";
}
```