

Hochschule Ulm



A Data Mining Approach to Compare Java with Kotlin

Author: Simon Hellbrück

Supervisor: Senior Lecturer Juhu Pekka Kämäri

Degree Program: Information Technology

Number of Pages: 32

Thesis Submitted in Fulfillment of the Requirements for the Degree of
Bachelor of Science

January 21, 2019

Abstract

In early 2018, Google announced Kotlin to be one of the official languages for developing Android applications (next to Java). Moreover, several studies reveal that Kotlin is about to make its way to an important language in the developer community. Developers praise the conciseness, short syntax and null pointer safety. No question, a competition has started, but can Kotlin replace Java in the long run? Several factors (combined) will determine if a developer decides learning and/or using Kotlin over Java. Of course, it is also a question of investment. Furthermore, one of the major questions is: Does the short syntax and conciseness of Kotlin causes less occurrences of bad practices? In order to shed light on this question, an experiment was conducted. Java and Kotlin repositories were downloaded from GitHub, bad practices in the respective projects were analysed, counted, summed up and eventually saved in CSV files. After that, several well established statistical methods were applied to the data to get meaningful results. The main conclusion is: A substantial difference in the occurrence of bad practices can not be confirmed.

Contents

1	Introduction	1
2	Research Design	1
3	Theoretical Background	1
3.1	Java	1
3.1.1	Architecture	2
3.2	Kotlin	2
3.2.1	Architecture	3
3.3	Syntactic Differences	3
3.3.1	Variable Declaration and Initialization	3
3.3.2	Conditional Expressions	4
3.3.3	Plain Old Java Objects	5
3.4	Null Pointer Safety	5
3.5	Comparative Overview	6
3.5.1	Java Issues Addressed in Kotlin	6
3.5.2	Only Available in Java	7
3.5.3	Only Available in Kotlin	8
3.6	Hybrid Programming	9
3.7	Trends and Statistics	11
3.8	Code Smells	14
4	Experimental Setup	15
5	Experiment	16
5.1	Business Understanding	16
5.2	Data Understanding	16
5.3	Data Preparation	19
5.4	Modelling	20
5.4.1	Assumptions for ANCOVA	22
5.4.2	ANCOVA	27
5.5	Evaluation	30
5.6	Deployment	30
6	Discussion, Related Work	31
7	Limitations, Future Research	31
8	Conclusion	32
	Appendices	36
A	Main Bash Script	36
B	Python Program	37
C	Bash Scripts for Kotlin Repositories	38
D	Bash Scripts for Java Repositories	39
E	R Code Part 1	40
F	R Code Part 2	41

List of Figures

1	Java Architecture (Adopted from [1])	2
2	Kotlin Architecture [2]	3
3	Most 'Loved' Programming Languages [3]	11
4	Most Wanted Programming Languages [3]	12
5	Correlated Technologies [3]	13
6	Usage of Kotlin [4]	13
7	Greatest Hits of Kotlin [4]	14
8	The Cross-Industry Standard Process for Data Mining [5]	15
9	Correlation Matrix of Java Projects (Training Data)	20
10	Correlation Matrix of Kotlin Projects (Training Data)	20
11	LOC and Total Amount of Smells in Java and Kotlin Projects (Training Data)	21
12	Diagnostic Plots of the Multidimensional Regression (Java Projects)	22
13	Diagnostic Plots of the Multidimensional Regression (Kotlin Projects)	23
14	Lambda and Corresponding Likelihood	23
15	Lambda and Corresponding Likelihood	24
16	Diagnostic Plots of the Regression Using a Logarithmic Fit (Java Projects)	25
17	Diagnostic Plots of the Regression Using a Logarithmic Fit (Kotlin Projects)	25
18	LOC vs Total Amount of Smells	26
19	Diagnostic Plots for the Multidimensional Regression (Java and Kotlin)	28
20	Histogram of the Residuals	28

List of Tables

1	Java Issues Addressed in Kotlin (adopted from [6])	7
2	Features and Concepts Only Available in Java (adopted from [6])	8
3	Code Smells	14
4	Code Smells Found in Kotlin Projects (Data Collected by Flauzino et al. [7])	17
5	Code Smells Found in Java Projects (Data Collected by Flauzino et al. [7]) .	18
6	Excerpt of Java Code Smells (Data Collected on November 27th)	19
7	Excerpt of Kotlin Code Smells (Data Collected on November 27th)	19
8	Box-Cox Transformations (According to [8])	24
9	Results of the Shapiro-Wilk Test	26
10	Concatenated Data Frame with Language Coded as 1 and 0	27
11	Results of the Shapiro-Wilk Test	29
12	Arithmetic Means of Total Amount of Smells (Without Controlling Covariates)	29
13	Arithmetic Means of Total Amount of Smells (When Controlling Covariates)	29
14	Descriptive Statistics: Summary of the Response Variable (Total Amount of Smells)	30

Listings

1	Value Assignment in Kotlin Using 'val'	3
2	Value Assignment in Kotlin Using 'var'	3
3	Value Assignment in Java Using a Datatype Declaration	3
4	Exception of Using 'val' and Being Immutable	4
5	Using the 'lateinit' Keyword for a Variable	4
6	Checking if a Variable has been Initialized	4
7	Short Version of a Conditional Branch in Java	4
8	Expression Statement in Kotlin for Conditional Branching	4
9	Switch Statement in Java	4
10	When Expression in Kotlin	5
11	Car Object in Java	5
12	Car Object in Kotlin	5
13	Example of a Null Pointer Exception in Java	6
14	Kotlin Compiler Telling the Problem of Having a Null Value	6
15	Explicit Declaration of a Null Value in Kotlin	6
16	Explicit Declaration and Call of a Null Value in Kotlin	6
17	Car Object in Java	9
18	Main Method in Kotlin	9
19	Car Object in Kotlin	9
20	Main Method Converted from Kotlin to Java	10
21	Bash Script Used to Execute the Python Program and All Bash Scripts at Once	36
22	Python Program Used to Download the Repository Names	37
23	Bash Script Used to Download the Kotlin Repositories	38
24	Bash Script Used to Apply Detekt to the Repositories	38
25	Bash Script Used to Read Log Files and Write the Results to a CSV File	38
26	Bash Script Used to Download the Java Repositories	39
27	Bash Script Used to Apply PMD to the Repositories	39
28	Bash Script Used to Read Log Files and Write the Results to a CSV File	39
29	R Code Used for Statistical Analysis	40
30	R Code Used for Statistical Analysis	41

List of Abbreviations

ANCOVA Analysis of Variance with Covariates

API Application Programming Interface

CRISP-DM Cross-Industry Standard Process for Data Mining

ES ECMAScript

IDE Integrated Development Environment

1 Introduction

Java is a general-purpose language developed by Sun in the 90s. Rumours are still going on, whether the language was named after the coffee or the Indonesian island – most probably it is the coffee.[9] Kotlin is an island in front of St. Petersburg and the name of the programming language refers to that island. The developing team works on that language – and on that island – for nine years now. The first official release was in 2016.[10][11] In the area of Android application development, Java has been the only official language for a long time. However, a couple of months ago, Google announced Kotlin to be an official language for developing Android applications as well.[12] Moreover, previous studies have shown that Kotlin is growing not only in prominence but also in terms of popularity compared to other, well established programming languages: Stackoverflow registers an increase in the percentage of questions, which are asked on the platform.[13] Furthermore, a survey on Stackoverflow in 2018 revealed that Kotlin is one of the most popular languages.[3] But what are the benefits of using Kotlin over Java? Kotlin developers praise the short syntax, null safety and the compatibility to existing Java projects.[14][15][16] But are those sufficient reasons to learn a new language, to convert whole projects from one language to another? And one of the major questions is of course: Are Kotlin projects easier to maintain due to less occurrences of bad practices? This study will investigate, whether Java or Kotlin projects contain more occurrences of bad practices.

The next chapters cover the following: Chapter 2 includes the research question, the research method as well as the expected output. Chapter 3 will address the theoretical background: the architecture of the languages, syntactic differences, null pointer safety, a comparative overview, hybrid programming as well as current trends. Moreover, the term code smells will be explained and relevant examples for this study are explained further. In chapter 4, the cross-industry standard process for data mining will be introduced as a framework for the experiment. The conducted experiment will be shown in chapter 5. According to the conducted experiment, a discussion will follow (6). Also, work which is related to this study will be discussed in chapter 6. In chapter 7, limitations of the conducted experiment and starting points for future research are provided. In chapter 8, a conclusion will be drawn based on the results found in the experiment.

2 Research Design

Research Question: Which programming language causes less occurrences of bad practices?

An experiment was conducted in order to answer the research question. GitHub repositories were examined with regard to the frequency of bad practices. Furthermore, additional data such as the rating, the number of contributors and the number of code lines of the respective project were collected. Based on the collected data, several statistical methods were applied in order to obtain reliable results, which then show what programming language contains less occurrences of bad practices on average.

3 Theoretical Background

In the following chapters, the historical background of the programming languages Java and Kotlin, their architectures and the main differences regarding syntax and the behavior at compile and run time will be addressed. Statistics will reveal current trends. Furthermore, the term code smells and some metrics regarding code smells will be introduced.

3.1 Java

Since the original name for a new language (Oak) was taken by Oak Technologies, members working on a project for Sun in the early 1990s decided to name their project Java. This name is most probably related to the coffee 'Java'. [9] Java is a programming language, which has been designed in such a way that many developers can apply it easily. Furthermore, it is concurrent, class-based, object-oriented and suited for general purposes. The syntax of Java is similar to C as well as to C++. Moreover, the creator of Java, James Gosling, avoided to

include components, which have not been tested before.[17] The current Java version is 11 and has been released in September 2018 as a long-term support release.[18]

3.1.1 Architecture

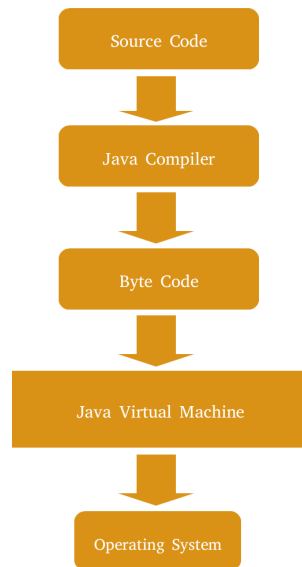


Figure 1: Java Architecture (Adopted from [1])

The source code written by the developer is translated into byte code by the Java compiler. At runtime, the hardware specific Java Virtual Machine generates machine code, which then can be executed by the actual machine.[1] Hence the Java Virtual Machine does depend on the platform, whereas the programming language Java does not – the byte code produced by the compiler can be transferred to every computer running a Java Virtual Machine.[19]

3.2 Kotlin

The programming language Kotlin is statically typed and named after an island in front of St. Petersburg. The Kotlin project was drawn up in 2010 by the Czech software development company JetBrains, which is well-known for several IDEs such as the IntelliJ IDEA.[11] Both functional programming as well as object-oriented programming is supported. Andrey Breslav, the lead language designer of Kotlin, states out that Kotlin is a language, which is completely compatible with Java by using Java libraries, providing Java APIs and integrating Java frameworks. When talking about Java libraries, Andrey Breslav emphasizes that Kotlin improves the existing Java libraries by extensions, but also by techniques supported by the compiler such as collections, primitives and arrays. In short, Kotlin adapts to the existing Java ecosystem. However, the main goal of the Kotlin creators is to provide a language, which is more concise and flexible compared to the existing ones.[10]

Andrey Breslav claims Kotlin to be a general-purpose language just like Java. According to that, Kotlin can be used in the following fields.

- Server-side applications
- Android applications
- Desktop applications

The first stable version of Kotlin was released in February 2016. The current version of Kotlin is 1.3 and has been released in October 2018. Moreover, Kotlin is an Open Source language developed on GitHub.[20] In 2017, Google announced Kotlin as an official programming language for developing Android applications.[12]

3.2.1 Architecture

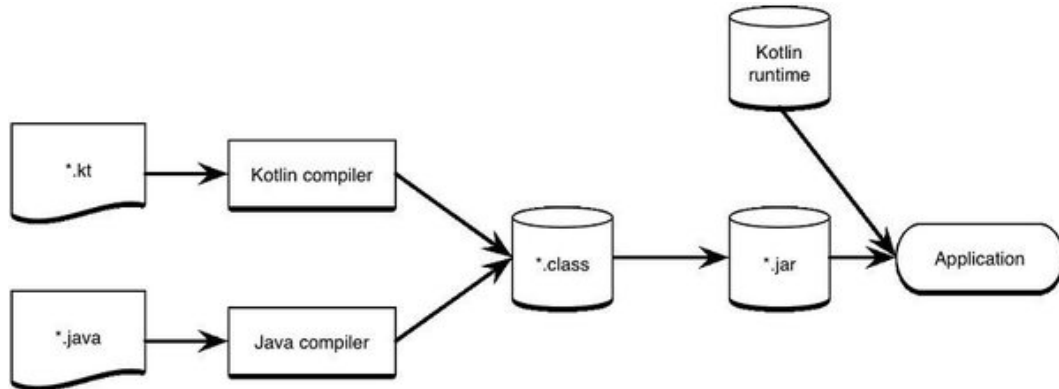


Figure 2: Kotlin Architecture [2]

The way the source code – written in Kotlin – is compiled into machine code depends on the language which is targeted. Supposing the targeted language is Java, the Kotlin compiler generates byte coded files out of the source code. This byte code is the same as the byte code which is generated by the Java compiler thus the generated files from both compilers are able to communicate with each other. This makes Kotlin files inter-operable with Java files.

Moreover, by using LLVM (a compiler framework), the Kotlin compiler is able to create code for several platforms. For instance, if JavaScript is targeted, the source code is converted into ES5.1 hence a compatible code for JavaScript is created.[21]

3.3 Syntactic Differences

In the following chapters, the most significant syntactic differences between Java and Kotlin are elaborated. Minor differences such as declaration of bindings, lambda expressions and interfaces will not be considered any further, since the differences are too small and therefore not worth mentioning.

3.3.1 Variable Declaration and Initialization

In Kotlin, two keywords are used to declare variables. The keyword 'var' is used to declare variables, which are mutable, while 'val' is used to declare variables, which are immutable. As it can be seen in the following example, the compiler is also aware of the data type (string) without having it explicitly declared.

```
1 val word = "hello"
2 word = "bye" // Value cannot be reassigned
```

Listing 1: Value Assignment in Kotlin Using 'val'

Using the 'var' keyword instead, makes it possible to change the value of the variable.

```
1 var word = "hello"
2 word = "bye" // Value of 'word' is now 'bye'
```

Listing 2: Value Assignment in Kotlin Using 'var'

In Java, however, it is necessary to declare the data type the developer wants to use.

```
1 String word = "hello"
2 word = "bye"
```

Listing 3: Value Assignment in Java Using a Datatype Declaration

When using the 'val' keyword as a class property as shown in the following listing, the value of the property depends on the parameters given to the method. This means that the value of the variable declared with the 'val' keyword changes every time the method is used with different parameters.

```

1 class MutableValExample(val sum: Int, val amount: Int) {
2     val average: Int
3     get() = (sum/amount)
4 }

```

Listing 4: Exception of Using 'val' and Being Immutable

By using the keyword 'lateinit' it is possible to initialize a variable later. However, the variable should be initialized before calling it, otherwise a null pointer exception will be thrown at runtime. In this case, it is up to the developer to take care of a possible null pointer exception.[22]

```

1 lateinit var questionTextView: TextView

```

Listing 5: Using the 'lateinit' Keyword for a Variable

To check if the variable has already been initialized, the function *.isInitialized()* can be called.[23]

```

1 if (foo :: questionTextView.isInitialized) {
2     println(foo.questionTextView)
3 }

```

Listing 6: Checking if a Variable has been Initialized

3.3.2 Conditional Expressions

In Java, the question mark operator can be used to shorten an if else block.

```

1 int max = (a > b) ? a : b;

```

Listing 7: Short Version of a Conditional Branch in Java

However, Kotlin offers a slightly different way to do the same.

```

1 val max = if (a > b) a else b

```

Listing 8: Expression Statement in Kotlin for Conditional Branching

Furthermore, a traditional switch statement looks like the following in Java.

```

1 int day = 5;
2 String dayString;
3 switch (day) {
4     case 1: dayString = "Monday";
5         break;
6     case 2: dayString = "Tuesday";
7         break;
8     case 3: dayString = "Wednesday";
9         break;
10    case 4: dayString = "Thursday";
11        break;
12    case 5: dayString = "Friday";
13        break;
14    case 6: dayString = "Saturday";
15        break;
16    case 7: dayString = "Sunday";
17        break;
18    default: dayString = "Invalid day";
19        break;
20 }

```

Listing 9: Switch Statement in Java

In Kotlin, when expressions replace the switch statements.

```

1 var x = 5
2 var day = "No valid day"
3 when (x) {
4     1 -> day = "Monday"
5     2 -> day = "Tuesday"
6     3 -> day = "Wednesday"
7     4 -> day = "Thursday"
8     5 -> day = "Friday"
9     6 -> day = "Saturday"
10    7 -> day = "Sunday"
11 }
12 print(day)

```

Listing 10: When Expression in Kotlin

3.3.3 Plain Old Java Objects

When declaring an object, the conciseness of Kotlin stands out. In order to declare the following object, 29 lines in Java would be needed.

```

1 class Car {
2     private String color;
3     private String manufacturer;
4     private int horsePower;
5
6     public Car(String color, String manufacturer, int horsePower) {
7         this.color = color;
8         this.manufacturer = manufacturer;
9         this.horsePower = horsePower;
10    }
11    public String getColor() {
12        return color;
13    }
14    public void setColor(String color) {
15        this.color = color;
16    }
17    public String getManufacturer() {
18        return manufacturer;
19    }
20    public void setManufacturer(String manufacturer) {
21        this.manufacturer = manufacturer;
22    }
23    public int getHorsePower() {
24        return horsePower;
25    }
26    public void setHorsePower(int horsePower) {
27        this.horsePower = horsePower;
28    }
29 }

```

Listing 11: Car Object in Java

However, in Kotlin it would be just one line.

```

1 data class Car (var color:String, var manufacturer:String, var horsePower: Int)

```

Listing 12: Car Object in Kotlin

3.4 Null Pointer Safety

When Tony Hoare was developing an object-oriented programming language in the 1960s, he decided to use null references, which are not safely checked by the compiler. At runtime, a reference to a null value, i.e. a value that is in fact no value, leads to throwing the well-known null pointer exception. Technically spoken, the problem of having null values in a code is, that null pretends to be a value, which is actually no value. Tony Hoare apologised in 2009 for what he called *the billion dollar mistake*. He stated out that this invention was eventually very expensive in the last decades in terms of maintainance, development, debugging and, of course, money.[24] In fact, the main reason behind the crashes of many Android applications are caused by unhandled null pointer exceptions.[25] In Java programs, the type checking ensures that the used variables will be run successfully when the program

is executed – except the variable is of type null and that will become a problem at run time. The following Java code will produce a null pointer exception at runtime.

```
1 String string = null; // No compilation error
2 System.out.println(string.toUpperCase()); // Throws a null pointer exception
```

Listing 13: Example of a Null Pointer Exception in Java

Whereas Kotlin provides null pointer safety; meaning that the type checking would already ring the alarm bells, since the example string is not explicitly declared as null.

```
1 var string: String = null // Compiler already tells what is wrong
2 println(string.toUpperCase()) // This is not even reached
```

Listing 14: Kotlin Compiler Telling the Problem of Having a Null Value

The question mark operator helps to explicitly declare a variable as nullable.

```
1 val string: String? = null // Variable is now null
2 println(string.toUpperCase()) // A regular call is not allowed on a nullable string
```

Listing 15: Explicit Declaration of a Null Value in Kotlin

Using doubled exclamation mark supports calling the null variable, so that eventually a null pointer exception is thrown. However, this should be used carefully.

```
1 val string: String? = null // Variable is now null
2 println(string!!.toUpperCase()) // The !! operator allows to eventually call the nullable string
```

Listing 16: Explicit Declaration and Call of a Null Value in Kotlin

In conclusion, it can be said that Kotlin made null pointer exceptions to compile time errors rather than to runtime errors, however it is still up to the developer to take care of potential null pointer exceptions. If a variable is declared as lateinit, it can still be null when reading it at run time. Therefore, the .isInitialized() function should be called as stated out earlier.

3.5 Comparative Overview

In addition to the syntactic differences already mentioned, the following chapters will give an overview of Java issues addressed in Kotlin, features and concepts which only Java has and vice versa. This comparison has been adopted from the official documentation of Kotlin.[6]

3.5.1 Java Issues Addressed in Kotlin

In the following table Java issues addressed in Kotlin are summarized. An explanation helps to understand the respective issue.

Java Issue Addressed in Kotlin	Explanation
Null References	As stated out earlier, the issue of having null references at run time are minimized by having a type system informing the developer about the issue at compile time.
No Raw Types	When it comes to converting code from Java to Kotlin, the syntax for generics is slightly different. For instance, <i>List</i> is converted into <i>List<*>!</i> .
Arrays	Arrays in Kotlin are invariant. It is not possible to assign <i>Array<String></i> to <i>Array<Any></i> , for example. By that, a possible failure at run-time is avoided.
Function Types	In Kotlin, function types such as <i>(Int) -> String</i> are used.
Use-Site Variance	Use-site variance and wild cards allow types which cannot be assigned to anything. By disallowing this feature, future problems are avoided.
Checked Exceptions	In Kotlin, no exceptions are checked at compile time for the purpose of better productivity and less occurrences of bad practices in large projects. If having checked exceptions is good or not, is a major topic of debate among developers.[26]

Table 1: Java Issues Addressed in Kotlin (adopted from [6])

3.5.2 Only Available in Java

In the following table features which are only available in Java are listed. Due to the intention of the developers of Kotlin to address issues in Java, of course, table 2 will contain concepts and features which were addressed (as issues) in table 1.

Only available in Java	Explanation
Checked Exceptions	As mentioned before, Kotlin does not have checked exceptions for the purpose of productivity and enhanced code quality in big projects.
Primitive Types	Java has primitive types such as int, byte and double. In Kotlin, the primitive types from Java are derived from the class <i>Number</i> .
Static Members	Static methods such as <i>public static void main(String[] args)</i> do not exist in Kotlin.
Non-Private Fields	Fields in Java can be publicly accessed, while in Kotlin properties are used.
Wildcard-Types	Unknown types of generics are not supported by Kotlin since class cast exceptions may occur at runtime.
Ternary-Operator	Ternary operators are not used in Kotlin and replaced by expressions as explained in the previous chapter.

Table 2: Features and Concepts Only Available in Java (adopted from [6])

3.5.3 Only Available in Kotlin

Furthermore, according to the official documentation of Kotlin, the following features are features only available in Kotlin (and not in Java).[6]

- Performant Custom Control Structures
- Extension Functions
- Null-safety
- Smart Casts
- String Templates
- Properties
- Primary Constructors
- First-class Delegation
- Type Inference for Variable and Property Types
- Singletons
- Declaration-site Variance and Type Projections
- Range Expressions
- Operator Overloading
- Companion Objects
- Data Classes
- Separate Interfaces for Read-only and Mutable Collections
- Coroutines

3.6 Hybrid Programming

As explained earlier, the architecture of Kotlin makes it possible to use Kotlin files as well as Java files in a single project. By using the 'Car' example again, an object is added to the project.

```
1 class Car {
2     private String color;
3     private String manufacturer;
4     private int horsepower;
5
6     public Car(String color, String manufacturer, int horsepower) {
7         this.color = color;
8         this.manufacturer = manufacturer;
9         this.horsePower = horsepower;
10    }
11    public String getColor() {
12        return color;
13    }
14    public void setColor(String color) {
15        this.color = color;
16    }
17    public String getManufacturer() {
18        return manufacturer;
19    }
20    public void setManufacturer(String manufacturer) {
21        this.manufacturer = manufacturer;
22    }
23    public int getHorsePower() {
24        return horsepower;
25    }
26    public void setHorsePower(int horsepower) {
27        this.horsePower = horsepower;
28    }
29 }
```

Listing 17: Car Object in Java

Now, an instance of the object can be created by using Kotlin. Also, the getter methods can be used by means of the dot operator.

```
1 fun main(args: Array<String>) {
2     val beetle = Car("Red", "VW", 90)
3
4     println (beetle.getManufacturer()) // VW
5     println (beetle.getColor()) // Red
6     println (beetle.getHorsePower()) // 90
7 }
```

Listing 18: Main Method in Kotlin

In some cases it makes sense to convert single files from Java to Kotlin and vice versa. JetBrains, as the developer of Kotlin, supports these conversions in the IntelliJ IDEA (Java integrated development environment). Using the built-in tools would change the aforementioned code snippets to the following. The class Car would be converted, as demonstrated before, to a simple one-liner.

```
1 data class Car (var color:String, var manufacturer:String, var horsepower: Int)
```

Listing 19: Car Object in Kotlin

While the main function written in Kotlin is converted to the following.


```

1 public final class MainKt {
2     public static final void main(@NotNull String[] args) {
3         Intrinsic .checkParameterIsNotNull(args, "args");
4         Car beetle = new Car("red", "VW", 90);
5         String var2 = beetle.getManufacturer();
6         System.out.println(var2);
7         var2 = beetle.getColor();
8         System.out.println(var2);
9         int var3 = beetle.getHorsePower();
10        System.out.println(var3);
11    }
12 }

```

Listing 20: Main Method Converted from Kotlin to Java

At this point, some annotations have to be made. The names of the auxiliary variables being used for printing the attributes of the object have no meaning and are usually considered as bad practice. In a real case scenario, an experienced programmer would never use 'var2' or 'var3' in order to name a variable. Furthermore, no class declaration is being used for Kotlin, while the Java code includes the final attribute in the class declaration as well as in the main method, meaning that inheritance is prohibited. However, this is only the case when converting the main function from Kotlin code to Java code. Also, it is not easily recognizable at first sight, what the Kotlin code for the Java class 'Car' actually includes. When seeing this piece of code (listing 19), developers have to bear in mind that the same functions are available as in the counterpart in Java.

3.7 Trends and Statistics

A survey conducted by Stackoverflow in 2018 shows that many developers expressed their sympathy for Kotlin. The following graphs represent the percentage of developers who are currently using the respective programming language and want to carry on working with it. As it can be seen, Kotlin is in the second place with 75.1 percent while roughly half of the developers using Java want to continue programming with it.

Most Loved, Dreaded, and Wanted Languages

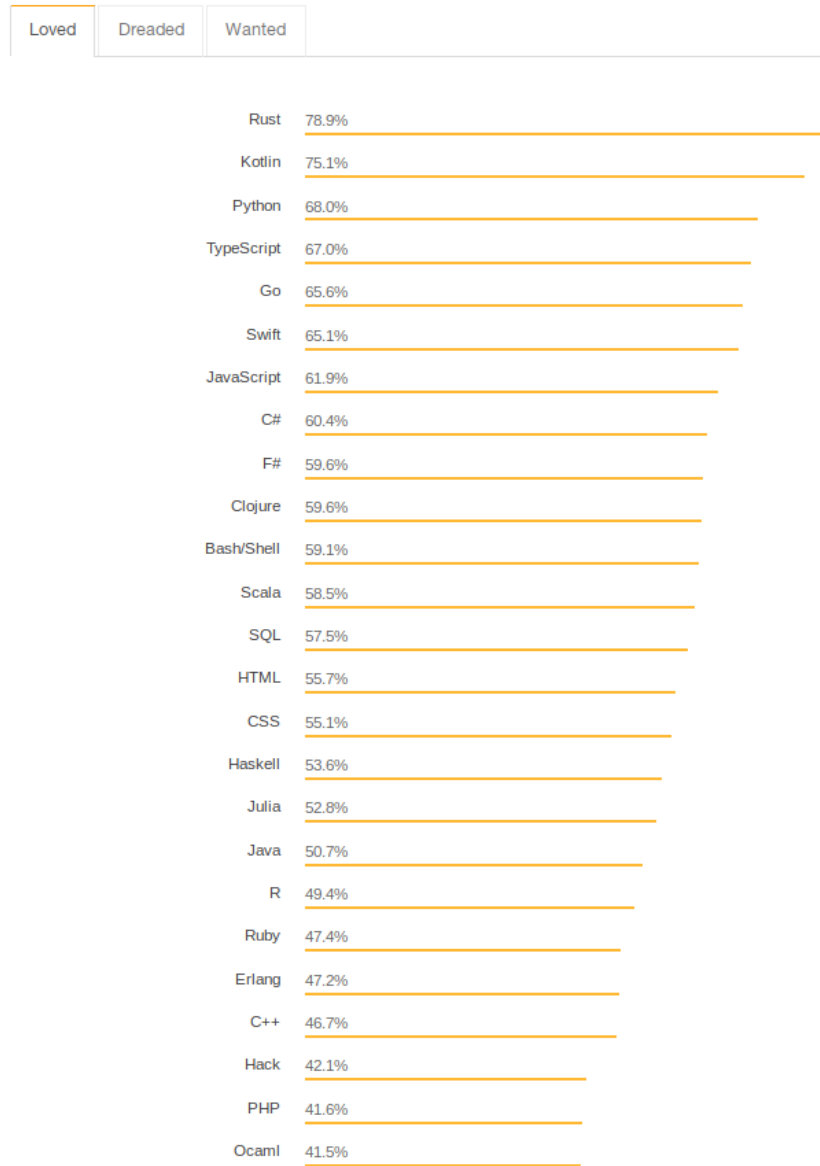


Figure 3: Most 'Loved' Programming Languages [3]

Most Loved, Dreaded, and Wanted Languages

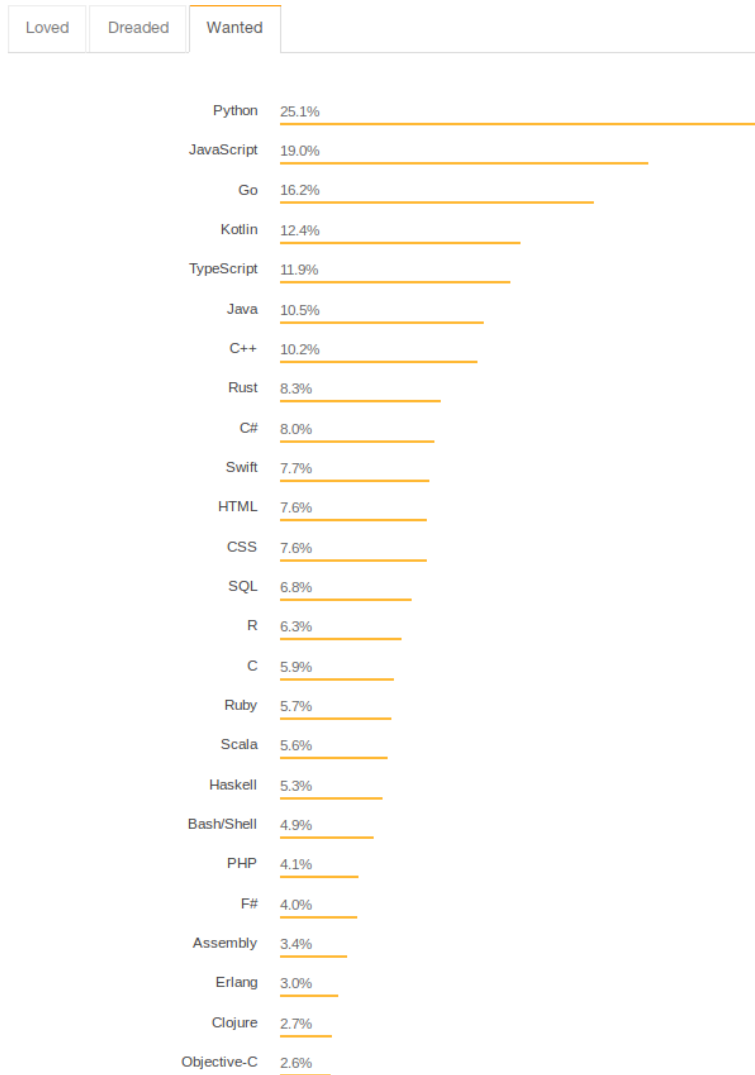


Figure 4: Most Wanted Programming Languages [3]

While there is a difference of roughly 25 percentage points regarding 'Loved Languages', the difference between Java and Kotlin in the ranking of 'Wanted Languages' is quite small. In this case, the percentage for each programming language represents developers who would like to learn the respective language.

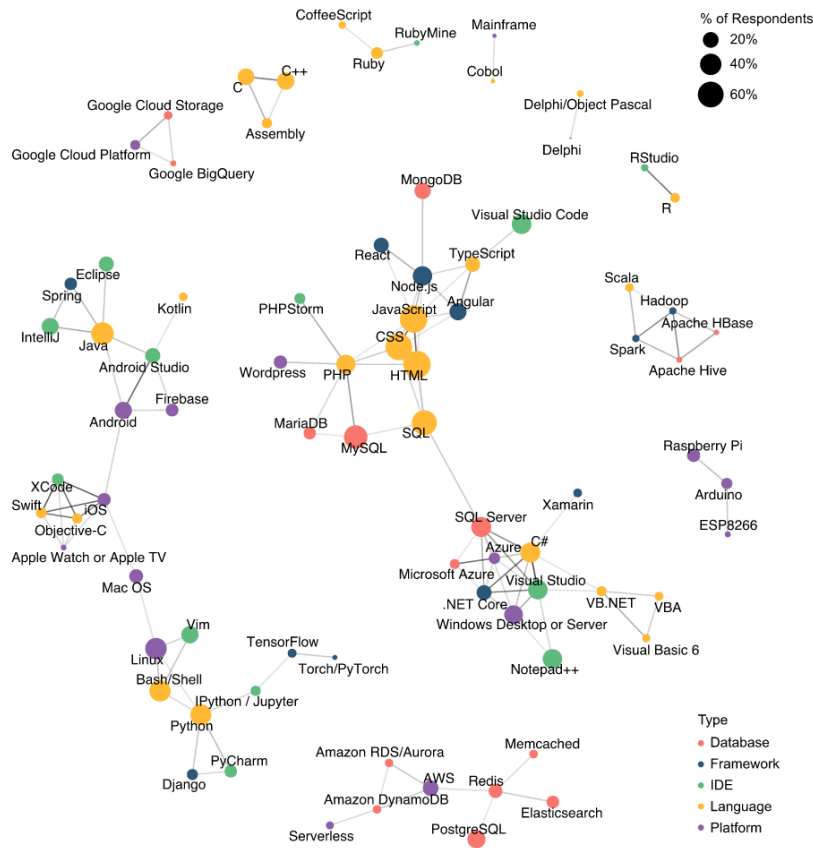


Figure 5: Correlated Technologies [3]

The cluster in figure 5 shows that developers tend to use Java with the well-known Spring Framework as well as with the IDEs Eclipse and IntelliJ. Furthermore, Java seems to be used for more than developing Android applications, while most of the developers using Kotlin are mainly using it for Android application development, even though it can be used for the same purposes (as mentioned earlier) as Java is being used.

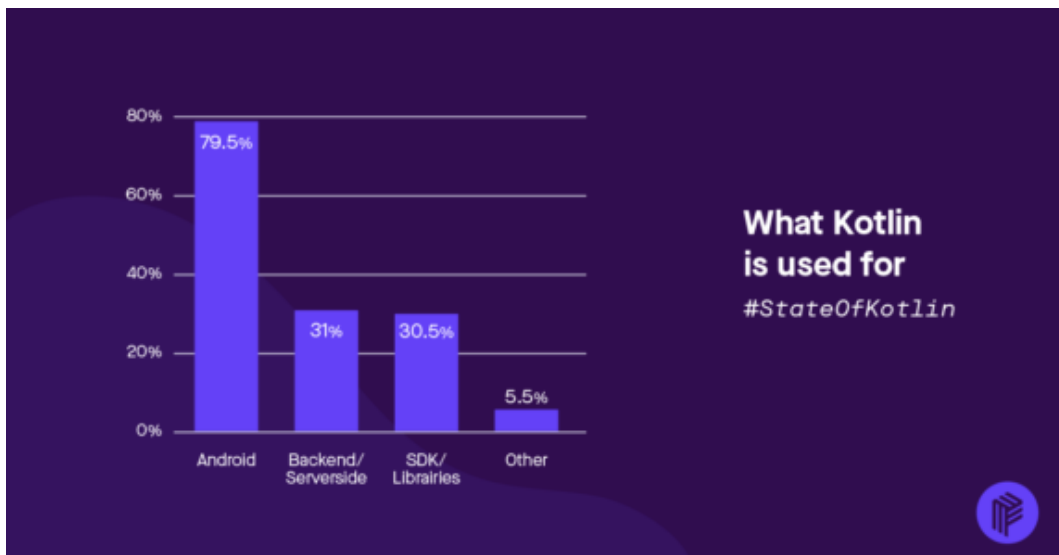


Figure 6: Usage of Kotlin [4]

More than 2700 people participated in a survey conducted from January to March 2018 by Pusher (a development platform). Here too, it becomes clear that most of the developers using Kotlin use it for the purpose of developing Android applications.

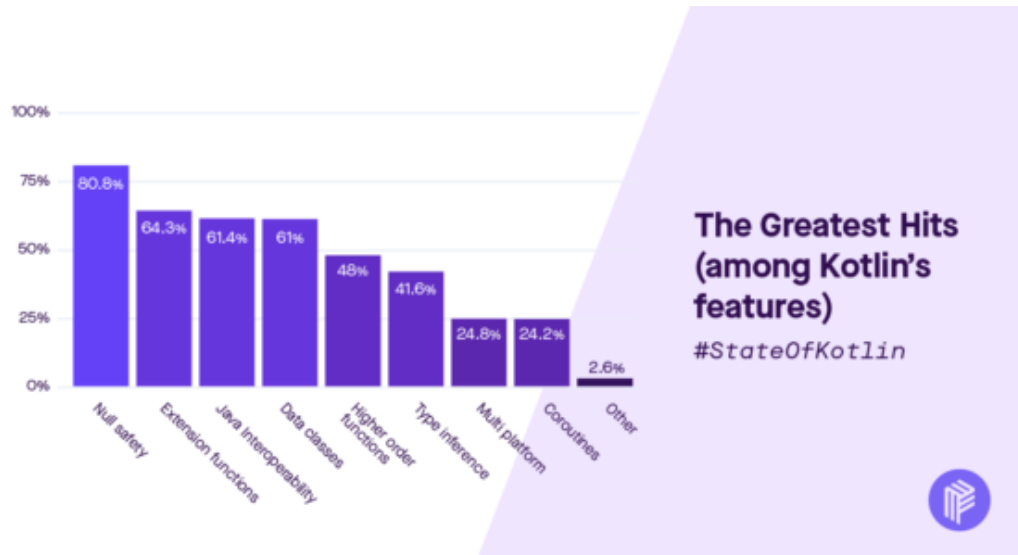


Figure 7: Greatest Hits of Kotlin [4]

As stated previously, null pointer exceptions are the main cause for crashes of Android applications.[25] It is therefore understandable that having null safety in Kotlin is considered as the most popular feature according to the survey conducted by Push.

3.8 Code Smells

Having bad practices in a source code is also known as having code smells. Code smells make it hard to maintain a current project or to develop it any further.[27] According to Fowler et al. [28], 22 code smells can be identified in source codes. For this study, the following code smells are relevant. The code smell *Long Filename* is being introduced just in this study.

Code Smell	Abbreviation	Explanation
Long Filename	LF	This smell refers to a filename that is too long. For one thing, long filenames impair the readability and secondly, long names might be an indication for having a file with too many responsibilities.
Data Class	DC	A class that is only used as a data holder and does not contain any further logic operations other than getters and setters.
Large Class	LC	Classes which have many lines of code are an indication for too many responsibilities. Moreover, long classes impair intuitive reading.
Long Method	LM	Methods which have many lines of code are also an indication for too many responsibilities. Long methods impair intuitive reading, too.
Parameter List	LPL	A method should be able to find parameters it needs rather than getting all parameters it needs. A method should therefore not receive a lot of parameters.
Too Many Methods	TMM	Here again, too many methods are an indication of a class having many responsibilities. Outsourcing similar methods would be an option to solve this issue.

Table 3: Code Smells

4 Experimental Setup

As the experiment which was conducted represents a data mining problem definition, a well-proven framework was chosen that is commonly used in real-case scenarios in the industry to ensure quality results.[29]

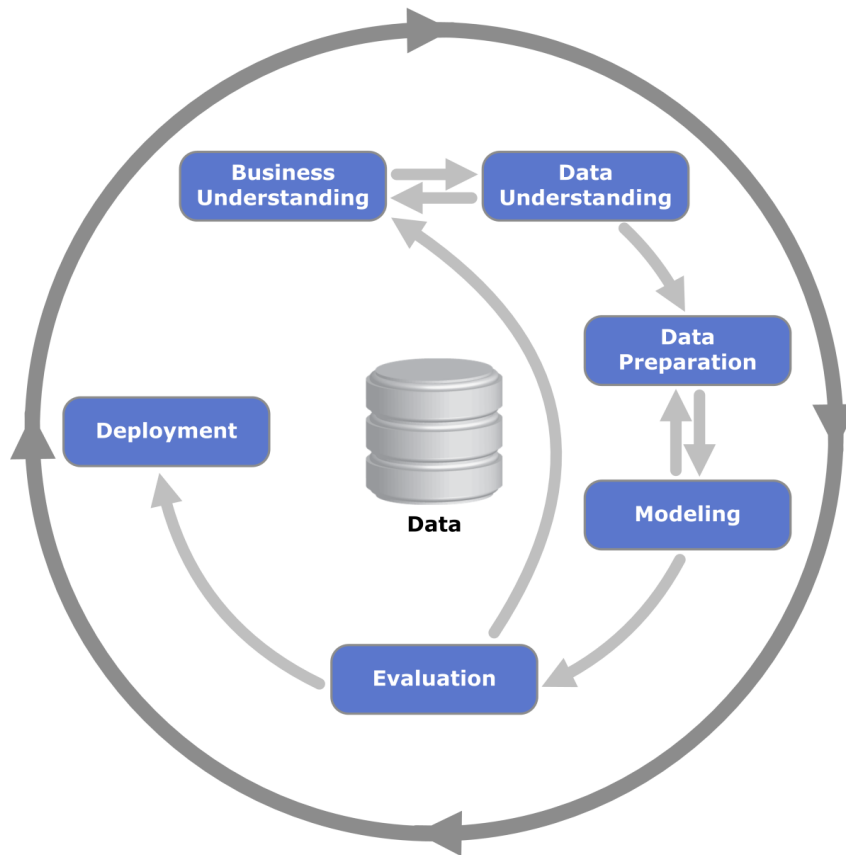


Figure 8: The Cross-Industry Standard Process for Data Mining [5]

The cross-industry standard process for data mining (CRISP-DM) is considered as flexible, robust and very practical when it comes to complex business cases. Moreover, the shown order between each single step is not always strictly applied, also slightly different models exist.[30] Since it is common practice to apply the CRISP-DM in the industry, it will be applied in this study as a framework in order to have a logically structured experiment. Based on the idea of the study "Are you still smelling it?" [7], an experiment was conducted. The individual steps of the experiment are derived from the CRISP-DM, which can be seen in figure 8. In the first step, the superordinate problem or rather the business problem has to be converted into a data mining problem definition. Before that, the main goal of the study must be clear as well as the requirements by taking the business perspective. In the Data Understanding phase, data are collected and considered more precisely. Especially interesting findings as well as lacks of data quality are particularly interesting in this context – especially with regard to the previous step. Before one is able to work with the data, the raw data have to be prepared in such a manner, one can work with it in the steps afterwards. The methods and techniques applied in the modelling phase depend on the structure of the data and vice versa – this explains the loop back to the previous phase: Data Preparation. In order to be able to tell if one of the applied models is performing well towards predicting yet unknown data, an evaluation is necessary. The outcome of this phase will be one model giving a good forecast. If no well performing model was found, it might be appropriate to go back to the first step: Business Understanding. When deploying the model that was developed during the previous steps, it usually means that in the future a forecast predicts unseen data and information or rather solutions to meet the requirements identified in the first step when the business perspective was taken. Of course, a code representation should then process the new raw data as it has when developing the model.[31]

5 Experiment

The experiment is composed of six phases according to the CRISP-DM. In the next chapters, the data mining model will be used as a framework to analyse Java and Kotlin repositories on GitHub. The repositories were downloaded together with meta data such as lifespan, number of open issues as well as the stargazers count (rating of a repository). Code smells could be identified by using the tools PMD (a code analysing tool for Java), Detekt (a code analysing tool for Kotlin) and a bash script for counting the characters of filenames.[32][33]

5.1 Business Understanding

Flauzino et al. [7] analysed 100 GitHub projects: 50 Java projects and 50 Kotlin projects on the basis of five code smells:

- Data Class
- Large Class
- Long Method
- Long Parameter List
- Too Many Methods

The main goal of the study was to find out whether the analysed Java projects or Kotlin projects contain more code smells. Furthermore, the number of commits, lines of code as well as the lifespan of a project were fetched. According to Flauzino et al. [7], several scripts were used to download and analyse the GitHub repositories. Based on this idea, a similar experiment was conducted.

5.2 Data Understanding

The main reason for conducting a similiar experiment was that the study conducted by Flauzino et al. [7] was deficient in that the figures it mentioned were not reliable, firstly due to the reason that Detekt and PMD are counting lines of code differently. Detekt counts only plain code – this means neither curly brackets appearing alone in a line nor method declarations are counted as lines of code. However, PMD does count these lines – as a matter of fact, even empty lines are counted as lines of code. It can therefore be concluded that using code smells like LC or LM will lead to distorted results. Secondly, the code smell Data Class is discussed controversially by professionals.[34] Therefore this study will not take any of these code smells into consideration. Last but not least, thresholds for respective code smells are not discussed in the aforementioned study. Of course, there is no such thing as fixed thresholds for code smells and it should be therefore declared which are being used. In case of using default thresholds, a comparison would be misleading again: PMD uses 1000 as a maximum for code lines of classes, 100 for methods, while Detekt uses 150 for classes and 20 for methods.[35][36] Table 4 shows code smells in Kotlin projects found by Flauzino et al.[7] Unfortunately, explanations for thresholds (e.g. 20 lines for a method or 50 for a class) and why a whole column is filled with zeros (DC) are not provided.

Program Name	Bad Smell					Total	#Commits	LOC	Lifespan
	DC	LC	LM	LPL	TMM				
awesome-kotlin	0	0	0	0	0	0	814	35825	1016
Design-Patterns-In-Kotlin	0	0	0	0	0	0	101	405	610
GankClient-Kotlin	0	0	0	0	0	0	46	894	553
gradle-play-publisher	0	0	0	0	0	0	357	915	1391
kotlin-examples	0	0	0	0	0	0	154	1545	1898
Kotlin-for-Android-Developers	0	0	0	0	0	0	22	685	1057
Kotlin-Tutorials	0	0	0	0	0	0	118	10398	597
profile-summary-for-github	0	0	0	0	0	0	194	350	170
SdkSearch	0	0	0	0	0	0	450	2488	166
transitioner	0	0	0	0	0	0	33	131	172
android-architecture-components	0	0	0	0	1	1	168	6174	385
android-clean-architecture-boilerplate	0	0	0	0	1	1	65	2181	293
Colorful	0	0	0	1	0	1	53	264	602
JellyToolbar	0	0	0	0	1	1	20	425	427
kotlin-koans	0	0	0	0	1	1	214	1696	1531
kotterknife	0	0	0	0	1	1	82	251	1317
Android-TextView-LinkBuilder	0	0	0	0	2	2	111	703	1070
sqldelight	0	0	0	0	2	2	860	11024	951
Bandhook-Kotlin	0	0	0	1	2	3	107	3167	1165
dexcount-gradle-plugin	0	2	1	0	1	4	300	1177	1090
Fuel	0	0	0	0	5	5	423	5581	1090
TourGuide	0	1	0	0	4	5	190	1760	1152
mapdb	0	0	0	0	6	6	2100	2190	2111
SearchFilter	0	2	1	0	3	6	23	1381	579
kotlinconf-app	0	1	0	0	6	7	6	3879	215
Fotoapparat	0	0	0	0	9	9	582	5703	432
kotlinpoet	0	2	0	1	6	9	1111	10398	454
p3c	0	0	0	4	5	9	172	5302	340
android-topeka	0	1	0	0	9	10	239	4077	1117
shadowsocks-android	0	1	2	1	8	12	2412	5354	1990
flexbox-layout	0	2	9	1	2	14	329	8206	755
RxDownload	0	1	0	0	13	14	456	2723	578
spek	0	0	0	5	11	16	586	3819	2031
okio	0	4	0	1	12	17	663	5733	1534
android-ktx	0	2	2	0	14	18	438	5894	182
Exposed	0	2	0	2	14	18	951	8333	1764
SuperSLiM	0	3	0	11	7	21	177	4338	1258
muzei	0	5	3	3	13	24	1496	12153	1573
kotlin-dsl	0	0	0	8	17	25	1902	15207	762
intellij-rust	0	5	0	1	21	27	5495	60348	553
kotlinx.coroutines	0	6	1	0	21	28	804	31567	706
RxKotlin	0	0	0	30	5	35	211	1850	1379
tachiyomi	0	9	1	2	58	70	1417	24902	965
ktor	0	5	1	32	34	72	2013	38172	1030
tornadofx	0	9	0	37	37	83	2535	21082	874
anko	0	1	0	8	125	134	994	38169	1349
corda	0	27	13	66	88	194	5814	106854	600
Twidere-Android	0	37	45	66	94	242	2888	69499	1426
arrow	0	0	0	294	51	345	3672	22829	431
kotlin-native	0	55	200	251	208	714	3771	178650	743

Table 4: Code Smells Found in Kotlin Projects (Data Collected by Flauzino et al. [7])

However, table 5 shows Java projects, whereas the column DC is not filled with zeros only.

Program Name	Bad Smell					Total	#Commits	LOC	Lifespan
	DC	LC	LM	LPL	TMM				
CircleImageView	0	0	0	0	0	0	122	356	1593
Material-Animations	0	0	0	0	1	1	76	992	1178
RxAndroid	0	0	0	0	1	1	461	1181	1379
PhotoView	0	0	1	0	0	1	427	1589	2142
interviews	0	0	1	0	0	1	390	7300	469
AndroidSwipeLayout	0	1	1	0	1	3	175	2667	1373
ViewPagerIndicator	0	0	1	0	3	4	241	3072	2490
SlidingMenu	0	0	2	0	3	5	723	3212	2159
Android-CleanArchitecture	3	0	0	0	3	6	243	2777	1379
leakcanary	2	0	1	0	5	8	454	4243	1126
EventBus	2	0	0	0	10	12	480	5333	2143
BaseRecyclerViewAdapterHelper	7	1	1	0	3	12	908	5672	779
android-async-http	3	1	3	0	7	14	874	3446	2655
material-dialogs	0	2	4	0	8	14	1560	7584	1303
Android-Universal-Image-Loader	3	1	1	0	11	16	1025	5276	2375
butterknife	0	2	8	1	7	18	836	10653	1911
picasso	0	1	1	2	16	20	1198	9632	1841
SmartRefreshLayout	1	1	9	2	10	23	723	17064	361
stetho	2	0	2	0	19	23	528	18214	1223
lottie-android	11	0	3	4	11	29	925	11214	600
retrofit	1	3	2	0	23	29	1572	19419	2822
java-design-patterns	31	0	1	0	5	37	2060	27426	1389
plaid	12	1	9	6	16	44	453	16921	942
AndroidUtilCode	6	8	5	2	28	49	863	22481	668
MPAndroidChart	5	3	25	2	15	50	1983	24910	1495
androidannotations	16	0	5	1	45	67	2774	37211	2335
greenDAO	30	0	0	5	33	68	844	19663	2411
zxing	18	0	21	9	26	74	3431	105107	2421
jadx	22	1	9	0	46	78	662	44942	1898
tinker	8	2	35	16	26	87	295	31826	630
iosched	18	9	25	2	53	107	1548	57854	1519
Hystrix	17	5	23	23	44	112	2106	50510	2017
okhttp	7	11	6	4	88	116	3159	56228	2136
fresco	6	1	5	10	98	120	1657	428112	1184
selenium	8	2	8	2	110	130	22349	85807	1961
glide	1	3	7	9	117	137	2211	70355	2411
Signal-Android	25	9	10	17	79	140	3543	80634	2357
zheng	75	3	4	0	76	158	1222	30057	602
incubator-dubbo	89	11	40	4	146	290	2308	103349	2170
ExoPlayer	48	19	35	58	149	309	4788	113451	1446
RxJava	0	54	55	8	364	481	5366	271036	1967
spring-boot	276	10	2	2	245	535	16829	247388	2048
netty	22	35	84	26	485	652	8797	250399	2758
fastjson	490	19	89	5	80	683	2705	149361	2399
druid	297	44	209	1	176	727	5812	295170	2399
Telegram	54	72	523	62	192	903	311	325744	1677
libgdx	70	37	141	266	462	976	13254	274702	2118
spring-framework	362	53	81	17	818	1331	16597	602324	2729
guava	10	82	49	34	1167	1342	4725	381338	1461
elasticsearch	351	86	415	214	1266	2332	39203	1055917	3032

Table 5: Code Smells Found in Java Projects (Data Collected by Flauzino et al. [7])

In order to find out which programming language causes more code smells, the experiment was conducted in a similar way, whereas the smells DC, LC and LM have not been taken into account due to the reasons mentioned earlier. However, the following smells are considered in this study, whereas t represents the threshold for each smell. $t=2$ for LPL means that a method or function that takes more than two parameters is counted as a smell. The length of filenames is determined by counting characters.

- Long Filename ($t=30$)
- Long Parameter List ($t=2$)
- Too Many Methods ($t=5$)

Of course, these thresholds represent a personal subjective opinion. In fact, there is no general rule.[37] But there has to be a limit somewhere. Table 6 and 7 show an excerpt of the collected data. In total, 1000 Java and 1000 Kotlin repository names were downloaded from GitHub using a Python program, which can be seen in appendix B. In order to avoid duplicated code, projects with exactly the same name were removed and the one with the most stargazers remained. As a result, 991 Java repositories and 995 Kotlin repositories were downloaded. As it can be seen in the tables 6 and 7, the projects are sorted by stargazers in descending order. Several scripts were then being executed to analyse the downloaded

repositories and to write the results in CSV files. All scripts and the Python program used for downloading and counting smells can be found in the appendices A, B, C and D.

Project	LF	LPL	TMM	Total	Lifespan	Issues	LOC	Commits	Contributors	Stargazers
iluwatar/java-design-patterns	8	97	30	135	1569	180	64318	2743	196	41682
ReactiveX/RxJava	103	1130	670	1903	2147	29	343496	6524	324	36456
elastic/elasticsearch	847	10577	3068	14492	3212	1829	1490246	73023	1404	36282
spring-projects/spring-boot	1458	1119	580	3157	2228	389	382812	19876	611	31331
square/retrofit	8	139	42	189	3002	70	25133	1759	154	30429
kdn251/interviews	26	92	0	118	649	31	13903	438	35	30317
square/okhttp	3	392	152	547	2316	170	72217	3928	220	29776
google/guava	260	598	553	1411	1641	788	669615	6332	300	28230
PhilJay/MPAndroidChart	7	262	56	325	1675	1300	32250	2007	84	24980
spring-projects/spring-framework	1471	4248	1598	7317	2909	179	930140	21524	384	24891
bumptech/glide	29	487	179	695	1966	143	79992	2557	149	24048
airbnb/lottie-android	6	126	17	149	780	44	12705	1110	64	23272
apache/incubator-dubbo	37	654	278	969	2350	250	137141	3081	178	22649
JakeWharton/butterknife	0	86	12	98	2091	83	12110	968	90	22571
square/leakcanary	4	41	16	61	1306	22	7374	632	69	21179
Blankj/AndroidUtilCode	1	455	80	536	848	30	36786	961	29	21109
zxing/zxing	19	442	75	536	2601	8	57910	3482	136	20766
greenrobot/EventBus	27	35	12	74	2323	98	7004	504	27	19748
proxyee-down-org/proxyee-down	0	22	7	29	396	110	4038	476	16	18660
ReactiveX/RxAndroid	0	2	2	4	1559	3	1451	475	69	17135

Table 6: Excerpt of Java Code Smells (Data Collected on November 27th)

Project	LF	LPL	TMM	Total	Lifespan	Issues	LOC	Commits	Contributors	Stargazers
JetBrains/kotlin	4746	11202	2896	18844	2477	96	1030909	81451	370	25332
shadowsocks/shadowsocks-android	5	10	20	35	2170	10	7815	2614	44	19057
tipsy/profile-summary-for-github	0	0	0	0	350	11	361	198	10	18323
google/iosched	29	45	30	104	1699	22	25634	2348	67	17697
afollestad/material-dialogs	0	30	12	42	1483	2	5219	1724	90	14675
Kotlin/anko	5	937	222	1164	1529	206	51278	1089	76	12883
google/flexbox-layout	3	4	4	11	935	41	9066	570	23	12866
alibaba/p3c	17	38	21	76	520	13	7033	210	27	12293
googlesamples/android-architecture-components	1	7	9	17	565	205	10084	230	40	10849
googlesamples/android-UniversalMediaPlayer	0	0	4	4	1356	18	2739	268	26	10428
JakeWharton/RxBinding	51	0	0	51	1342	33	4691	555	46	8171
KotlinBy/awesome-kotlin	0	2	0	2	1196	59	1472	905	160	6059
square/okio	0	45	19	64	1714	26	10067	809	47	5710
JetBrains/kotlin-native	13	727	201	941	923	137	101833	4933	89	5164
ReactiveX/RxKotlin	0	65	0	65	1559	10	2513	307	49	5012
googlesamples/android-topeka	0	16	11	27	1297	8	5976	241	22	4993
googlesamples/android-sunflower	2	0	1	3	186	45	1971	260	14	4979
agrosner/DBFlow	1	78	78	157	1530	39	22484	2876	66	4411
ktorio/ktor	11	272	80	363	1210	207	56695	2685	68	4113
Tapadoo/Alerter	0	2	3	5	660	16	1578	303	24	4048

Table 7: Excerpt of Kotlin Code Smells (Data Collected on November 27th)

5.3 Data Preparation

By using R (a statistical programming language for computing and graphics) the collected data are processed further. The complete code used for that can be found in the appendices E and F. A common method in machine learning is the practice of dividing data into training and test data.[38] Thus the training data can be used to develop a model and the test data to evaluate it. 400 Projects of both programming languages were randomly selected for training data. The rest of the data were used as test data.

5.4 Modelling

As already stated out by Flauzino et al. [7], code smells strongly correlate with the lines of code.[7] Furthermore, lifespan, number of open issues, number of commits, the amount of contributors as well as the number of stargazers correlate with the total amount of smells. Figure 9 and 10 show the correlation coefficients between the variables in Java and Kotlin projects.

	LF	LPL	TMM	Total	Lifespan	Issues	LOC	Commits	Contributors	Stargazers
LF	1.00	0.44	0.58	0.56	0.20	0.31	0.61	0.25	0.37	0.12
LPL	0.44	1.00	0.93	0.99	0.27	0.33	0.91	0.76	0.71	0.14
TMM	0.58	0.93	1.00	0.97	0.29	0.45	0.96	0.70	0.73	0.20
Total	0.56	0.99	0.97	1.00	0.28	0.37	0.94	0.74	0.72	0.16
Lifespan	0.20	0.27	0.29	0.28	1.00	0.30	0.28	0.25	0.39	0.17
Issues	0.31	0.33	0.45	0.37	0.30	1.00	0.36	0.16	0.32	0.38
LOC	0.61	0.91	0.96	0.94	0.28	0.36	1.00	0.79	0.77	0.18
Commits	0.25	0.76	0.70	0.74	0.25	0.16	0.79	1.00	0.87	0.12
Contributors	0.37	0.71	0.73	0.72	0.39	0.32	0.77	0.87	1.00	0.30
Stargazers	0.12	0.14	0.20	0.16	0.17	0.38	0.18	0.12	0.30	1.00

Figure 9: Correlation Matrix of Java Projects (Training Data)

	LF	LPL	TMM	Total	Lifespan	Issues	LOC	Commits	Contributors	Stargazers
LF	1.00	0.58	0.24	0.65	0.04	0.06	0.47	0.30	0.09	0.03
LPL	0.58	1.00	0.71	0.99	0.14	0.29	0.71	0.52	0.38	0.06
TMM	0.24	0.71	1.00	0.78	0.19	0.40	0.84	0.75	0.45	0.09
Total	0.65	0.99	0.78	1.00	0.15	0.31	0.78	0.60	0.39	0.07
Lifespan	0.04	0.14	0.19	0.15	1.00	0.21	0.17	0.28	0.24	0.22
Issues	0.06	0.29	0.40	0.31	0.21	1.00	0.35	0.41	0.48	0.13
LOC	0.47	0.71	0.84	0.78	0.17	0.35	1.00	0.83	0.51	0.14
Commits	0.30	0.52	0.75	0.60	0.28	0.41	0.83	1.00	0.64	0.25
Contributors	0.09	0.38	0.45	0.39	0.24	0.48	0.51	0.64	1.00	0.42
Stargazers	0.03	0.06	0.09	0.07	0.22	0.13	0.14	0.25	0.42	1.00

Figure 10: Correlation Matrix of Kotlin Projects (Training Data)

When plotting the lines of code a project has as the explanatory variable and the total number of code smells a project has as the response variable, the connection becomes clear. Figure 11 shows Java and Kotlin projects together in one plot.

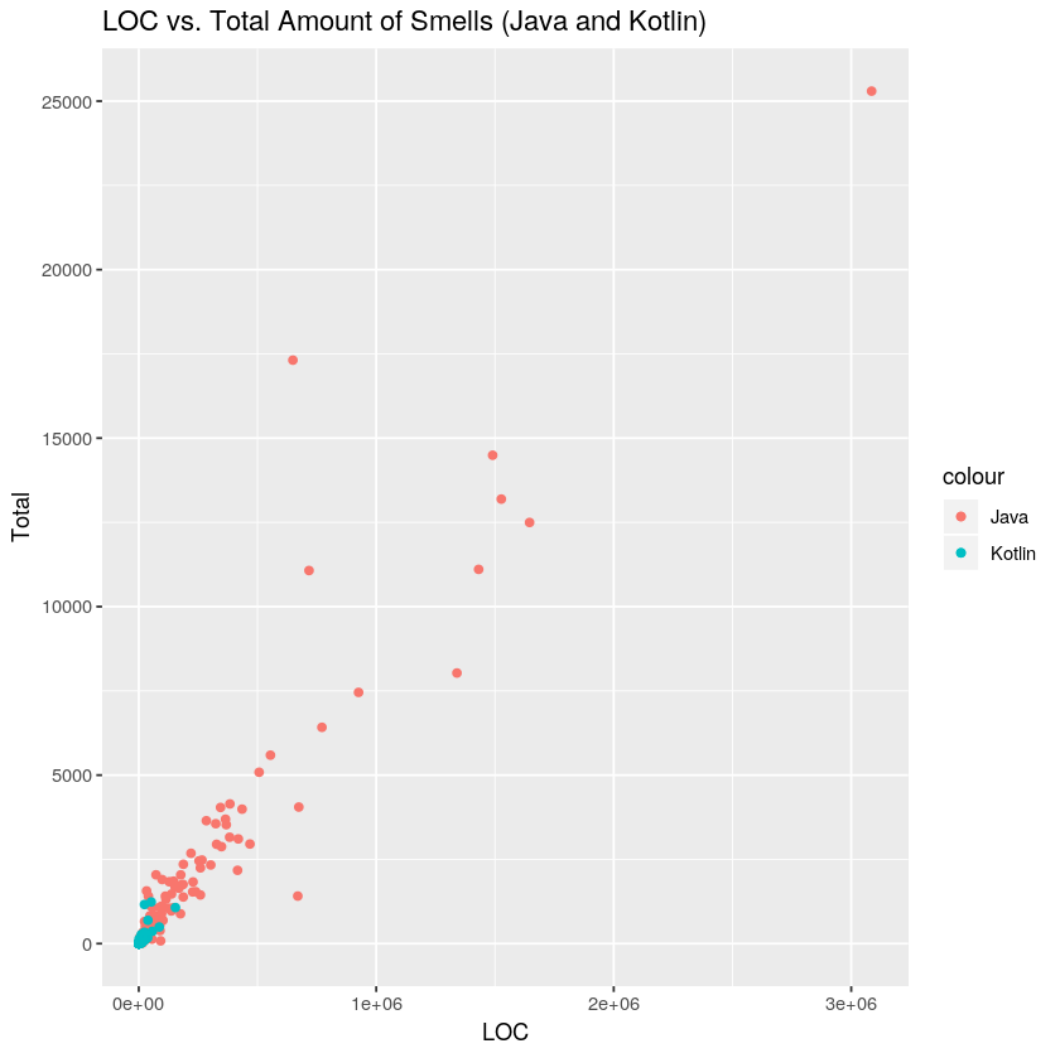


Figure 11: LOC and Total Amount of Smells in Java and Kotlin Projects (Training Data)

Since figure 11 and the correlation matrices indicate that the amount of smells a project has is substantially influenced by more variables than just the programming language, these influencing factors should not be discounted in order to get an authentic result. To find out to which extent a linear relationship between dependent variable and independent variables exist, a multidimensional regression of the shape

$$\hat{Y} = b_0 + b_1X_1 + b_2X_2 + \dots + b_pX_p$$

was used as it is a well established technique in statistical data analysis to predict future data.[39] \hat{Y} stands for predicted values of the total amount of smells and the X values for the explanatory variables shown in figure 9 and 10 (lifespan, issues, lines of code, contributors and stargazers). Since there is a strong correlation between number of commits and lines of code, the number of commits would not deliver new insights and remains therefore unconsidered. Furthermore, an established statistical method to take care of the uncontrolled independent variables is the analysis of variance with covariates (ANCOVA).[40] By making use of this method, the results will be truly robust and comparable. In order to be able to conduct an ANCOVA, specific assumptions have to be met.[41]

5.4.1 Assumptions for ANCOVA

The assumptions for conducting an ANCOVA in this study are as follows.

- **Assumption 1:** Covariate and dependent variable should be measured on a continuous scale, the covariate can also be categorical. All the covariates as well as the dependent variable (total amount of smells) are on a continuous scale as it can be seen in table 6 and 7.
- **Assumption 2:** The observations should be independent from one another. GitHub projects are independent from one another, except when developers are forking repositories. By deleting duplicates when downloading the repositories, only GitHub projects left which should be largely independent from each other.
- **Assumption 3:** The data should be approximately normally distributed. The original data set fetched from GitHub did not look normally distributed. The Q-Q plots in the figures 12 and 13 should show a line going from the lower left corner to the upper right corner in order to meet the criteria of normality.[42]

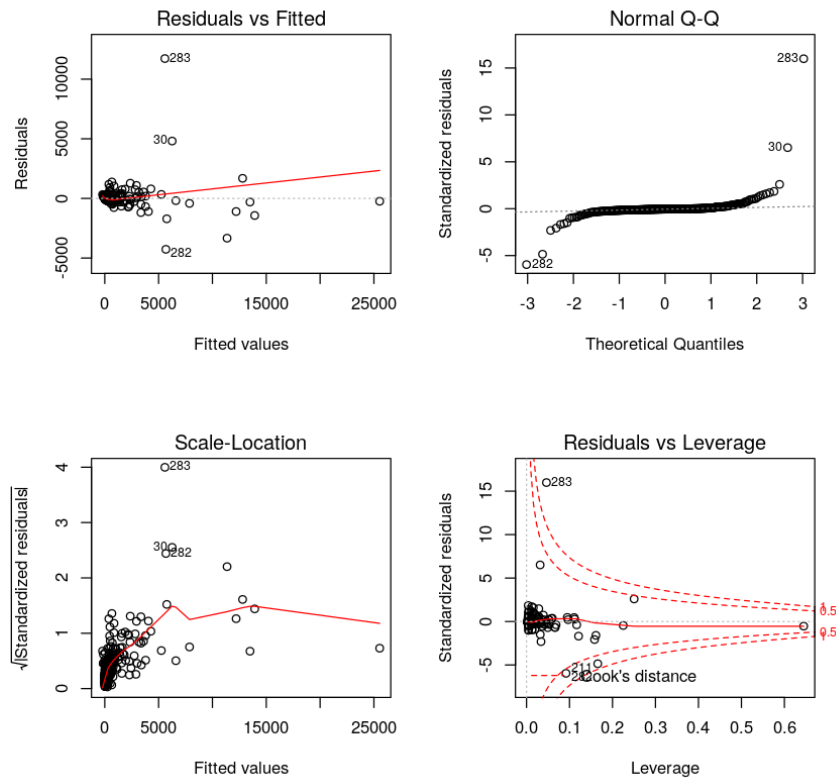


Figure 12: Diagnostic Plots of the Multidimensional Regression (Java Projects)

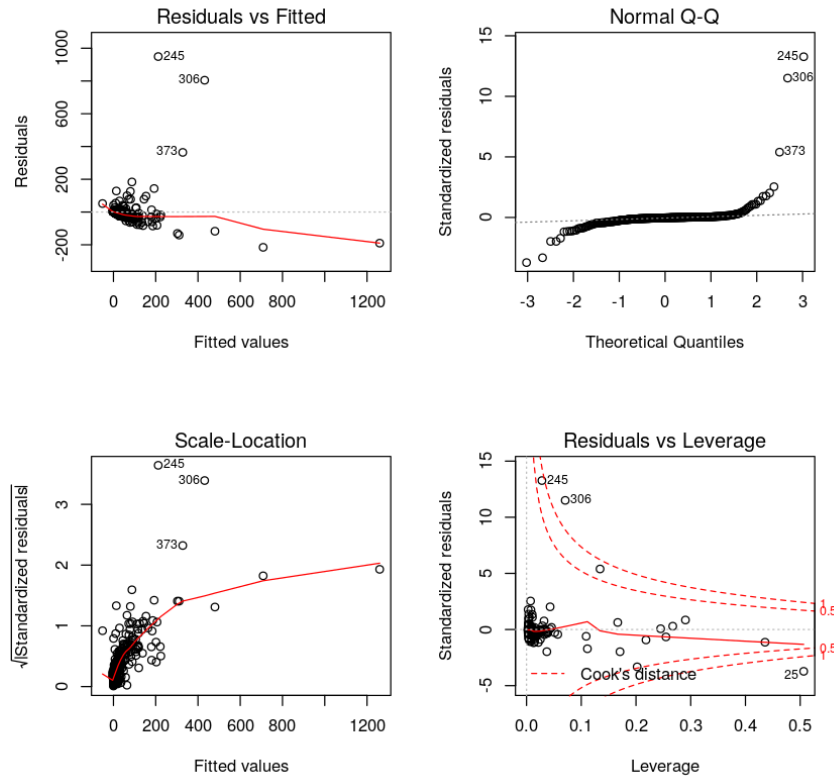


Figure 13: Diagnostic Plots of the Multidimensional Regression (Kotlin Projects)

Since this is one of the assumptions for the ANCOVA yet not met, a Box-Cox transformation was conducted. The Box-Cox transformation is a power transformation that makes it possible to apply a certain function to the data in order to get approximately a normal distribution.[8] The Box-Cox power transformation is defined as the following.[43]

$$x'_\lambda = \frac{x^\lambda - 1}{\lambda}$$

λ (lambda) is estimated by using the profile likelihood function.[44] The highest likelihood shows the best λ for reaching normality. Figure 14 and the tables in figure 15 show that λ is near zero for the highest likelihood for both data sets.

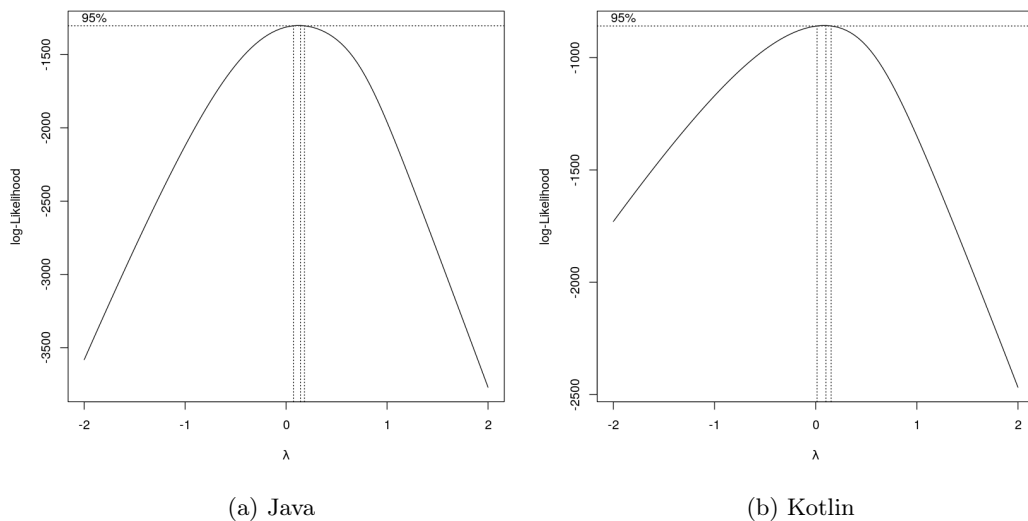


Figure 14: Lambda and Corresponding Likelihood

Sorting the results in descending order by likelihood shows a λ of 0.14 for Java and 0.10 for Kotlin data. Both is closer to zero than to 0.5

lambda	lik	lambda	lik
0.14141414	-1301.560	0.10101010	-857.4145
0.10101010	-1301.790	0.06060606	-857.5357
0.18181818	-1303.468	0.14141414	-858.6224
0.06060606	-1304.177	0.02020202	-858.9026
0.22222222	-1307.500	0.18181818	-861.2650
0.02020202	-1308.751	-0.02020202	-861.4381

(a) Java

(b) Kotlin

Figure 15: Lambda and Corresponding Likelihood

According to the Box-Cox transformation, the function applied to the training data of Java and Kotlin projects is the natural logarithm as it can be seen in the following table.

λ	-2	-1	-0.5	0	0.5	1	2
Function	$1/Y^2$	$1/Y$	$1/\sqrt{Y}$	$\log(Y)$	\sqrt{Y}	None	Y^2

Table 8: Box-Cox Transformations (According to [8])

The figures 16 and 17 show diagnostic plots for the multidimensional fits, whereas the logarithmic function was applied to the response variables. After deleting some outliers which are not representative, the Q-Q plots in the diagnostic plots indicate a normal distribution (almost a 45 degree line) for Kotlin and Java projects.

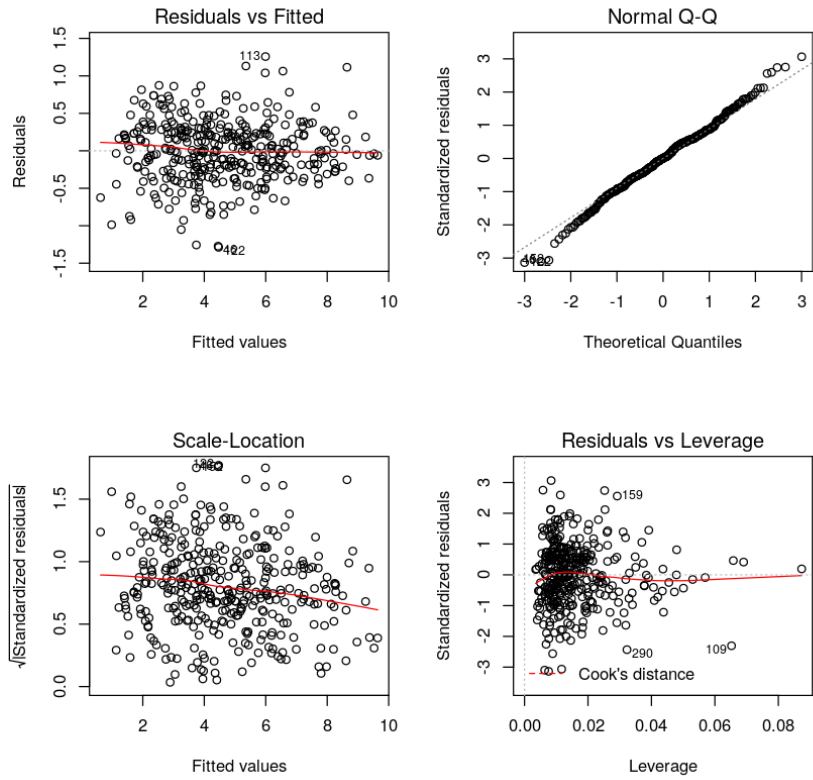


Figure 16: Diagnostic Plots of the Regression Using a Logarithmic Fit (Java Projects)

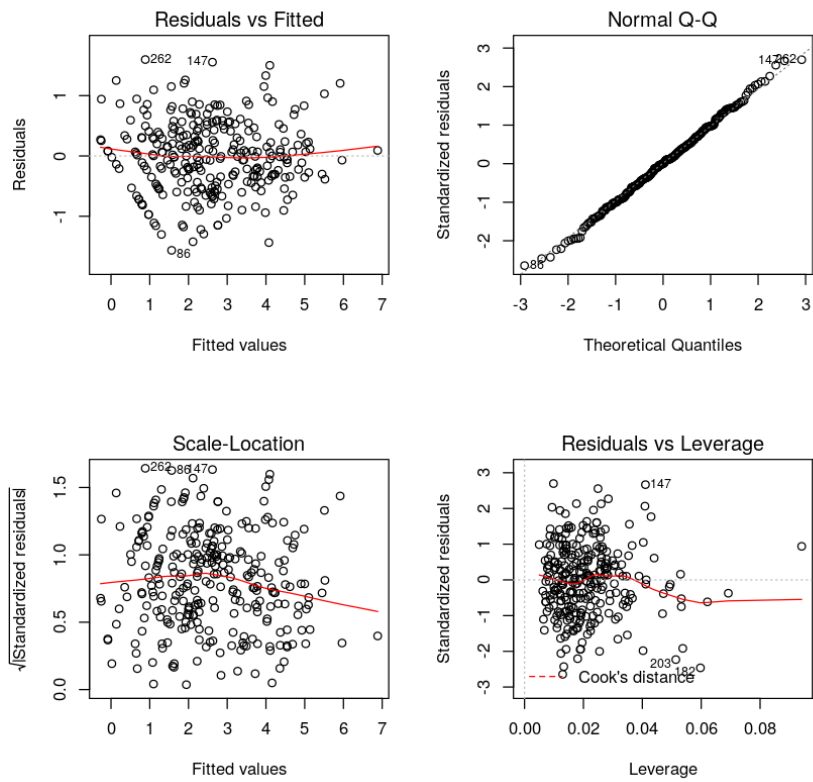


Figure 17: Diagnostic Plots of the Regression Using a Logarithmic Fit (Kotlin Projects)

The results of the Shapiro-Wilk test show that the data are normally distributed by using a logarithmic fit for the multidimensional regression and removing some outliers. $Total_j$ stands for the total amount of smells as the response variable in Java projects, while $Total_k$ stands for the total amount of smells as the response variable in Kotlin projects.

Table 9: Results of the Shapiro-Wilk Test

$Total_j$		$Total_k$	
w	p-value	w	p-value
0.99461	0.2157	0.99753	0.9482

The null hypothesis (with an alpha level of 0.05) can therefore not be rejected, since both p-values are greater than 0.05. Meaning that it can not be said that the data are not normally distributed (double negative). Figure 18 shows a logarithmic fit for (logarithmised values of) total amount of smells as the response variable and lines of code as the explanatory variable. The regression looks like a fairly good fit for the training data.

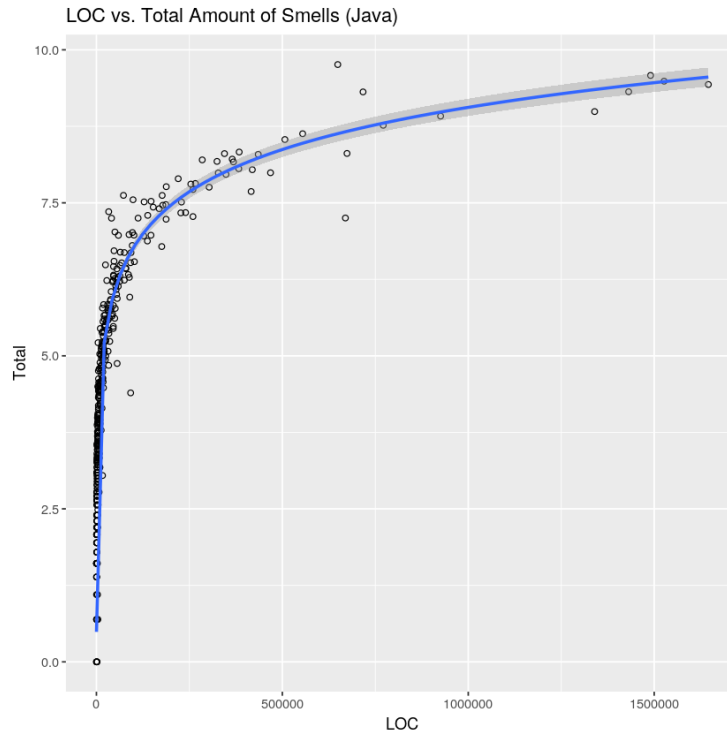


Figure 18: LOC vs Total Amount of Smells

After looking at the Q-Q plots in the figures 16 and 17 and the results of the Shapiro-Wilk test, it can be concluded that condition three (the data should be approximately normally distributed) is satisfied.

- **Assumption 4:** The relationship between the independent variable and the dependent variables should be a linear regression relationship. When taking a look at the figures 16 and 17 at the upper left plots (residuals vs fitted), the red line seems to be fairly straight and therefore it can be conducted that linearity is given.
- **Assumption 5:** Homoscedasticity (constant variance or homogeneity of variance) should be given. When taking a look at the figures 16 and 17 again, but this time at the bottom left plot, it can be seen that the scattering around the line does not seem to vary too much, it can be therefore concluded that a constant variance is roughly given.

The complete analysis can be found in the appendices E and F.

5.4.2 ANCOVA

A table containing the independent variable, covariates of the respective language as well as a column indicating the language is needed to conduct an ANCOVA. By using a few commands in R, the two data frames containing the data for the respective language were concatenated to one, while the column 'Language' tells which language it is (1 for Java, 0 for Kotlin).

	Language	Total	Lifespan	Issues	LOC	Contributors	Stargazers
355	1	3.583519	1345	224	2756	10	4468
356	1	6.519147	3010	424	67918	90	4497
357	1	5.379897	1063	3	15173	45	2457
358	1	2.708050	2841	53	739	5	3677
359	1	4.488636	1954	21	5065	18	3468
360	1	3.091042	750	11	4389	4	6676
361	1	1.386294	772	6	402	10	2401
362	1	4.510860	981	86	9640	5	5106
363	1	2.197225	1251	50	597	2	3079
364	1	3.526361	1099	69	4311	12	3068
365	1	5.365976	2991	1021	33539	272	8659
366	1	7.470794	3114	669	186812	104	4563
367	1	2.833213	1266	307	2043	100	4583
368	1	5.236442	307	102	35907	28	4992
369	1	2.302585	898	19	1062	7	2064
370	1	3.761200	1272	195	5237	13	3606
371	1	5.624018	2820	90	34761	76	7294
372	1	6.716595	872	136	47381	52	4943
373	1	3.828641	827	22	4110	18	3389
374	0	2.890372	767	25	528	3	383
375	0	4.158883	2166	4	6895	24	483

Table 10: Concatenated Data Frame with Language Coded as 1 and 0

As it can be seen in table 10, the first entries represent Java projects, from line 374 onwards Kotlin projects are represented. The diagnostic plots for the fit of the multidimensional regression of both languages can be seen in figure 19. After removing some outliers again, the upper left plot shows a line that indicates linearity, while the plot in the upper right corner seems to be a 45 degree line. Therefore normally distributed data can be assumed. The plot in the lower left corner shows that homoscedasticity is roughly met.

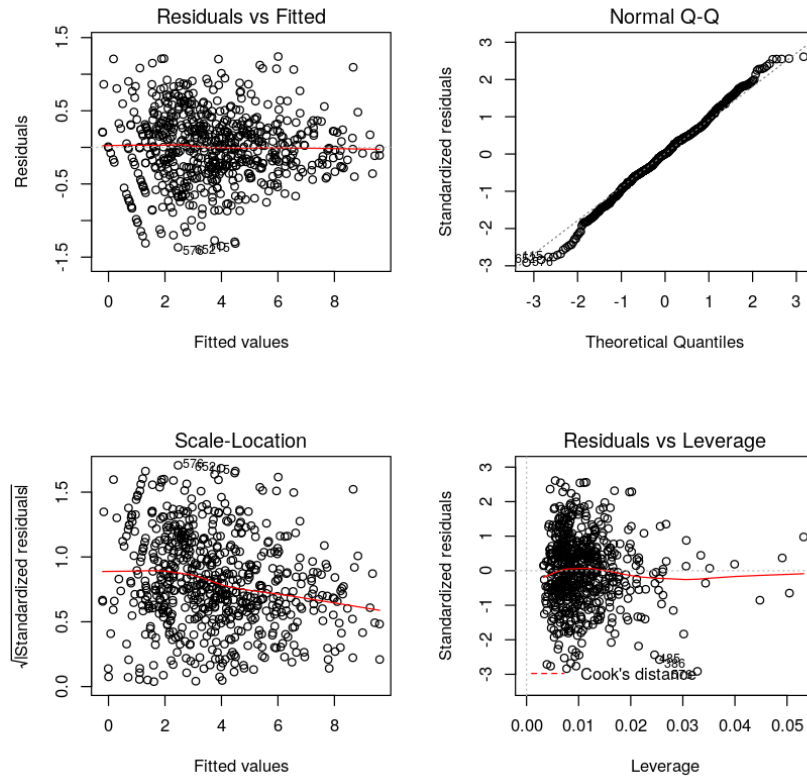


Figure 19: Diagnostic Plots for the Multidimensional Regression (Java and Kotlin)

Furthermore, the Shapiro-Wilk test (with an alpha level of 0.05) shows that the null hypothesis cannot be rejected. Once again, it can not be denied that the data are not normally distributed.

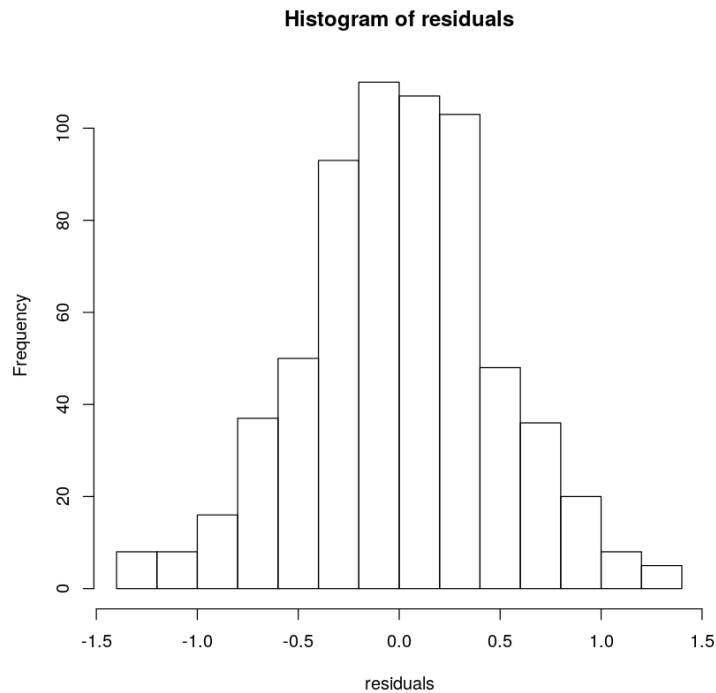


Figure 20: Histogram of the Residuals

Table 11: Results of the Shapiro-Wilk Test

<i>Total_{jk}</i>	
w	p-value
0.99566	0.06743

Performing descriptive statistical analysis without controlling the covariates (i.e. without taking additional independent variables into account) the following means are calculated for the total amount of smells in Java and Kotlin projects.

Table 12: Arithmetic Means of Total Amount of Smells (Without Controlling Covariates)

Java	Kotlin
4.622647	2.560392

An ANCOVA was conducted to analyse the impact of the programming languages Java and Kotlin on the occurrences of code smells while controlling the effect of the aforementioned variables. When doing so the following means are calculated.

Table 13: Arithmetic Means of Total Amount of Smells (When Controlling Covariates)

Java	Kotlin
5.996391	5.643317

The results in table 13 indicate that there is no great difference regarding smells in the two languages and the original means (table 12) are not reliable. Considering the difference between the results of the arithmetic means leads to assume that the influence of variables shown in the correlation matrices should be taken into account.

5.5 Evaluation

In order to evaluate the model found before, the root mean squared error (RMSE) is applied. The RMSE tells how far away the actual points are from the (logarithmic) regression on average. The test data are used for this evaluation. The RMSE is calculated by using the following formula.[45]

$$RMSE = \sqrt{\frac{\sum (z_{fi} - z_{oi})^2}{n}}$$

Whereas the difference between the predicted value (z_{fi}) and the actual (observed) value (z_{oi}) is calculated (equals the residuals). The squared sum of the residuals is then divided by the number of projects (n). After that, the square root of the result is calculated. When using the test data (which is $n - 400$ for the respective language) an RMSE of 0.57 (rounded to two decimals) is calculated. Now is that a good RMSE?

Min.	1st Qu.	Median	Mean	Mean 3rd Qu.	Max.
0.69	2.40	3.61	3.87	5.17	10.34

Table 14: Descriptive Statistics: Summary of the Response Variable (Total Amount of Smells)

These values as well as the RMSE are – of course – not the original values but were calculated before by using the logarithmic function. The values of the response variable vary from 0.69 to 10.34 with a mean of 3.87. A common way to normalize the RMSE is to divide it by the difference of the maximum of y and the minimum of y .

$$NRMSE = \frac{RMSE}{y_{max} - y_{min}}$$

When applying this to the data above, the result is the following.

$$\frac{0.57}{10.34 - 0.69} = 0.06$$

This result can be interpreted as a percentage: The logarithmic regression predicts data with an error rate of 6% which can be considered as pretty good.

5.6 Deployment

A code representation which processes new raw data as the data used in this study can be found in the appendix. The precondition, however is that the data are distributed in a similar way, otherwise the Box-Cox transformation might be misleading. Furthermore, the code representation is limited to Java and Kotlin projects and has yet not been tested by using other programming languages. When considering Java and Kotlin data in the future the model developed during the experiment should provide rather good results. However, this compliance applies with the restriction that data are fetched from GitHub sorted by stargazers. Results might differ when using another way of fetching data.

6 Discussion, Related Work

Even though a positive trend can be noticed by looking at the statistics in chapter 3.7, it is questionable if it is really worth it to learn a new programming language when one is already used to a well established one. To be able to answer this question, more aspects than pure syntax, code smells and certain features have to be taken into account. For example: Are there interesting employers offering positions for Kotlin developers? If not, why would someone want to learn a new language when there is no market?

As it can be seen in the results of the experiment, there is no great difference in the means of code smells between Java and Kotlin projects. Therefore the aforementioned aspects gain even more importance.

In 1999, Fowler et al. [28] introduced 22 codes smells in their study "Refactoring: Improving the Design of Existing Code". Only two of them were selected for this study due to comparability reasons. The study "Are you still smelling it?" by Flauzino et. al [7] is the most similar study found when researching for studies related to Kotlin and Java. However, as stated out earlier, "Are you still smelling?" provided results of a similar experiment which were de facto distorted and not reliable. That is precisely why in the presented experiment in this study not only more projects were taken into consideration. Furthermore, influencing variables (such as lines of code and contributors) affecting the amount of smells were controlled by conducting an ANCOVA.

7 Limitations, Future Research

The conducted experiment was only performed by using three code smells:

- Long Filename
- Long Parameter List
- Too Many Methods

In future research more code smells could be taken into consideration. Also, several analysis cycles could be performed based on different thresholds – these results might differ completely from the results shown in this study. Besides, correlation coefficients depend strongly on the thresholds. Moreover, only GitHub projects were taken into account. Consequently, some relevant projects are probably excluded. Furthermore, there might be a strong influence on smells caused by the area of application. As shown in chapter 3.7, Kotlin is mostly used for developing Android applications while the range of uses for Java is more diverse. Also, it is questionable if a comparison between two languages should be based on projects implemented by different developers, since code smells are mainly caused by developers. If a developer tends to use bad practices when programming in Kotlin, he most probably tends to do so when programming in Java and vice versa. There are two possible ways to address that issue in future research.

1. The interaction between syntactic differences and specific code smells could be examined by comparing exactly the same projects implemented in Kotlin and in Java (by using code conversion e.g.).
2. A survey among developers who use both languages could provide better and more authentic results.

8 Conclusion

In this study Kotlin and Java projects were downloaded from GitHub by using several scripts and a Python program developed before (all scripts and the Python program can be found in the appendix). Afterwards, the repositories were analysed regarding the code smells Long Filename, Long Parameter List and Too Many Methods by using the tools PMD and Detekt. The threshold used to detect the smells were 30 characters for filenames, 2 parameters for long parameter list and 5 methods for the maximum number of methods a class is allowed to have. Here again, several bash scripts were being used to automatically apply the aforementioned tools on the projects and to write the results in CSV files.

After all the data had been collected, correlation matrices provided clear evidence that the amount of smells a project has is dependent from specific variables such as lines of code, contributors and the lifespan. To take care of this issue, an ANCOVA was conducted by using the mathematical programming language R and a multidimensional regression. Unfortunately, the data did not show a normal distribution when plotting diagnostic graphs in the first place. A Box-Cox transformation then helped to find a transformation to get an approximately normally distributed set of data by applying the natural logarithm to the response variable (total amount of smells). After all the assumptions for an ANCOVA were met, the actual analysis could be conducted. A crucial difference in the means before and after controlling the effects of the covariates revealed that ignoring these covariates leads to wrong conclusions.

In fact, there is no substantial difference when taking a look at code smells in Kotlin and in Java. However, it should be noted that this is not the only measurement when deciding between two languages. For those who are interested in the occurrence of code smells and the evaluation of the latter, this work should provide a good starting point for research in Kotlin and Java related studies, especially when other methods of analysis are applied as suggested previously: a survey or a comparison of the same projects.

References

- [1] Java Architecture; 2015 (accessed November 07, 2018). Available from: <https://www.careerbless.com/java/basics/JavaArchitecture.php>.
- [2] Jemerov D, Isakova S. Kotlin in Action. Manning Publications; 2016.
- [3] Developer Survey Results 2018; 2018 (accessed November 12, 2018). Available from: <https://insights.stackoverflow.com/survey/2018#most-loved-dreaded-and-wanted>.
- [4] The State of Kotlin 2018; 2018 (accessed November 12, 2018). Available from: <https://pusher.com/state-of-kotlin>.
- [5] Process diagram showing the relationship between the different phases of CRISP-DM; (accessed January 5, 2019). Available from: https://en.wikipedia.org/wiki/Cross-industry_standard_process_for_data_mining#/media/File:CRISP-DM_Process_Diagram.png.
- [6] Comparison to Java Programming Language; (accessed November 12, 2018). Available from: <https://kotlinlang.org/docs/reference/comparison-to-java.html>.
- [7] Flauzino M, Veríssimo J, Terra R, Cirilo E, H S Durelli V, Durelli RS. Are you still smelling it? Journal of Experimental Algorithmics. 2018; Available from: <https://dl.acm.org/citation.cfm?id=3267186>.
- [8] Making Data Normal Using Box-Cox Power Transformation; (accessed December 28, 2018). Available from: <https://www.isixsigma.com/tools-templates/normality/making-data-normal-using-box-cox-power-transformation/>.
- [9] Murphy K. So why did they decide to call it Java?; 1996 (accessed November 07, 2018). Available from: <https://www.javaworld.com/article/2077265/core-java/so-why-did-they-decide-to-call-it-java-.html>.
- [10] Heiss JJ. The Advent of Kotlin: A Conversation with JetBrains' Andrey Breslav; 2013 (accessed November 07, 2018). Available from: <https://www.oracle.com/technetwork/articles/java/breslav-1932170.html>.
- [11] FAQ; (accessed December 28, 2018). Available from: <https://kotlinlang.org/docs/reference/faq.html>.
- [12] Lardinois F. Google makes Kotlin a first-class language for writing Android apps; 2017 (accessed November 07, 2018). Available from: <https://techcrunch.com/2017/05/17/google-makes-kotlin-a-first-class-language-for-writing-android-apps/?guccounter=1>.
- [13] Stack Overflow Trends; (accessed December 28, 2018). Available from: <https://insights.stackoverflow.com/trends?tags=kotlin>.
- [14] Leiva A. 12 reasons why you should start using Kotlin for Android today (KAD 30); (accessed December 29, 2018). Available from: <https://antonioleiva.com/reasons-kotlin-android/>.
- [15] Sommerhoff P. Kotlin vs. Java: 9 Benefits of Kotlin for Your Business; (accessed December 29, 2018). Available from: <https://business.udemy.com/blog/kotlin-vs-java-9-benefits-of-kotlin-for-your-business/>.
- [16] Dossey A. Java vs. Kotlin: Which is the Better Option for Android App Development?; (accessed December 29, 2018). Available from: <https://clearbridgemobile.com/java-vs-kotlin-which-is-the-better-option-for-android-app-development/>.
- [17] Gosling J, Joy B, Steele G, Bracha G, Buckley A. The Java ® Language Specification. Oracle America; 2015.
- [18] Chander S. Introducing Java SE 11; 2018 (accessed November 07, 2018). Available from: <https://blogs.oracle.com/java-platform-group/introducing-java-se-11>.

- [19] Is the JVM (Java Virtual Machine) platform dependent or platform independent? What is the advantage of using the JVM, and having Java be a translated language?; (accessed November 07, 2018). Available from: <https://www.programmerinterview.com/index.php/java-questions/jvm-platform-dependent/>.
- [20] Breslav A. Kotlin 1.0 Released: Pragmatic Language for JVM and Android; 2016 (accessed November 07, 2018). Available from: <https://blog.jetbrains.com/kotlin/2016/02/kotlin-1-0-released-pragmatic-language-for-jvm-and-android/>.
- [21] Kotlin - Architecture; (accessed November 07, 2018). Available from: https://www.tutorialspoint.com/kotlin/kotlin_architecture.htm.
- [22] Kerimbaev B. Kotlin: When to Use Lazy or Lateinit; 2017 (accessed November 01, 2018). Available from: <https://www.bignerdranch.com/blog/kotlin-when-to-use-lazy-or-lateinit/>.
- [23] Properties and Fields; (accessed November 05, 2018). Available from: <https://kotlinlang.org/docs/reference/properties.html>.
- [24] Hoare T. Null References: The Billion Dollar Mistake; 2009 (accessed October 17, 2018). Available from: <https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare>.
- [25] Levy A. Top 5 Crashes on Android; 2016 (accessed October 18, 2018). Available from: <https://www.apteligent.com/technical-resource/top-5-crashes-on-android/>.
- [26] Friesen J. The State of Kotlin 2018; 2016 (accessed November 12, 2018). Available from: <https://www.javaworld.com/article/3142626/core-java/are-checked-exceptions-good-or-bad.html>.
- [27] Sharma T, Spinellis D. A Survey on Software Smells. Manning Publications; 2016.
- [28] Fowler M, Beck K, Brant J, Opdyke W, Roberts d. Refactoring: Improving the Design of Existing Code. 1999;.
- [29] Vorhies W. CRISP-DM – a Standard Methodology to Ensure a Good Outcome; (accessed December 29, 2018). Available from: <https://www.datasciencecentral.com/profiles/blogs/crisp-dm-a-standard-methodology-to-ensure-a-good-outcome>.
- [30] What is the CRISP-DM methodology?; (accessed November 14, 2018). Available from: <https://www.sv-europe.com/crisp-dm-methodology/>.
- [31] Vorhies W. CRISP-DM – a Standard Methodology to Ensure a Good Outcome; 2016 (accessed November 14, 2018). Available from: <https://www.datasciencecentral.com/profiles/blogs/crisp-dm-a-standard-methodology-to-ensure-a-good-outcome>.
- [32] PMD; (accessed December 29, 2018). Available from: <https://pmd.github.io/>.
- [33] detekt; (accessed December 29, 2018). Available from: <https://arturbosch.github.io/detekt/>.
- [34] Is “Data class” really a code smell?; (accessed December 15, 2018). Available from: <https://stackoverflow.com/questions/16719270/is-data-class-really-a-code-smell>.
- [35] PMD Source Code Analyzer Project; (accessed December 15, 2018). Available from: https://pmd.github.io/pmd-6.3.0/pmd_rules_java_design.html#excessivemethodlength.
- [36] Complexity Rule Set; (accessed December 15, 2018). Available from: <https://arturbosch.github.io/detekt/complexity.html>.
- [37] How many lines of code should a function/procedure/method have?; (accessed January 5, 2019). Available from: <https://stackoverflow.com/questions/611304/how-many-lines-of-code-should-a-function-procedure-method-have>.

- [38] Shah T. About Train, Validation and Test Sets in Machine Learning; (accessed January 5, 2019). Available from: <https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7>.
- [39] Creech S. Linear Regression; (accessed January 5, 2019). Available from: <https://www.statisticallysignificantconsulting.com/RegressionAnalysis.htm>.
- [40] Analysis of covariance (ANCOVA); (accessed December 27, 2018). Available from: <https://www.statisticssolutions.com/analysis-of-covariance-ancova/>.
- [41] One-way ANCOVA in SPSS Statistics; (accessed December 29, 2018). Available from: <https://statistics.laerd.com/spss-tutorials/ancova-using-spss-statistics.php>.
- [42] Kabacoff I R. R in Action. Manning Publications; 2011.
- [43] Box-Cox Transformations; (accessed December 28, 2018). Available from: <http://onlinestatbook.com/2/transformations/box-cox.html>.
- [44] Power transform; (accessed January 5, 2019). Available from: https://en.wikipedia.org/wiki/Power_transform.
- [45] Statistics; (accessed December 29, 2018). Available from: <https://cirpwiki.info/wiki/Statistics>.

Appendices

A Main Bash Script

```
1 #!/usr/bin/bash
2
3 rm java_repository_names kotlin_repository_names # Removes the existing repositories names
4
5 # The following Python program loads repository names via the GitHub API and saves them in text files
6 # Use arguments to indicate the sample size you prefer: 100, 200, 300, 400, 500, 1000
7 # Example: #python3 load_repository_names.py 300
8 # When no argument is given, the default (5) sample size will be used
9 python3 load_repository_names.py
10
11 # By using the following commands, Kotlin projects will be downloaded and analysed
12
13 bash load_kotlin_repos # Clones repositories by reading the files
14 bash check_kotlin_smells # Uses Detekt to analyse the Kotlin repositories and saves the logs
15 bash read_write_kotlin_data # Checks the logs and sums up the code smells
16
17 # By using the following commands Java projects will be downloaded and analysed
18
19 bash load_java_repos # Clones repositories by reading the files
20 bash check_java_smells # Uses PMD to analyse the Java repositories and saves the logs
21 bash read_write_java_data # Checks the logs and sums up the code smells
```

Listing 21: Bash Script Used to Execute the Python Program and All Bash Scripts at Once

B Python Program

```
1 import sys
2 import requests
3 import json
4 import datetime
5 from datetime import date
6 import time
7
8 def test_case():
9     return 5, 1
10
11 def user_spec_sample(sample_size):
12     if (sample_size == 5):
13         print("Test case is running ...")
14         return test_case()
15     elif (sample_size == 100):
16         return sample_size, 1
17     elif (sample_size == 200):
18         return sample_size, 2
19     elif (sample_size == 300):
20         return sample_size, 3
21     elif (sample_size == 400):
22         return sample_size, 4
23     elif (sample_size == 500):
24         return sample_size, 5
25     elif (sample_size == 1000):
26         return sample_size, 10
27     else:
28         print("Please use 100, 200, 300, 400, 500 or 1000 as a sample size. Test case (sample size = 5) will be executed now.")
29     return test_case()
30
31 def getLifespanInDays(givenDate):
32     datetime_now = getDateFormat(datetime.datetime.today().strftime('%Y-%m-%d'))
33     delta = datetime_now - givenDate
34     return delta.days
35
36 def getDateFormat(givenDate):
37     year = int(givenDate[:6])
38     month = int(givenDate[5:7])
39     day = int(givenDate[8:10])
40     return (date(year, month, day))
41
42 def write_data_to_file(programming_language, input_type):
43     with open(programming_language+' repository_names', 'a') as text_file:
44         for page_number in range(len(input_type)):
45             for repo in input_type[page_number]['items']:
46                 date_repo = repo['created_at'][:10]
47                 lifespan = getLifespanInDays(getDateFormat(date_repo))
48                 text_file.write(repo['full_name']+', '+str(lifespan)+'+', '+str(repo['open_issues'])+', '+str(repo['
49                     stargazers_count'])
50                 text_file.write('\n')
51
52 input_java_repo = []
53 input_kotlin_repo = []
54
55 try:
56     user_argument = (int(sys.argv[1]))
57     sample_size_and_pages = user_spec_sample(user_argument)
58     number_of_pages = sample_size_and_pages[1]
59     per_page = sample_size_and_pages[0]
60 except IndexError:
61     print("Please use 100, 200, 300, 400, 500 or 1000 as an argument for the sample size. Test case (sample size = 5) will be
62         executed now.")
63     sample_size_and_pages = test_case()
64     number_of_pages = sample_size_and_pages[0]
65     per_page = sample_size_and_pages[1]
66
67 for page_number in range(number_of_pages):
68     url_java_repos = 'https://api.github.com/search/repositories?q=stars%3A%3E%3D10+language%3Ajava&sort=stars&order=
69         desc&page='+str(page_number+1)+'&per_page='+str(per_page)
70     url_kotlin_repos = 'https://api.github.com/search/repositories?q=stars%3A%3E%3D10+language%3A%3Akotlin&sort=stars&order
71         =desc&page='+str(page_number+1)+'&per_page='+str(per_page)
72     r_java = requests.get(url_java_repos)
73     r_kotlin = requests.get(url_kotlin_repos)
74     input_java_repo.append(json.loads(r_java.text))
75     input_kotlin_repo.append(json.loads(r_kotlin.text))
76     time.sleep(10)
77
78 write_data_to_file('java', input_java_repo)
79 write_data_to_file('kotlin', input_kotlin_repo)
```

Listing 22: Python Program Used to Download the Repository Names

C Bash Scripts for Kotlin Repositories

```
1 #!/usr/bin/bash
2 rm -r -f Repos_Kotlin
3 mkdir Repos_Kotlin
4 cd Repos_Kotlin
5 filename="./kotlin_repository_names"
6 while read -r line; do
7     complete_line="$line"
8     rep_name="$( cut -d ',' -f 1 <<< "$complete_line" )"
9     git clone https://github.com/$rep_name.git
10    echo +-----+
11    echo "cloned $rep_name"
12    echo +-----+
13 done < "$filename"
```

Listing 23: Bash Script Used to Download the Kotlin Repositories

```
1 #!/usr/bin/bash
2 rm -r Logs_Kotlin_Smells
3 mkdir Logs_Kotlin_Smells
4
5 cd Repos_Kotlin
6
7 echo Deleting empty lines in .kt files ...
8 find -type f -name '*.kt' -exec sed -i '/^\s*$/d' {} + # Deletes all empty lines in all Kotlin files in the subdirs
9
10 filename="./kotlin_repository_names"
11 while read -r line; do
12     complete_line="$line"
13     rep_name="$( cut -d ',' -f 1 <<< "$complete_line" )"
14     echo +-----+
15     echo Checking $rep_name
16     echo +-----+
17     java -jar ../detekt/detekt-cli/build/libs/detekt-cli-1.0.0-RC11-all.jar --input ../Repos_Kotlin/${rep_name##*/} >> ../
18     Logs_Kotlin_Smells/${rep_name##*/}
19 done < "$filename"
```

Listing 24: Bash Script Used to Apply Detekt to the Repositories

```
1 #!/bin/bash
2 csv_file=kotlin_code_smells.csv
3 rm $csv_file
4 rep_names="kotlin_repository_names"
5 echo Project, LF '(t=30)', LPL'(t=2)', TMM'(t=5)', Total, Lifespan, Issues, LOC, Commits, Contributors, Stargazers >>
6 $csv_file
7 while read -r line; do
8     complete_line="$line"
9     rep_name="$( cut -d ',' -f 1 <<< "$complete_line" )"
10
11     lifespan="$( cut -d ',' -f 2 <<< "$complete_line" )"
12     issues="$( cut -d ',' -f 3 <<< "$complete_line" )"
13     stargazers_count="$( cut -d ',' -f 4 <<< "$complete_line" )"
14     log_file="Logs_Kotlin_Smells/${rep_name##*/}"
15     cd Repos_Kotlin/${rep_name##*/}
16     contributors=$(git log --all --pretty="%an" | sort | uniq | wc -l)
17     commits=$(git rev-list --all --count)
18     loc=$(find . -type f -name '*.kt' -exec cat {} \; | sed '/^\s*$/d;/^\s*\//d' | wc -l)
19     cd ..
20     excessiveParameterListSmellCount=$(grep -c "LongParameterList" $log_file)
21     tooManyMethodsSmellCount=$(grep -c "TooManyFunctions" $log_file)
22
23     IFS=$'\n'; set -f
24     filename_length_counter=0
25     for filename in $(find Repos_Kotlin/${rep_name##*/} -name '*.kt'); do
26
27         filename_with_extension=$(echo "${filename}" | sed 's,*,/::')
28         filename_without_extension="$( cut -d ',' -f 1 <<< "$filename_with_extension" )"
29
30         if (( ${#filename_without_extension} > 30 )); then
31             ((filename_length_counter++))
32         fi
33     done
34     unset IFS; set +f
35
36     total=$((filename_length_counter+excessiveParameterListSmellCount+tooManyMethodsSmellCount))
37     echo $rep_name, $filename_length_counter, $excessiveParameterListSmellCount, $tooManyMethodsSmellCount, $total, $lifespan,
38     $issues, $loc, $commits, $contributors, $stargazers_count >> $csv_file
39 done < "$rep_names"
```

Listing 25: Bash Script Used to Read Log Files and Write the Results to a CSV File

D Bash Scripts for Java Repositories

```
1 #!/usr/bin/bash
2 rm -r -f Repos_Java
3 mkdir Repos_Java
4 cd Repos_Java
5 filename="./java_repository_names"
6 while read -r line; do
7     complete_line="$line"
8     rep_name="$( cut -d ',' -f 1 <<< "$complete_line" )"
9     git clone https://github.com/$rep_name.git
10    echo +-----+
11    echo "cloned $rep_name"
12    echo +-----+
13 done < "$filename"
```

Listing 26: Bash Script Used to Download the Java Repositories

```
1 #!/usr/bin/bash
2 rm -r Logs_Java_Smells
3 mkdir Logs_Java_Smells
4
5 cd Repos_Java
6
7 find -type f -name '*.java' -exec sed -i '/^\s*/d' {} + # Deletes all empty lines in all Java files in the subdirs
8
9 filename="./java_repository_names"
10 while read -r line; do
11     complete_line="$line"
12     rep_name="$( cut -d ',' -f 1 <<< "$complete_line" )"
13     echo +-----+
14     echo Checking $rep_name ...
15     echo +-----+
16     ../pmd-bin-6.9.0/bin/run.sh pmd -d ../Repos_Java/${rep_name###*/} -no-cache -f xml -R ../rules.xml >> ../
17     Logs_Java_Smells/${rep_name###*/}
18 done < "$filename"
```

Listing 27: Bash Script Used to Apply PMD to the Repositories

```
1 #!/bin/bash
2 csv_file=java_code_smells.csv
3 rm $csv_file
4 rep_names="java_repository_names"
5 echo Project, LP '(t=30)', LPL'(t=2)', TMM'(t=5)', Total, Lifespan, Issues, LOC, Commits, Contributors, Stargazers >>
6 $csv_file
7 while read -r line; do
8     complete_line="$line"
9     rep_name="$( cut -d ',' -f 1 <<< "$complete_line" )"
10    lifespan="$( cut -d ',' -f 2 <<< "$complete_line" )"
11    issues="$( cut -d ',' -f 3 <<< "$complete_line" )"
12    stargazers_count="$( cut -d ',' -f 4 <<< "$complete_line" )"
13    log_file="Logs_Java_Smells/${rep_name###*/}"
14    cd Repos_Java/${rep_name###*/}
15    contributors=$(git log --all --pretty="%an" | sort | uniq | wc -l)
16    commits=$(git rev-list --all --count)
17    loc=$(find . -type f -name '*.java' -exec cat {} \; | sed '/^\s*/d;/^\s*/\|/d' | wc -l)
18    cd ..
19    excessiveParameterListSmellCount=$(grep -c "ExcessiveParameterList" $log_file)
20    tooManyMethodsSmellCount=$(grep -c "TooManyMethods" $log_file)
21
22    IFS=$'\n'; set -f
23    filename_length_counter=0
24    for fileName in $(find Repos_Java/${rep_name###*/} -name '*.java'); do
25
26        filename_with_extension=$(echo "${filename}" | sed 's,*,/:;')
27        filename_without_extension="$( cut -d ',' -f 1 <<< "$filename_with_extension" )"
28
29        if (( ${#filename_without_extension} > 30 )); then
30            ((filename_length_counter++))
31        fi
32
33    done
34    unset IFS; set +f
35
36    total=$((filename_length_counter+excessiveParameterListSmellCount+tooManyMethodsSmellCount))
37    echo $rep_name, $filename_length_counter, $excessiveParameterListSmellCount, $tooManyMethodsSmellCount, $total, $lifespan,
38    $issues, $loc, $commits, $contributors, $stargazers_count >> $csv_file
39 done < "$rep_names"
```

Listing 28: Bash Script Used to Read Log Files and Write the Results to a CSV File

E R Code Part 1

```
1 library(effects)
2 library(devtools)
3 library(easyGgplot2)
4 library("MASS")
5
6 java_code_smells = read.csv("java_code_smells.csv", header = TRUE)
7 kotlin_code_smells = read.csv("kotlin_code_smells.csv", header = TRUE)
8
9 set.seed(37)
10 df_java_projects_ran <- java_code_smells[sample(nrow(java_code_smells)),]
11 df_kotlin_projects_ran <- kotlin_code_smells[sample(nrow(kotlin_code_smells)),]
12
13 rownames(df_java_projects_ran) <- NULL
14 rownames(df_kotlin_projects_ran) <- NULL
15
16 java_projects_training <- df_java_projects_ran[1:400,]
17 kotlin_projects_training <- df_kotlin_projects_ran[1:400,]
18
19 java_projects_training_w_projects <- subset(java_projects_training, select = c(2,3,4,5,6,7,8,9,10,11))
20 res <- cor(java_projects_training_w_projects)
21 round(res, 2)
22
23 kotlin_projects_training_w_projects <- subset(kotlin_projects_training, select = c(2,3,4,5,6,7,8,9,10,11))
24 res <- cor(kotlin_projects_training_w_projects)
25 round(res, 2)
26
27 LOC <- java_projects_training$LOC
28 Total <- java_projects_training$Total
29
30 ggplot() +
31   geom_point(data=java_projects_training, aes(LOC, Total, color='Java')) +
32   geom_point(data=kotlin_projects_training, aes(kotlin_projects_training$LOC, kotlin_projects_training$Total, color='Kotlin')) +
33   ggtitle("LOC vs. Total Amount of Smells (Java and Kotlin)")
34
35 fit <- lm(Total ~ LOC + Stargazers + Contributors + Issues + Lifespan, java_projects_training)
36
37 summary(fit)
38 par(mfrow=c(2,2))
39 plot(fit)
40
41 fit <- lm(Total ~ LOC + Stargazers + Contributors + Issues + Lifespan, kotlin_projects_training)
42
43 summary(fit)
44 par(mfrow=c(2,2))
45 plot(fit)
46
47
48 java_projects_training <- java_projects_training[java_projects_training$Total != 0, ]
49 java_projects_training <- java_projects_training[java_projects_training$Lifespan != 0, ]
50 java_projects_training <- java_projects_training[java_projects_training$Issues != 0, ]
51 java_projects_training <- java_projects_training[java_projects_training$Contributors != 0, ]
52 java_projects_training <- java_projects_training[java_projects_training$Stargazers != 0, ]
53
54 b=boxcox(Total ~ Lifespan + Issues + LOC + Contributors + Stargazers, data=java_projects_training)
55 lambda=b$x
56 lik=b$y
57 bc=cbind(lambda,lik)
58 head(bc[order(-lik),])
59
60 kotlin_projects_training <- kotlin_projects_training[kotlin_projects_training$Total != 0, ]
61 kotlin_projects_training <- kotlin_projects_training[kotlin_projects_training$Lifespan != 0, ]
62 kotlin_projects_training <- kotlin_projects_training[kotlin_projects_training$Issues != 0, ]
63 kotlin_projects_training <- kotlin_projects_training[kotlin_projects_training$Contributors != 0, ]
64 kotlin_projects_training <- kotlin_projects_training[kotlin_projects_training$Stargazers != 0, ]
65
66 b=boxcox(Total ~ Lifespan + Issues + LOC + Contributors + Stargazers, data=kotlin_projects_training)
67 lambda=b$x
68 lik=b$y
69 bc=cbind(lambda,lik)
70 head(bc[order(-lik),])
71
72 java_projects_training$Total <- log(java_projects_training$Total)
73
74 rownames(java_projects_training) <- NULL
75
76 Language <- java_projects_training$Language
77 Lifespan <- java_projects_training$Lifespan
78 Issues <- java_projects_training$Issues
79 LOC <- java_projects_training$LOC
80 Contributors <- java_projects_training$Contributors
81 Stargazers <- java_projects_training$Stargazers
82
83 fit <- lm(Total ~ log(Lifespan) + log(Issues) + log(LOC) + log(Contributors) + log(Stargazers), data= java_projects_training)
84 par(mfrow=c(2,2))
85 plot(fit)
86
87 fit <- lm(Total ~ log(Lifespan) + log(Issues) + log(LOC) + log(Contributors) + log(Stargazers), data= java_projects_training
88 [-c(106, 347, 293, 48, 248, 313, 3, 145, 86, 243, 274, 114, 81, 16, 234),])
89 par(mfrow=c(2,2))
90 plot(fit)
91
92 java_projects_training <- java_projects_training[-c(106, 347, 293, 48, 248, 313, 3, 145, 86, 243, 274, 114, 81, 16, 234),]
93
94 resid <- resid(fit)
95 shapiro.test(resid)
96
97 rownames(kotlin_projects_training) <- NULL
98
99 kotlin_projects_training$Total <- log(kotlin_projects_training$Total)
100
101 Language <- kotlin_projects_training$Language
102 Lifespan <- kotlin_projects_training$Lifespan
103 Issues <- kotlin_projects_training$Issues
104 LOC <- kotlin_projects_training$LOC
105 Contributors <- kotlin_projects_training$Contributors
106 Stargazers <- kotlin_projects_training$Stargazers
107
108 fit <- lm(Total ~ log(Lifespan) + log(Issues) + log(LOC) + log(Contributors) + log(Stargazers), data= kotlin_projects_training
109 )
110 par(mfrow=c(2,2))
111 plot(fit)
```

Listing 29: R Code Used for Statistical Analysis

F R Code Part 2

```
1 fit <- lm(Total ~ log(Lifespan) + log(Issues) + log(LOC) + log(Contributors) + log(Stargazers), data= kotlin_projects_training
2           [-c(256, 133, 173, 90, 181, 246, 1),])
3 par(mfrow=c(2,2))
4 plot(fit)
5
6 kotlin_projects_training <- kotlin_projects_training[-c(256),]
7
8 resid <- resid(fit)
9 shapiro.test(resid)
10
11 java_projects_training["Language"] <- 1
12 kotlin_projects_training["Language"] <- 0
13 java_kotlin_mul = data.frame(rbind(cbind(java_projects_training$Language, java_projects_training$Total, java_projects_training$Contributors, java_projects_training$Lifespan, java_projects_training$Issues, java_projects_training$LOC, java_projects_training$Stargazers), cbind(kotlin_projects_training$Language, kotlin_projects_training$Total, kotlin_projects_training$Lifespan, kotlin_projects_training$Issues, kotlin_projects_training$LOC, kotlin_projects_training$Contributors, kotlin_projects_training$Stargazers)))
14
15 java_kotlin_mul <- setNames(java_kotlin_mul, c("Language", "Total", "Lifespan", "Issues", "LOC", "Contributors", "Stargazers"))
16
17 java_kotlin_mul <- java_kotlin_mul[java_kotlin_mul$Lifespan != 0, ]
18 java_kotlin_mul <- java_kotlin_mul[java_kotlin_mul$Issues != 0, ]
19 java_kotlin_mul <- java_kotlin_mul[java_kotlin_mul$LOC != 0, ]
20 java_kotlin_mul <- java_kotlin_mul[java_kotlin_mul$Contributors != 0, ]
21 java_kotlin_mul <- java_kotlin_mul[java_kotlin_mul$Stargazers != 0, ]
22
23 java_kotlin_mul[355:375,]
24
25 fit <- lm(Total ~ Language + log(Lifespan) + log(Issues) + log(LOC) + log(Contributors) + log(Stargazers), data= java_kotlin_mul)
26 par(mfrow=c(2,2))
27 plot(fit)
28
29 rownames(java_kotlin_mul) <- NULL
30
31 fit <- lm(Total ~ Language + log(Lifespan)+log(Issues) + log(LOC)+log(Contributors) + log(Stargazers), data= java_kotlin_mul[-c(463, 506, 546, 554, 619, 374, 634, 480, 520, 459, 440, 555, 590),])
32 par(mfrow=c(2,2))
33 plot(fit)
34
35 java_kotlin_mul <- java_kotlin_mul[-c(463, 506, 546, 554, 619, 374, 634, 480, 520, 459, 440, 555, 590),]
36
37 java_kotlin_mul$Language <- factor(java_kotlin_mul$Language, label = c("Kotlin", "Java"))
38
39 aggregate(java_kotlin_mul$Total, by=list(java_kotlin_mul$Language), FUN=mean)
40
41 java_kotlin_mul_lang <- java_kotlin_mul$Language
42
43 result_of_ancova <- aov(Total ~ java_kotlin_mul_lang + log(Lifespan) + log(Issues) + log(LOC) + log(Contributors) + log(Stargazers), data = java_kotlin_mul)
44 summary(result_of_ancova)
45
46 residuals <- resid(result_of_ancova)
47 shapiro.test(residuals)
48 hist(residuals)
49
50 effect("java_kotlin_mul_lang", result_of_ancova)
51
52 summary(java_kotlin_mul)
53
54 RMSE = function(residuals){
55   sqrt(mean((residuals)^2))
56 }
57
58 java_projects_test <- df_java_projects_ran[401:NROW(df_java_projects_ran),]
59 kotlin_projects_test <- df_kotlin_projects_ran[401:NROW(df_kotlin_projects_ran),]
60
61 java_projects_test["Language"] <- 1
62 kotlin_projects_test["Language"] <- 0
63
64 java_kotlin_mul_test = data.frame(rbind(cbind(java_projects_test$Language, java_projects_test$Total, java_projects_test$Contributors, java_projects_test$Lifespan, java_projects_test$Issues, java_projects_test$LOC, java_projects_test$Stargazers), cbind(kotlin_projects_test$Language, kotlin_projects_test$Total, kotlin_projects_test$Lifespan, kotlin_projects_test$Issues, kotlin_projects_test$LOC, kotlin_projects_test$Contributors, kotlin_projects_test$Stargazers)))
65
66 java_kotlin_mul_test <- setNames(java_kotlin_mul_test, c("Language", "Total", "Lifespan", "Issues", "LOC", "Contributors", "Stargazers"))
67
68 java_kotlin_mul_test$Total <- log(java_kotlin_mul_test$Total)
69
70 java_kotlin_mul_test <- java_kotlin_mul_test[java_kotlin_mul_test$Lifespan != 0, ]
71 java_kotlin_mul_test <- java_kotlin_mul_test[java_kotlin_mul_test$Issues != 0, ]
72 java_kotlin_mul_test <- java_kotlin_mul_test[java_kotlin_mul_test$LOC != 0, ]
73 java_kotlin_mul_test <- java_kotlin_mul_test[java_kotlin_mul_test$Contributors != 0, ]
74 java_kotlin_mul_test <- java_kotlin_mul_test[java_kotlin_mul_test$Stargazers != 0, ]
75 java_kotlin_mul_test <- java_kotlin_mul_test[java_kotlin_mul_test$Total != 0, ]
76 java_kotlin_mul_test <- factor(java_kotlin_mul_test$Language, label = c("Kotlin", "Java"))
77
78 java_kotlin_mul_test <- java_kotlin_mul_test[Reduce('&', lapply(java_kotlin_mul_test, is.finite)),]
79
80 rownames(java_kotlin_mul_test) <- NULL
81 head(java_kotlin_mul_test)
82
83 fit <- lm(Total ~ Language + log(Lifespan) + log(Issues) + log(LOC) + log(Contributors) + log(Stargazers), data= java_kotlin_mul_test)
84 resid <- resid(fit)
85 round(RMSE(resid),2)
86
87 round(summary(java_kotlin_mul_test$Total),2)
```

Listing 30: R Code Used for Statistical Analysis