



^b
**UNIVERSITÄT
BERN**

Recognising structural patterns in code

A k-means clustering approach

Bachelor Thesis

Cédric Walker

from

Luzern LU, Switzerland

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

23. August 2016

Prof. Dr. Oscar Nierstrasz

Haidar Osman

Software Composition Group

Institut für Informatik

University of Bern, Switzerland

Abstract

For processing data, it is necessary for the computer to parse it into a readable format. The building of a parser requires a model of the grammar of the input. For many new programming language dialects and log files, these models are sometimes not available. For creating a model, the structure and grammar information must be inferred from the source code, which is a time consuming process. The goal of this project is to automatically provide an overview of different structures of the source code language. We introduce in this thesis a clustering approach with k-means. Through different representations – numerical vectors inferred from the textual information of segments of the input – structural information should be exploited and assigned to different clusters according to their similarity.

For splitting the input into representation vectors, the smallest instance that can hold structure is found. These are then clustered using the k-means algorithm. This should help developers get a better overview of the data and in a perfect case, make them able to create parser rules for each cluster based on its nearest assigned statements.

The used approach has some promising results, assigning different patterns into different clusters. For measuring the performance of our representation methods, we calculate v-measure scores ranging from 0 to 1, with 1 being a complete match. Out of 24 possible combination of representation, four representations achieved good results, having a v-measure score equal or higher than 0.66 for Java and another 4 with satisfying results, having equal or better scores than 0.4. The representation methods which do well on Java, also do a good job, clustering C#, XML and ABACUS log files, but perform poorly on Brainfuck, with v-measure scores around 0.

Having the concept of clustering languages, the resulting clusters were used to calculate a precision value for classifying the degree of relation between two languages. The precision value is calculated measuring the overlap of the clusters of a first language with the vector representations of a second language. Unfortunately, our approach did not perform well, as it classified every language with the exception of XML, as very closely related.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Goal and Focus	2
1.3	Outline	2
2	Related Work	4
3	Methods	5
3.1	Statements and Tokens	5
3.2	Representers	6
3.2.1	Type Structure Representer	7
3.2.2	Type Specific Representer	7
3.2.3	Distance Zero Representer	8
3.2.4	Distance All Representer	9
3.2.5	Weight Summary Representer	9
3.2.6	Weight Sum First Representer	10
3.2.7	Weight Inverse Representer	10
3.3	K-Means	10
3.4	The Elbow Method	12
3.5	Language Differentiation	13
4	Implementation	15
4.1	Technology	15
4.2	K-Means Pipeline	15
4.3	Language Differentiation Pipeline	17
4.4	Execution Time	19
4.5	Testing	19
5	Evaluation	20
5.1	Evaluation Setup	20
5.1.1	V-Measures	21

5.1.2	The Elbow Method	22
5.2	Evaluating the Representations	23
5.2.1	V-Measure scores for Java	23
5.2.2	Discussion	25
5.2.3	Type Structure vs. Type Specific	26
5.3	Applicability for other languages	27
5.3.1	C#	28
5.3.2	C++	29
5.3.3	Python	29
5.3.4	XML	30
5.3.5	ABACUS log file	31
5.3.6	Brainfuck	31
5.3.7	Discussion	32
5.4	Evaluation the Output	33
5.4.1	Discussion	35
5.5	Evaluating Language Differentiation	35
6	Conclusions and Future Work	38
6.1	Future Work	39
7	Anleitung zu wissenschaftlichen Arbeiten	45
7.1	Introduction	45
7.2	Getting Started	45
7.3	Structure Finder	46
7.4	Structure Analyzer	50
7.5	Language Differentiator	52

1

Introduction

1.1 Motivation

Writing software means reading software code [11]. For developers it is crucial to understand the state of the system in order to carry out development and software evolution tasks. As a consequence, developers often use as much time reading and understanding the written code as actually writing new code. Analysis tools for integrated development environments can help developers understand code and reduce time for assessing software code [13]. For such tools to be created, a software model of the language the tool is built for, must exist in the form of a parser. For many dialects of programming language these models do not exist. Aside from programming languages, log files are a very important part of software applications. Many software applications do produce auxiliary text files as output. These log files are used in various ways, for example in debugging and profiling purposes. While generating log files is a very simple and straightforward process, the understanding of log files can be quite hard, as these can be very large files with complex structure [17]. If a model of the grammar and the underlying structure does not exist, it is necessary to infer these properties from the source code. The inferring of a software model and the building of a custom parser is a complex task that can take several people several months to accomplish. In this bachelors project, we try to build a tool that can distinguish different pattern and structures automatically from software code or log files [7].

1.2 Goal and Focus

The goal of this thesis is to develop a software tool that can atomically find structure in (unknown) software code or log files and generate different sets consisting of related pattern, so that a parser rule can be inferred. The process of creating the software tool is divided into three different sub-problems that had to be addressed:

1. **Statement creation.** Structure manifests itself in an ordered sequence of different tokens. For finding different patterns, the smallest statement had to be found which is able to hold structural information. In this thesis, we assume that carriage returns indicate the beginning of a new pattern. A statement is then split into tokens. Multiple tokens are either separated by blank spaces or by special characters and contain one or more character. Special characters are single character tokens.
2. **Representation and differentiation.** For algorithmically deciding and quantifying the difference between statements, a mathematical representation of the statements needs to be introduced. The approaches used in these methods, differ from creating a vector out of defining features, but instead focuses on the type of symbols used in a token.
3. **Clustering.** A programming language consists of a finite number of different patterns that are used for describing software. These patterns are therefore used multiple times, only differing in the used variables and parameters. Patterns that are similar or differ only in the variables they contain must be filtered. Filtering is achieved through k-means clustering. Patterns should be clustered according to their similarity.

To achieve a solution to these sub-problems, certain assumptions about the commonalities of languages and log files had to be made. We try to keep these assumptions as minimal as possible, so that a wide variety of programming languages and log files could be analysed:

- A language consists of a finite number of patterns that differ only in used variables and parameters.
- Software code contains indents and carriage returns, as it is best practice for many programming languages amongst most developers, to make them more human readable.

1.3 Outline

The rest of the thesis is structured as follows:

Chapter 2 introduces the components of StructureFinder and explains how the used methods work.

Chapter 3 describes the design of the StructureFinder and how the methods introduced in chapter 2 are integrated.

Chapter 4 uses experiments to gather information about the performance of our implemented approach and discusses those findings.

Chapter 5 concludes the work and discusses topics for future expanding StructureFinder.

2

Related Work

Recognizing Patterns is an easy task for humans, as we learn at a very young age to recognize patterns such as digits whether they are handwritten, machine printed, bold, italic or even rotated. Decision making relies on correctly interpreting the patterns of your surroundings [9]. The same statements are also true for computers. For this reason, there is extensive research in supervised and unsupervised classification algorithms. This thesis applies techniques that are also used in many other fields such as Data Mining, Document Classification and even bioinformatics [9].

Most of the work done in the field of automatic recognition of languages is related to natural language processing (NLP). There are different papers regarding unsupervised parsing of natural languages, using part-of-speech (POS) tags or data oriented parsing (DOP) [4] [14] [5] [6]. In regards to automatic recognition of structure in software languages there are not many research articles to be found. Joel Guggisberg [8] used static code analysis to create heuristics for finding keywords of programming languages from source code, but had to make assumptions on the placement of keywords in software code. Our approach in learning about the source code language differs in limiting the assumptions made on the input, allowing to also analyse log files, and making it not a static but a statistical problem. To the best of our knowledge this is the first attempt to use unsupervised clustering algorithm to cluster structure according to similarities.

3

Methods

In this chapter we describe in detail the used methods for creating and tokenizing the input data and put forward the different representations used in the created tool (subsequently also referred to as StructureFinder). In our context, a method is defined as a process, which transforms the input data and if replaced, would result in a different output of StructureFinder.

3.1 Statements and Tokens

To be able to cluster our input data, which are lines of codes, it first needs to be processed into smaller instances that are comparable to each other. As mentioned in the introduction, we use in this project two instances of segmentation: Statements and tokens, where a statement is consisting of one or more tokens. A statement can be created in different ways. Because the main effort of this thesis lies in the structural clustering using the k-means algorithm, the creation of statements is simplified.

A statement is a line of code. Carriage returns and line feeds are a very important way to make software code readable, creating a structure the human eye can perceive more easily. The way of using carriage returns and line feeds for defining a statement, tries to exploit the way we make software code and log files more readable for humans.

For visualising our methods, we introduce a simplified Java example. In this example code, each new line becomes a statement, creating three different statements from this code snippet:

```
if (x > 1) {return 0;}  
if (x <= 0) {return 1;}  
else {return v1;}
```

Listing 1: Statements in Java example

A token is a single word, numeral or punctuation. A statement is further processed into tokens, which are words, numerals, or punctuations.

- A **Word** is a sequence of at least one letter and can contain numerals but cannot contain punctuation.
- A **Numeral** is a sequence of at least one digit. A numeral cannot contain letters or punctuation.
- A **Punctuation** is one special character which cannot be a letter nor a digit.

In our example tokens would be made as follows:

```
if (x > 1) {return 0;}  
if (x <= 0) {return 1;}  
else {return v1;}
```

Listing 2: Tokens in statements

In this example, the words are marked in orange, numerals are in red and punctuations are coloured blue. Tokens are divided by spaces. Punctuation that is not encapsulated by spaces will still be tokenized as an separate special characters. This means, as seen in the previous example that "<=" will be tokenized into two tokens: "<" and "=". Variables such as "v1" are tokenized as a word.

3.2 Representers

As mentioned in the introduction, vector representation is needed for clustering statements. A representer calculates a vector representation out a statement or modifies already existing representations. There are three different categories of representers: Type representers, distance representers and weight representers.

1. **Type Representer** creates vector representations according to the characteristics of the tokens of the statement and assigns numbers to the vector accordingly.

2. **Distance Representer** calculates distance information from the given representation of a Type Representer.
3. **Weight Representer** functions as a noise cancelling or filtering mechanism and can assign weight to specific parts of a representation.

Representers can be concatenated in the order shown above. A Type Representer is always needed, but can optionally be followed by a Distance Representer or a Weight Representer or both.

3.2.1 Type Structure Representer

The first Type Representer to be looked at, is the Type Structure Representer. It assigns a strict value according to the type of the token.

Example: In our implementation we assigned the values 0 for words, 1 for numerals and 2 for Punctuation. For our previously used Java snippet in Listing 1

```
if (x > 1) {return 0;}
if (x <= 0) {return 1;}
else {return v1;}
```

Listing 3: Simplified Java snippet

Type Structure Representer gives us the vector representations:

```
(0 2 0 2 1 2 2 0 1 2 2)
(0 2 0 2 2 1 2 2 0 1 2 2)
(0 2 0 0 2 2)
```

Listing 4: Type Structure Representation example as vectors

3.2.2 Type Specific Representer

The Type Specific Representer uses the same structural information for calculating the vector representation as the Type Structure Representer. But instead of assigning a single value per type, the Type Specific Representer assigns a value per encountered unique token, with the exception of bracket symbols, for which opening and closing brackets are treated as one unique token.

To maintain the structural information of the statement, the assigned value per unique token is then normed into intervals, corresponding to the defined type values in subsection 3.2.1. An element of the vector representation is then calculated as follows:

$$vector_element = \frac{1}{assigned_token_value} + type_value$$

with *vector_element* being the vector element of the representation at the position of the encountered token, *assigned_token_value* the value of the encountered token and *type_value* a predefined value for each different token type.

Example: As in the Type Structure Representer we assign the values 0 for words, 1 for numerals and 2 for Punctuation. A punctuation token therefore is always in the interval (2, 3].

For our previously used Java snippet in Listing 1,

```
if (x > 1) {return 0;}
if (x <= 0) {return 1;}
else {return v1;}
```

Listing 5: Type Specific Representation example

this gives us the vector representations:

$$\begin{aligned} & (\frac{1}{1} + 0 \quad \frac{1}{2} + 2 \quad \frac{1}{3} + 0 \quad \frac{1}{4} + 2 \quad \frac{1}{5} + 1 \quad \frac{1}{2} + 2 \quad \frac{1}{6} + 2 \quad \frac{1}{7} + 0 \quad \frac{1}{8} + 1 \quad \frac{1}{9} + 2 \quad \frac{1}{6} + 2) \\ & (\frac{1}{1} + 0 \quad \frac{1}{2} + 2 \quad \frac{1}{3} + 0 \quad \frac{1}{4} + 2 \quad \frac{1}{10} + 2 \quad \frac{1}{8} + 1 \quad \frac{1}{2} + 2 \quad \frac{1}{6} + 2 \quad \frac{1}{7} + 0 \quad \frac{1}{8} + 2 \quad \frac{1}{9} + 2 \quad \frac{1}{6} + 2) \\ & (\frac{1}{11} + 0 \quad \frac{1}{6} + 2 \quad \frac{1}{7} + 0 \quad \frac{1}{12} + 0 \quad \frac{1}{9} + 2 \quad \frac{1}{6} + 2) \end{aligned}$$

Listing 6: Type Specific Representation as vectors example

3.2.3 Distance Zero Representer

The Distance Zero Representer tries to exploit distance information of statement tokens. It measures the distances between the occurrences of the same values given by the used Type Representer. Measurement starts with zero as a new type value is first encountered. If the same type value is encountered after that, the new vector element will be the distance between the first encounter and the current position of the vector element. Example: For simplifying the example, Type Structure Representer is used for assigning type values.

```
(0 2 0 2 1 2 2 0 1 2 2)
(0 2 0 2 2 1 2 2 0 1 2 2)
(0 2 0 0 2 2)
```

Listing 7: Type Representation as calculated in Listing 4

Applying Distance Zero Representation to the above Java snippet gives:

```
(0 0 2 2 0 4 5 7 4 8 9)
(0 0 2 2 3 0 5 6 8 4 9 10)
(0 0 2 3 3 4)
```

Listing 8: Distance Zero Representation of Type Structure Representation

3.2.4 Distance All Representer

The Distance All Representer calculates the maximal distance between the same type values. Type values are given by a Type Representer. It then sets the maximum distance as the new value for replacing the type value.

Example: Type Structure Representer is used for assigning type values:

```
(0 2 0 2 1 2 2 0 1 2 2)
(0 2 0 2 2 1 2 2 0 1 2 2)
(0 2 0 0 2 2)
```

Listing 9: Type Representation as calculated in Listing 4

Applying Distance All Representer gives the following representations:

```
(5 3 5 3 4 3 3 5 4 3 3)
(6 3 6 3 3 4 3 3 6 4 3 3)
(2 3 2 2 3 3)
```

Listing 10: Distance All Representation vector representation example

3.2.5 Weight Summary Representer

The Weight Summary Representer is used for reducing noise in the statements. Multiple subsequent occurrences of the same type values are summarized into one vector element of this type value. Example: In the following example benefit of this technique can be seen in reducing the difference between statement 1 and statement 2:

```
(0 2 0 2 1 2 2 0 1 2 2)
(0 2 0 2 2 1 2 2 0 1 2 2)
(0 2 0 0 2 2)
```

Listing 11: Type Representation as calculated in Listing 4

Applying WeightSummaryRepresenter gives the following representations:

```
(0 2 0 2 1 2 0 1 2)
(0 2 0 2 1 2 0 1 2)
(0 2 0 2)
```

Listing 12: Weight Summary Representation vector representation example

3.2.6 Weight Sum First Representer

In code, important information such as keywords can often be found in the beginning of a line of code. Weight Sum First Representer adds all type values or distance values, following the current position, giving the vector elements in the beginning a higher weight.

Example: For our previously used Java snippet in Listing 4, applying Weight Sum First Representer gives the following representations.

```
(14 14 12 12 10 9 7 5 5 4 2)
(16 16 14 14 12 10 9 7 5 5 4 2)
(6 6 4 4 2 2)
```

Listing 13: Weight Sum Representation vector representation example

3.2.7 Weight Inverse Representer

The Weight Inverse Representer inverses every vector element. This reduces the variance of the data and inverses the weight of the statement representation. Example: The representation of the Type Structure Representer as shown in Listing 4 is inverted. A Zero value, as possible in Type Structure Representation, is ignored in the calculation:

```
(0 1/2 0 1/2 1/2 1/2 1/2 0 1/2 1/2 1/2)
(0 1/2 0 1/2 1/2 1/2 1/2 1/2 1/2 1/2 1/2)
(0 1/2 0 0 1/2 1/2)
```

Listing 14: Type Specific Representation as vectors example

3.3 K-Means

K-means is an unsupervised clustering algorithm. It clusters the given data in k different clusters, with k being a known parameter, according to the vector representation of the data. For clustering the data, k different centroids are randomly initialized and the algorithm labels the data according to the nearest centroid. At the end of the algorithm, the centroids will have minimal distance to its assigned data points. This is achieved by moving the centroids to the euclidean mean of all the assigned data points. The mathematical expression, given the trainings set $\{x^{(1)}, \dots, x^{(m)}\}$ and $x^{(i)} \in \mathbb{R}$ of the k-means algorithm is as follows:

1. Initialize **cluster centroids** $\mu_1, \mu_2, \dots, \mu_k \in \mathbb{R}^n$ randomly.
2. Repeat until convergence: {

For every i , set

$$c^{(i)} := \arg \min_j \|x^{(i)} - \mu_j\|^2.$$

For each j , set

$$\mu_j := \frac{\sum_{i=1}^m 1\{c^{(i)} = j\} x^{(i)}}{\sum_{i=1}^m 1\{c^{(i)} = j\}}.$$

Figure 3.1: Mathematical expression of k-means [12].

A practical example of the k-means algorithm:

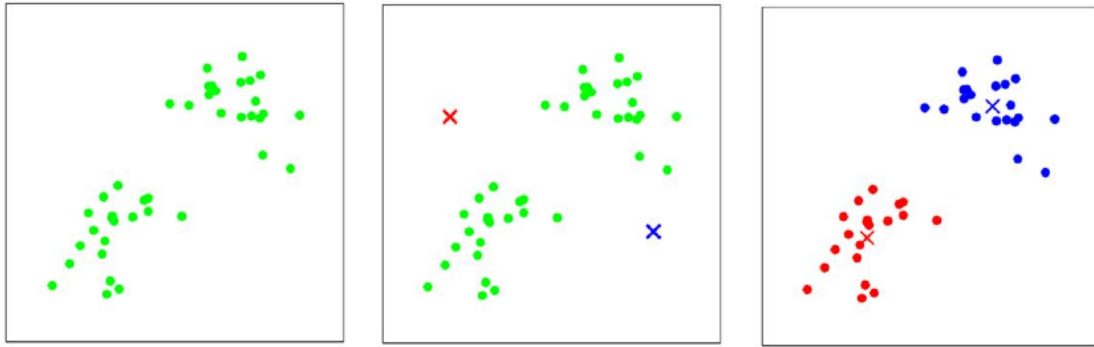


Figure 3.2: Example of k-means iteration [12].

The red and blue crosses visualize the centroids. They are initialized randomly at the start of the algorithm. The centroids will then in every iteration gradually move nearer to the mean of the final clusters, minimizing the cost of the distortion function:

$$J(c, \mu) = \sum_{i=1}^m \|x^{(i)} - \mu_{c^{(i)}}\|^2.$$

J measures the sum of the squared distances between each training example $x^{(i)}$ and the centroid of the cluster $\mu_{c^{(i)}}$ to which it has been assigned.

3.4 The Elbow Method

One problem of the k-means algorithm is that the parameter k must be known, which in our case is not known. K is dependent on the given input on which we want to limit our assumptions. Therefore, a way to algorithmically calculate k is needed. The idea behind the Elbow Method, is to find the number of centroids, so that if you add one centroid, the fit of the centroids to the data – measured by the cost function J – does not improve significantly any more. In the case of k equals one, the position of the centroid would be the mean of all data points and in the case of k equals the number of all data points, the centroids would be the points themselves. As a consequence, k equals one results in the worst fit and while k equals the number of all data points gives perfect fit [10]. This can be illustrated by the following example where we normed J by k :

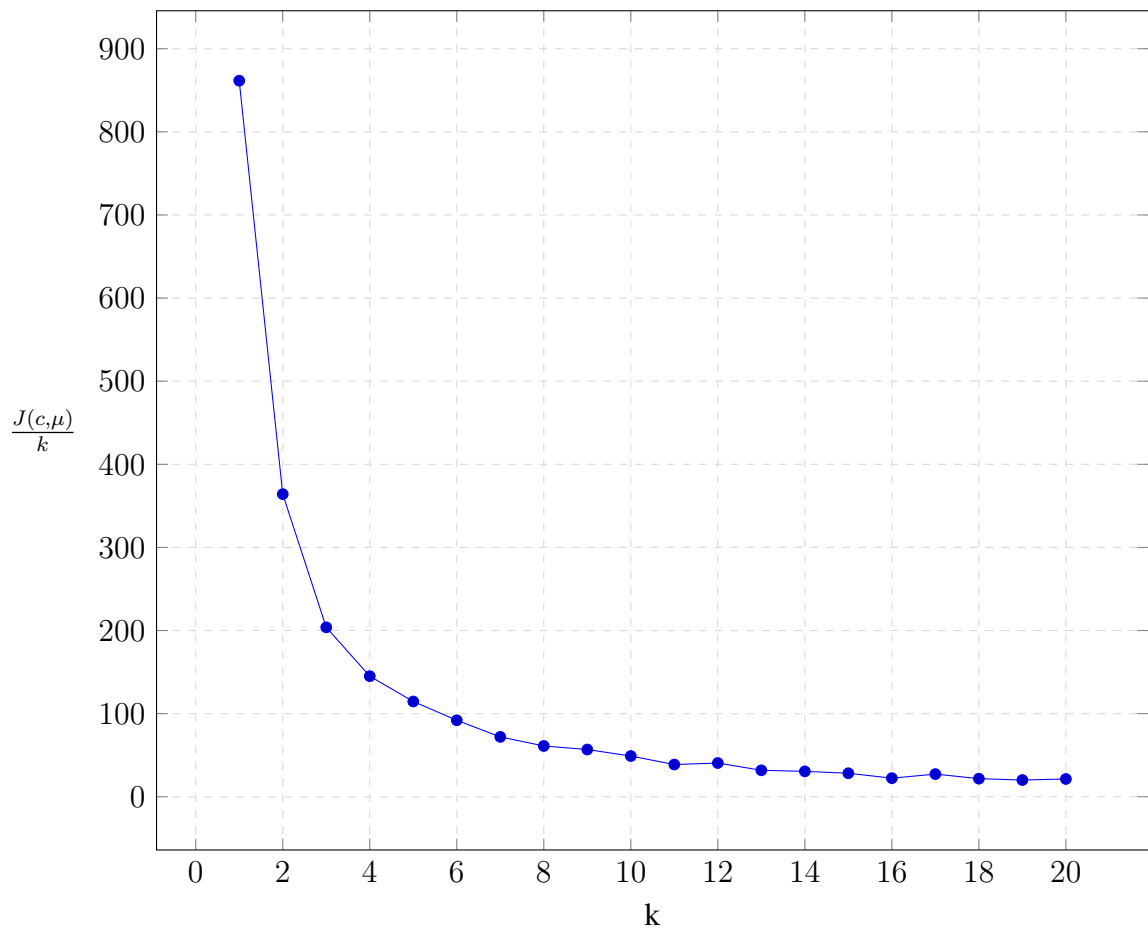


Figure 3.3: Average Fit per centroid per k , calculated with StructureFinder using Java file input and TypeStructureRepresenter.

As seen in the figure above, the improvement in the data modelling decreases as k becomes larger. By calculating the partial derivative $\frac{c_{i-1} - c_{i+1}}{k_{i-1} - k_{i+1}}$, the decrease can be measured. The partial derivative is maximized until it reaches a threshold α , with $\alpha \in \mathbb{R}^-$. This value defines our guess for k .

3.5 Language Differentiation

Based on the representation based k-means clustering, this thesis also provides a method to calculate a precision value of how closely related two languages are. For this, one language is clustered with the k-means algorithm. For each cluster the radius is calculated ranging from the coordinate of the centroid to the assigned data point which is the furthest away from the centroid. For comparing a second language to the first, it suffices, to only calculate the vector representation of the data as discussed in section 3.2. For each represented data point of the second language, it is then checked, if it is inside any circle spanned from the previously calculated radii. The precision value describing the relation is calculated as follows:

$$language_relation = \frac{number_of_data_points_in_radius}{number_of_data_points}$$

This means, that for any given two languages, the result 1 indicates that the two languages are related closely and 0 indicates that the languages are completely different. The two dimensional visualisation can be found in Figure 3.4. The red and blue crosses represent the data points of the clusters of the clustered language, with the centroid as the centre and the green crosses being the data points of the second language.

In this example, only one of two green crosses, the data point of the second language, inside one of the two circles spanned from the radii of the previously calculated k-means clusterings blue and red. This would result in a precision value of $\frac{1}{2} = 0.5$.

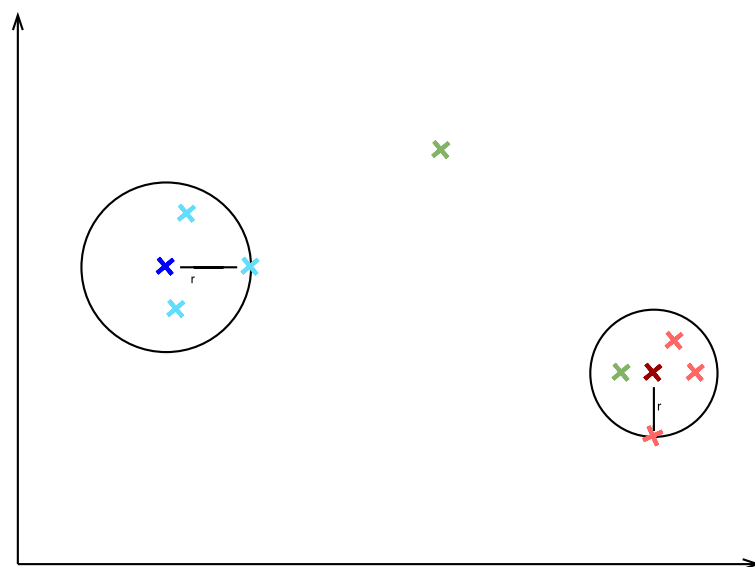


Figure 3.4: Two dimensional example of Language Differentiation

4

Implementation

This chapter gives an overview of the implementation of the k-means clustering tool StructureFinder. First, we look at used resources. Second, we look at the k-means and Language Differentiator pipeline and how the data is processed.

4.1 Technology

StructureFinder is developed in Pharo 4.0 [1]. Statements and tokens are separated with Petit Parser [3], which is also used for recognizing the types of the tokens.

4.2 K-Means Pipeline

The first implementation to verify our ideas of clustering statements of software code was a simple k-means clustering algorithm with a single static k as a parameter.

Known k . The implementation of the k-means clustering for calculating a single cluster is quite straightforward. A code file is loaded by a File Handler object. With the help of Petit Parser the loaded code file is then split into statements and tokens by the New Line Statementmaker. A Representer Runner is responsible for handling the different representation methods and calls to the loaded Representers. The Representers then successively create the vector representation. These representations are clustered by the implemented k-means algorithm. The processed clusters are then sent back to the

File Handler, which saves each cluster in a separate file, meaning it saves per centroid their assigned statements. The centroids are initialised in a range of values determined by the Representors used, so that no initialised value can be higher than any value of the calculated representations. This is done, to minimize the risk of getting centroids, without any data assigned to them, so called empty centroids. The data flow can be seen in the following figure:

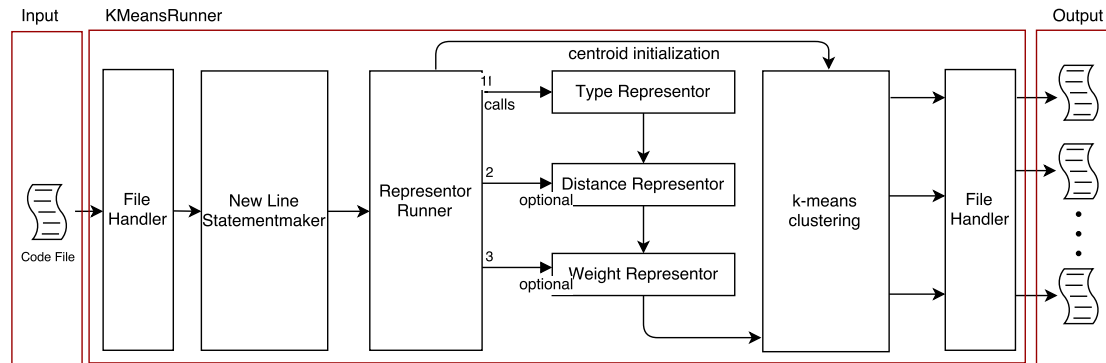


Figure 4.1: Simplified data flow chart of the single k k-means pipeline.

As discussed before, assumptions on the structure of the input should be minimal. Therefore we do not want to make assumptions about k , but want to have an algorithm to calculate the optimal number of clusters.

Interval. If the optimal number of clusters is not known, it is needed to calculate different clusterings with different numbers of clusters and compare them. For this, we implemented a way to create different clusters for a given interval. For each natural number in this interval, a set of clusters is calculated and then compared according to the elbow method discussed in section 3.4. Each calculated set of clusters is given to the File Handler and output is created for each set of clusters in the same way as in the implementation shown above. All results from the distortion function, as well as the used representation and the optimal value of k is saved in a log file. The expansion results in this updated data flow:

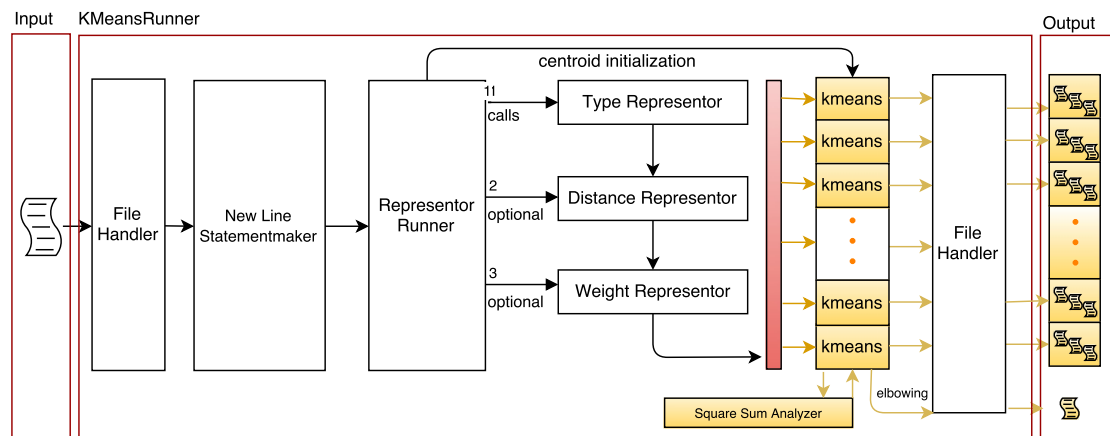


Figure 4.2: Simplified data flow chart of the expanded k-means pipeline, with highlighted differences to 4.1 in yellow.

The Square Sum Analyser calculates the distortion function for the elbow method for guessing k for the different k-means clusterings.

For the updated implementation of StructureFinder, there is also a very important addition to the output: the nearest statements of the centroids of the best clustering. These statements are also saved to the log file. The bigger our input files are, the bigger the output files get. This makes it harder to get an overview of found structures based on lists of all the statements. The nearest statements of the centroids serve as perfect example and should incorporate the pattern featured in the corresponding centroid.

4.3 Language Differentiation Pipeline

As discussed in section 3.5, it is possible to use the implementation of the k-means clustering to calculate a precision value, measuring how closely related different languages are. The implementation is expanded by adding a control object for the language differentiation. The data flow of the Language Differentiation looks accordingly:

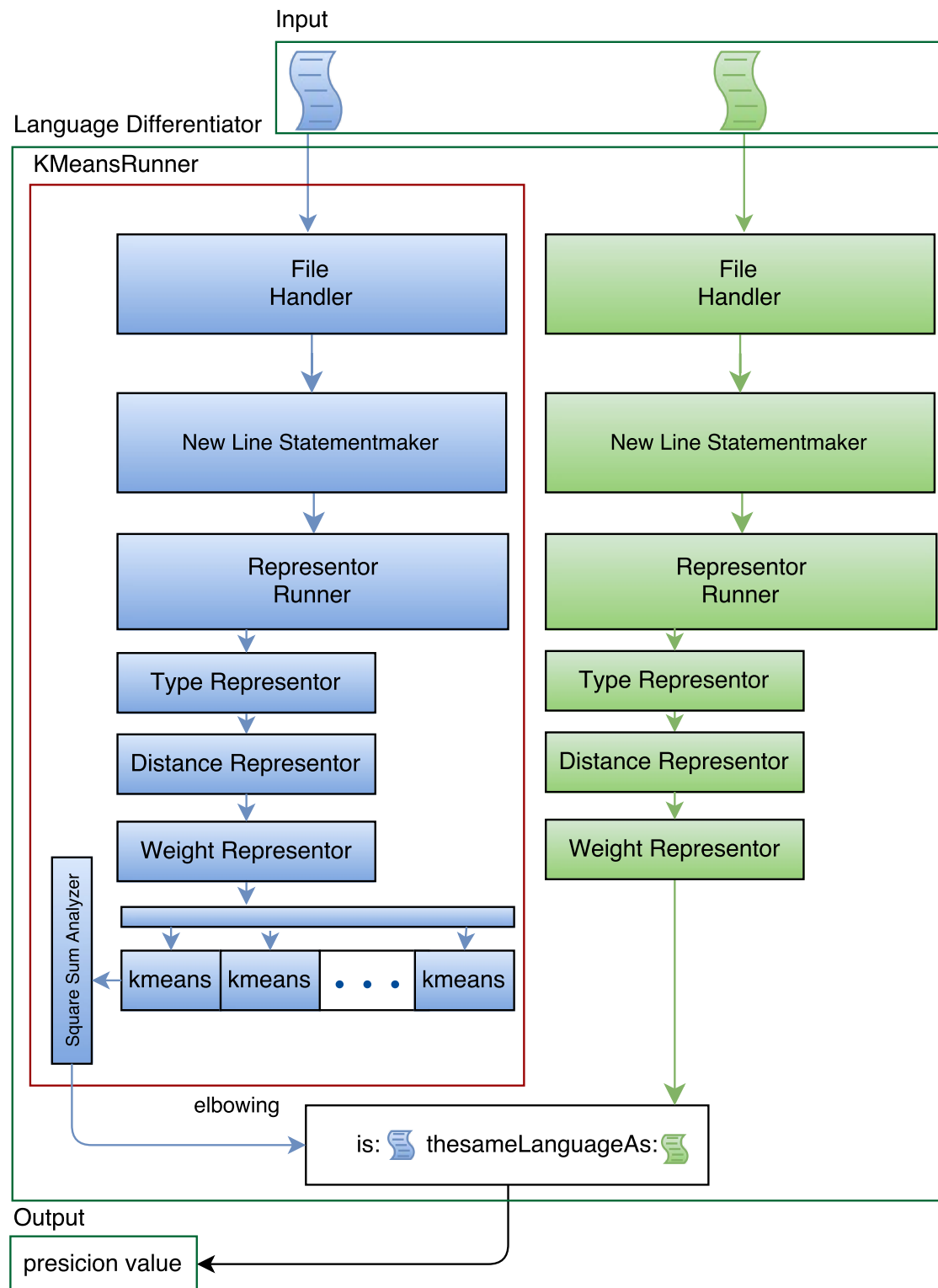


Figure 4.3: Simplified data flow chart of the Language Differentiator pipeline.

4.4 Execution Time

The execution time of StructureFinder is dependent on different parameters of the input and configuration. The parameters the tool depends on, are the number of iterations, the size of the training data, the number of clusters, as well as the number of attributes, meaning the length of the vector representations.

While running the algorithm, we experienced significant slow downs for calculating the clusterings, using the Type Structure Representer or with calculation using the Weight Inverse Representer. These two representations are the only ones using float calculation, instead of integer calculations. Because there is a lot of vector calculation involved, it leads to the conclusion that floating point calculation is the cause of the slowdowns.

4.5 Testing

Testing is done in SUnit, which is integrated in Pharo 4.0. A test suite was implemented testing vector calculations and the correct functioning of the creation of the representation.

5

Evaluation

In this chapter, we evaluate how well the ideas we implement in this thesis work. For this, we first need to introduce a way to measure the performance of k-means clusterings and take a look how the elbow method should be configured. Second, we try to answer the question of which representation introduced in section 3.2 performs the best. Third, we run StructureFinder on different languages to see if it can handle different structural concepts and last, we look at how the clusters created from StructureFinder look like and what we can learn about the structure of input data out of its clustering. After the evaluation of StructureFinder, we look shortly at the performance of our expansion for classifying the degree of relation between two languages.

5.1 Evaluation Setup

Evaluating an unsupervised learning algorithm does have the problem that there is no labelled data to test against. But since we know what the goal of the algorithm is supposed to be, we are able to manually cluster files into a reference clustering, making the cluster assignment ourselves to test our methods against. In a first phase, we can then introduce v-measures for calculating a score based on a reference clustering. For calculating clusters we need to have a elbow threshold parameter α (introduced in section 3.4) for guessing k, as we do not want to define k. So in a second phase we experimentally look for a desirable value of α for further computations.

5.1.1 V-Measures

Given the knowledge of the ground truth class assignments of the samples, which we manually created, it is possible to define a metric, using conditional entropy analysis. In particular Rosenberg and Hirschberg [16] define the following two desirable objectives for any cluster assignment:

- **homogeneity:** each cluster contains only members of a single class.
- **completeness:** all members of a given class are assigned to the same cluster.

Homogeneity and completeness scores are formally given by:

$$h = 1 - \frac{H(C|K)}{H(C)}$$

$$c = 1 - \frac{H(K|C)}{H(K)}$$

with the conditional entropy of the classes, given the cluster assignments $H(C|K)$ being defined as:

$$H(C|K) = - \sum_{c=1}^{|C|} \sum_{k=1}^{|K|} \frac{n_{c,k}}{n} \log \left(\frac{n_{c,k}}{n_k} \right)$$

and the conditional entropy of the classes is given by:

$$H(C) = - \sum_{c=1}^{|C|} \frac{n_c}{n} \log \left(\frac{n_c}{n} \right)$$

where n_k is the number of samples in cluster k and $n_{c,k}$ is the number of samples from class c assigned to cluster k .

The conditional entropy of clusters given class $H(K|C)$ and the entropy of clusters $H(K)$ are defined in a symmetric manner.

The V-Measure is defined as the harmonic mean of homogeneity and completeness:

$$v = 2 \frac{h * c}{h + c}$$

This gives us values for h , c and v ranging from $[0, 1]$. These values can be used as a score to compare how well our representations performed, with 1 being a perfect score and 0 being the lowest score [16].

5.1.2 The Elbow Method

The elbow method, as seen in Section 3.4, is a heuristic for finding a good value for the parameter k . In our manually clustered Java data, we were able to cluster it into 13 different clusters, making this our desired output of the elbow method. For the Type Structure and the Type Specific Representer we run the elbow methods with different threshold parameter α on Java the previously mentioned Java code file:

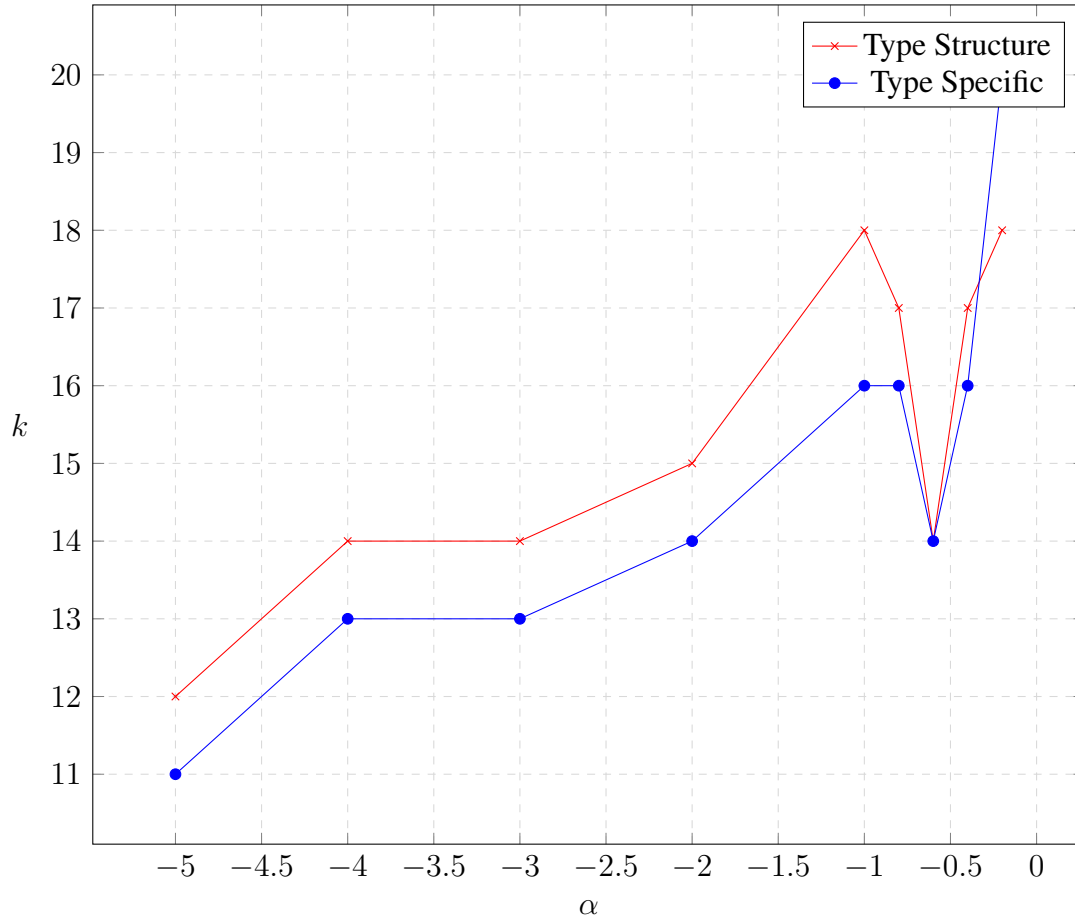


Figure 5.1: Plot of elbow method. Heuristic value of k , given α .

The smaller α , the smaller is k . These two plots show, that a value of -3 is needed to get near the desired value for k . Therefore -3 will be used for further calculations. It is important to notice that the calculated k differs for the same value of α for the two different used representation methods. Also we have to address the problem of statistical outliers, as can be seen in this graph at around $\alpha = -0.5$. Given that the underlying

k-means algorithm is statistical, outliers are to be expected. Problems related to the k-means implementation are discussed in section 5.2.

5.2 Evaluating the Representations

Representations are an important part of StructureFinder, because they change the behaviour of the k-means algorithm. For further evaluation we want to know, which Representers perform well and should therefore be used if a new language is clustered.

5.2.1 V-Measure scores for Java

Using the v-measure scores, we can calculate two tables containing all the possible representations introduced in this thesis. Table 5.1 is clustered using the Type Structure representation, while table 5.2 is calculated using Type Specific representation. The two tables are hierarchically structured. All the calculations in the two tables were made with clusterings for $k \in [10, 25]$ and with the elbow threshold parameter $\alpha = -3$. Aside from the calculations of the scores for homogeneity, completeness and v-measures, the tables also feature the elbowed value for k as "perfect k". Using k-means clustering, it is possible for centroids to not be assigned to any data points. Because of this, the number of empty centroids encountered in the clustering is featured as well. While computing the v-measure scores, there was a problem with the range of the values computed. Under the condition of empty centroids present, there was a possibility for the scores to be below zero. Fortunately this was only the case if there were many centroids without assigned data points. For simplicity, these values were rounded to zero.

Table 5.1: Java v-measures of all possible representations with Type Structure Representer

Representer	homogeneity	completeness	v-measure	perfect k	empty centroids
Type Structure	0,61	0,9	0,73	13	0
Dist Zero	0,47	0,35	0,4	14	4
Weight Summary	0,22	0,42	0,29	16	5
Weight Sum First	0,14	0,11	0,12	12	7
Weight Inverse	0,72	0	0	11	9
Dist All	0,89	0,53	0,66	13	4
Weight Summary	0,08	0,29	0,13	13	6
Weight Sum First	0,1	0	0	20	13
Weight Inverse	0,57	0	0	11	5
Weight Summary	0,25	0,23	0,24	14	8
Weight Sum First	0,21	0,27	0,24	11	5
Weight Inverse	0,77	0,56	0,65	11	0

Table 5.2: Java v-measures of all possible representations with Type Specific Representer

Representer	homogeneity	completeness	v-measure	perfect k	empty centroids
Type Specific	0,68	0,91	0,77	11	0
Dist Zero	0,56	0	0	13	6
Weight Summary	0,45	0	0	15	8
Weight Sum First	0,12	0	0	20	10
Weight Inverse	0,65	0,65	0,62	11	6
Dist All	0,36	0	0	14	9
Weight Summary	0,6	0	0	13	8
Weight Sum First	0,1	0	0	20	9
Weight Inverse	0,87	0	0	11	9
Weight Summary	0,62	0,95	0,75	14	0
Weight Sum First	0,27	0	0	15	9
Weight Inverse	0,47	0,73	0,58	20	6

The best results were achieved by using only the two Type Representers. They scored extremely well in completeness with values up to 0.91, if used without any other representation. The best scores however were achieved by adding the Weight Summary Representer, which as discussed in section 3.2, reduces noise in the type representation.

With the addition of Weight Summary representation, the algorithm scores up to 0.95 in completeness and works extremely well if used with the Type Specific representation, but not as good with the Type Structure representation. If we look at the highest scores, Type Structure Representation is the better choice having both the highest score with the representation that uses only Type Representation and the highest score overall with the use of Type Specific Representation and the Weight Summary Representer. We will for further evaluation concentrate on the Type Representors and the Weight Summary Representer and will take a look on how they can cluster different languages.

The heuristic guess for k does extremely well. As mentioned before, $\alpha = -3$ was used for the calculations of the elbow method. Of interest are all the combinations of Representers that do not result in empty centroids. For those representations, the guessed k varies maximally by 2 from the manually created example. For the best combination, being composed of the Type Specific and the Weight Summary, it only differs by one additional cluster.

5.2.2 Discussion

While there are pretty good results from StructureFinder, most of them have only low scores. This section tries to find an explanation as to why.

Assumptions. The main goal of this thesis was to find structural patterns in software code files or log files, while keeping the assumptions about the structure of the input files as minimal as possible. One of the problems of using a clustering algorithm is, that said algorithm does make assumptions about the data it clusters. For k-means, this assumption lies in the goal of the algorithm to minimize the squared sum of the distances from the data points to the centroids. This might mean that it always tries to find clusters that are circular in shape. This might not be the case for the representations we create in this thesis, as the representations were created as seen fit for representing certain characteristics of structures in a statement.

Variance. If we look at the scores in tables 5.1 and 5.2, it is striking that methods do poorly which allow for high variance in the representations. This is the case for all the Distance Representers, as well as for representations such as Weight Sum First. It is interesting to see, that the Distance Representers do better, if we lower the variance of the data, which is the case if we additionally use the Weight Inverse Representer. This could be the case, again, because the k-means algorithm tries to minimize the squared sum distances from data points to centroids. With a big variance in the data, this can cause the k-means algorithm to give more weight to clusters that are more dense and thus neglecting smaller clusters with lower density [15].

Empty Centroid. In the results, we needed to feature a column for empty centroids, as this is a big problem that needs to be addressed. Centroids are randomly initialized with uniform distribution over all dimensions of the vector, scaled to the highest vector element found in the data. In contrast to featurization, representations allow for vectors

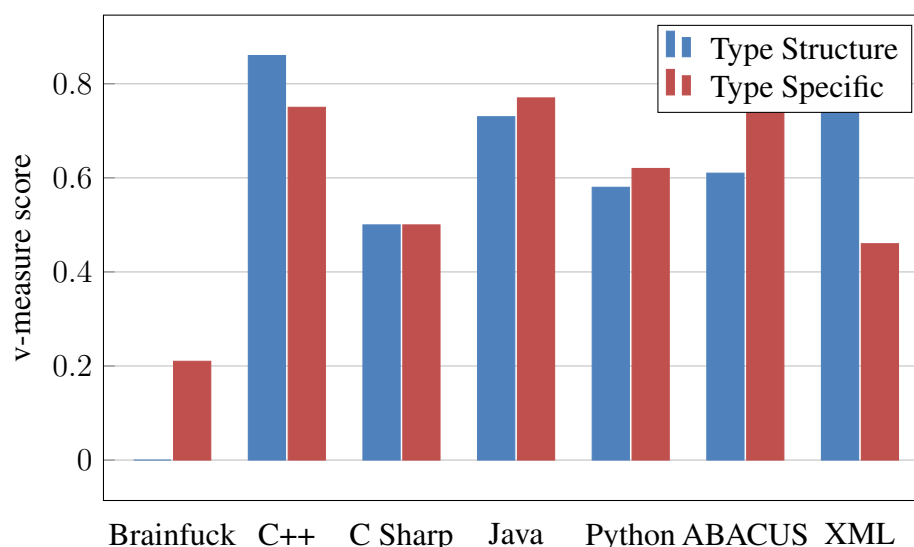
of different sizes. For vector calculations this means that smaller vectors need to be padded with zeros to compare them. Therefore, more relevant data lies in the lower dimensions of a vector. If the centroids are uniformly initialised in every dimension, initialized centroids can be further away from the data points as any other centroid. We can see this in Weight Sum First, where we have high values at the beginning and low values at the end. So Weight Sum First increases this phenomena, but every method that allows for high variance in the representation is therefore more prone to have empty centroids.

Convergence. Scores rely on the convergence of the k-means algorithm. For the algorithm to be guaranteed to converge to the optimal result, certain assumptions of the data must be met. If these assumptions are not met, it is possible that the algorithm converges to a local instead of a global optimum. In our thesis, it is unfortunately the case that these assumptions are not always met. This can result in different results, as in the tables above. This does not necessarily mean that our calculated data is not valid. If the Representers are run multiple times, the algorithm that did well in the scores were actually the most robust ones, having the lowest variance in the scores and the distortion function. This can be traced back to the robustness in relation to the empty centroids, meaning that if a representation is more likely to have empty centroids, it is also to be expected to have higher variance of the v-measures and in the distortion function.

Summary. The best representations in our evaluation were clearly the Type Structure and the Type Specific. It seems that they best fit in the assumptions made by the k-means algorithm and that their limited variance prevents them from suffering from empty centroids. For clustering new languages one of these two representation is likely to achieve satisfying results. For creating new representation methods the discussed topics above should be taken into account.

5.2.3 Type Structure vs. Type Specific

The two Type Representers are performing reasonably well. In this section we want to find out which of the two is expected to perform better. For this we use data calculated for the evaluation of the applicability for other languages in section 5.3.



For Java it was clear that only the Type Representers worked well. Weight Summary could improve the results, but was not guaranteed to. For evaluating which of the two Type Representers performs better, we calculated v-measure scores for both Type Representers on 6 additional Languages. It is hard to see which one of the two Type Representers does overall perform better, as their performance relies heavily on the input. This also stands out if we calculate the averages of all the scores of the Type Structure and Type Specific representation. The average of all scores of the Type Structure Representer is 0.57, while the average of all the Type Specific representation lies by 0.58. But if we look at the average of the best performed representations per language, we get a score of 0.64, and if we ignore Brainfuck, which we were not good in clustering at all, we get a score of 0.71. This makes it hard to tell which of the two Representers is overall expected to perform better, as scores depend more on input language as on the Representer. Predictions based on the languages are very hard. As we saw in this chapter, even closely related languages can behave very differently in the clustering process.

5.3 Applicability for other languages

The main evaluation of the k-means algorithm was done for Java. As a result of section 5.2.1, we can now discard certain representations as not suitable. We will now take a look at the performance of the representations for other languages than Java, including an Abacus [2] log file, for evaluating how our tool varies for different languages. It is desirable that it performs well on other languages, because we want StructureFinder to be able to cluster unknown languages reasonably. All the input files used for clustering contain source code randomly selected from Github. Each file contains roughly about 300 lines of code or more from different source files.

5.3.1 C#

Table 5.3: C Sharp v-measures

Representor	homogeneity	completeness	v-measure	perfect k	empty centroid
Type Structure	0,86	0,75	0,8	13	2
WeightSummary	0,23	0,61	0,34	12	5
Type Specific	0,61	1	0,75	15	2
WeightSummary	0,9	0,15	0,26	11	4

If we look at the taxonomy of programming languages, we see that C# is very close to Java. Therefore, it is to be expected that many structural elements are the same for both languages. It is not a surprise that the results of our methods are not that different to those calculated for Java. The heuristic for k is not as good as for Java. While we manually classified C# into 17 clusters, even the best guess missed two clusters. This makes $\alpha = -3$ not the best choice for C#. Nevertheless, the scores for the single use of the Type Representers are satisfying with the best v-measure around 0.8. But it stands out that the Weight Summary Representer – our noise reduction – does not perform as well as in Java. This is surprising, given the close relation between Java and C#. While it resulted in good scores for Java, giving the best score with Type Specific representation, the scores for C# are remarkably low. Differences can lie in the composition of the used data. For this thesis around 300 lines of code for each language are used in the clustering of the language. A big challenge in clustering is the correct assignment of comments. If a code contains lots of comments that are formatted in different ways or even software code that is commented out, this will be clustered differently from intended. The C# code used for clustering shows indeed lots of different comments, that were not clustered as intended.

It is also surprising, that we have way more empty centroids than in Java. The clustering of Java resulted in neither of the used representations used for C# in empty centroids. It is not clear as to why this happens. One thing that seems to be standing out, is that the use of the noise reduction, in general either improves the result, or causes it to have more empty centroids. This can be the case because reducing noise also means that information is removed, lowering the dimension of the vectors. If too much information is removed, the clustering cannot work properly, as variance of the data may become too low, so that data points cannot be distinguished well enough.

5.3.2 C++

Table 5.4: C++ v-measures

Representor	homogeneity	completeness	v-measure	perfect k	empty centroid
Type Structure	0,35	0,84	0,5	16	0
WeightSummary	0,67	0,27	0,39	12	3
Type Specific	0,4	0,65	0,5	16	4
WeightSummary	0,15	0,41	0,22	17	7

C++, like C#, is very closely related to Java. So similar results are to be expected. With Type Structure and Type Specific representation, this is the case, even though they score lower. It is suspected that there are the same problems in clustering as in C#. C++ does also contain different ways for commenting that are difficult to handle. The scores of C++ are more similar to C# than to those calculated for Java.

Like C#, the manual clustered example contained 17 clusters. The number of clusters was guessed better than for C#, with Type Structure and Type Specific only missing one cluster and Type Specific with applied Weight Summary guessing the correct k.

5.3.3 Python

Table 5.5: Python v-measures

Representor	homogeneity	completeness	v-measure	perfect k	empty centroid
Type Structure	0,47	0,76	0,58	13	0
WeightSummary	0,66	0,41	0,5	13	3
Type Specific	0,51	0,8	0,62	16	2
WeightSummary	0,27	0,53	0,37	18	1

Python does have similar constructs to Java, but is not as closely related to it compared to C++ or C#. The evaluation of Python shows reasonable clusterings for Type Structure and Type Specific, but does also not well, if Weight Summary representation is applied. If we include the guessed value for k and the number of centroids, Type Structure does better than Type Specific representation, even though the Type Specific alone does have

a higher score. Unfortunately, k is not guessed as well for Type Structure as for Type Specific. The number of clusters in our manually created clustering is 12. Elbowing does achieve a good value for the Type Structure Representer, with or without the application of Weight Summary Representer, creating only one extra cluster. For Type Specific, k is guessed too big, having 4, respectively 6 additional clusters than the manually created clustering. If we take a look at the used code, we see lots of different statements beginning with "print" and should be assigned to the same cluster. Having just one word at the beginning of the statement characterising the whole statement works the same way as comments do. This makes it hard for the clustering algorithm to assign them properly.

5.3.4 XML

Table 5.6: XML v-measures

Representer	homogeneity	completeness	v-measure	perfect k	empty centroid
Type Structure	0,7	0,86	0,77	11	0
WeightSummary	0,6	0,45	0,51	11	5
Type Specific	0,58	0,38	0,46	13	5
WeightSummary	0,83	0,51	0,63	11	2

All the previously evaluated programming languages have similar constructs. This cannot be said about XML, as it is a hierarchical mark-up language. The best results were scored with the Type Structure Representer, scoring high in homogeneity and completeness. The other results are not conclusive, because it cannot be determined, if noise reduction does help or not.

The manually clustered example contains 15 clusters. This means, that for XML, the elbowing parameter $\alpha = -3$ does not seem to be the best choice, as the guess for k is too low over all the evaluated representations.

5.3.5 ABACUS log file

Table 5.7: ABACUS log v-measures

Representor	homogeneity	completeness	v-measure	perfect k	empty centroid
Type Structure	0,84	0,48	0,61	14	6
WeightSummary	0,37	0,33	0,35	14	7
Type Specific	0,77	0,41	0,54	12	3
WeightSummary	0,89	0,65	0,75	11	7

The ABACUS log file does achieve good scores, even though there are many empty centroids in the final clusterings. Only one representation method has a score lower than 0.5. In the scores gathered, it stands out that Weight Summary representation only improves the score, if used with Type Specific representation. For Type Structure Representer this does not apply. The number of clusters would have been guessed pretty well by the elbow method. But if we subtract the empty centroids from it, we get too few clusters.

It is not certain why the clustering of the ABACUS log files results in this many empty centroids, but it could mean that there are a lot of the same types in the same part of the statements, causing problems with the initialization of the centroids, as was discussed before. The problem with this explanation is that Type Structure with Weight Summary does not perform better, which should be expected in that case.

5.3.6 Brainfuck

Table 5.8: Brainfuck v-measures

Representor	homogeneity	completeness	v-measure	perfect k	empty centroid
Type Structure	0,5	0	0	12	5
WeightSummary	0,2	0,04	0,07	11	8
Type Specific	0,62	0,01	0,21	13	5
WeightSummary	0,28	0	0	11	6

Brainfuck differs from any other programming language evaluated in this thesis. Brainfuck uses only eight different special characters as instructions, with everything else

being treated as comments. This makes clustering the statements very hard, as there is very little difference between statements. Even though we remarked in the evaluation of Java that lower variance can result in better scores, it is clear that too low variance cannot be clustered correctly, as all statement representation vectors gather around the same points. Because we initialize centroids uniformly over all dimensions, this results in many empty centroids for Brainfuck. Also, it is important to see, that noise reduction goes awry as well. Noise reduction reduces the dimension of the representations remarkably, because Brainfuck only uses one type of token for all instructions. Reducing dimensions, does also mean that we lose information. This may work well for Java, as it really reduces noise, but in Brainfuck, it loses too much information to still be clustered correctly.

5.3.7 Discussion

We can see that StructureFinder can cluster different languages reasonably. For this we picked languages that are not all closely related. This can help us anticipate if StructureFinder is able to cluster languages we do not have a lot of information about. And as discussed for each language clustered the results are satisfying. Nevertheless there are some problems that need to be addressed which can affect the clustering.

Comments. Comments are hard to find. In programming languages there is often more than one way a comment can be defined, as well as the possibilities for using single-line or multi-line commenting. A comment is most of the time defined by the first few tokens in a statement, with the rest of the statement containing arbitrary structural information and sometimes even out-commented software code. This makes them very hard to cluster correctly.

Formatting. Our approach for creating statements (taking every new line as a statement, see section 3.1) is depended on the correct formatting of the software code. Not correctly formatted code can cause structure to be split into statements incorrectly.

Statements. In StructureFinder, carriage return is the used delimiter for creating statements. This is a very simple approach and does not need to map the structure of loop based programming languages adequately.

Dimensionality. There are different dimensions in the data that need to be looked at: vector dimension, number of clusters expected and the number of lines in the input. If one of those mentioned dimensions is too low, it can result in not proper clustering by the k-means algorithm, because it does not have enough difference in the data to assign them to different clusters. More dimensions can contain more information. Low vector dimensions could be causing the problems in clustering Brainfuck, as it contains almost only punctuation tokens.

Summary. Different Languages behave differently in our clustering approach. But with the exception of Brainfuck the v-measure scores for the clusterings are satisfying. It is unfortunately not possible at the moment to make predictions of how the composition

of the language affects the outcome of the clustering algorithm, because related languages behave differently. This difference in behaviour could also be caused by the limited size of the input. It is important to note that the size of the input used for the evaluation was only around 300 lines of code each. Even though they were gathered from different source files, the results calculated in this thesis are purely experimental and do not allow for statistical conclusions on what to do or not to do in regards to the contents of the input.

5.4 Evaluating the Output

If we take a look at the tables 5.1 and 5.2, we can see that our calculated scores indicate that the clusterings made from StructureFinder to some extent matches our manually created clusterings. In this section we look now at the output of the tool -as described in section 4.2- to find out, what we can learn from the nearest statements of the centroids and if these statements can help to gain an overview of the structure the languages used in our input files and thus help a developer in creating parser rules. For this we look at the nearest statements for Java input files, created using first, Type Structure Representer and second Type Specific Representer with elbow parameter $\alpha = -3$.

Nearest Statments to Centroids
public class BootReceiver extends BroadcastReceiver {
import ch . xonix . mensa . unibe . model . Mensa ;
*
public abstract class AbstractInvitationsFragment extends Fragment {
return container ;
import java . util . ArrayList ;
public class CriteriaMatcher {
/ * *
protected void onCreate (Bundle savedInstanceState) {
public void onDestroy () {
android . R . layout . simple _ list _ item _ 1) ;
container = new ArrayList < Criteria > () ;
crit . setCriteriaName (criteria) ;

Figure 5.2: Nearest statements per centroids, created witch Type Structure Representer

Each line of figure 5.2 represents a centroid and with this the perfect example of the

pattern a centroid should be composed of. Therefore it would be desired that these statements differ in their pattern. As we can see different statements with different constructs in figure 5.2, there are multiple occurrences of similar statements. The first thing that stands out is the occurrence of two different lines which are related to commenting. It is clear that comments are problematic in our clustering approach, because the beginning of the statements and not the representation of the whole information defines the cluster it should be assigned to. Also, there are two almost identical statements on line 1 and line 4 with the difference of one token. This means, that the length of the representation does play a big part in the assignments of the statements to clusters. It is to be noted here that this would be the same statement if Weight Summary Representer would have been used with the Type Structure Representer. Also we have two centroids declaring imports and two method declarations. Also interesting is to note what is missing. There are no conditionals (if) and no while or for loops in our generated figure 5.2. In our manual clustering we assigned all if statements to one centroid.

Nearest Statments to Centroids
import ch . xonix . mensa . unibe . model . Mensa ;
Intent intent = new Intent (this , DrawerMenuActivity . class) ;
public class BootReceiver extends BroadcastReceiver {
public abstract class AbstractInvitationsFragment extends Fragment {
container = new ArrayList < Criteria > () ;
import java . util . Set ;
tenOClock . set (Calendar . YEAR , Calendar . MONTH , Calendar . DAY _ OF _ MONTH , 10 , 0 , 0) ;
else {
}) ;
if (LoginService . isLoggedIn ())
adapter = createAdapter () ;
public class CriteriaMatcher {
* This simple abstract class has the functionality which is common for both

Figure 5.3: Nearest statements per centroids, created witch Type Specific Representer

Type Structure Representer does a better job, assigning the comments to the same clusters. We still have the problem here that we have two nearly identical patterns, starting with the import token. It would be desired to have them assigned to the same cluster. The only difference of those two statements lies in the lengths of the statement and furthermore only in a series of a repeating pattern. It should clear that these two should be assigned to the same cluster. This is a huge drawback of our implemented approaches. Also we have the same problem as in figure 5.2 above, here in line 3 and 4 in figure 5.3, with statements that only differ in one word.

5.4.1 Discussion

It is possible to learn about key-constructs of the language clustered with StructureFinder. A problem is that there are different centroids with the same patterns as their nearest data point. This means that even though we have a good matching to our manually clustered example – as can be seen in the v-measure scores in section 5.2.1 – the centres of our clusters are not necessarily at the desired position. This limits the number of different constructs we can learn from our nearest statements. Also, since we limit our assumption on the language, we cannot say how much structural constructs are missing from the created list and so we never have a complete overview of the language using the nearest statements of the centroids. Alternatively, we could look at the complete feed of statements per centroid which is also saved. But it is not efficient to sort through the same number of lines of the input.

5.5 Evaluating Language Differentiation

After evaluation the k-means algorithm, we now take a look at the performance of our expansion for calculating a precision value for how close languages are related. For this we calculate a matrix containing all the precision values between pairs of languages. But this does not make it possible to infer parser rules from the nearest statements of the clusters.

Table 5.9: Cross Language Differentiation

languages	Brainfuck	C#	C++	Java	Abacus log	Python	Xml
Brainfuck	0,998	1	0,998	1	0,996	1	1
C#	1	1	1	1	0,997	1	1
C++	0,995	1	0,894	1	0,997	1	1
Java	0,65	0,972	0,81	0,998	0,843	0.94	0.87
Abacus log	0,998	1	1	1	0,999	1	1
Phyton	0,88	1	0,947	1	0,925	1	0.97
Xml	0,52	0,957	0,495	0,89	0,62	0.89	0.88

The results of the Language Differentiator are unfortunately not as expected. It was expected that languages like Java and C# would be similar, because they both are a C type language, as are C++ and Python. But it was not expected that nearly all values calculated, resulted near or are one. This applies also to the ABACUS log file, which is not even a language. It would not suffice just to reference the problems mentioned in section 5.2.2. There must be a general flaw in the idea.

The underlying idea of the Language Differentiator was that if we can cluster statements in a satisfying way, clusters of different statements in a single language would need to be further apart from statements of other clusters than from statements of the same cluster. The conclusion from this is that different languages should result in different clusterings, with different centroids.

Overlap. While it is not possible in k-means that clusters overlap. But it is possible for clusters of two languages to overlap, because they are created independently. If the variance of the data is small, then all data points of the statement representation of the two different languages are located in the same space. Therefore, if this is the case, our way of calculating language differentiation would not be able to find differences between the languages, because they would count as positive findings in our calculations.

This can be visualized in a two dimensional example.

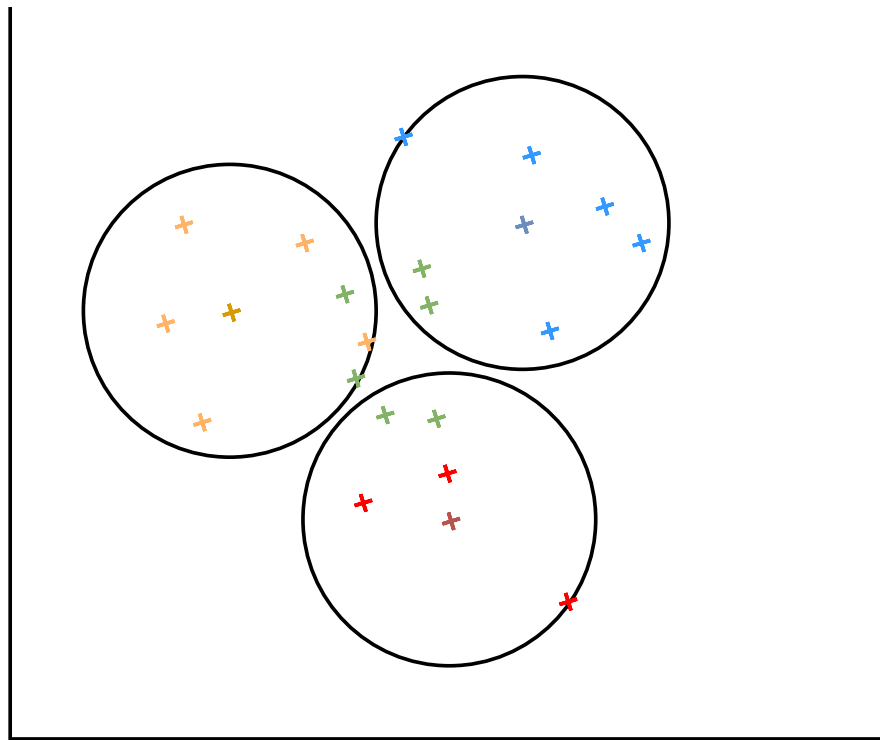


Figure 5.4: Overlap of clusters in Language Differentiator

The orange, blue and red crosses visualize the vector points of the representation of the first language and the black circles are spanned from the radii from the centroids to the point of the cluster with longest distance. The green crosses visualize the same for

the second language. It is clear that these languages are not a close match, because the green crosses are divided to all three clusters created from the first language. But because the variance of the clusters is low, it is possible that all the green crosses lie in the radii of the clusters of the first language. This gives us for this example a precision value of 1, even though if we would span a circle around the green crosses it would intersect with all three circles spanned from the first language. A better approach for language differentiation with k-means could be to calculate a precision value not based on the vector data points, but on the centroids of the clustering of the second language.

6

Conclusions and Future Work

Finding structural patterns in code or log files can help infer grammar rules, as an important part of building a parser. In this thesis we introduce pattern recognition, using the unsupervised learning algorithm k-means and presented ways for representing the data for the used algorithm. Our goal was to minimize assumptions on the input, therefore allowing StructureFinder to work on any given code or log files. We evaluated these methods to see if they perform as expected.

Representations were calculated using different representations, which can be chained together using one Representer of each category, categorized in Type Representers, Distance Representers and Weight Representers.

The evaluation of the representations indicate, that using Type representations without any other representation performs best for almost all. Using Weight Summary representation for noise reduction can further improve the results on some languages, but is not guaranteed to perform better. If we take a better look at the Type Representers, Type Specific does perform slightly better for any tested language.

Apart from the results given by the Type Representers and/or Weight Summary representation, the other representations did not do well in the evaluation, having low scores and were not robust regarding their computations, having high variance in multiple runs and resulting in empty centroids.

If we look at the centroids and the nearest statements located to those centroids, we have an overview of different structures of the clustered language. Unfortunately, despite good v-measure scores it is possible to have statements, which we manually assigned to the same clusters, featured as nearest statements of two different centroids. Nevertheless, key features of the language can be identified, but it is not possible to know what patterns

are not featured in this list of statements. This gives us no complete overview of the language and makes it hard to infer parser rules from the nearest statements.

In this thesis it stands out that k-means for clustering structural patterns in code clearly has limitations, as k-means itself does make assumptions on the data while clustering, as it always tries to find circular shaped clusters.

Our introduced approach for calculating how closely related languages are did not perform well. It resulted almost in every case in a precision value of one, meaning that both languages should be the same.

6.1 Future Work

Regarding the problems of k-means, different unsupervised learning algorithms, using single linkage or hierarchical clustering work better for certain languages, as they make different assumptions regarding their input.

The execution of the k-means implementation is quite expensive for big data sets and in particular, if the vectors contain floating point numbers. Better implementation of vector computations could certainly improve the run time of our tool.

In this thesis, the main focus lies on the different representations and the implementation of the k-means pipeline. The statements were created as single lines, as we wanted to limit our assumptions. Most programming languages have block structures, allowing for encapsulating their statements. The new line approach does not allow for encapsulation. Improvement on the creation of statements could improve the quality of the output of StructureFinder.

With the knowledge about the assumption made by the k-means algorithm, we could also try to make more representations, which could exploit those assumptions better.

It would also be interesting to see how the results could be improved, if we add heuristics for finding keywords. Keywords are a very important construct of programming languages, that cannot be found with our representation approach. Comments or print statements can add different structure after a keyword, but should be assigned to the same cluster. Adding information about keywords, for example as a fourth type for tokens, could improve the clusterings.

For language differentiation the approach could be changed to calculate the precision value based on the centroid of a cluster and not on the statements representation vector points of a language. This could reduce the problem of the general overlap of data with low variance.

List of Tables

5.1	Java v-measures of all possible representations with Type Structure Representer	24
5.2	Java v-measures of all possible representations with Type Specific Representer	24
5.3	C Sharp v-measures	28
5.4	C++ v-measures	29
5.5	Python v-measures	29
5.6	XML v-measures	30
5.7	ABACUS log v-measures	31
5.8	Brainfuck v-measures	31
5.9	Cross Language Differentiation	35

Listings

1	Statements in Java example	6
2	Tokens in statements	6
3	Simplified Java snippet	7
4	Type Structure Representation example as vectors	7
5	Type Specific Representation example	8
6	Type Specific Representation as vectors example	8
7	Type Representation as calculated in Listing 4	8
8	Distance Zero Representation of Type Structure Representation	8
9	Type Representation as calculated in Listing 4	9
10	Distance All Representation vector representation example	9
11	Type Representation as calculated in Listing 4	9
12	Weight Summary Representation vector representation example	9
13	Weight Sum Representation vector representation example	10
14	Type Specific Representation as vectors example	10

List of Figures

3.1	Mathematical expression of k-means [12].	11
3.2	Example of k-means iteration [12].	11
3.3	Average Fit per centroid per k, calculated with StructureFinder using Java file input and TypeStructureRepresenter.	12
3.4	Two dimensional example of Language Differentiation	14
4.1	Simplified data flow chart of the single k k-means pipeline.	16
4.2	Simplified data flow chart of the expanded k-means pipeline, with high- lighted differences to 4.1 in yellow.	17
4.3	Simplified data flow chart of the Language Differentiator pipeline. . . .	18
5.1	Plot of elbow method. Heuristic value of k, given α	22
5.2	Nearest statements per centroids, created witch Type Structure Representer	33
5.3	Nearest statements per centroids, created witch Type Specific Representer	34
5.4	Overlap of clusters in Language Differentiator	36
7.1	Folder structure for setting up the StructureFinder tool, testFiles is the root folder	46
7.2	Pharo 4.0 playground for configuring and running StructureFinder . . .	47
7.3	Output Folders and Files under Windows	49
7.4	Example of "configuration_used.txt" log file.	50
7.5	Set up of perfect example for analysing	51
7.6	Preconfigured StructureAnalyzer Playground	51
7.7	Log file "configuration_used", added performance information from analysis.	52
7.8	Language Differentiator Playground	53
7.9	Language Differentiator compare folder structure	54
7.10	Language Differentiator cross comparison	55

Bibliography

- [1] <http://pharo.org/download>, 2016. [Online; accessed 20-August-2016].
- [2] <http://www.abacus.ch/>, 2016. [Online; accessed 21-August-2016].
- [3] Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. *Deep Into Pharo*. Lulu. com, 2013.
- [4] Rens Bod. An all-subtrees approach to unsupervised parsing. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th Annual Meeting of the Association for Computational Linguistics*, ACL-44, pages 865–872, Stroudsburg, PA, USA, 2006. Association for Computational Linguistics.
- [5] Rens Bod. Unsupervised parsing with u-dop. In *Proceedings of the Tenth Conference on Computational Natural Language Learning*, CoNLL-X '06, pages 85–92, Stroudsburg, PA, USA, 2006. Association for Computational Linguistics.
- [6] Simon Dennis. An exemplar-based approach to unsupervised parsing. In *Proceedings of the 27th Conference of the Cognitive Science Society*, 2005.
- [7] Alpana Dubey, Pankaj Jalote, and Sanjeev Kumar Aggarwal. Inferring grammar rules of programming language dialects. In *International Colloquium on Grammatical Inference*, pages 201–213. Springer, 2006.
- [8] Joël Guggisberg, Oscar Nierstrasz, and Phd Jan Kurš. Automatic token classification. 2015.
- [9] Anil K Jain, Robert P. W. Duin, and Jianchang Mao. Statistical pattern recognition: A review. *IEEE Transactions on pattern analysis and machine intelligence*, 22(1):4–37, 2000.
- [10] Trupti M Kodinariya and Prashant R Makwana. Review on determining number of cluster in k-means clustering. *International Journal*, 1(6):90–95, 2013.
- [11] Thomas D LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, pages 492–501. ACM, 2006.

- [12] Andrew Ng. Cs229 lecture notes.
- [13] Oscar Nierstrasz and Mircea Lungu. Agile software assessment. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 3–10. IEEE, 2012.
- [14] Roi Reichart and Ari Rappoport. Automatic selection of high quality parses created by a fully unsupervised parser. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning*, pages 156–164. Association for Computational Linguistics, 2009.
- [15] David Robinson. K-means clustering is not a free lunch. <http://varianceexplained.org/r/kmeans-free-lunch/>, 2015. [Online; accessed 30-July-2016].
- [16] Andrew Rosenberg and Julia Hirschberg. V-measure: A conditional entropy-based external cluster evaluation measure. In *EMNLP-CoNLL*, volume 7, pages 410–420, 2007.
- [17] Jan Valdman. Log file analysis. *Department of Computer Science and Engineering (FAV UWB).*, *Tech. Rep. DCSE/TR-2001-04*, 2001.

7

Anleitung zu wissenschaftlichen Arbeiten

This chapter contains additional documentation for the StructureFinder tool, implemented in Pharo 4.0 and also consists of a users guide to get the data used in this thesis.

7.1 Introduction

The main effort in this bachelor thesis lies in the implementation of the clustering and representation mechanics. There are three different parts in our finished StructureFinder tool:

- **StructureFinder.** Clusters the given input.
- **StructureAnalyzer.** Analysing mode of StructureFinder, clusters the input and returns additional information about how well it performed.
- **LanguageDifferentiator.** Expansion of StructureFinder, calculates a precision value for two input languages or can additional be used for cross comparison of different languages.

7.2 Getting Started

For replicating the results of this thesis or to play around with the clustering mechanism Pharo 4.0 and the Pharo 4.0 Image File containing the StructureFinder source code is

needed.

Pharo 4.0 with a standard Pharo 4.0 Image can be downloaded from:

https://github.com/countschokula/BA_StructurFinder

The Image containing the source code of this thesis is stored on a git repository and can be cloned using the git command:

```
$ git clone https://github.com/countschokula/BA_StructurFinder
```

7.3 Structure Finder

The StructureFinder is the implemented k-means clustering tool. For the execution, a folder structure needs to be created, containing the input files and folders for the output files. For proper functioning, the folder structure needs to look exactly like this. This needs to be manually created. If a new language is added, folders must be created in the folders files and result. "testFiles" is the root folder. In an input Folder there can be more than one file present. If that is the case, StructureFinder will concatenate them to one file in the clustering process.

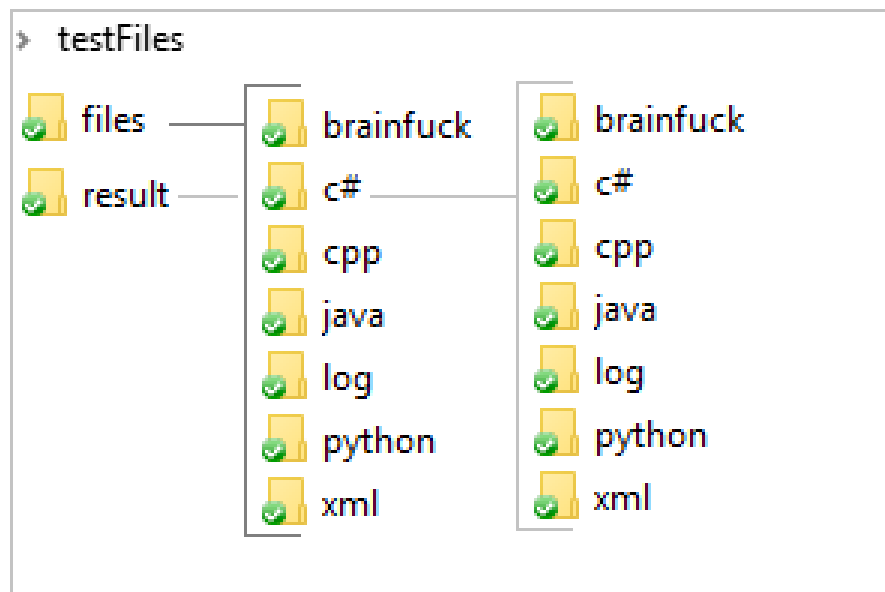
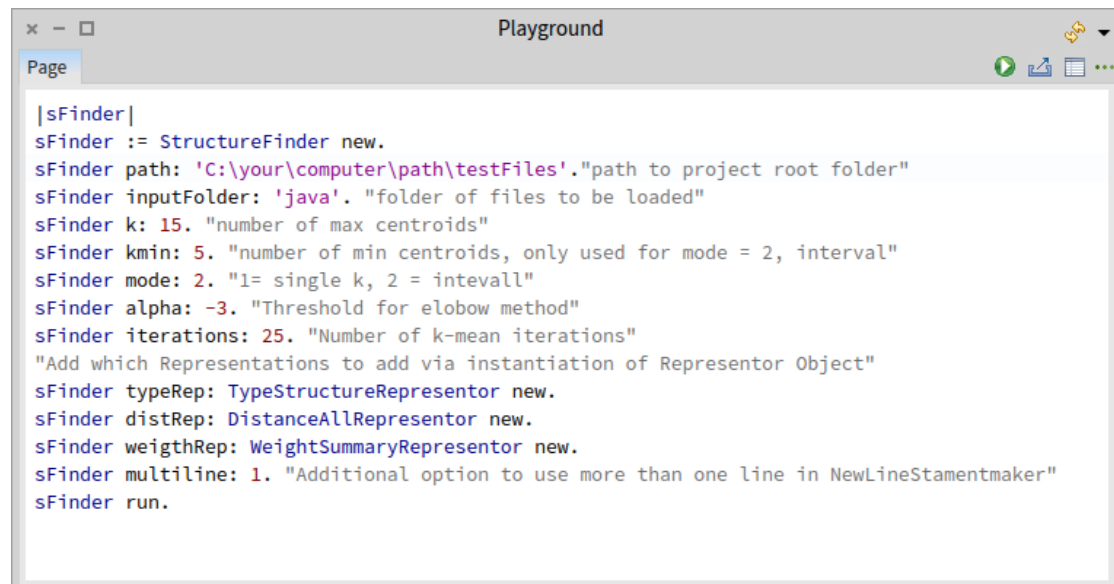


Figure 7.1: Folder structure for setting up the StructureFinder tool, testFiles is the root folder

In the uploaded Image there should be preconfigured playgrounds. If there is no playground open, one needs to be opened.

The preconfigured playground for the StructureFinder contains all necessary parameters for running the clustering algorithm:



```

|sFinder|
sFinder := StructureFinder new.
sFinder path: 'C:\your\computer\path\testFiles'."path to project root folder"
sFinder inputFolder: 'java'."folder of files to be loaded"
sFinder k: 15."number of max centroids"
sFinder kmin: 5."number of min centroids, only used for mode = 2, interval"
sFinder mode: 2."1= single k, 2 = intervall"
sFinder alpha: -3."Threshold for elobow method"
sFinder iterations: 25."Number of k-mean iterations"
"Add which Representations to add via instantiation of Representor Object"
sFinder typeRep: TypeStructureRepresentor new.
sFinder distRep: DistanceAllRepresentor new.
sFinder weigthRep: WeightSummaryRepresentor new.
sFinder multiline: 1."Additional option to use more than one line in NewLineStamentmaker"
sFinder run.
  
```

Figure 7.2: Pharo 4.0 playground for configuring and running StructureFinder

For better understanding, the preconfigured playground contains information on the parameters that can be set. We now take a closer look at the parameters of the StructureFinder.

- **path.** This parameter is necessary for correct loading and storing of the input respectively the output files. The path should point to the root of the folder structure as defined above.
- **mode** It is possible to run two different modes of StructureFinder. Parameter "1" creates a single run of the clustering for input k, while parameter "2" runs the clustering for every natural number between k2 and k.
- **inputFolder.** It is possible to have different folders, containing different languages in your project root folder. The message "inputFolder:" makes the tool use the input of the folder. The folder is given as a string. Output is saved in the corresponding folder under results.
- **k.** Number of clusters for single clustering. If interval mode is activated, k is the higher limit of the interval.

- **kmin.** If interval mode is activated, kmin is the lower limit of the interval.
- **iterations** Gives the kMeansRunner the number of iterations it should run the k-means algorithm, 25 is a low, fast and reasonable guess for the number of iteration. If no iteration value is set, the default value is 50.
- α . Defines the threshold for the elbow method, the heuristic for guessing k. This parameter is only used, if the tool is run on an interval.
- **typeRep.** A Type Representer must be set. Representers are given to the StructureFinder as Instances. There are two Type Representers available. TypeStructure and TypeSpecific.
- **distRep.** A Distance Representer is optional. It can be set by giving the StructureFinder an Instance of the Distance Representer you want to use. Currently there are two Distance Representers available. DistanceZero- and DistanceAllRepresenter.
- **weightRep.** Adding a Weight Representer is optional. Currently there are three different Weight Representers available. WeightSummary-, WeightSum- First and WeightInverseRepresenter.

Output. The statements of a cluster are saved to .txt files. A file per centroid is created and the files are enumerated. They are saved under results in the corresponding folder the input file is located and are put in a folder named with the timestamp of the execution of the algorithm. For interval execution, for each created clustering the result will be saved:

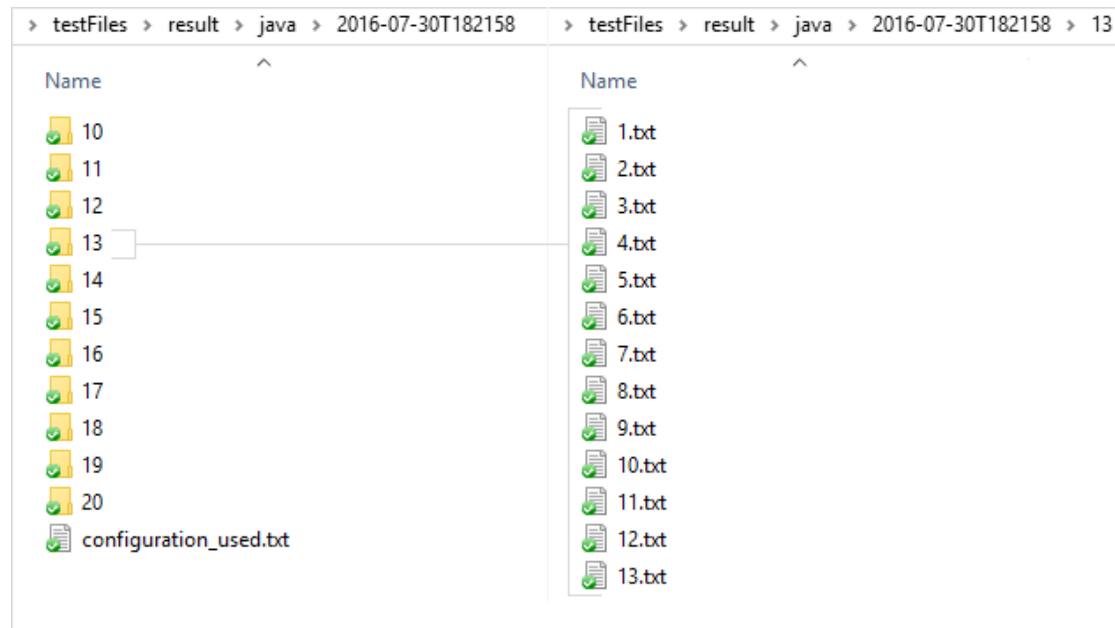


Figure 7.3: Output Folders and Files under Windows

As seen in Figure 7.3 a file named `configuration_used` will be created. It contains the parameters used in calculating the clusters, as well as the value of the distortion functions averaged over k , for all sets of clusters. Also, the nearest statements to the centroids are listed.

```

1 Folder: java
2
3 Representors:
4 Type Structure Representor
5 Distance All Small Representor
6 Weight Inverse Representor
7
8 Distortion Function values:
9 105.097198154773
10 107.85233046512776
11 73.97848952007175
12 61.20784924288378
13 51.41051669574161
14 50.22390051699675
15 40.1699284252772
16 34.6681874520477
17 31.112434434379516
18 32.90443814492816
19 26.4433594020361
20
21 Guess for K: 13
22
23 nearest Statements per centroid:
24 default :
25 PendingIntent . FLAG _ CANCEL _ CURRENT ) ;
26 if ( daily != null ) {
27 menuItem = item ;
28 public class CriteriaMatcher {
29 protected static final String MENSA _ ID _ KEY = " ch . xonix . mensa . unibe : : mensa _ ID " ;
30 showMessage . setText ( R . string . not _ logged _ in ) ;
31 public void onReceive ( Context context , Intent intent ) {
32 for ( Menu menu : daily . getMenus ( ) ) {
33 import ch . xonix . mensa . unibe . model . Mensa ;
34 * predefined values made in the Settings , if there are now settings made ,
35 crit . setCriteriaName ( criteria ) ;
36 public class BootReceiver extends BroadcastReceiver {

```

Figure 7.4: Example of "configuration_used.txt" log file.

7.4 Structure Analyzer

Structure Analyzer allows for calculating the v-measure scores. V-measure is explained in detail in section 5.1.1.

For calculating v-measures, it is necessary to have a manually created clustering of your input data to test against. The input file needs to be split into the clusterings wanted to be achieved and saved into different text files. The assigned statements must be only featured once in the manually created clusters. Also, the cluster set must be stored in the correct folder. For this a folder named "perfect" needs to be created in the root folder of your project path. In the folder "perfect" a folder with the same name as the input folder has to be created and this is where the manually assigned clusters need to be stored. The input of the files has to be on the same location used for the StructureFinder.

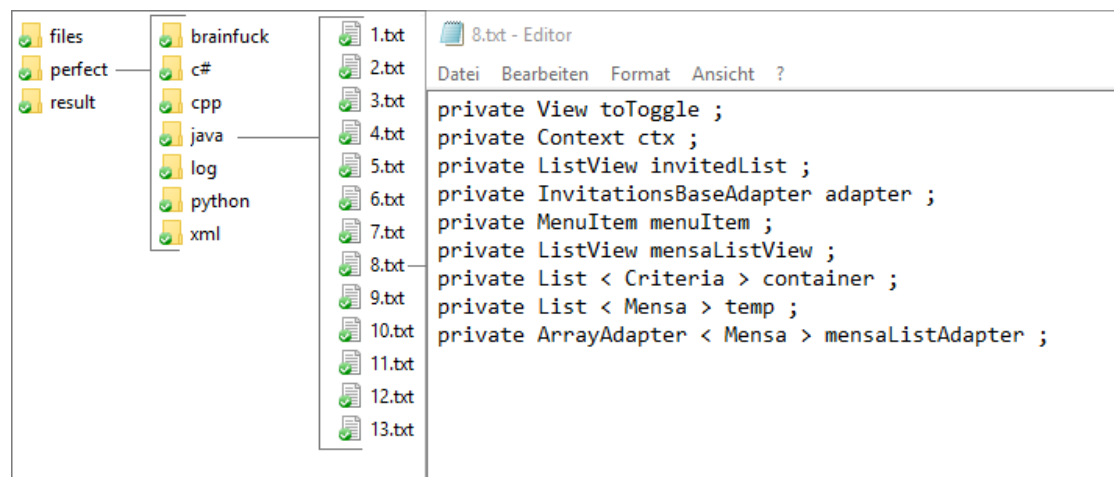


Figure 7.5: Set up of perfect example for analysing

The configuration is exactly the same as in section 7.3 for the StructureFinder.

The preconfigured playground for the StructureAnalyzer looks like this:

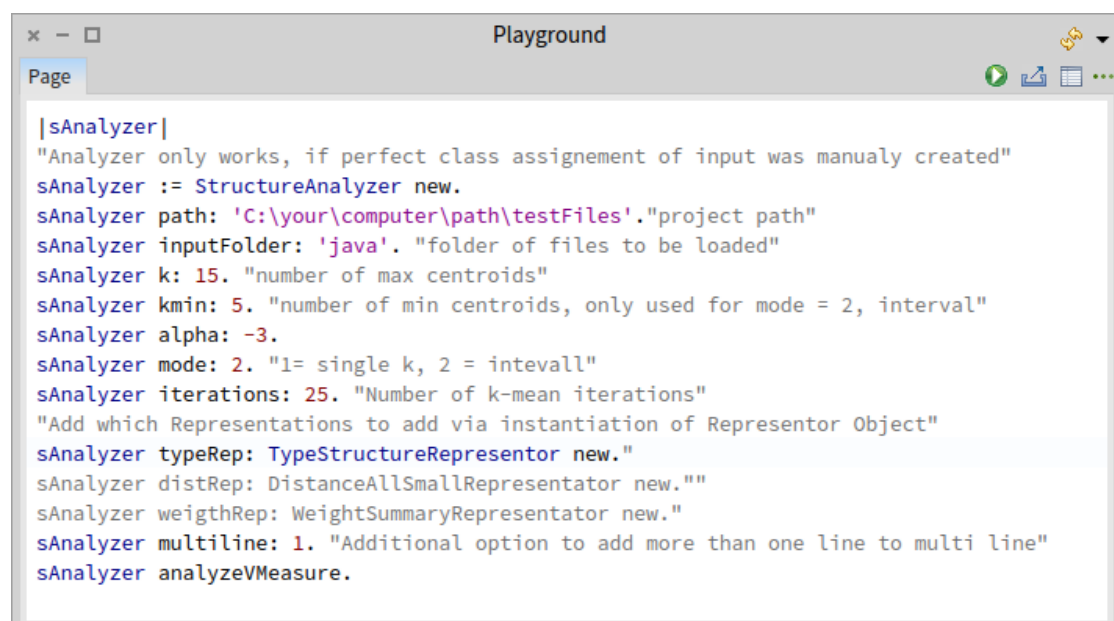


Figure 7.6: Preconfigured StructureAnalyzer Playground

Output The StructureAnalyzer has the same output as the StructureFinder in section 7.3. The exception is, that the additional information about the performance of the clusterings are added to the log file.

```
20 homogeneity: 0.5781070081099207  
21 completeness: 0.9104945606931953  
22 v-measure: 0.70719163195013
```

Figure 7.7: Log file "configuration_used", added performance information from analysis.

7.5 Language Differentiator

There are two different ways for calculating the Language Differentiator values. The first, calculates the precision value of how close two languages are related, using two normal input files from the files directory, as described in section 7.3. The second approach, which was used for calculating the results in this thesis, as can be found in section 7.3, uses a different test and trainings set per language, allowing to test the relation of two different input sets for the same language.

We now take a look at the first approach and the preconfigured Pharo playground.

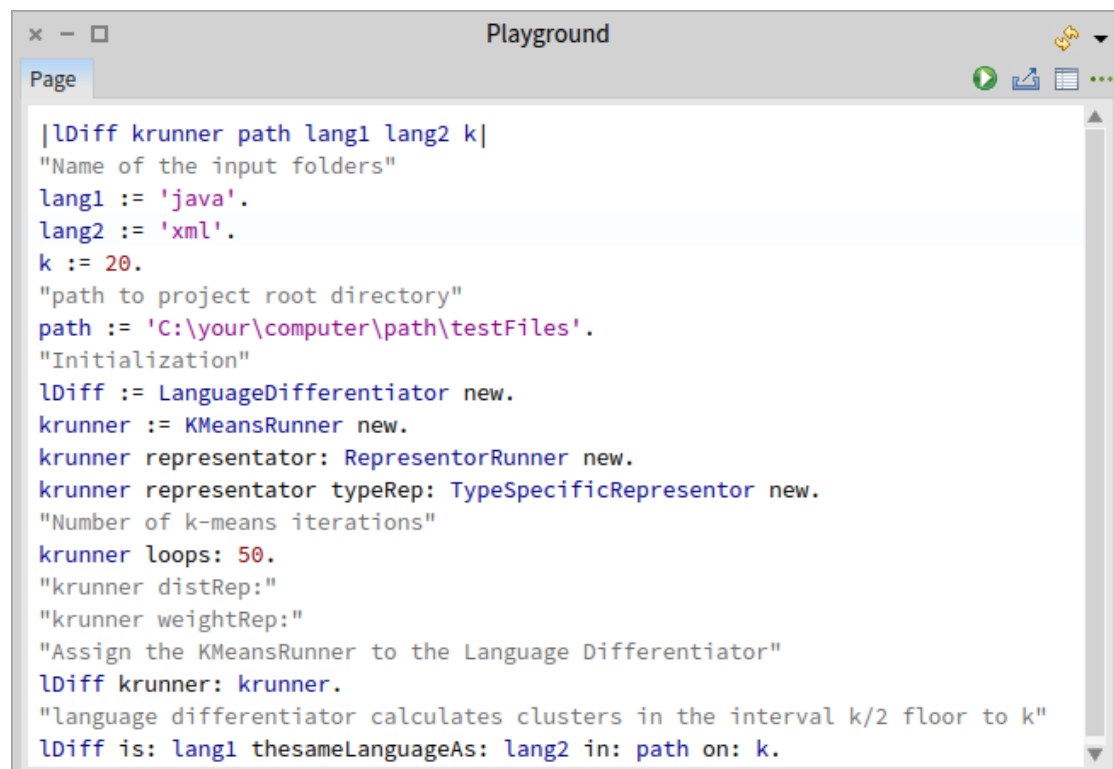


Figure 7.8: Language Differentiator Playground

The initialization of the LanguageDifferentiator works nearly the same as for the StructureFinder and the StructureAnalyzer. The exception is, that the kMeansRunner Object is not wrapped in the LanguageDifferentiator and must be initialised separately.

Output. For the playground to show the results, press Ctrl+A and Ctrl+P. The clusters are saved in the normal result folder.

The second approach needs a separate folder structure:

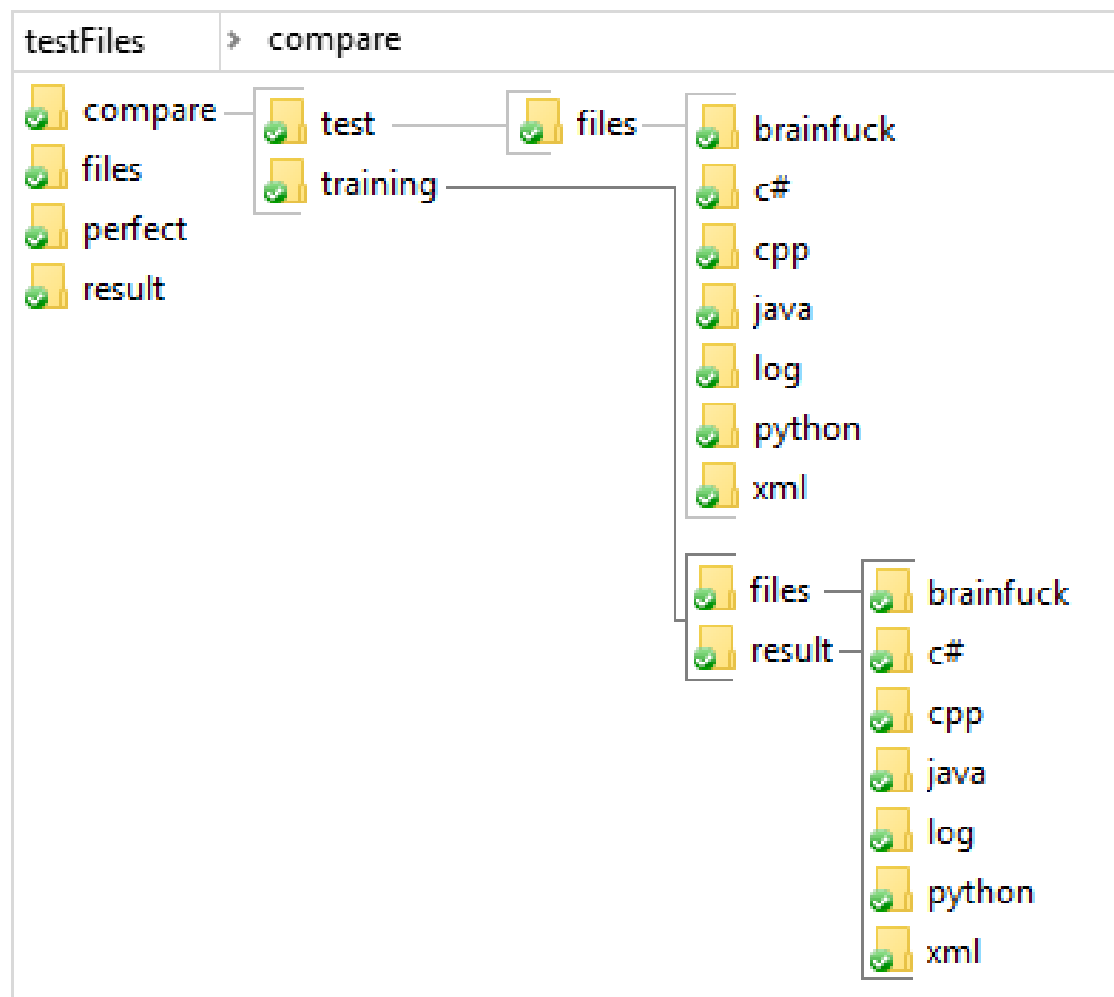
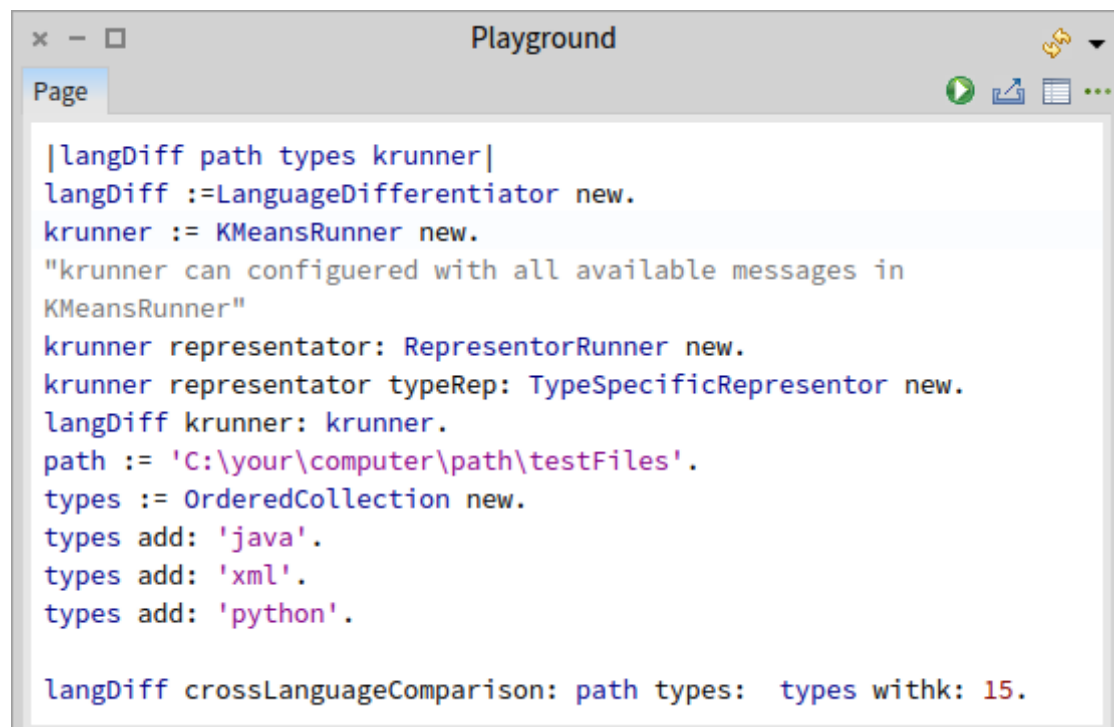


Figure 7.9: Language Differentiator compare folder structure

There is also a preconfigured playground for the "crossLanguageComparison:"

The image shows a screenshot of a software window titled "Playground". The window has a standard macOS-style title bar with a close button (X), a minimize button (-), and a maximize button (square). Below the title bar, there is a tab labeled "Page". The main area of the window contains a text editor with Scala code. The code is as follows:

```
|langDiff path types krunner|
langDiff := LanguageDifferentiator new.
krunner := KMeansRunner new.
"Krunner can be configured with all available messages in
KMeansRunner"
krunner representator: RepresentorRunner new.
krunner representator typeRep: TypeSpecificRepresentor new.
langDiff krunner: krunner.
path := 'C:\your\computer\path\testFiles'.
types := OrderedCollection new.
types add: 'java'.
types add: 'xml'.
types add: 'python'.

langDiff crossLanguageComparison: path types: types withk: 15.
```

The code is color-coded: keywords like "new.", "add:", and "withk:" are in red; strings are in purple; and other identifiers are in blue. The window also features a toolbar on the right side with icons for running (a green play button), saving (a floppy disk), and other functions.

Figure 7.10: Language Differentiator cross comparison

The KMeansRunner is configured in the same way as above. All the folders that we want to compare with each other can be added in string form to the types collection with the message "add:".

Output. Output is generated using the playground command Ctrl+P. Results of the clusterings are saved to the results folder in compare>training>result.