Comparison of Clang Abstract Syntax Trees using String Kernels
HPCS 2018, Orleans, France

Raul Torres, Julian M. Kunkel, Manuel F. Dolz, Thomas Ludwig
1. Motivation

2. Background
   - Intermediate representations
   - String kernels

3. Proposed solution
   - Creating strings from ASTs
   - Finding similarities with a novel string kernel

4. Evaluation
   - Experiment configuration
   - Blended spectrum kernel
   - Kast spectrum kernel
   - Kast1 spectrum kernel

5. Conclusions and future work
1. Motivation
Motivation

Code similarity

- Computer programs exhibit similarities that can be detected before, during, and after execution time.
Motivation

Code similarity

- Computer programs exhibit similarities that can be detected before, during, and after execution time.
- Programs that are similar tend to behave in similar manner too.
Motivation

Code similarity

- Computer programs exhibit similarities that can be detected before, during, and after execution time.
- Programs that are similar tend to behave in similar manner too.
- Detecting program similarities may have the following applications:
Motivation

Code similarity

- Computer programs exhibit similarities that can be detected before, during, and after execution time.
- Programs that are similar tend to behave in similar manner too.
- Detecting program similarities may have the following applications:
  - Analyze and improve the overall performance of a set of similar programs.
Motivation

Code similarity

• Computer programs exhibit similarities that can be detected before, during, and after execution time.
• Programs that are similar tend to behave in similar manner too.
• Detecting program similarities may have the following applications:
  – Analyze and improve the overall performance of a set of similar programs.
  – Assist the programmer in finding code that is already implemented instead of coding from scratch.
Motivation

Code similarity

• Computer programs exhibit similarities that can be detected before, during, and after execution time.
• Programs that are similar tend to behave in similar manner too.
• Detecting program similarities may have the following applications:
  – Analyze and improve the overall performance of a set of similar programs.
  – Assist the programmer in finding code that is already implemented instead of coding from scratch.
  – Find common mistakes at designing or writing programs: code smells.
Motivation

Code similarity

- Computer programs exhibit similarities that can be detected before, during, and after execution time.
- Programs that are similar tend to behave in similar manner too.
- Detecting program similarities may have the following applications:
  - Analyze and improve the overall performance of a set of similar programs.
  - Assist the programmer in finding code that is already implemented instead of coding from scratch.
  - Find common mistakes at designing or writing programs: code smells.
  - Detect plagiarism.
The similarity between two programs can be determined by the amount of clones they share. There are four distinct types of them:

• Type-1: this type of clones stand for pieces of code containing differences in the layout, spaces and comments.
• Type-2: these codes also present differences in data types and identifiers.
• Type-3: these clone types also include additions, modifications and deletions of lines of code.
• Type-4: represent codes that present different implementation but the same functionality.
The similarity between two programs can be determined by the amount of clones they share. There are four distinct types of them:

- **Type-1**: this type of clones stand for pieces of code containing differences in the layout, spaces and comments.

- **Type-2**: these codes also present differences in data types and identifiers.

- **Type-3**: these clone types also include additions, modifications and deletions of lines of code.

- **Type-4**: represent codes that present different implementation but the same functionality.
The similarity between two programs can be determined by the amount of clones they share. There are four distinct types of them:

- **Type-1**: this type of clones stand for pieces of code containing differences in the layout, spaces and comments.
- **Type-2**: these codes also present differences in data types and identifiers.
The similarity between two programs can be determined by the amount of clones they share. There are four distinct types of them:

- **Type-1**: this type of clones stand for pieces of code containing differences in the layout, spaces and comments.
- **Type-2**: these codes also present differences in data types and identifiers.
- **Type-3**: these clone types also include additions, modifications and deletions of lines of code.
Motivation

Code clones

The similarity between two programs can be determined by the amount of clones they share. There are four distinct types of them:

- **Type-1**: this type of clones stand for pieces of code containing differences in the layout, spaces and comments.
- **Type-2**: these codes also present differences in data types and identifiers.
- **Type-3**: these clone types also include additions, modifications and deletions of lines of code.
- **Type-4**: represent codes that present different implementation but the same functionality.
The similarity between two programs can be determined by the amount of clones they share. There are four distinct types of them:

- **Type-1**: this type of clones stand for pieces of code containing differences in the layout, spaces and comments.
- **Type-2**: these codes also present differences in data types and identifiers.
- **Type-3**: these clone types also include additions, modifications and deletions of lines of code.
- **Type-4**: represent codes that present different implementation but the same functionality.
The similarity between two programs can be determined by the amount of clones they share. There are four distinct types of them:

- **Type-1**: this type of clones stand for pieces of code containing differences in the layout, spaces and comments.
- **Type-2**: these codes also present differences in data types and identifiers.
- **Type-3**: these clone types also include additions, modifications and deletions of lines of code.
- **Type-4**: represent codes that present different implementation but the same functionality.

2. Background
Intermediate representations (IRs)

Generalities

- The IR of a program is the central data structure in a compiler. Around it, analysis, transformations and optimizations are performed.
Intermediate representations (IRs)

Generalities

- The IR of a program is the central data structure in a compiler. Around it, analysis, transformations and optimizations are performed.
- Broad categories:
Intermediate representations (IRs)

Generalities

- The IR of a program is the central data structure in a compiler. Around it, analysis, transformations and optimizations are performed.

- Broad categories:
  - *Graphical IRs*: This IR type stores the program information in a graph-like data structure.
Intermediate representations (IRs)

Generalities

- The IR of a program is the central data structure in a compiler. Around it, analysis, transformations and optimizations are performed.

- Broad categories:
  - *Graphical IRs*: This IR type stores the program information in a graph-like data structure.
  - *Linear IRs*: This IR type makes use of simple linear sequences to store operations, similar to machine code.
Intermediate representations (IRs)

Generalities

• The IR of a program is the central data structure in a compiler. Around it, analysis, transformations and optimizations are performed.

• Broad categories:
  – Graphical IRs: This IR type stores the program information in a graph-like data structure.
  – Linear IRs: This IR type makes use of simple linear sequences to store operations, similar to machine code.
  – Hybrid IRs: Hybrid IRs combine elements of the previous two categories.
Intermediate representations (IRs)

Generalities

• The IR of a program is the central data structure in a compiler. Around it, analysis, transformations and optimizations are performed.

• Broad categories:
  – *Graphical IRs*: This IR type stores the program information in a graph-like data structure.
  – *Linear IRs*: This IR type makes use of simple linear sequences to store operations, similar to machine code.
  – *Hybrid IRs*: Hybrid IRs combine elements of the previous two categories.

• Complex compiler infrastructures might work with different interconnected IRs, some of them closer to the source code, others closer to the machine instruction level.
Intermediate representations (IRs)
Abstract syntax trees (ASTs)

• ASTs are a graph-based intermediate representation.
Intermediate representations (IRs)

Abstract syntax trees (ASTs)

- ASTs are a graph-based intermediate representation.
- They are defined as contractions of parse trees.
Intermediate representations (IRs)

Abstract syntax trees (ASTs)

- ASTs are a graph-based intermediate representation.
- They are defined as contractions of parse trees.
- Most non-terminal symbols are ignored.
Intermediate representations (IRs)

Abstract syntax trees (ASTs)

- ASTs are a graph-based intermediate representation.
- They are defined as contractions of parse trees.
- Most non-terminal symbols are ignored.
- The precedence and the meaning of the expressions are preserved.
Intermediate representations (IRs)

Abstract syntax trees (ASTs)

- ASTs are a graph-based intermediate representation.
- They are defined as contractions of parse trees.
- Most non-terminal symbols are ignored.
- The precedence and the meaning of the expressions are preserved.
- Their level of abstraction is not far from the original source code.
Intermediate representations (IRs)

Abstract syntax trees (ASTs)

- ASTs are a graph-based intermediate representation.
- They are defined as contractions of parse trees.
- Most non-terminal symbols are ignored.
- The precedence and the meaning of the expressions are preserved.
- Their level of abstraction is not far from the original source code.
- In this work we used the AST from Clang, a LLVM frontend for C/C++/Objective C programs.
Intermediate representations (IRs)

Abstract syntax trees (ASTs)

- ASTs are a graph-based intermediate representation.
- They are defined as contractions of parse trees.
- Most non-terminal symbols are ignored.
- The precedence and the meaning of the expressions are preserved.
- Their level of abstraction is not far from the original source code.
- In this work we used the AST from Clang, a LLVM frontend for C/C++/Objective C programs.
Intermediate representations (IRs)

Abstract syntax trees (ASTs)

- ASTs are a graph-based intermediate representation.
- They are defined as contractions of parse trees.
- Most non-terminal symbols are ignored.
- The precedence and the meaning of the expressions are preserved.
- Their level of abstraction is not far from the original source code.
- In this work we used the AST from Clang, a LLVM frontend for C/C++/Objective C programs.

How to compare these data structures? direct tree comparison? flatten into strings? extract attribute set?
String kernels

Generalities

• Strings are a common and useful form of representing data (e.g. DNA sequences).
String kernels

Generalities

- Strings are a common and useful form of representing data (e.g. DNA sequences).
- String kernels can be intuitively understood as functions measuring the similarity of pairs of strings.
Strings are a common and useful form of representing data (e.g. DNA sequences).

String kernels can be intuitively understood as functions measuring the similarity of pairs of strings.

The more similar two strings $A$ and $B$ are, the higher the value of a string kernel $K(A, B)$ will be.
String kernels

Generalities

- Strings are a common and useful form of representing data (e.g. DNA sequences).
- String kernels can be intuitively understood as functions measuring the similarity of pairs of strings.
- The more similar two strings $A$ and $B$ are, the higher the value of a string kernel $K(A, B)$ will be.
- In particular, string kernels check the number of shared substrings among a collection of strings.
Some kernel functions have been proposed:

- The bag-of-characters kernel: Performs single-character matching.
- The bag-of-words kernel: Searches for shared words among strings.
- The $k$-spectrum kernel: Counts only sub-strings of length $k$.
- The $k$-blended spectrum kernel: It includes all strings whose length is minor than or equal to a given number $k$. 

What is our contribution?
Some kernel functions have been proposed:

- The bag-of-characters kernel:
- The bag-of-words kernel:
- The $k$-spectrum kernel:
- The $k$-blended spectrum kernel:
String kernels

Examples

Some kernel functions have been proposed:

- **The bag-of-characters kernel:**
  - Performs single-character matching.

- **The bag-of-words kernel:**
  - Searches for shared words among strings.

- **The $k$-spectrum kernel:**
  - Counts only sub-strings of length $k$.

- **The $k$-blended spectrum kernel:**
  - It includes all strings whose length is minor than or equal to a given number $k$. 

What is our contribution?
Some kernel functions have been proposed:

- **The bag-of-characters kernel:**
  - Performs single-character matching.

- **The bag-of-words kernel:**

- **The k-spectrum kernel:**

- **The k-blended spectrum kernel:**
  - It includes all strings whose length is minor than or equal to a given number $k$. 

What is our contribution?
Some kernel functions have been proposed:

- **The bag-of-characters kernel:**
  - Performs single-character matching.

- **The bag-of-words kernel:**
  - Searches for shared words among strings.
String kernels

Examples

Some kernel functions have been proposed:

- **The bag-of-characters kernel:**
  - Performs single-character matching.

- **The bag-of-words kernel:**
  - Searches for shared words among strings.

- **The $k$-spectrum kernel:**
String kernels

Examples

Some kernel functions have been proposed:

- **The bag-of-characters kernel:**
  - Performs single-character matching.

- **The bag-of-words kernel:**
  - Searches for shared words among strings.

- **The $k$-spectrum kernel:**
  - Counts only sub-strings of length $k$. 
String kernels

Examples

Some kernel functions have been proposed:

- **The bag-of-characters kernel:**
  - Performs single-character matching.

- **The bag-of-words kernel:**
  - Searches for shared words among strings.

- **The $k$-spectrum kernel:**
  - Counts only sub-strings of length $k$.

- **The $k$-blended spectrum kernel:**
String kernels

Examples

Some kernel functions have been proposed:

- **The bag-of-characters kernel:**
  - Performs single-character matching.

- **The bag-of-words kernel:**
  - Searches for shared words among strings.

- **The $k$-spectrum kernel:**
  - Counts only sub-strings of length $k$.

- **The $k$-blended spectrum kernel:**
  - It includes all strings whose length is minor than or equal to a given number $k$. 
String kernels

Examples

Some kernel functions have been proposed:

- **The bag-of-characters kernel:**
  - Performs single-character matching.

- **The bag-of-words kernel:**
  - Searches for shared words among strings.

- **The $k$-spectrum kernel:**
  - Counts only sub-strings of length $k$.

- **The $k$-blended spectrum kernel:**
  - It includes all strings whose length is minor than or equal to a given number $k$. 
String kernels
Examples

Some kernel functions have been proposed:

- **The bag-of-characters kernel:**
  - Performs single-character matching.

- **The bag-of-words kernel:**
  - Searches for shared words among strings.

- **The $k$-spectrum kernel:**
  - Counts only sub-strings of length $k$.

- **The $k$-blended spectrum kernel:**
  - It includes all strings whose length is minor than or equal to a given number $k$.

What is our contribution?
3. Proposed solution
Overview

1. Convert the trees into weighted strings.
Overview

1. Convert the trees into weighted strings.
2. Compress the strings in order to save some space.
Overview

1. Convert the trees into weighted strings.
2. Compress the strings in order to save some space.
3. Obtain a similarity matrix a.k.a. kernel matrix using a novel string kernel function.
Overview

1. Convert the trees into weighted strings.
2. Compress the strings in order to save some space.
3. Obtain a similarity matrix a.k.a kernel matrix using a novel string kernel function.
4. Use a clustering algorithm to extract knowledge from the kernel matrix.
Overview

1. Convert the trees into weighted strings.
2. Compress the strings in order to save some space.
3. Obtain a similarity matrix a.k.a kernel matrix using a novel string kernel function.
4. Use a clustering algorithm to extract knowledge from the kernel matrix.

This work extends previous research from the authors, where they proposed a string kernel for the detection of patterns in I/O traces: "A novel string representation and kernel function for the comparison of I/O access patterns," in Parallel Computing Technologies.
Overview

1. Convert the trees into weighted strings.
2. Compress the strings in order to save some space.
3. Obtain a similarity matrix a.k.a kernel matrix using a novel string kernel function.
4. Use a clustering algorithm to extract knowledge from the kernel matrix.

This work extends previous research from the authors, where they proposed a string kernel for the detection of patterns in I/O traces:

From trees to strings

a) AST.

b) Extracted tokens.

<table>
<thead>
<tr>
<th>Tokens</th>
<th>Repetitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>[IfStmt]</td>
<td>1</td>
</tr>
<tr>
<td>[Binary Operator]</td>
<td>1</td>
</tr>
<tr>
<td>[Binary Operator]</td>
<td>1</td>
</tr>
<tr>
<td>[ImplicitCastExpr]</td>
<td>1</td>
</tr>
<tr>
<td>[LEVEL_UP]</td>
<td>2</td>
</tr>
<tr>
<td>[Binary Operator]</td>
<td>1</td>
</tr>
<tr>
<td>[ImplicitCastExpr]</td>
<td>1</td>
</tr>
<tr>
<td>[LEVEL_UP]</td>
<td>3</td>
</tr>
<tr>
<td>[Binary Operator]</td>
<td>1</td>
</tr>
<tr>
<td>[ImplicitCastExpr]</td>
<td>1</td>
</tr>
<tr>
<td>[DeclRefExpr]</td>
<td>1</td>
</tr>
<tr>
<td>[LEVEL_UP]</td>
<td>1</td>
</tr>
<tr>
<td>[DeclRefExpr]</td>
<td>1</td>
</tr>
<tr>
<td>[LEVEL_UP]</td>
<td>3</td>
</tr>
<tr>
<td>[CompoundStmt]</td>
<td>1</td>
</tr>
<tr>
<td>[ReturnStmt]</td>
<td>1</td>
</tr>
<tr>
<td>[IntegerLiteral]</td>
<td>1</td>
</tr>
<tr>
<td>[LEVEL_UP]</td>
<td>4</td>
</tr>
</tbody>
</table>
Compression of the string

1. Similar consecutive tokens:
Compression of the string

1. Similar consecutive tokens:
   - $[\text{BinaryOp}]_1 [\text{BinaryOp}]_1 [\text{BinaryOp}]_1 [\text{BinaryOp}]_1$
     \[ \downarrow \]
     $[\text{BinaryOperator}]_4$
Compression of the string

1. Similar consecutive tokens:
   - $[\text{BinaryOp}]_1[\text{BinaryOp}]_1[\text{BinaryOp}]_1[\text{BinaryOp}]_1$
     \[ \downarrow \]
     $[\text{BinaryOperator}]_4$

2. Delete specific tokens:
1. **Similar consecutive tokens:**
   - \([\text{BinaryOp}]_1 [\text{BinaryOp}]_1 [\text{BinaryOp}]_1 [\text{BinaryOp}]_1\)
     \[\downarrow\]
     \([\text{BinaryOperator}]_4\)

2. **Delete specific tokens:**
   - \([\text{CStyleCastExpr}]_1 [\text{CallExpr}]_1 [\text{ImplicitCastExpr}]_1 [\text{DeclRefExpr}]_1\)
     \[\downarrow\]
     \([\text{DeclRefExpr}]_4\)
Compression of the string

1. Similar consecutive tokens:
   - \([\text{BinaryOp}]_1[\text{BinaryOp}]_1[\text{BinaryOp}]_1[\text{BinaryOp}]_1\)
     \[\downarrow\]
     \[\text{BinaryOperator}]_4\]

2. Delete specific tokens:
   - \([\text{CStyleCastExpr}]_1[\text{CallExpr}]_1[\text{ImplicitCastExpr}]_1[\text{DeclRefExpr}]_1\)
     \[\downarrow\]
     \[\text{DeclRefExpr}]_4\]

3. Simplify declaration tokens:
Compression of the string

1. Similar consecutive tokens:
     \downarrow
     [BinaryOperator]_4

2. Delete specific tokens:
   - `[CStyleCastExpr]_1[CallExpr]_1[ImplicitCastExpr]_1
     [DeclRefExpr]_1
     \downarrow
     [DeclRefExpr]_4

3. Simplify declaration tokens:
   - `[DeclStmt]_1[VarDecl]_1[DeclRefExpr]_1[LEVEL_UP]_4
     \downarrow
     [DeclStmt]_3[LEVEL_UP]_4
Compression of the string

1. Similar consecutive tokens:
   - \([\text{BinaryOp}]_1 [\text{BinaryOp}]_1 [\text{BinaryOp}]_1 [\text{BinaryOp}]_1\)
   \[\downarrow\]
   \([\text{BinaryOperator}]_4\)

2. Delete specific tokens:
   - \([\text{CStyleCastExpr}]_1 [\text{CallExpr}]_1 [\text{ImplicitCastExpr}]_1 [\text{DeclRefExpr}]_1\)
   \[\downarrow\]
   \([\text{DeclRefExpr}]_4\)

3. Simplify declaration tokens:
   - \([\text{DeclStmt}]_1 [\text{VarDecl}]_1 [\text{DeclRefExpr}]_1 [\text{LEVEL\_UP}]_4\)
   \[\downarrow\]
   \([\text{DeclStmt}]_3 [\text{LEVEL\_UP}]_4\)

4. Compress in pairs:
1. Similar consecutive tokens:
     ↓
     [BinaryOperator]_4

2. Delete specific tokens:
   - `[CStyleCastExpr]_1 [CallExpr]_1 [ImplicitCastExpr]_1
     [DeclRefExpr]_1
     ↓
     [DeclRefExpr]_4

3. Simplify declaration tokens:
   - `[DeclStmt]_1 [VarDecl]_1 [DeclRefExpr]_1 [LEVEL_UP]_4
     ↓
     [DeclStmt]_3 [LEVEL_UP]_4

4. Compress in pairs:
   - `[IntegerLiteral]_1 [LEVEL_UP]_5 [IntegerLiteral]_1 [LEVEL_UP]_2
     ↓
     [IntegerLiteral]_2 [LEVEL_UP]_7
KastX spectrum kernel family

Definitions

Having two strings $A$ and $B$:
KastX spectrum kernel family

Definitions

Having two strings $A$ and $B$:

1. The algorithm requires a minimum weight or “cut weight” value as parameter.
KastX spectrum kernel family

Definitions

Having two strings $A$ and $B$:

1. The algorithm requires a minimum weight or “cut weight” value as parameter.

2. The aim is to find the longest matching substrings of $A$ and $B$, whose weights are greater than or equal to the cut weight. They are called *valid matching substrings*. 

Having two strings $A$ and $B$:

1. The algorithm requires a minimum weight or “cut weight” value as parameter.
2. The aim is to find the longest matching substrings of $A$ and $B$, whose weights are greater than or equal to the cut weight. They are called *valid matching substrings*.
3. A valid matching substring can appear more than once in each string.
KastX spectrum kernel family

Definitions

Having two strings $A$ and $B$:

1. The algorithm requires a minimum weight or “cut weight” value as parameter.
2. The aim is to find the longest matching substrings of $A$ and $B$, whose weights are greater than or equal to the cut weight. They are called *valid matching substrings*.
3. A valid matching substring can appear more than once in each string.
4. A valid matching substring must not be a substring of another valid matching substring in at least one of the original strings.
KastX spectrum kernel family
An example with cut weight = 4

a) $S_1$ is the largest substring found on both examples.

\[
A_{64} = a_3 \ b_2 \ c_4 \ d_2 \ e_1 \ f_5 \ g_1 \ h_1 \ i_1 \ j_2 \ k_1 \ l_3 \ m_1 \ n_2 \ f_3 \ g_1 \ h_2 \ o_1 \ p_1 \ q_1 \ r_2 \ s_1 \ t_5 \ u_9 \ b_7 \ c_2
\]

\[
B_{52} = v_2 \ a_5 \ b_1 \ c_1 \ d_3 \ e_1 \ f_4 \ g_1 \ h_1 \ w_2 \ x_2 \ y_1 \ a_1 \ b_2 \ c_6 \ d_1 \ e_3 \ f_1 \ g_1 \ h_3 \ z_1 \ b_5 \ c_1 \ f_1 \ g_1 \ h_1
\]
KastX spectrum kernel family

An example with cut weight = 4

a) $S_1$ is the largest substring found on both examples.

\[ A_{64} = a_3 b_2 c_4 d_2 e_1 f_5 g_1 h_1 \]

\[ B_{52} = v_2 a_5 b_1 c_1 d_3 e_1 f_4 g_1 h_1 \]

b) $S_2$ appears once as an independent case.

\[ A_{64} = a_3 b_2 c_4 d_2 e_1 f_5 g_1 h_1 \]

\[ B_{52} = v_2 a_5 b_1 c_1 d_3 e_1 f_4 g_1 h_1 \]
KastX spectrum kernel family
An example with cut weight = 4

a) $S_1$ is the largest substring found on both examples.

\[ A_{64} = a_3 \ b_2 \ c_4 \ d_2 \ e_1 \ f_5 \ g_1 \ h_1 \ i_1 \ j_2 \ k_3 \ m_1 \ n_2 \ f_3 \ g_1 \ h_2 \ o_1 \ p_1 \ q_1 \ r_2 \ s_1 \ t_5 \ u_9 \ b_7 \ c_2 \]

\[ B_{52} = v_2 \ a_5 \ b_1 \ c_1 \ d_3 \ e_1 \ f_4 \ g_1 \ h_1 \ w_2 \ x_2 \ y_1 \ a_1 \ b_2 \ c_6 \ d_1 \ e_3 \ f_1 \ g_1 \ h_3 \ z_1 \ b_5 \ c_1 \ f_1 \ g_1 \ h_1 \]

b) $S_2$ appears once as an independent case.

\[ A_{64} = a_3 \ b_2 \ c_4 \ d_2 \ e_1 \ \boxed{f_5 \ g_1 \ h_1} \ i_1 \ j_2 \ k_3 \ m_1 \ n_2 \ f_3 \ g_1 \ h_2 \ o_1 \ p_1 \ q_1 \ r_2 \ s_1 \ t_5 \ u_9 \ b_7 \ c_2 \]

\[ B_{52} = v_2 \ a_5 \ b_1 \ c_1 \ d_3 \ e_1 \ \boxed{f_4 \ g_1 \ h_1} \ w_2 \ x_2 \ y_1 \ a_1 \ b_2 \ c_6 \ d_1 \ e_3 \ f_1 \ g_1 \ h_3 \ z_1 \ b_5 \ c_1 \ \boxed{f_1 \ g_1 \ h_1} \]

3 (ignored)

\[ 6 \]

\[ 5 \]

\[ 6 \]

\[ 2 \ (\text{ignored}) \]

\[ 8 \]

\[ 6 \]

\[ 9 \]

\[ 18 \]

\[ 17 \]

\[ 19 \]

\[ 7 \]

\[ 5 \]

\[ 6 \]

\[ 9 \]

\[ 18 \]

\[ 17 \]

\[ 19 \]

\[ 7 \]

\[ 5 \]

\[ 6 \]

\[ 9 \]

\[ 18 \]

\[ 17 \]

\[ 19 \]

\[ 7 \]

\[ 5 \]

\[ 6 \]

\[ 9 \]
The \textit{kast1 spectrum kernel} has the following definition:

- Each valid matching substring embeds a new feature for $A$ and $B$.
- The similarity value corresponds to the inner product of the new feature vectors of $A$ and $B$.
- This kernel uses only the weight of the independent valid matching substrings.
- If the string does not present an independent occurrence of a particular valid matching substring, the feature value is set to 1, to avoid zero values when calculating the inner product.
The *kast1 spectrum kernel* has the following definition:

- Each valid matching substring embeds a new feature for *A* and *B*.
The *kast1 spectrum kernel* has the following definition:

- Each valid matching substring embeds a new feature for $A$ and $B$.
- The similarity value corresponds to the inner product of the new feature vectors of $A$ and $B$. 
Kast1 spectrum kernel

Description

The *kast1 spectrum kernel* has the following definition:

- Each valid matching substring embeds a new feature for *A* and *B*.
- The similarity value corresponds to the inner product of the new feature vectors of *A* and *B*.
- This kernel uses only the weight of the *independent* valid matching substrings.
The *kast1 spectrum kernel* has the following definition:

- Each valid matching substring embeds a new feature for $A$ and $B$.
- The similarity value corresponds to the inner product of the new feature vectors of $A$ and $B$.
- This kernel uses only the weight of the *independent* valid matching substrings.
- If the string does not present an independent occurrence of a particular valid matching substring, the feature value is set to 1, to avoid zero values when calculating the inner product.
Kast1 spectrum kernel: an example (I)

New feature vector for $A$

\[ A_{64} = a_3 b_2 c_4 d_2 e_1 f_5 g_1 h_i j_{19} k_1 l_3 m_1 n_2 f_3 g_1 h_2 o_1 p_1 q_1 r_2 s_1 t_5 u_9 b_7 c_2 \]

\[ A_{64} = a_3 b_2 c_4 d_2 e_1 f_5 g_1 h_i j_{7} k_1 l_3 m_1 n_2 f_3 g_1 h_2 o_1 p_1 q_1 r_2 s_1 t_5 u_9 b_7 c_2 \]

\[ A_{64} = a_3 b_2 c_4 d_2 e_1 f_5 g_1 h_i j_{6} k_1 l_3 m_1 n_2 f_3 g_1 h_2 o_1 p_1 q_1 r_2 s_1 t_5 u_9 b_7 c_2 \]
Kast1 spectrum kernel: an example (I)

New feature vector for A

\[ A_{64} = a_3 b_2 c_4 d_2 e_1 f_5 g_1 h_i j k_1 l m_1 n_2 f_3 g_1 h_2 o_1 p_1 q_1 r_2 s_1 t_5 u_9 b_7 c_2 \]

\[ A_{64} = a_3 b_2 c_4 d_2 e_1 f_5 g_1 h_i j k_1 l m_1 n_2 f_3 g_1 h_2 o_1 p_1 q_1 r_2 s_1 t_5 u_9 b_7 c_2 \]

\[ A_{64} = a_3 b_2 c_4 d_2 e_1 f_5 g_1 h_i j k_1 l m_1 n_2 f_3 g_1 h_2 o_1 p_1 q_1 r_2 s_1 t_5 u_9 b_7 c_2 \]

\[ \text{weight}_{k1w \geq 4}(S_1)_A = 19 \] (1)
Kast1 spectrum kernel: an example (I)

New feature vector for A

\[ A_{64} = a_3 b_2 c_4 d_2 e_1 f_5 g_1 h_1 i_1 j_2 k_1 l_3 m_1 n_2 f_3 g_1 h_2 o_1 p_1 q_1 r_2 s_1 t_5 u_9 b_7 c_2 \]

\[ A_{64} = a_3 b_2 c_4 d_2 e_1 f_5 g_1 h_1 i_1 j_2 k_1 l_3 m_1 n_2 f_3 g_1 h_2 o_1 p_1 q_1 r_2 s_1 t_5 u_9 b_7 c_2 \]

\[ A_{64} = a_3 b_2 c_4 d_2 e_1 f_5 g_1 h_1 i_1 j_2 k_1 l_3 m_1 n_2 f_3 g_1 h_2 o_1 p_1 q_1 r_2 s_1 t_5 u_9 b_7 c_2 \]

weight_{k1}w \geq 4(S_1)_A = 19 \quad (1)

weight_{k1}w \geq 4(S_2)_A = 6 \quad (2)
Kast1 spectrum kernel: an example (I)

New feature vector for $A$

\[
A_{64} = a_{3} b_{2} c_{4} d_{2} e_{1} f_{5} g_{1} h_{1} \quad i_{1} j_{2} k_{1} l_{3} m_{1} n_{2} f_{3} g_{1} h_{2} o_{1} p_{1} q_{1} r_{2} s_{1} t_{5} u_{9} b_{7} c_{2}
\]

\[
A_{64} = a_{3} b_{2} c_{4} d_{2} e_{1} f_{5} g_{1} h_{1} \quad i_{1} j_{2} k_{1} l_{3} m_{1} n_{2} f_{3} g_{1} h_{2} o_{1} p_{1} q_{1} r_{2} s_{1} t_{5} u_{9} b_{7} c_{2}
\]

\[
A_{64} = a_{3} b_{2} c_{4} d_{2} e_{1} f_{5} g_{1} h_{1} \quad i_{1} j_{2} k_{1} l_{3} m_{1} n_{2} f_{3} g_{1} h_{2} o_{1} p_{1} q_{1} r_{2} s_{1} t_{5} u_{9} b_{7} c_{2}
\]

\[
\text{weight}_{k1_{w \geq 4}}(S_{1})_{A} = 19 \quad (1)
\]

\[
\text{weight}_{k1_{w \geq 4}}(S_{2})_{A} = 6 \quad (2)
\]

\[
\text{weight}_{k1_{w \geq 4}}(S_{3})_{A} = 9 \quad (3)
\]
Kast1 spectrum kernel: an example (I)

New feature vector for A

\[
A_{64} = \begin{array}{ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
Kast1 spectrum kernel: an example (II)

New feature vector for B

\[
B_{52} = v_2 a_5 b_1 c_1 d_3 e_1 f_4 g_1 h_1 \quad w_2 x_2 y_1 a_1 b_2 c_6 d_1 e_3 f_1 g_1 h_3 \quad z_1 b_5 c_1 f_1 g_1 h_1
\]

\[
B_{52} = v_2 a_5 b_1 c_1 d_3 e_1 f_4 g_1 h_1 \quad w_2 x_2 y_1 a_1 b_2 c_6 d_1 e_3 f_1 g_2 h_3 \quad z_1 b_5 c_1 f_1 g_1 h_1
\]
Kast1 spectrum kernel: an example (II)

New feature vector for \( B \)

\[
\text{weight}_{k1}^{w \geq 4}(S_1)_B = 17 + 18 = 35
\]
Kast1 spectrum kernel: an example (II)

New feature vector for \( B \)

\[
\begin{align*}
B_{52} &= v_2 \begin{array}{ccccccc}
\color{red}{a_5} & b_1 & c_1 & d_3 & e_1 & f_4 & g_1 & h_1 \\
\end{array} \ w_2 \ x_2 \ y_1 \begin{array}{ccccccc}
\color{red}{a_1} & b_2 & c_6 & d_1 & e_3 & f_1 & g_1 & h_3 \\
\end{array} \ z_1 \ b_5 \ c_1 \ f_1 \ g_1 \ h_1 \\
B_{52} &= v_2 \begin{array}{ccccccc}
\color{red}{a_5} & b_1 & c_1 & d_3 & e_1 & f_4 & g_1 & h_1 \\
\end{array} \ w_2 \ x_2 \ y_1 \begin{array}{ccccccc}
\color{red}{a_1} & b_2 & c_6 & d_1 & e_3 & f_1 & g_1 & h_3 \\
\end{array} \ z_1 \ b_5 \ c_1 \ f_1 \ g_1 \ h_1 \\
B_{52} &= v_2 \begin{array}{ccccccc}
\color{red}{a_5} & b_1 & c_1 & d_3 & e_1 & f_4 & g_1 & h_1 \\
\end{array} \ w_2 \ x_2 \ y_1 \begin{array}{ccccccc}
\color{red}{a_1} & b_2 & c_6 & d_1 & e_3 & f_1 & g_1 & h_3 \\
\end{array} \ z_1 \ b_5 \ c_1 \ f_1 \ g_1 \ h_1
\end{align*}
\]

\[
weight_{k1_{w\geq4}}(S_1)_B = 17 + 18 = 35
\]

\[
weight_{k1_{w\geq4}}(S_2)_B = 1
\]
Kast1 spectrum kernel: an example (II)

New feature vector for B

\[
B_{52} = v_2 a_5 b_1 c_1 d_3 e_1 f_4 g_1 h_1 \quad w_2 x_2 y_1 a_1 b_2 c_6 d_1 e_3 f_1 g_1 h_3 \quad z_1 b_5 c_1 f_1 g_1 h_1
\]

\[
B_{52} = v_2 a_5 b_1 c_1 d_3 e_1 f_4 g_1 h_1 \quad w_2 x_2 y_1 a_1 b_2 c_6 d_1 e_3 f_1 g_2 h_3 \quad z_1 b_5 c_1 f_1 g_1 h_1
\]

\[
B_{52} = v_2 a_5 b_1 c_1 d_3 e_1 f_4 g_1 h_1 \quad w_2 x_2 y_1 a_1 b_2 c_4 d_1 e_3 f_1 g_1 h_3 \quad z_1 b_5 c_1 f_1 g_1 h_1
\]

\[
weight_{k1w\geq 4}(S_1)_B = 17 + 18 = 35 \quad (5)
\]

\[
weight_{k1w\geq 4}(S_2)_B = 1 \quad (6)
\]

\[
weight_{k1w\geq 4}(S_3)_B = 6 \quad (7)
\]
Kast1 spectrum kernel: an example (II)

New feature vector for B

\[ B_{52} = v_2 a_5 b_1 c_1 d_3 e_1 f_4 g_1 h_1 w_2 x_2 y_1 a_1 b_2 c_6 d_1 e_3 f_1 g_1 h_3 z_1 b_5 c_1 f_1 g_1 h_1 \]

\[ B_{52} = v_2 a_5 b_1 c_1 d_3 e_1 f_4 g_1 h_1 w_2 x_2 y_1 a_1 b_2 c_6 d_1 e_3 f_1 g_1 h_3 z_1 b_5 c_1 f_1 g_1 h_1 \]

\[ B_{52} = v_2 a_5 b_1 c_1 d_3 e_1 f_4 g_1 h_1 w_2 x_2 y_1 a_1 b_2 c_6 d_1 e_3 f_1 g_1 h_3 z_1 b_5 c_1 f_1 g_1 h_1 \]

\[
weight_{k1_{w \geq 4}}(S_1)_B = 17 + 18 = 35
\]  
\[
weight_{k1_{w \geq 4}}(S_2)_B = 1
\]  
\[
weight_{k1_{w \geq 4}}(S_3)_B = 6
\]  
\[ f_{1_{w \\geq 4}}(B) = \{35, 1, 6\} \]
Kast1 spectrum kernel: an example (III)

Similarity calculation

\[ k_{1_{w \geq 4}}(A, B) = \langle \{19, 6, 9\}, \{35, 1, 6\} \rangle = 725 \]  \hfill (9)
Kast1 spectrum kernel: an example (III)

Similarity calculation

\[ k_{1w \geq 4}(A, B) = \langle \{19, 6, 9\}, \{35, 1, 6\} \rangle = 725 \quad (9) \]

\[ \bar{k}_{1w \geq 4}(A, B) = \frac{725}{\sqrt{k_{1w \geq 4}(A, A) \times k_{1w \geq 4}(B, B)}} \quad (10) \]

According to this kernel, Strings A and B are approximately 21.78% similar.
Kast1 spectrum kernel: an example (III)

Similarity calculation

\[ k_{1w \geq 4}(A, B) = \langle \{19, 6, 9\}, \{35, 1, 6\} \rangle = 725 \]  (9)

\[ \bar{k}_{1w \geq 4}(A, B) = \frac{725}{\sqrt{k_{1w \geq 4}(A, A) \times k_{1w \geq 4}(B, B)}} \]  (10)

\[ k_{1w \geq 4}(A, B) = \frac{725}{64 \times 52} \approx 0.2178 \]  (11)

According to this kernel, Strings A and B are approximately 21.78% similar.
Kast1 spectrum kernel: an example (III)

Similarity calculation

\[ k_{1w \geq 4}(A, B) = \langle \{19, 6, 9\}, \{35, 1, 6\} \rangle = 725 \quad (9) \]

\[ \bar{k}_{1w \geq 4}(A, B) = \frac{725}{\sqrt{k_{1w \geq 4}(A, A) \times k_{1w \geq 4}(B, B)}} \quad (10) \]

\[ \bar{k}_{1w \geq 4}(A, B) = \frac{725}{64 \times 52} \quad (11) \]

\[ \bar{k}_{1w \geq 4}(A, B) = \frac{725}{3328} \quad (12) \]

According to this kernel, Strings A and B are approximately 21.78% similar.
Kast1 spectrum kernel: an example (III)

Similarity calculation

\[ k_{1\geq 4}(A, B) = \langle \{19, 6, 9\}, \{35, 1, 6\} \rangle = 725 \quad (9) \]

\[ \bar{k}_{1\geq 4}(A, B) = \frac{725}{\sqrt{k_{1\geq 4}(A, A) \times k_{1\geq 4}(B, B)}} \quad (10) \]

\[ \bar{k}_{1\geq 4}(A, B) = \frac{725}{64 \times 52} \quad (11) \]

\[ \bar{k}_{1\geq 4}(A, B) = \frac{725}{3328} \quad (12) \]

\[ \bar{k}_{1\geq 4}(A, B) \approx 0.2178 \quad (13) \]

According to this kernel, Strings A and B are approximately 21.78% similar.
\[ k_{1\geq 4}(A, B) = \langle \{19, 6, 9\}, \{35, 1, 6\} \rangle = 725 \quad (9) \]

\[ \bar{k}_{1\geq 4}(A, B) = \frac{725}{\sqrt{k_{1\geq 4}(A, A) \times k_{1\geq 4}(B, B)}} \quad (10) \]

\[ \bar{k}_{1\geq 4}(A, B) = \frac{725}{64 \times 52} \quad (11) \]

\[ \bar{k}_{1\geq 4}(A, B) = \frac{725}{3328} \quad (12) \]

\[ \bar{k}_{1\geq 4}(A, B) \approx 0.2178 \quad (13) \]

According to this kernel, Strings A and B are approximately 21.78% similar.
4. Evaluation
Experiment configuration

Code samples (I)

20 functions X 5 versions (Original, Type-1, Type-2, Type-3 and Type-4 clones) classified as follows:

- (A) Matching functions:
  - K-spectrum kernel.
  - Blended spectrum.
  - Bag-of-characters kernel.
  - Bag-of-words kernel.
  - Bag-of-sentences kernel.

- (B) Sort functions:
  - Bubble sort.
  - Insert sort.
  - Selection sort.
  - Heap sort.
  - Merge sort.
Experiment configuration

Code samples (I)

20 functions X 5 versions (Original, Type-1, Type-2, Type-3 and Type-4 clones) classified as follows:

- **(A) Matching functions.**
Experiment configuration

Code samples (I)

20 functions X 5 versions (Original, Type-1, Type-2, Type-3 and Type-4 clones) classified as follows:

• (A) Matching functions.
  – K-spectrum kernel.
Experiment configuration
Code samples (I)

20 functions X 5 versions (Original, Type-1, Type-2, Type-3 and Type-4 clones) classified as follows:

- **(A) Matching functions.**
  - K-spectrum kernel.
  - Blended spectrum.
20 functions X 5 versions (Original, Type-1, Type-2, Type-3 and Type-4 clones) classified as follows:

- (A) Matching functions.
  - K-spectrum kernel.
  - Blended spectrum.
  - Bag-of-characters kernel.
20 functions X 5 versions (Original, Type-1, Type-2, Type-3 and Type-4 clones) classified as follows:

• (A) Matching functions.
  ‒ K-spectrum kernel.
  ‒ Blended spectrum.
  ‒ Bag-of-characters kernel.
  ‒ Bag-of-words kernel.
20 functions X 5 versions (Original, Type-1, Type-2, Type-3 and Type-4 clones) classified as follows:

- **(A) Matching functions.**
  - K-spectrum kernel.
  - Blended spectrum.
  - Bag-of-characters kernel.
  - Bag-of-words kernel.
  - Bag-of-sentences kernel.
Experiment configuration

Code samples (I)

20 functions X 5 versions (Original, Type-1, Type-2, Type-3 and Type-4 clones) classified as follows:

• (A) Matching functions.
  – K-spectrum kernel.
  – Blended spectrum.
  – Bag-of-characters kernel.
  – Bag-of-words kernel.
  – Bag-of-sentences kernel.

• (B) Sort functions.
20 functions X 5 versions (Original, Type-1, Type-2, Type-3 and Type-4 clones) classified as follows:

- **(A) Matching functions.**
  - K-spectrum kernel.
  - Blended spectrum.
  - Bag-of-characters kernel.
  - Bag-of-words kernel.
  - Bag-of-sentences kernel.

- **(B) Sort functions.**
  - Bubble sort.
Experiment configuration

Code samples (I)

20 functions X 5 versions (Original, Type-1, Type-2, Type-3 and Type-4 clones) classified as follows:

- **(A) Matching functions.**
  - K-spectrum kernel.
  - Blended spectrum.
  - Bag-of-characters kernel.
  - Bag-of-words kernel.
  - Bag-of-sentences kernel.

- **(B) Sort functions.**
  - Bubble sort.
  - Insert sort.
Experiment configuration

Code samples (I)

20 functions X 5 versions (Original, Type-1, Type-2, Type-3 and Type-4 clones) classified as follows:

- **(A) Matching functions.**
  - K-spectrum kernel.
  - Blended spectrum.
  - Bag-of-characters kernel.
  - Bag-of-words kernel.
  - Bag-of-sentences kernel.

- **(B) Sort functions.**
  - Bubble sort.
  - Insert sort.
  - Selection sort.
20 functions X 5 versions (Original, Type-1, Type-2, Type-3 and Type-4 clones) classified as follows:

- **(A) Matching functions.**
  - K-spectrum kernel.
  - Blended spectrum.
  - Bag-of-characters kernel.
  - Bag-of-words kernel.
  - Bag-of-sentences kernel.

- **(B) Sort functions.**
  - Bubble sort.
  - Insert sort.
  - Selection sort.
  - Heap sort.
Experiment configuration

Code samples (I)

20 functions X 5 versions (Original, Type-1, Type-2, Type-3 and Type-4 clones) classified as follows:

- **(A) Matching functions.**
  - K-spectrum kernel.
  - Blended spectrum.
  - Bag-of-characters kernel.
  - Bag-of-words kernel.
  - Bag-of-sentences kernel.

- **(B) Sort functions.**
  - Bubble sort.
  - Insert sort.
  - Selection sort.
  - Heap sort.
  - Merge sort.
Experiment configuration

Code samples (II)

• (C) 3D stencils.
Experiment configuration

Code samples (II)

- (C) 3D stencils.
  - Compact stencil.
Experiment configuration

Code samples (II)

- (C) 3D stencils.
  - Compact stencil.
  - Side stencil.
Experiment configuration

Code samples (II)

- (C) 3D stencils.
  - Compact stencil.
  - Side stencil.
  - Edge stencil.
Experiment configuration

Code samples (II)

- (C) 3D stencils.
  - Compact stencil.
  - Side stencil.
  - Edge stencil.
  - Vertex stencil.
Experiment configuration

Code samples (II)

- (C) 3D stencils.
  - Compact stencil.
  - Side stencil.
  - Edge stencil.
  - Vertex stencil.
  - Non-compact stencil 1 layer.
Experiment configuration
Code samples (II)

• (C) 3D stencils.
  – Compact stencil.
  – Side stencil.
  – Edge stencil.
  – Vertex stencil.
  – Non-compact stencil 1 layer.

• (D) 2D stencils.
Experiment configuration

Code samples (II)

• (C) 3D stencils.
  – Compact stencil.
  – Side stencil.
  – Edge stencil.
  – Vertex stencil.
  – Non-compact stencil 1 layer.

• (D) 2D stencils.
  – Compact stencil.
Experiment configuration

Code samples (II)

- **(C) 3D stencils.**
  - Compact stencil.
  - Side stencil.
  - Edge stencil.
  - Vertex stencil.
  - Non-compact stencil 1 layer.

- **(D) 2D stencils.**
  - Compact stencil.
  - Edge stencil.
• (C) 3D stencils.
  – Compact stencil.
  – Side stencil.
  – Edge stencil.
  – Vertex stencil.
  – Non-compact stencil 1 layer.

• (D) 2D stencils.
  – Compact stencil.
  – Edge stencil.
  – Vertex stencil.
Experiment configuration

Code samples (II)

- (C) 3D stencils.
  - Compact stencil.
  - Side stencil.
  - Edge stencil.
  - Vertex stencil.
  - Non-compact stencil 1 layer.

- (D) 2D stencils.
  - Compact stencil.
  - Edge stencil.
  - Vertex stencil.
  - Non-compact stencil 1 layer.
Experiment configuration

Code samples (II)

• (C) 3D stencils.
  – Compact stencil.
  – Side stencil.
  – Edge stencil.
  – Vertex stencil.
  – Non-compact stencil 1 layer.

• (D) 2D stencils.
  – Compact stencil.
  – Edge stencil.
  – Vertex stencil.
  – Non-compact stencil 1 layer.
  – Non-compact stencil 2 layers.
Experiment configuration
Other setups

- The selected cut weight values were the following:
  - \( \{2^0, 2^1, \ldots, 2^k\} : k = 9 \).

- The clustering algorithm here used was:
  - Hierarchical Clustering.
Baseline kernel 1

Blended spectrum kernel

Cut weight = 16.
Baseline kernel 2 (previous work)

Kast spectrum kernel

Cut weight = 64.
Proposed kernel
Kast1 spectrum kernel

Cut weight = 16.
5. Conclusions and Future Work
Conclusions

- The proposed *kast1 spectrum kernel* and the *kast spectrum kernel* had similar clustering performance.
Conclusions

- The proposed \textit{kast1 spectrum kernel} and the \textit{kast spectrum kernel} had similar clustering performance.
- They showed a consistent formation of three clusters: matching functions, sorting functions and stencils (3D and 2D).
Conclusions

- The proposed *kast1 spectrum kernel* and the *kast spectrum kernel* had similar clustering performance.
- They showed a consistent formation of three clusters: matching functions, sorting functions and stencils (3D and 2D).
- They yielded better results than the *blended spectrum kernel* as the clustering showed no misplaced examples.
Conclusions

- The proposed *kast1 spectrum kernel* and the *kast spectrum kernel* had similar clustering performance.
- They showed a consistent formation of three clusters: matching functions, sorting functions and stencils (3D and 2D).
- They yielded better results than the *blended spectrum kernel* as the clustering showed no misplaced examples.
- This indicates that this novel comparison method can be promisingly utilized to find similarities in source code snippets.
Future work

• **Automatic selection of the cut weight.**
Future work

- Automatic selection of the cut weight.
- Analysis of the intra-cluster distances between clone types.
Future work

- Automatic selection of the cut weight.
- Analysis of the intra-cluster distances between clone types.
- Comparison against a tree kernel applied directly over the ASTs.
Future work

- Automatic selection of the cut weight.
- Analysis of the intra-cluster distances between clone types.
- Comparison against a tree kernel applied directly over the ASTs.
- Study the linear intermediate representation delivered by the LLVM Compiler Infrastructure.
Thanks!