Supporting Materials

For paper "Recommending Code Snippets for Code Completion Based on a Large-Scale API Usage Base"

Qian Wu§, Yingnong Dang§, Chenglin Zhong§, Han Yin§, Qianxiang Wang§, Dongmei Zhang§, Tao Xie³
§Peking Univ., China, §Microsoft Research Asia, China, ³University of Illinois at Urbana-Champaign, U.S.A.

This document is the supporting material of our paper “Recommending Code Snippets for Code Completion Based on a Large-Scale API Usage Base”. This document includes three sections. Section 1 presents two examples as supporting materials for Section 5.2.3 of our paper. Section 2 presents an example of our code completion with low precision, which we use as the supporting material for Section 5.3.4 of our paper. Section 3 presents the questionnaire we used for our user study. The results of the user study can be found in Section 6 of our paper.

1. Validation of Using Invocation Set to Characterize Snippets

In Section 5.2.3 of our paper, we validate our hypothesis that snippets with the same set of method invocations tend to have similar semantics (proposed in Section 2.4.1 of our paper). As presented in Section 5.2.3, we obtained statistics of the snippet groups we extracted from a large codebase. The snippets in each group have identical sets of method invocations. We show the statistics in Table 1 of our paper, which indicate that the snippets in a majority (77.4%) of groups have the same semantic because they share the exact same Program Dependency Graph (PDG). Table 1 of our paper also indicates that the snippets in the remaining groups (22.6%) have 2 or more distinct graphs. The average similarity of the graphs in these groups ranges from 0.62 to 0.86.

As presented in Section 5.2.3, to better understand groups with two or more distinct graphs, we further break them down into two types, with type A being groups with the same method invocation sequences, and type B with different method invocation sequences.

As mentioned in Section 5.2.3 of our paper, we manually examined 10 groups (randomly sampled) with type A and type B, respectively. We found that all the groups sampled have the same or similar semantics. In the following, we provide 4 sample groups we examined, 2 with type A and 2 with type B.

1.1 Type A (Same Method Invocation Sequence, Different PDG)

Among the 10 sampled groups of Type A, the largest group has 78 snippets, and the smallest one has 4 snippets. We carefully examined these 10 groups, and found that code snippets in all of these groups have very similar semantics. There are four major types of differences among code snippets in same group.

a. Parameter preparation. The two example code snippets in Figure 1 below have the same set of method invocations. They also have the same method invocation sequence. The semantic of both is to call a stored procedure to retrieve data from a database (ToString and ToInt32 are in the predefined list of trivial methods and are thus neglected). One difference between these two code snippets is that they use different ways to specify the name of the stored procedure. The name of the stored procedure is assigned to command.CommandText in the snippet in Figure 1(a) (Line 4), while the name of the stored procedure is directly specified when sqlCommand is initialized in the snippet in Figure 1(b) (Line 1). These two different approaches of specifying the name of the stored procedure have the exact same semantic, while having different program dependency graphs.

```
1. using (SqlCommand com = new SqlCommand())
2. {
3.    com.CommandType = CommandType.StoredProcedure;
5.    com.Parameters.Add(new SqlParameter("@RunId", RunId));
6.    com.Connection = conn;
7.    SqlDataReader reader = com.ExecuteReader(CommandBehavior.CloseConnection);
8.    while (reader.Read()) {
9.        bucketNumber = Convert.ToInt32(reader["BucketNumber"]);
10.   }
11. }
```

(a)
b. **Data manipulation.** The two snippets in Figure 1 also process the data read from SQL databases in different ways. Such differences are task specific. For example, one column is read in the snippet in Figure 1 (a), while two columns are read in the snippet shown in Figure 1 (b). Such task-specific differences lead to the two snippets having different PDGs. However, such task-specific contrast do not result in any big differences in code recommendation, i.e., either of them can be used as a reference for engineers who want to implement similar logic.

```
1. SqlCommand command = new SqlCommand("GetLocationCountByBuckets", connection);
2. command.CommandType = System.Data.CommandType.StoredProcedure;
3. command.Parameters.Add(new SqlParameter("@VersionID", versionID));
4. using (SqlDataReader dataReader = command.ExecuteReader())
5. {
6.   if (dataReader.HasRows)
7.   {
8.     while (dataReader.Read())
9.     {
10.    if (dataReader["LocationCount"] != DBNull.Value)
11.    {
12.       locationBucket.LocationCount = Convert.ToInt32(dataReader["LocationCount"].ToString());
13.    }
14.    }
15.    locationBucket.ShortName = dataReader["ShortName"].ToString();
16.  }
17. }
18. }
```

```
1. SqlCommand myCommand = new SqlCommand();
2. myCommand.Connection = sqlConnection;
3. if (myCommand.CommandType == CommandType.StoredProcedure && null != sqlParams)
4. {
5.   foreach (KeyValuePair<string, object> sqlParam in sqlParams)
6.   {
7.     SqlParameter sqlParameter = new SqlParameter(sqlParam.Key, sqlParam.Value);
8.     myCommand.Parameters.Add(sqlParameter);
9.   }
10. }
11. try
12. {
13.   objReturn = myCommand.ExecuteScalar();
14. }
15. catch (SqlException e)
16. {
17.   throw e;
18. }
```

Figure 1. Example code snippets with the same set of method invocation set and sequence, but different program dependency graphs; they have very similar semantics

(b)

1. SqlCommand command = new SqlCommand("GetLocationCountByBuckets", connection);
2. command.CommandType = System.Data.CommandType.StoredProcedure;
3. command.Parameters.Add(new SqlParameter("@VersionID", versionID));
4. using (SqlDataReader dataReader = command.ExecuteReader())
5. {
6.   if (dataReader.HasRows)
7.   {
8.     while (dataReader.Read())
9.     {
10.    if (dataReader["LocationCount"] != DBNull.Value)
11.    {
12.       locationBucket.LocationCount = Convert.ToInt32(dataReader["LocationCount"].ToString());
13.    }
14.    }
15.    locationBucket.ShortName = dataReader["ShortName"].ToString();
16.  }
17. }
18. }

1. SqlCommand myCommand = new SqlCommand();
2. myCommand.Connection = sqlConnection;
3. if (myCommand.CommandType == CommandType.StoredProcedure && null != sqlParams)
4. {
5.   foreach (KeyValuePair<string, object> sqlParam in sqlParams)
6.   {
7.     SqlParameter sqlParameter = new SqlParameter(sqlParam.Key, sqlParam.Value);
8.     myCommand.Parameters.Add(sqlParameter);
9.   }
10. }
11. try
12. {
13.   objReturn = myCommand.ExecuteScalar();
14. }
15. catch (SqlException e)
16. {
17.   throw e;
18. }

Figure 2. Example of code snippets with different coding styles and control-flows, while having very similar semantics.

(c) **Coding styles.** The use or absence of try/catch, and using are two typical examples. Such a difference does not impact the major semantic of the code snippets, while leading to different PDGs. For example, the code in Figure 2 (a) uses try/catch, but
the snippet in Figure 2 (b) doesn't. These two code snippets have very similar semantics.

d. **Control-flow.** Some code snippets are different in the use of control-flow logic while having very similar semantics. For example, the code snippet in Figure 2(a) adds parameters to the list myCommand.Parameters when myCommand.CommandType is equal to CommandType.StoredProcedure and the input parameter list sqlParams is not null. However, the code snippet in Figure 2(b) directly sets command.CommandType to be System.Data.CommandType.StoredProcedure and adds a new parameter to the list command.Parameters by directly constructing a new instance of SqlParameter. Such differences in control-flow logic are task specific and not semantically significant. Another example is that the code shown in Figure 2 (a) iteratively adds a collection of parameters using a for loop, while the code in Figure 2 (b) only adds one parameter without a for loop. Such a difference can be viewed as task-specific.

In summary, code snippets with the same invocation sequence, i.e., code snippets of a group of Type A are likely to share the same semantics.

**1.2 Type B (Different Method Invocation Sequence, Different PDG)**

We manually examined 10 groups with Type B. The code snippets within each group have the same sets of method invocations but different invocation sequences. We found that the code snippets in all 10 groups have the same or similar semantics.

One of these 10 groups contains 252 snippets, with the invocation set {SqlConnection.new, CreateCommand, Open, Close}. These code snippets have 11 different invocation sequences, as shown in Table 1 below. The frequency of each sequence, i.e., the number of snippets with a method invocation sequence, is also shown in Table 1.

<table>
<thead>
<tr>
<th>ID</th>
<th>Method Invocation Sequence</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SqlConnection.new -&gt; CreateCommand -&gt; open -&gt; Close</td>
<td>190</td>
</tr>
<tr>
<td>2</td>
<td>SqlConnection.new -&gt; Open -&gt; CreateCommand -&gt; Close</td>
<td>19</td>
</tr>
<tr>
<td>3</td>
<td>SqlConnection.new -&gt; CreateCommand -&gt; Open -&gt; Close -&gt; CreateCommand -&gt; Open -&gt; Close</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>SqlConnection.new -&gt; CreateCommand -&gt; Open -&gt; CreateCommand -&gt; Close</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>SqlConnection.new -&gt; CreateCommand -&gt; Open -&gt; Close -&gt; CreateCommand -&gt; Open -&gt; Close -&gt; CreateCommand -&gt; Open -&gt; Close</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>SqlConnection.new -&gt; CreateCommand -&gt; Open -&gt; CreateCommand -&gt; Close -&gt; CreateCommand -&gt; Open -&gt; CreateCommand -&gt; Close</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>SqlConnection.new -&gt; CreateCommand -&gt; Open -&gt; Close -&gt; CreateCommand -&gt; Open</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>SqlConnection.new -&gt; CreateCommand -&gt; Open -&gt; Close -&gt; SqlConnection.new</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>SqlConnection.new -&gt; Open -&gt; CreateCommand -&gt; CreateCommand -&gt; CreateCommand -&gt; Close</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>SqlConnection.new -&gt; Open -&gt; CreateCommand -&gt; Close -&gt; Close -&gt; Close -&gt; Close -&gt; Close -&gt; Close -&gt; Close</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>Close -&gt; SqlConnection.new -&gt; Open -&gt; CreateCommand</td>
<td>1</td>
</tr>
</tbody>
</table>

There are two reasons leading to snippets with the same set but different invocation sequences.

(1) **Duplicate method invocations.** There are still clones remaining in some snippets. The reason is that we used a rather conservative code-clone removal technique (Section 2.3.3 in our paper). These clones serve for some task-specific purposes, but are not meaningful for code recommendation purpose.

The invocation sequences 3-8 in Table 1 can be derived by adding duplicate calls to the first sequence (1). Figure 3 and Figure 4 show the example code snippet with method invocation sequences 1 and 3, respectively. Sequences 9 and 10 can be derived by adding duplicate invocations to 2.

(2) **Different orders of method invocations.** There are three different orders of method invocations in the code snippet group shown in Table 1. Sequence 1, 2, and 11 are representatives of these three orders of method invocation. The example code snippets of these three sequences are shown in Figures 3, 5 and 6, respectively.

The semantics of the code snippet in Figures 3 and 5 are the same, i.e., the order of method invocation CreateCommand and Open does not matter. The semantics of the code snippet in Figure 6 is different from the other two. In the code snippet in Figure 6, the status of ConnectionState is verified first (Line 3), and then conn.Close() is invoked if conn.State is not closed. However, the frequency of this sequence is only 1, indicating that such an invocation sequence is used in rare cases. Therefore, we prefer to recommend sequence 1, which is very popular.

```
1. SqlConnection conn = new SqlConnection(strConn);
2. using (SqlCommand sqlCmd = conn.CreateCommand())
3. {
4.   conn.Open();
5.   conn.Close();
6. }
```

**Figure 3. A code-snippet with method invocation sequence 1 in Table 2: SqlConnection.new -> CreateCommand -> Open -> Close**
Figure 4. A code-snippet with method invocation sequence 3 in Table 2: SqlConnection.new -> CreateCommand -> Open -> Close -> CreateCommand -> Open -> Close

```csharp
7. SqlConnection conn = new SqlConnection(strConn);
8. using (SqlCommand sqlCmd = conn.CreateCommand())
9. {
10.   conn.Open();
11.   conn.Close();
12. }
13. for (int i = 0; i < phases.Count; i++)
14. {
15.   using (SqlCommand cmdInsert = conn.CreateCommand())
16.   {
17.     conn.Open();
18.     conn.Close();
19.   }
20. }
```

Figure 5. A code-snippet with method invocation sequence 2 in Table 2: SqlConnection.new -> Open -> CreateCommand -> Close

```csharp
1. try
2. {
3.     SqlConnection thisConnection = new SqlConnection(s_AlexandriaConnectionString);
4.     thisConnection.Open();
5.     SqlCommand thisCommand = thisConnection.CreateCommand();
6.     thisConnection.Close();
7. }
8. catch (SqlException e)
9. {
10. }
```

Figure 6. A code-snippet with method invocation sequence 11 in Table 2: Close -> SqlConnection.new -> Open -> CreateCommand

```csharp
1. if (conn == null || conn.State != ConnectionState.Open)
2. {
3.   if (conn != null & conn.State != ConnectionState.Closed)
4.     conn.Close();
5.   conn = new SqlConnection(Analyzer.InterleavingDB);
6.   conn.Open();
7. }
8. performanceCommand = conn.CreateCommand();
```

2. Example of Code Completion with Low Precision

We analyzed the reason for leading a low precision result of our code completion approach in Section 5.3.4 of our paper. We present an example of low precision results returned by our tool in this section, as shown in Figure 7. In this example, we simulate a real use case. We assume an engineer asks for help from our tool after he inputs the first statement (Line 1 in Figure 7(a)), where a new instance of object AlertMailSession, called currentMailSession, is created. Our tool recommends a code snippet that includes a number of method invocations and returns a number of attributes of currentMailSession. However, only one of these recommended method invocations is used in real code\(^1\) (Line 2-7 of Figure 7(a)). Consequently, a low precision score is obtained. In this case, our tool actually provides much more information than what is needed. It is meaningful to engineers since we provide full documentation about how to get different attributes of the object AlertMailSession.

3. Questionnaire

We presented our user study result in Section 6 of our paper. The questions in the questionnaire we used in this study are all open questions, as shown below.

1. How many years of c# coding experience do you have?

\(^{1}\) Please refer to Section 5.3.1 of our paper about how we build the testing set.
2. In general, how many (code snippet) suggestions do you want to browse?
3. What kind of help do you think this prototype could provide you with?
4. What kind of capabilities do you expect such a tool to have?
5. Can you provide more comments / suggestions about this code completion prototype?

(a) Test method

```
using (AlertMailSession currentMailSession = new AlertMailSession(startTime, endTime) {
    try{
        currentMailSession.GetBuildBreakCount(out totalBuildBreakMailCount, out threadIds);
    }
    catch (System.Exception e){
    }
}
```

(b) Recommended code snippet

```
AlertMailSession mailSession = new AlertMailSession();
customerArray = mailSession.GetCustomers();
clusterArray = mailSession.GetClusterInfo();
alertArray = mailSession.GetAlertInfo();
threadStarterArray = mailSession.GetThreadStarts();
...
autoMessageCount = mailSession.GetAutoMessageCount();
buildBreakThreadCount = mailSession.GetBuildBreakCount(out buildBreakMailCount, out threadIds);
buildBreakerArray = mailSession.GetBuildBreakers();
...
```

Figure 7. An example of code-completion with low precision