Inconsistency Detection in Software Component Source Code using Ant Colony Optimization and Neural Network Algorithm

Amit Verma*, Srishti Gupta and Iqbaldeep Kaur

Computer Science and Engineering, Chandigarh Engineering College, Landran, Mohali – 140307, Punjab, India; damitverma.cu@gmail.com, srishtigupta221@gmail.com, iqbaldeepkaur.cu@gamil.com

Abstract

Objectives: Inconsistency detection is one of the major challenges in source code for the software developers. There is the need of consistent identifiers to reduce the code inconsistencies. So, developers should either have the knowledge to create conceptual identifiers or the knowledge to detect the inconsistencies in source code. Methods/Statistical Analysis: There is the availability of a list of tools for the detection of different types of inconsistencies. But the existing tools are not much appropriate for Semantic, Syntactic Inconsistencies and Part of Speech tagging. Findings: In the paper, an autonomous tool Automatic Bad Code Detector (ABCD) is developed to detect semantic, syntactic and part of speech inconsistency in the source code. ABCD tool identifies the inconsistencies in the source code based on the detected Code Clones. These Clones are detected by matching the test code with Code Repository. A java project based code repository is considered for experimentation. ABCD is evaluated for different java projects in order to find inconsistencies in source code. In ABCD tool main inconsistency detector are Ant Colony Optimization and Neural Network Back Propagation algorithm. Further, ABCD is useful in re-implementing the new versions of the java code. Applications/Improvements: The current concept is evaluated for the Semantic, Syntactic, POS-Word and POS-Phrase inconsistencies based on evaluation parameter of precision. The efficiency of ABCD is evaluated as an overall value for the precision, recall and f-measure.

Keywords: Ant Colony Optimization, Automatic Bad Code Detector, Code Repository, Inconsistency Detection, Neural Network Back Propagation Algorithm

1. Introduction

Inconsistency may be defined as mismatching between the code identifiers but code will not be considered as inconsistent if they mean the same\(^1\). There is the need of efficient tools to detect the inconsistent identifiers. Developers should also be trained to identify the difference between the consistent and inconsistent identifiers so that inconsistencies in the code can be identified\(^2\). Inconsistency can be in the form of Semantic, Syntactic or POS inconsistency. Section 2 presents some of the existing methods for the detection of these inconsistencies.

In this current work, “Automatic Bad Code Detector” (ABCD) is developed to detect inconsistency and maintain consistency in the source code. ABCD tool works in order to detect inconsistencies due to the change in the source code on the basis of Code Repository that result into Code Clones. Code Clones are mainly of two types, exact clones and renaming clones. Exact clones are those Code Clones in which two or more code fragments found to be same with code repository. Renaming clones are syntactically similar code fragments due to change in data types or renaming identifiers. In our experimentation, ABCD detects the inconsistency in code based on the Exact and Renaming Clones.

ABCD tool is also useful to find changes in the java source code. An outline for ABCD tool is prepared by following three phases. In the first phase, developer uploads the java projects and creates the Code Repository that comprises of specific variables and methods in a

\(^{1}\) Author for correspondence
class. In the second phase, the tool uploads the target java project. It will track the changes made to the source code and check the inconsistency on the basis of Code Repository. This approach uses two algorithms “Ant Colony Optimization Technique” and “Neural Network Back Propagation Algorithm” to scan the source code changes and to detect inconsistencies in the source code of java project. During the last phase, if there will be changes detected, then the tool offers two options to opt from, either it will update the changes to the source code or it will make a new version of the java project and save into the Code Repository. Current approach is also useful in reusing the source code, since java files (components) are stored into the Code Repository and can be retrieved for further use.

In addition with inconsistencies detection (due to the changes in code), ABCD is also helpful in detecting bad smell from the source code. Bad Smell in a source code can be defined as violation condition or more precisely, errors that make code inefficient. A list of well known inconsistencies, bad smells and bugs is presented in Table 1.

<table>
<thead>
<tr>
<th>Type of Bugs/Inconsistencies/Bad Smell</th>
<th>Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntactic Inconsistency Bugs</td>
<td>Bugs (code changes), code smell, style smells (Change in design), and intentional inconsistent edits.</td>
</tr>
<tr>
<td>Misspelling Inconsistencies</td>
<td>Spelling mistakes in code. e.g. Methods, identifiers etc.</td>
</tr>
<tr>
<td>Lexicon Bad smells</td>
<td>Violation of grammatical rules Identifier mapping, Naming rules, Type name indication.</td>
</tr>
<tr>
<td>Type Checking Bad smells</td>
<td>Badsmell in conditional statements- (if/switch statements)</td>
</tr>
<tr>
<td>Homonyms Inconsistencies</td>
<td>Badsmell in conditional statements- (if/switch statements)</td>
</tr>
<tr>
<td>Synonyms Inconsistencies</td>
<td>Different words with same meaning. Mapping of semantics with semantics of identifier.</td>
</tr>
<tr>
<td>Concise mapping</td>
<td>Syonyms. Same word sequence.</td>
</tr>
</tbody>
</table>

Rest of the section is described in the following manner: Section 2 presents the existing work for inconsistencies detection. Section 3 describes the basic concepts used like Component Based Software Engineering, Code Repository, Neural Network and Artificial Ant Colony Optimization. Section 4 discusses about the ABCD tool. Section 5 describes the evaluated results with comparative analysis of the ABCD tool with other considered tools/concepts and Section 6 concludes the paper.

2. Literature Survey

In this computing era, researchers have developed many different tools to automatically detect the inconsistency in the source code. Existing work for the source code inconsistency detection by some authors is presented here. In it has been discussed about the spelling checking in programming code. As per authors, there is the need to pre-check the spelling of each word that programmers types during coding work. In it has been described the way to present the identifiers. Authors suggested for the proper naming of the identifiers that address the code and instances to avoid the misunderstanding for developers. In extension of the work is done by inducing the semantic identifiers for inconsistencies detection. Here the previous rule of conciseness and consistency is extended with its compatibility from names to concepts without any mapping. Authors have also introduced Part of Speech tagging on the basis of words.

In the concept of lexicon bad smell detection in source code has introduced. Lexicon bad smell are some construction problems in lexicon that usually addressed with renaming or refactoring of code. This method is efficient to find the inconsistencies in lexicon that usually addressed with phrase based part of speech identifiers. Further, in an autonomous approach to find suitable identifiers from source code has proposed. It also supports the automatic completion of identifiers during writing and for replacement if the existing identifier can be replaced with more appropriate one. This approach supports both the semantic and syntactic inconsistent identifiers. In semantic porting analyses algorithm for the detection and characterization of porting inconsistencies has proposed. But there was the need of improvement to reduce the false positive rate of concept.

In a code dictionary based concept for the detection of inconsistencies in the code has developed. Authors have given a GUI based tool CodeAmigo for the detection.
of inconsistencies in code using code dictionary. Authors have evaluated the concept for the semantic, syntactic and Part of Speech (PoS) inconsistent identifiers on the basis of parameters of precision and recall. Overall CodeAmigo shows efficient results as compared to existing approaches. But authors have not suggested any further tool for overall software maintenance. Also there is need for the code reusability.

Different authors presented their work for the different inconsistency types. In current work, ABCD tool is developed that have the capability to detect the Syntactic, Semantic and PoS (word, phrase) inconsistencies. These inconsistencies are identified for the java project codes with the detection of Exact and Renaming Code Clones. The basic concepts considered for the experimentation are discussed in the next section.

3. Basic Concepts

This section describes the basic concepts of Component Based Software Engineering (CBSE), Code Repository, Ant Colony Optimization and Neural Network Back Propagation algorithm.

3.1 Component Based Software Engineering (CBSE)

Component Based Software Engineering (CBSE) can be defined as a component reuse-based approach that defines implements and compose loosely coupled independent components into the systems. Components communicate through transmitting interfaces. Component-Based Software Engineering (CBSE) is a process that emphasizes the design and construction of component-based systems using reusable software\(^\text{13}\). The following text consists of software components, reusability of components and the CBSE process.

A software component is a software element that conforms to a component model and can be individually executed and composed without making any changes\(^\text{14}\). A software component is a unit of composition with contractually specified interfaces and explicit context dependencies. A software component can be executed independently\(^\text{15}\). Components can be reused for further development.

Reusable software components are the modules that are designed to be useful in several programs\(^\text{16}\). Component reusability or Code reusability can be defined as the use of existing Code Repository to build new software. One can also reuse Design, Documents and Test Cases etc. Software components are used for development of new software instead of ruining time and cost of firms\(^\text{17}\). A reusable software component is a logically cohesive, loosely coupled module that denotes a single abstraction\(^\text{18}\).

Component-Based Software Engineering is based on rapid assembly and maintenance of component-based systems, where components and platforms have properties and these properties provide the basis for predicting properties of systems built from components\(^\text{19}\).

In the current work, java software components are used. Components are scanned and java files are extracted from java software components. Finally, these components in the form of java files are stored into the Code Repository

3.2 Code Repository

Code repository can be termed as a “Database” that catalogs identifiers from the source code. It consists of identifier’s name, meaning and relationships with other identifiers in the database.

To build a Code Repository, the developer scans its own trusted documents (java documents). Basically, this approach collects class, methods, attributes and identifiers from the java documents. The developer can add other java documents or remove existing documents in order to check the Code Repository function\(^\text{10}\).

In the current approach, Code Repository is build up from input java projects. Java files are extracted from the source code in order to develop the repository. Repository mainly consists of name of the java files. With the help of “White Space Tokenization Technique”, splitting of identifiers in java file is done and those identifiers are stored into the code repository. Each java file comprise of class name and its associated method names, attributes, interfaces, parameters and constants. The criterion for repository is that each class should be encapsulated in separate java file. The schema for building a code repository is given in Figure 1.

<table>
<thead>
<tr>
<th>File name</th>
<th>Array List</th>
<th>Complete code</th>
</tr>
</thead>
</table>

Figure 1. Code Repository Schema.

File name consists of java files with extension of .java. Array list consist of details for each java file i.e. Class
name and its methods, attributes, interfaces, parameters and constants. Complete code carries the whole source code in order to track exact line number when any inconsistency is detected or any changes are made.

Now, to scan the identifiers (class and its methods, attributes etc), an optimization technique is required. Ant Colony Optimization technique will be used in the current approach.

### 3.3 Ant Colony Optimization

Ant Colony Optimization is an artificial intelligence based concept introduced by Marco Dorigo for the optimization of the NP hard problems. In ACO, a number of artificial ants build up the solutions to an optimization problem and exchange information on their quality via a communication scheme. Ant Colony Optimization is based on the optimization behavior of social species ants that work in local and global experience sharing behavior. In ACO, ants use a substance name pheromone to share their experience with the path of other ants.

Ant colonies or more accurately social insect societies are distributed systems that, in spite of behaving individually, present a highly structured social organization. As a result of this organization, ant colonies can perform complex tasks. The “ant colony algorithm” is derived from the observation of real ants and their behavior. The main idea is that the self-organizing principles which allow the highly coordinated behavior of real ants can be exploited to coordinate populations of artificial agents that collaborate to solve computational problems.

The approach behind ACO can be summarized as an iterative process in which a population of simple agent's recursively constructs candidate solutions; this construction process is guided by heuristic information over the given problem. ACO has been applied to a various range of hard combinatorial problems. Here, problems are defined in terms of components and states, which are basically the sequences of components. Ant Colony Optimization incrementally generates solution paths for the given component and it adds new components to a state. Memory contains all the transitions between pairs of solution components and also the degree of desirability is associated to each transition depending on the quality of the solutions in which it occurred so far. While a new solution is generated, a component ‘c’ is included in a state, with a probability that is proportional to the desirability of the transition between the last component included in the state, and c itself. Pseudo code for basic working of Ant Colony Optimization technique is given as:

```plaintext
Set initial_state;
Initialize pheromone trail and heuristic value;
do
{
    MOVE_neighbour states;
    BUILD_feasible solution;
    COMPUTE_heuristic value;
    EVALUATE_solution;
    UPDATE_pheromone;
}
while (current state solution == destination state solution)
Termination condition.

Initial state is Starting state, ant moves towards its Destination.

Pheromone trail is a chemical substance produced and released into the environment by an ant in order to trace the path.

Heuristic Information represents the candidate solution.

In the current approach, Ant Colony Optimization Technique is used to enhance the performance for the clone matching to detect the changes and inconsistencies in the source code.

Further, to detect inconsistencies in the source code an autonomous technique of Neural Network Automated Back Propagation Algorithm is used to detect inconsistencies in source code based on the code repository.

### 3.4 Neural Network Back Propagation Algorithm (NN-BPA)

Neural Network comprises of an input, output and one or more hidden layers. The connection between these layers is like, each node from input layer is connected to a node from hidden layers and every node from hidden layers is connected to a node in output layer. A basic model for neural network is shown in Figure 2. To every connection a weight is associated to it. Input layer consists of the raw information that is fed into the network. This part of network never changes its values. Every single input to the network is duplicated and sent down to the nodes in hidden layer. Hidden layer accepts data from the input layer. It uses input values and modifies them using some weight value, this new value is then sent to the output
layer but here also some modification by some weight is done from connection between hidden and output layer. Output layer processes the information received from the hidden layer and produces an output. This output is then processed by activation function.

\[ H \rightarrow I \rightarrow D \rightarrow D \rightarrow E \rightarrow N \]

The following steps can describe the back-propagation algorithm\(^27\):

- Initialize.
- Set of inputs,
  \[ A = [a_1, a_2, a_3, \ldots, a_n] \]
  \[ A = [a_1, v_1, a_2, v_2, a_3, v_3, \ldots, a_n, v_n] \]
- Submit pattern.
  \( T_p \) - Target Pattern \((a_1, v_1, a_2, v_2, a_3, v_3, \ldots, a_n, v_n)\)
  \( E_p \) - Existing Pattern \((e_1, e_2, e_3, \ldots, e_n)\)
  \( T_p \) is submitted in order to make comparison between \( T_p \) and \( E_p \).
  - Compare for all input elements
    Input Elements are: \[ A[a_1], A[a_2], A[a_3], \ldots, A[a_n] \]
    if \[ T_p = E_p \] {  
      (pattern gets match);  
    }  
    else  
    {  
      (pattern matching error);  
    }
  - Back-propagation of an error and weight modification.
    if \[ T_p \neq E_p \] then
    add erroneous result to the list of array (learns from its mistakes)
  
  Next, Output to the input elements are,
  \[ b_1 = [h_1 w_1] \],
  \[ b_2 = [h_2 w_2] \],
  \[ b_3 = [h_3 w_3] \],
  
  \[ b_n = [h_n w_n] \]
  Array of output is given as,
  \[ B = [b_1, b_2, b_3, \ldots, b_n] \]
- End of pattern selection from the training set.
- End of learning process.

In the current approach, the objective of the algorithm is to generate patterns in order to detect the changes and inconsistencies in the source code.

This Algorithm is also used to generate patterns to make comparisons, in order to detect inconsistencies and Bad-smell in the source code.

\[ V_1 \rightarrow V_2 \rightarrow V_3 \rightarrow \ldots \rightarrow V_n \]

Figure 2. Basic Model of Neural Network.

### 3.4.1 Activation Function

The actual output is evaluated by activation function. As shown in the Figure 3, SUM is collection of the output nodes from hidden layer that have been multiplied by connection weights, added to get single number and put through activation function. Activation function gets mentioned together with learning rate, momentum and pruning\(^25\).

One of the most popular Neural Network algorithms is Back-Propagation Algorithm. Back-propagation is the most widely used algorithm for supervised learning with multi-layered feed-forward networks\(^26\).

\[ a_1 \rightarrow V_1 \rightarrow b_1 \]

\[ a_2 \rightarrow V_2 \rightarrow b_2 \]

\[ a_3 \rightarrow V_3 \rightarrow b_3 \]

\[ \ldots \]

\[ a_n \rightarrow V_n \rightarrow b_n \]

Figure 3. Activation function.
4. **Automatic Bad Code Detector (ABCD)**

In the current work, ABCD tool is developed to detect the inconsistency in the source code. The functioning of ABCD tool is described below with three modes as explained:

**Mode 1: Developing Code Dictionary**
- Start.
- Developer upload java project in zip format into the tool one by one.
- Then, Developer unzips the considered java projects.
- Java files from project (.java extension files are extracted) are extracted using ant colony algorithm.
- Array list for each java file is created.
- Splitting of each line into identifiers is done with the help of white space tokenizer which will be stored into array list.
- Each array list of java file with class name and path name is stored into the code repository (database).
- Stop.

To store java project into the code repository above steps are repeated recursively.

Database (Code Repository) consist of file name, Array list and complete code and here java file contains class name, attributes, methods, parameters, identifiers etc.

**Mode 2: Changes and Inconsistency checking in source code.**
- Start.
- Target program is considered as an input file in zip format into the tool.
- Then, Developer unzip the considered java project.
- Extraction of java files from project is done with the help of Ant Colony algorithm. It iterate till the end of the source code. ACO scans code line by line and iterate the speed of code scanning.
- Then, each line of code is split using white space tokenizer and extracted identifiers are stored into the array list.
- Now, Neural network back propagation algorithm is applied to find inconsistencies due to changes in the code.

Further, for the

**Mode 3: New version to the source code (java project).**

If there will be any changes in the code, then with the help of ant colony algorithm, class name with its line number will be stored in array list. Finally ABCD presents two options:
- Tool will update changes to the source code and store updates into the code repository.
- Tool will reconstruct a new version of the source code and it saves into the code repository.

The above steps explain the working to develop an inconsistency tool (ABCD). Following section will give pseudo code for the development of ABCD.

START
UPLOAD Java_Project ;
EXTRACT Java_File [ACO];
CREATE Arr[];
SPLIT Identifiers [Space-Tokenizer];
STORE Identifiers_Code Repository;
DETECT Inconsistency (NN-BPA);
COMPARE Arr[] & Code Repository
if (Arr[] == Code Repository)
{
("No Inconsistent Code Detected");
}
else
{
("Inconsistent Code Detected");
}
REPORT Inconsistent Code;
CHECK_Code_Changes
if (targ_code==exist_code)
{
if (Code_Changes= Commit)
{
Make a new version;
}
else (Code_Changes= Rollback)
{
Make changes to the exist_code;
}
}
else
{
No Code_Changes;
}
TERMINATE;
Where,
Arr[] represents arraylist of identifiers.
targ_code represents Target code.
ext_code represents Existing code in code repository.

4.1 Key Points
- Detection of inconsistency will be done by checking into the code repository. Comparison between the current array list and code repository will be made, if there is any mismatch then the tool will show inconsistent result.
- Patterns will be made in order to detect changes in the code. If there will be any mismatch in the patterns then it may easily detect changes into the source code.
- Tool will report inconsistencies and changes made with file name (class name and line number) by using ant colony algorithm.

The complete data flow for the implemented concept are shown in the Figures 4 to 7 in form of Context DFD, Level 0 and Level 1 DFD.

In Figure 4, Context data flow diagram has considered the end components as Developer/Tester and code repository. Initially developer uploads the data projects into ABCD tool and mapping of code is performed to check the consistency of data by making a match with another end component code repository. Then ABCD tool decides whether the code is consistent or inconsistent.

In Figure 5 level 1 DFD, there are two data stores named Array List and Repository. Four processes as considered are ‘Scan and Split Identifiers’, ‘Inconsistency Detecting’, ‘Detect Code changes’ and ‘Create New Version’. Initially, java project will be uploaded. These projects are scanned and identifiers are splitted to check their consistency. Identifiers are compared from the Array List and Repository that whether the names of object, class, attributes, parameters are well defined as per code concept. In Array list data flow to second process of ‘Inconsistency detecting’, if code found to be inconsistent, then third process of ‘Detect Code Changes’ is performed and code changes are saved. As per the code changes, a new version of the code can be generated which is finally stored in Repository.

In Level 1 DFD is split into two parts Level 1.1 and Level 1.2 for the concrete understanding of ABCD tool.

As shown in Figure 6, Level 1.1 DFD introduced the concept of Ant Colony Optimization for the scanning of java project files. Ant Colony Optimization uses the local and global optimization based iterative concept to identify the inconsistent code identifiers and clones. Based on the ACO approach, it distinguishes the code identifiers and split identifiers using the space tokenization and the match consistent identifiers with the Array List store.

The output for the level 1.1 DFD is used in Level 1.2 DFD to check further the consistency of code. These identifiers are tested by generating code pattern using Neural Network back Propagation Algorithm as shown in Figure 7. Generated code pattern for the considered java code is compared with the existing code patterns. The existing code patterns are available in Array List and Repository. Finally a comparison is made. If the considered code found to be mismatched with the code repository, then code is considered to be inconsistent code.

Figure 4. Context data flow diagram.
5. Experimental Results and Discussion

This section evaluates the performance of the ABCD tool for the inconsistencies of Semantic, Syntactic and POS (word and phrase) types. Windows 7 based system with 4 GB of RAM, 500 GB of HDD, an Intel(R) Core(TM) i7 CPU, is used for conducting the experiments. Java based platform, Net Beans IDE 8.1 is used for implementation. For overall evaluation of implemented concept, parameters of precision, recall and f-measure are considered. These parameters are described as:

5.1 Evaluation Parameters

There are various available metrics used to evaluate different techniques. In current work, performance parameters of Precision, Recall and F-Measure have been used. To calculate the Precision, Recall and F-Measure, there is the need to understand the following terms as below:
• **TP (True Positive):** Is the percentage of inconsistent identifiers, which is classified as inconsistent.

• **FN (False Negative):** Is the percentage of consistent identifiers, which is classified as inconsistent.

• **TN (True Negative):** Is the percentage of inconsistent identifiers which is classified as consistent.

• **FP (False Positive):** Is the percentage of consistent identifiers which classified as consistent.

Based on these important measures, values of Precision ($p$), and Recall ($r$), F-measure are evaluated. The basic formulation of these terms is as follows:

- **Precision:** It denotes the proportion to measure the positiveness of concept. In other words, it can be defined as the method to have the accurate value of inconsistency detection as per actual repository code. This can be calculated as:
  \[ \text{Precision} = \frac{TP}{TP + FP} \]

- **Recall:** It measure the relevance from positive concept. This can be calculated as:
  \[ \text{Recall} = \frac{TP}{TP + FN} \]

- **F-Measure:** F-measure is the harmonic mean of precision and recall. It can be calculated as below:
  \[ \text{F-Measure} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \]

In this implemented approach, ABCD tool is tested for 12 java based projects. For different java projects, results vary. The evaluated values of 12 different projects with their clone type, matching, inconsistency precision (Semantic, Syntactic, POS-Word and POS-Phrase), Overall Precision, Recall and F-measure values are shown in Table 2.

From the evaluated results, it can be seen that different projects have different output value with different type of code clones. Code clones are found to be of exact and rename type. Precision value of POS-Word inconsistency is 81%. Precision value of POS-phrase inconsistency varies from 97.94% to 98.95%. Precision value of Semantic Inconsistency value varies from 68.18% to 74.76%. Precision value of Syntactic Inconsistency value varies from 83.3% to 83.33%. Overall Precision value varies from 82.76% to 84.51%. Overall Recall value is 75.33%. F-Measure value varies from 80.87% to 81.65%. Table 2 presents the average value for all the 12 java projects. The overall value of Precision, Recall and f-measure are shown with Figure 8 by evaluating the mean of considered 12 java projects.

### Table 2. Evaluated results for considered java projects

<table>
<thead>
<tr>
<th>Components/Features</th>
<th>Evaluated Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Projects</td>
<td>Total 12 Projects</td>
</tr>
<tr>
<td>Clone Type</td>
<td>Exact and Rename</td>
</tr>
<tr>
<td>Precision based POS-Word Inconsistency (%)</td>
<td>81 %</td>
</tr>
<tr>
<td>Precision based POS-Phrase Inconsistency (%)</td>
<td>98.36 %</td>
</tr>
<tr>
<td>Precision based Semantic Inconsistency (%)</td>
<td>72.72 %</td>
</tr>
<tr>
<td>Precision based Syntactic Inconsistency (%)</td>
<td>83.33 %</td>
</tr>
<tr>
<td>Overall Precision (%)</td>
<td>83.88 %</td>
</tr>
<tr>
<td>Recall (%)</td>
<td>75.33 %</td>
</tr>
<tr>
<td>F-Measure (%)</td>
<td>81.14 %</td>
</tr>
</tbody>
</table>

---

**5.2 Comparative Analysis**

To check the efficiency of ABCD tool, a comparative analysis is made with CodeAmigo tool\(^{10}\). The considered concept is evaluated based on the precision values for inconsistencies of Semantic, Syntactic, POS-Word and POS-Phrase basis. ABCD tool is compared with CodeAmigo inconsistency detection tool. Precision value of POS-Word inconsistency is 81%. Precision value of POS-phrase inconsistency varies from 97.94% to 98.95%. Precision value of Semantic Inconsistency value varies from 68.18% to 74.76%. Precision value of Syntactic Inconsistency value varies from 83.3% to 83.33%. To make the comparative analysis, an average precision value is considered. This Comparative analysis is shown in Figure 9.

---

[Add Figure 8 and Table 2 here]
Inconsistency detection is an important aspect in software development. The entire code depends upon the Semantic and Syntactic identifiers. But due to lack of developer’s knowledge and some other factors, some inconsistencies remains in the source code that interrupt the further development of code for a new developer. In the paper, the work on Semantic, Syntactic, POS-word and PSO-Phrase based inconsistencies detection is presented. The present work uses the ABCD tool for inconsistency detection. A java project based code repository is created. In this repository, Java files from java source code are stored. To find the inconsistencies in the changed source code, Ant Colony Optimization and Neural Network Back Propagation algorithm are used. Ant Colony Optimization Technique is used to enhance the performance for the clone matching to detect the changes and inconsistencies in the source code. Neural Network Automated Back Propagation Algorithm is used to detect inconsistencies in source code based on the code repository. This algorithm is also used to generate patterns to make comparisons, in order to detect inconsistencies and Bad-smell in the source code. Results are efficient as compared to CodeAmigo tool for Semantic and POS-Word inconsistency detection but somewhere lacks for the POS-phrase and Syntactic Inconsistency. Overall results are evaluated in the form of precision, recall and f-measure. So, it can be seen that ABCD tool is efficient enough to detect inconsistent identifiers.

7. References

25. Cilimkovic M. Neural Networks and Back Propagation Algorithm. Institute of Technology; Blanchardstown, North Dublin. 2015.