University of Essex
Department of Mathematical Sciences

MA981: DISSERTATION

Executable Decision Trees

Sandhya Lekshmana Kammath
2200835

Supervisor: Danilo Petti

August 25, 2023
Colchester

Abstract

The objective of the project is to create a software engineering tool that addresses the growing need for streamlined deployment of machine learning models across diverse execution environments, such as embedded devices, Internet of Things (IoT) devices, enterprise infrastructure, and handheld devices. The Decision Tree Model serves as a versatile supervised learning technique, adept at making predictions for alphanumeric (tabular) data. One nice property of Decision Tree Algorithm is the potential for generating rules organised as a series of conditional constructs organised in a nested manner. The tool explores the exciting possibility of generating shared objects (.so) or dynamic link libraries (.dll) after generating C/C++ from the rules generated from decision trees. The tool also generates Java bytecode by leveraging a public domain bytecode generation library. The resulting code in the form of a .class file will be packaged within an executable Java ARchive (JAR) file.

As a versatile and powerful rule-based machine learning algorithm, decision trees facilitate the efficient decision-making process by relying on a series of rules and conditions derived from input features. These rules are then parsed by a compiler, leading to the generation of corresponding Java bytecode and C/C++ code through transcompilation. The generated C/C++ code can be transformed to executable code using the GNU Compiler Collection (GCC) compiler. The Java bytecode is generated directly.

The software engineering tool's potential impact spans across diverse domains, including finance, healthcare, marketing, and others, offering a versatile solution for efficient model deployment. By leveraging decision trees and generating executable code, the tool enhances the development and deployment process, enabling seamless integration into various execution environments, including resource-constrained embedded systems. This capability opens up opportunities for on-device inference and decision-making, reducing reliance on external servers and promoting independent and efficient machine learning model operation.

Contents

List of Figures

List of Tables

List of Abbreviations

CART Classification And Regression Tree

MLOps Machine Learning Operations

.so Shared Object

.dll Dynamic Link Library

IoT Internet of Things

GCP Google Cloud Platform

AWS Amazon Web Services

PMML Predictive Model Markup Language

ONNX Open Neural Network Exchange

JAR Java ARchive

JVM Java virtual machine

AST Abstract Syntax Tree

GCC GNU Compiler Collection

CLR Common Language Runtime

BCEL Byte Code Engineering Library

CSV Comma-Separated Values

DSL Domain Specific Language

CFG Context Free Grammar

BNF Backus Naur Form

AIOps Artificial Intelligence for IT Operations

SLANG Simple LANGuage

ANTLR ANother Tool For Language Recognition

CHAPTER 1

Introduction

There is increasing demand for efficient deployment of machine learning models in various execution environments. The objective is to offer executable code as a versatile and accessible solution for deploying machine learning models. This approach aims to enhance the efficiency of model deployment, benefiting various applications and industries that depend on streamlined and effective deployment processes. By incorporating the executable code into the Machine Learning Operations (MLOps) pipeline, the model can be efficiently deployed, monitored, and managed alongside other components of the pipeline. It promotes automation, reproducibility, and version control of the model deployment process, ensuring smooth integration into the overall production environment.

Another exciting possibility is that the executable code, as Shared Object (.so) and Dynamic Link Library (.dll) can be utilized in embedded environments. Thus leveraging the performance and efficiency benefits of these languages in resource-constrained and embedded systems. This opens up opportunities for deploying machine learning models directly on edge devices, Internet of Things (IoT) devices, or other embedded platforms, where real-time or low-latency processing is crucial. These files, when integrated into embedded applications, allow on-device inference and decision-making. This reduces the reliance on external servers or cloud-based processing, enabling more efficient and independent operation of machine learning models in embedded environments.

The problem arises from the common approach of distributing the majority of machine learning models as resource files on the specified destination devices. The need to transfer Machine Learning engines to edge devices introduces a range of challenges, encompassing concerns such as version management, logistical complexities during transportation, and the requirement to accommodate extensive runtimes on these devices. This tool provides a straightforward resolution for deploying specific machine learning models in the form of Java ARchive (JAR) files within enterprise environments. Additionally, it simplifies deployment on edge devices through the utilization of .so or .dll files.

The research undertaking involved a meticulous exploration of the prevailing practices and methodologies related to the deployment of machine learning models and solutions in a multitude of industries. This endeavor

encompassed a comprehensive investigation into the techniques, challenges, and advancements employed to effectively integrate machine learning technologies into real-world applications. The majority of public cloud providers have developed MLOps pipelines to facilitate the large-scale deployment of machine learning models. We thoroughly assessed the offerings provided by Google Cloud Platform (GCP), Amazon Web Services (AWS), and Azure. In each of these instances, the models undergo conversion into a transfer format, Predictive Model Markup Language (PMML), PICKLE, or Open Neural Network Exchange (ONNX) for neural networks, before they are deployed within the intended target environment. Another notable feature offered by these platforms is the provision of runtimes specifically designed to interpret and execute the models within the designated target environment. This resource-oriented approach, centered on transferring files, necessitates further enhancement by offering deployment models that eliminate the requirement for runtimes on the target machines. The concept revolves around utilizing native code shared objects on mobile, handheld, and IoT devices, while leveraging executable JAR files in enterprise environments. This approach is chosen due to the assured existence of the Java virtual machine (JVM) across the majority of enterprise infrastructures.

This project's core emphasis lies in the development of a software engineering tool designed to streamline the process of generating Java bytecode and C/C++ code from tabular data. The tool will encompass a range of components and processes, working together cohesively to facilitate a smooth and efficient transformation of tabular data into both Java bytecode and C/C++ code.

The steps involved in the process includes

(a) Incorporating Jython for data ingestion within the machine learning program

(b) Generation of decision tree using the Classification And Regression Tree (CART) algorithm through Jython

(c) Generating the rules using Jython

(d) Converting rules into an Abstract Syntax Tree (AST) through the Simple LANGuage (SLANG) compiler (SLANG is a programming language).

(e) Transcompilation of AST to C/C++ code via SLANG compiler

(f) Compiling AST into executable JAR files using the SLANG Compiler

CHAPTER 2

Literature Review

Machine learning models have gained widespread adoption and utilization in the application environments be it the enterprise applications, desktop applications, handheld devices, IoT devices or the cloud environments. Consequently, this has led to an increased demand for the deployment of these models in a scalable manner. The current regime is based on a resource oriented approach (storing and transmitting models as files), when it comes to large deployments. This has led to a lot of anomalies in the process. The wider issues have been summarised in the following articles ([5], [16]). The development and effective deployment of Artificial Intelligence for IT Operations (AIOps) / MLOps solutions in practical, real-world scenarios present significant challenges that cut across various facets, including both technical and non-technical aspects. This underscores the complexity and breadth of the obstacles that need to be addressed for the seamless integration and utilization of AIOps solutions [5]. According to Paleyes et al [16], the principal contribution of the article resides in its exposition of the diverse challenges that practitioners encounter across every dimension of the machine learning deployment workflow. In the context of model deployment, a comprehensive exploration reveals a range of intricate challenges that span three pivotal phases: integration, monitoring, and updating. These distinct stages of the deployment process underscore the complex nature of deploying machine learning models in real-world scenarios.

Certain machine learning models can be transformed into executable files suitable for native formats. One way to package these models is by utilizing .dll for Windows or .so for Linux/Unix environments. For the enterprise environment, since there is presence of JVM, the models can be shipped as .class files embedded inside executable JAR files. The current project is an attempt to test the feasibility of creating a pipeline which takes data in tabular format to generate models which can be embedded inside executable files as code.

During the implementation of the pipeline, our journey encompassed extensive research across a diverse array of topics, spanning an expansive range from Comma-Separated Values (CSV) parsing to the complications of code generation within compilers. The exploration traversed a multifaceted landscape, delving into subjects as varied as CSV parsing techniques, the conversion of CSV files into Python or Jython lists, the dynamic inject of data into Jython scripts from Java language, the implementation of the CART algorithm, the generating rules from the decision trees to the SLANG format, and the nuanced realms of lexical analysis. Moreover,

our exploration ventured into the realm of recursive descent parsing, the construction and utilization of AST, the meticulous design of symbol tables, the recursive functions, the judicious application of design patterns, the process of transcompilation, the inner workings of the JVM, and bytecode generation using Byte Code Engineering Library (BCEL).

Roua Jabla et al. [11], developed a rule generation module that conducts pre-processing on a candidate dataset. This module creates decision trees and derives well-performing association rules. The goal is to enhance a rule knowledge base and enhance real-time decision-making processes. Taking idea from the above article, rules were generated from a custom cart algorithm as a mixture of Domain Specific Language (DSL) and SLANG format. We tweaked the SLANG compiler grammar to support DSL notation. A DSL is a specialized language designed for a particular problem domain or specific application. DSLs are often more concise, expressive, and user-friendly than general-purpose languages ([10], [8]).

The SLANG compiler as referenced in [1] encompasses various stages and components similar to those in general compiler construction, including Lexical Analysis, Syntax Analysis (Parsing), Semantic Analysis, Code Generation, Symbol Tables and Error Handling. During the Parsing phase, the compiler constructs the AST, which captures the essential syntactic information, from the input source code. The AST was structured utilizing composite pattern, as detailed in notable design pattern references ([9] and [15]). To handle the processing of this AST, the visitor pattern was adopted, drawing insights from the same above design pattern references ([9], [15]). This structured AST was subsequently transcompiled into ANSI C/C++ to facilitate the creation of executable files and compiled to Java bytecode.

Transcompilation, often referred to as transpiling, is an essential tool in modern software development, helping developers write code using the latest language features while ensuring that the code runs reliably across various platforms and environments. According to Branco et al. [3], discusses a tool called "CLARA" which is a transpiler designed to export machine learning models trained using the Scikit-Learn library to plain C code. The primary objective of CLARA is to enable the transpilation of Scikit-Learn machine learning models into the Resource Scare Environment Systems (RSES) framework. RSES is a software package and modeling framework used for analyzing and simulating complex ecological systems, specifically in the context of resource-scarce environments. Furthermore, CLARA offers multiple optimization techniques that enhance the performance of the transpiled models. The concept of transpiling machine learning models to a different framework or language is important for interoperability and integration purposes, allowing models to be deployed and utilized within different systems and environments.

The generation of the Java bytecode [7] was a rather involved process. This necessitated the study of JVM internals and JVM architecture. The JVM is a stack based virtual machine architecture which uses hotspot optimization. The knowledge of JVM architecture was gleaned from the JVM specification [12].

This extensive effort covered a wide range of subjects, each adding an important element to the complex process of implementing the pipeline. Our research journey, spanning basic principles to advanced methods, showcased the complexity of our pursuit as we built a strong and adaptable solution across various fields.

During our research, we actively delved into the feasibility of integrating ensemble methods such as bagging, boosting, and stacking into our approach [4]. This exploration encompassed a thorough examination of their potential applications and benefits. Within the context of stacking, a range of techniques, including simple

linear regression, multiple linear regression, and association algorithms [20] like Apriori, emerged as promising candidates.

Linear regression models are statistical techniques used to analyze and model the relationship between a dependent variable (also referred to as the target or response variable) and one or more independent variables (also known as predictor or feature variables). The main goal of linear regression is to find the best-fitting linear relationship that explains the variation in the dependent variable based on the values of the independent variables. In simple linear regression, the model involves only one independent variable. Multiple linear regression captures the relationships and interactions between multiple independent variables and the dependent variable.

In the stacking scenario, it was observed that linear regression models, be it simple linear regression or multiple linear regression, offered a viable avenue for implementation ([2], [13]). Linear regression models can be encoded as mathematical expressions involving scalar values (vectors in the case of simple linear regression and matrices in the case of multiple linear regression). The Apriori algorithm is a classic data mining technique used for association rule mining in large datasets and helps to identify frequent item sets and their associated rules from transactional databases. Notably, the Apriori algorithm emerged as a particularly suitable contender for generating rules within the SLANG framework, akin to the approach taken by the current tool. This insight underscored the potential for leveraging existing techniques to seamlessly integrate with our approach, thereby enhancing its versatility and potential.

In essence, our research ventured beyond the confines of our core approach, extending into the realm of ensemble methods and algorithmic combinations. This comprehensive exploration enriched our understanding and paved the way for innovative avenues that could potentially augment the capabilities of our tool in addressing complex real-world scenarios.

CHAPTER 3

Methodology

There are several algorithms commonly used to generate decision trees, each with its own approach to selecting the best attributes for splitting and creating the tree nodes. Some of the notable decision tree algorithms are ID3 (Iterative Dichotomiser 3), C4.5, CART, Random Forest.

The CART algorithm and its variants are widely used machine learning algorithms for creating decision tree models. The CART algorithm is flexible and capable of addressing both classification and regression tasks. When dealing with algorithms like the CART, the input for the decision tree model is structured as a tabular dataset organized in a row-major sequence, accompanied by a corresponding set of attributes. It is a common convention to place the target variable, also known as the dependent variable, in the final column of the dataset though it is not a strict rule. This arrangement is often preferred for consistency, ease of reference, compatibility and visualization.

The objective is to ascertain the feasibility of constructing a pipeline capable of converting tabular data into executable code through the utilization of a decision tree. The project structure is depicted in the illustration 3.1.

The process of generating executable code from a model involves the following steps.

3.1 Generating Decision Tree model using custom implementation of Classification And Regression Tree (CART) algorithm

The initial phase involves creating a decision tree from the tabular data. The tabular data is sourced as a CSV file format. Given the dataset's inclusion of both alphabetic and numeric data, we have chosen to utilize the CART algorithm for constructing the decision tree [19]. The initial stage involves constructing the representation of a tree. The CART algorithm initiates the process by iteratively analyzing the table of records to establish a root node. This node serves as the starting point for partitioning rows into two distinct segments, employing a boolean criterion to optimize the division. When dealing with columns that contain alphabetic data, our approach involves employing the equality (=) criterion. In contrast, for columns with numeric values, we opt for the greater than or equal to (>=) criterion. Following the initial split, the algorithm gives rise to a left

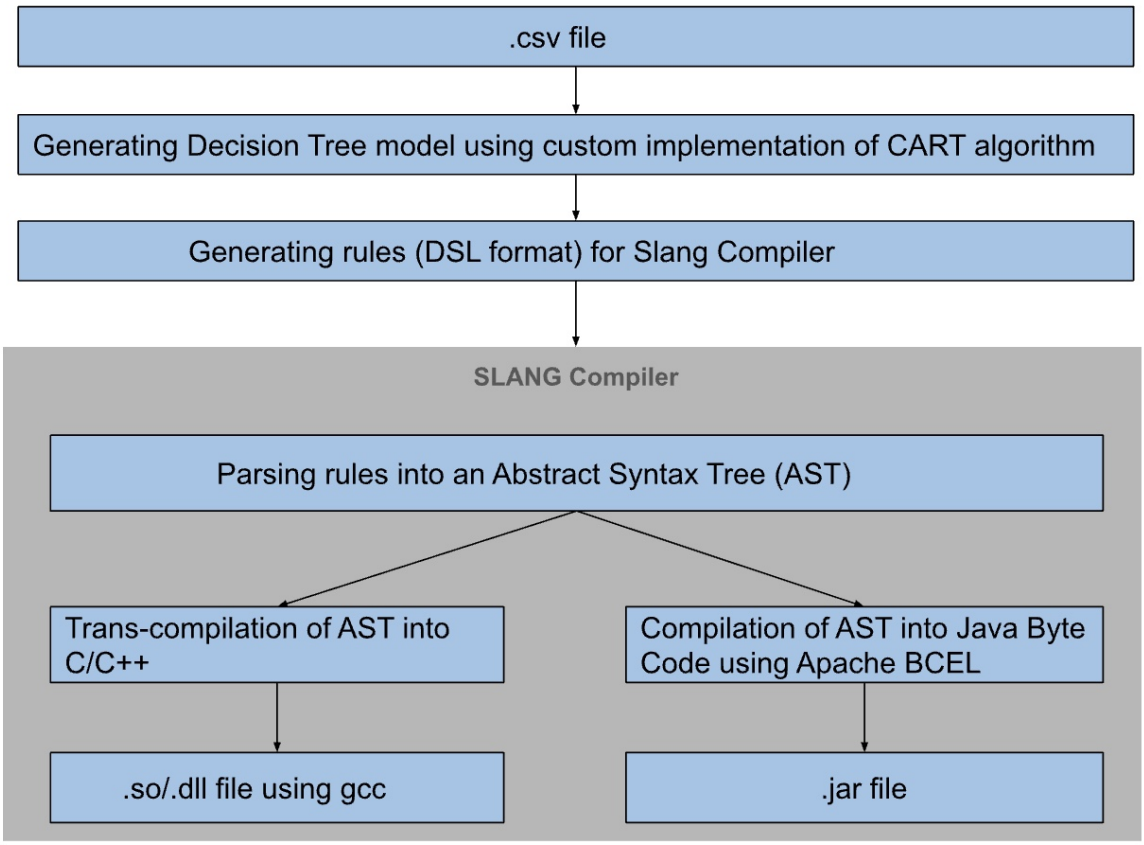

Figure 3.1: **Project Structure. Step 1:** Loading the tabular data as CSV file. **Step 2:** Generating Decision Tree Model from CSV file using Custom CART Algorithm. **Step 3:** Transforming decision tree to generate rules SLANG format **Step 4:** SLANG compiler parses rules into AST. **Step 5:** Tanscompilation of AST into C / C++ and compilation of AST into Java Bytecode. **Step 6:** Packaging C / C++ to .so /.dll and Java Bytecode to JAR file

subtree and a right subtree. Subsequently, this process is repeated recursively for the remaining columns. This iterative algorithm determines the most suitable column for partitioning a data table (or sub-table) by assessing information gain using metrics such as Entropy and Gini Impurity. Entropy and Gini impurity are integral components of decision tree algorithms, aiding in the selection of optimal splits and the construction of effective decision trees for classification tasks. The diagram 3.2 [6] illustrates the structure of a decision tree. Entropy and Gini Impurity are two commonly used metrics in decision tree algorithms for evaluating the quality of splits and making decisions about node divisions. The mathematical formula of entropy is given as 3.1.1 [19] and that of Gini impurity is given as 3.1.2 [19].

$$E(S) = \sum_{i=1}^{c} -p_i log_2 p_i \tag{3.1.1}$$

$$Gini(E) = 1 - \sum_{i=1}^{c} {p_i}^2 \tag{3.1.2}$$

Where c is the total number of classes and p_i is the probability of choosing a data point belonging to class i. Entropy is a measure of impurity or disorder in a set of data. In the context of decision trees, entropy is used

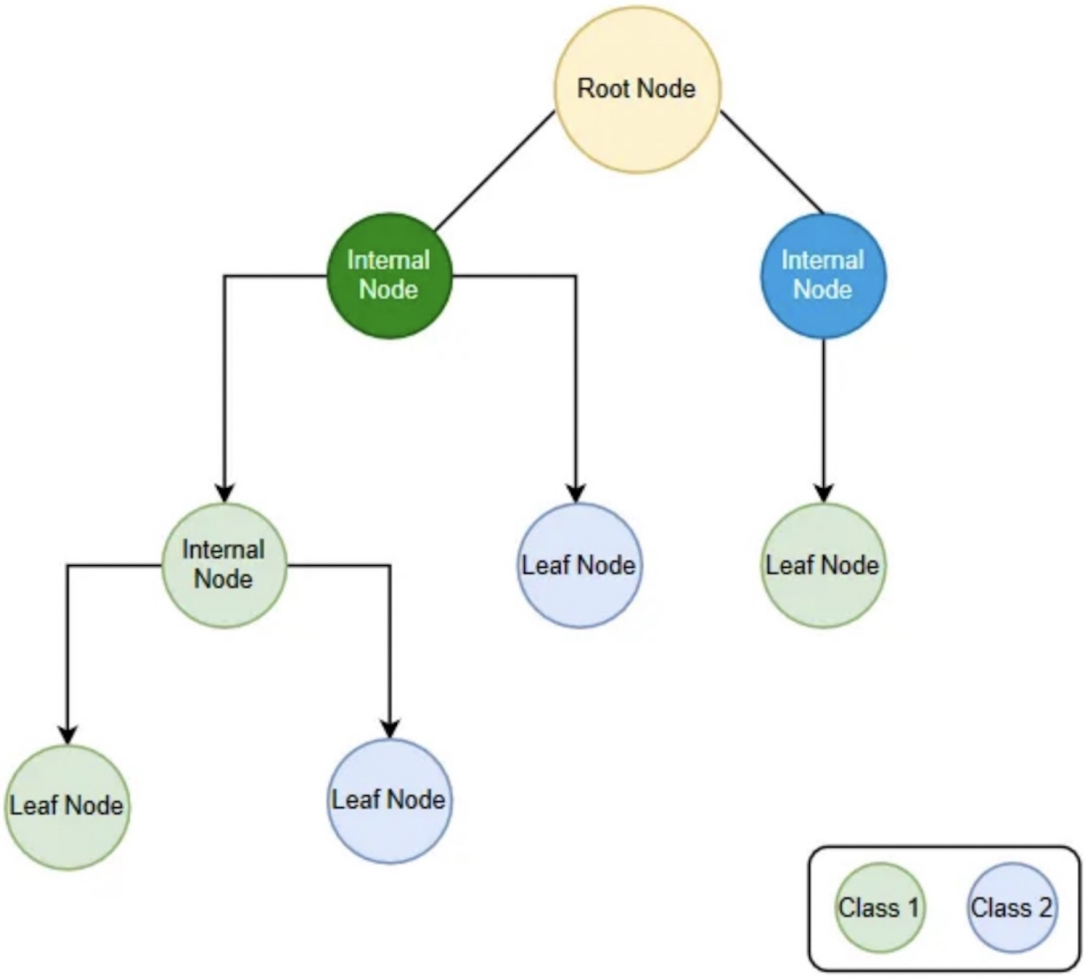

Figure 3.2: **Decision Tree Structure. Root Node:** It is the topmost node which serves as the starting point for making decisions. **Internal Node:** This also called as decision node as this is where node splits further based on the best attribute of the sub-group. **Leaf Node:** Final node from any internal node which holds the decision. **Class 1 and Class 2**: Leaf node corresponds to a specific class that the decision tree uses for prediction.

to calculate the impurity of a node by measuring the uncertainty or randomness of class labels within that node. A lower entropy indicates a more pure or homogeneous node, while higher entropy implies more diverse or mixed class labels. Gini Impurity is another measure of impurity used in decision tree algorithms. It quantifies the likelihood of a randomly selected data point being incorrectly classified based on the distribution of class labels in a node. A lower Gini Impurity value signifies a more pure node with predominantly one class label, while a higher value indicates a more mixed node.

Both entropy and Gini impurity are utilized during the process of determining the best attribute to split a node in a decision tree at a given level. The attribute with the lowest impurity after the split is chosen as the decision criterion. These metrics guide the decision tree algorithm in creating splits that maximize class separation and ultimately lead to more accurate predictions.

Upon identifying the optimal split at a specific level, the algorithm invokes itself recursively to further divide the left subtree and right subtree stemming from the designated node. This splitting and recursion

continue until a stopping criterion is met. This criterion could be a certain depth of the tree, a minimum number of samples per leaf, or other similar factors. But for current implementation we have done the split up to the leaf node.

When the recursion stops, the terminal nodes of the tree are called leaf nodes. Each leaf node represents a data item where a final decision is made. In a classification task, the majority class in the leaf node might be assigned as the predicted class. In a regression task, the mean or median value of the target variable in the leaf node could be used as the prediction. The figure 3.3 [18] shows the structure of the decision tree for a sample dataset. We decided to create a custom implementation for our project despite having the choice to use decision

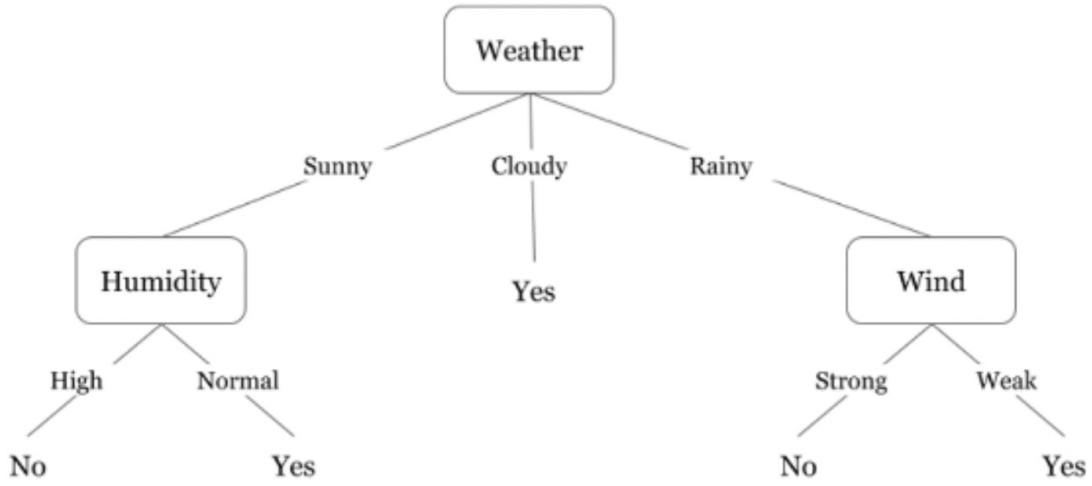

Figure 3.3: To determine if a person would partake in outdoor activities, the tree's initial decision enquired about the weather condition: whether it's sunny, cloudy, or rainy. If the answer is affirmative, the next step is to consider humidity and wind factors. If the wind is Weak or the humidity is within the normal range, this might lead to the decision that the individual can proceed outdoors.

tree modules like Scikit-learn or Weka. Scikit-learn or Weka are both popular open-source machine learning libraries. Our objective is to showcase the pipeline and a unique tool as part of our demonstration.

3.2 Generating Rules in Domain Specific Language (DSL) format

DSLs are powerful tools for addressing complex and specific problems within their designated domains. They can improve code readability, maintainability, and development speed when used appropriately.

Following the creation of the decision tree using the CART algorithm, a recursive function was implemented to traverse the tree in a depth-first manner. This function generates a sequence of if/else statements that accurately depict the structure of the tree. We chose a target language called SLANG, with a specific grammar based on the SlangForDotNet [14] compiler. At present, the compiler accommodates Integer, Double, String, and Boolean data types exclusively. However, there exists the potential to broaden its capability to encompass additional data types in the forthcoming stages.

3.3 Generating Abstract Syntax Tree (AST) using SLANG Compiler

The rules derived from the decision trees are generated as per the syntax of the SLANG programming language. It is a general-purpose DSL programming language designed to handle a wide range of tasks and cater to various domains. The SLANG which stands for Simple LANGuage is a programming language that includes built-in support for a range of data types, encompassing String, Boolean, Integer, and Double. The language provides features for conditional constructs such as if/else statements, supports looping through constructs like while loops, and facilitates the implementation of functions and recursive functions. The figure 3.4 depicts the various steps involved in the implementation of SLANG compiler.

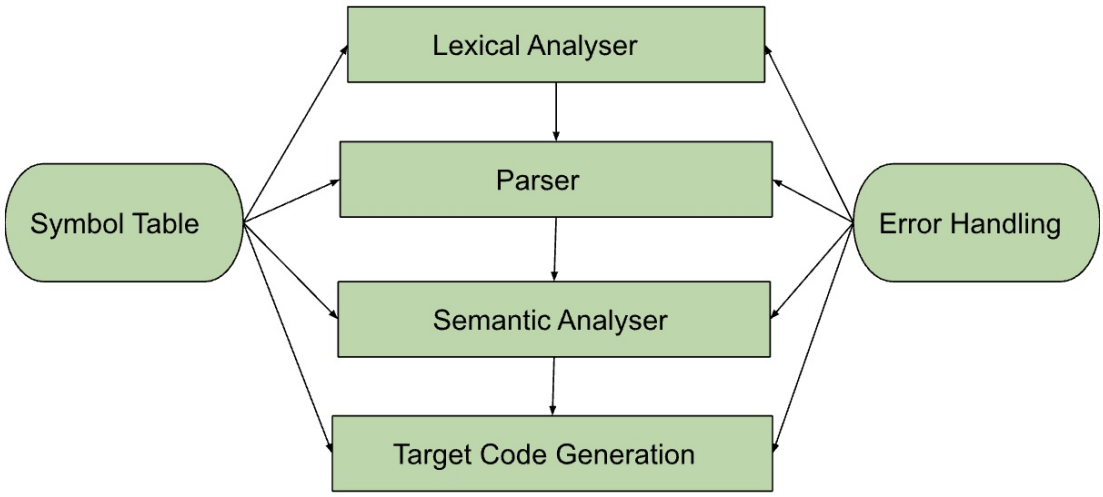

Figure 3.4: **SLANG Compiler Phases. Lexical Analyser:** Lexical Analyser breaks the source code into a series of Token. **Parser:** Parser works along with Lexical Analyser to construct Abstract Syntax Tree (AST). **Semantic Analyser:** Semantic Analyser does the type checking to check the correctness of the code. **Target Code Generation:** Target Code generation transforms AST into executable code (Java Bytecode and C/C++) **Error Handling:** Detecting syntax errors, semantic errors, and other issues in the code that violate the language rules throughout the process. **Symbol Table:** Symbol Table stores the information about variables and functions.

The syntax and semantics of the language are well-suited for the development of .DSL.

In a nutshell, we can describe SLANG language as follows

- Expressions are things evaluated for it's Value
- Statements are things which are executed for it's Effect
- Collection of Statements, Local Variables form a Procedure
- Collection of Procedures form a Module
- A Module is a Program

3.3.1 Lexical Analyser

The initial phase of every programming language processor endeavor involves constructing a Lexical Analyser. A Lexical Analyser, also known as a Lexer, is a piece of code which helps us to break the input into small pieces called Tokens. A token can represent various elements within a target language, such as keywords, string literals, identifiers, operators, colons, or any sequence of characters that holds a distinct meaning within the language.

The implementation of a Lexical Analyser often involves creating a custom code module, usually referred to as a "hand-crafted lexer". There is an option to write automatic Lexical Analysers by specifying regular expressions for token to a tool like JFLEX or ANother Tool For Language Recognition (ANTLR). But we chose a "hand-crafted lexer" to avoid certain minor complications which may come down the line. The custom Lexical Analyser module is responsible for breaking down the input source code into individual tokens. The lexer is structured around an infinite loop that iterates through the input source code until the entire code is processed. Within each iteration, the lexer identifies and extracts the next token. The lexer recognizes different types of tokens based on patterns. Like any other hand crafted lexer, whitespaces and comments are skipped before analyzing the lexemes. It begins by identifying single-character tokens, followed by two-character tokens and keywords, among other elements. The language encompasses nearly thirty distinct tokens. If the routine encounters characters or sequences that do not match any predefined token pattern, it may raise an error or issue a warning to indicate a potential lexical error in the source code. The Lexer functions as a coroutine to the Parser module, a topic that will be discussed in the subsequent sections.

3.3.2 Parser

The syntax of every programming language is defined using a Context Free Grammar (CFG). In common parlance, a CFG is a set of rules that specify how valid sentences of language can be formed by adhering to the guidelines set by the grammar. The grammar can be defined using a bottom-up approach, as seen in tools like Flex/Bison and JavaCC, or using a top-down approach, as demonstrated in ANTLR. The SLANG grammar is specified using a top-down grammar. Parsers can be implemented using automated parser generation tools such as ANTLR, or can choose to create a custom hand-crafted parser. In technical terms, the grammar of the SLANG has been designed in a way that it can be efficiently parsed by a top-down parser using a lookahead of one token. For the purpose of the thesis and to reduce the learning curve, a hand-crafted parser was selected as the preferred method.

The Parser uses a widely known method called Recursive Descent Parsing. The Recursive Descent Parser follows a top-down approach, beginning with the Goal Symbol (PROGRAM), and recursively descending through the production rules until reaching the leaf nodes. During the parsing process, the module generates a data structure known as Abstract Syntax Tree (AST), which will be elaborated upon in the subsequent explanation.

The entire grammar of the SLANG language is given the Appendix 6.1. But, a comprehensive understanding of the grammar is essential for a thorough explanation. The grammar serves as the foundation for parsing and plays a pivotal role in determining the structure and syntax of the language. It consists of a set of production

rules that define how valid sentences or expressions are constructed within the language. We utilize the Backus Naur Form (BNF) formalism to present the grammar for the SLANG language. BNF is a widely accepted notation for expressing context-free grammars and serves as a concise and precise method for describing the syntax of programming languages, command-line tools, and file formats, among others.

Productions for the Operators

```
<LOGIC_OP> := '&&' | '||'
<REL_OP> := '>' |' < '|' >=' |' <=' |' <>' |' =='
<MUL_OP> := '*' |' /'
<ADD_OP> := '+' |' -'
```

In every programming language, expressions are organized in a hierarchical manner. In the context of top-down parsing, operators with lower precedence are positioned at the root of the expression hierarchy. As the parsing process descends down the tree towards the leaf nodes, operators of higher precedence are represented.

The Expression hierarchy of the parser are

- Boolean Expressions
- Relational Expressions
- Term (things which you add often, as per Algebra)
- Factor (things which you multiply, as per Algebra)

Productions for the Expression

```
<expr> ::= <BExpr>
<BExpr> ::= <LExpr> LOGIC_OP <BExpr>
<LExpr> ::= <RExpr> REL_OP <LExpr>
<RExpr> ::= <Term> ADD_OP <RExpr>
<Term>::= <Factor> MUL_OP <Term>
<Factor> ::= <Numeric> | <String> | TRUE | FALSE | <variable> | '(' <expr> ')' |
    ↪ {+|-|!} <Factor> | <callexpr>
<callexpr> ::= funcname '(' actuals ')'
```

In any imperative programming language, statements are executed for its effects on the variable present in the current environment(context). Statements can be classified as immutable and mutable statements. The assignment statement is the only mutable statement supported by the system. The statement supported are

- Print Statement
- PrintLine Statement
- Return Statement
- Assignment Statement

- If Statement
- While Statement
- Function Call Statement

The BNF grammar for the Statements are

```
<stmts> := { stmt }+
{stmt} := <vardeclstmt> | <printstmt>|<printlnstmt>
<assignmentstmt>|<callstmt>|<ifstmt>|
<whilestmt> | <returnstmt>
<vardeclstmt> ::= <type> var_name;
<printstmt> := PRINT <expr>;
<assignmentstmt>:= <variable> = value;
<ifstmt>::= IF <expr> THEN <stmts> [ ELSE <stmts> ] ENDIF
<whilestmt>::= WHILE <expr> <stmts> WEND
<returnstmt>:= Return <expr>
```

In SLANG, procedures (functions) are modelled as a collection of statements with local variables. The collection of procedures are treated as a single compilation unit and are termed as modules. A module with at least one Main function is a program and thus far SLANG is a single module programming language. The organizing constructs supported by the programs are

- Procedures
- Module
- Program

Productions for the Procedure, Module and Program

```
<Module> ::= {<Procedure>}+;
<Procedure>::= FUNCTION <type> func_name '(' arglist ')'
<stmts>
END
<type> := NUMERIC | STRING | BOOLEAN | INTEGER
arglist ::= '(' {} ')' | '(' <type> arg_name [, arglist ] ')'
```

When a parse error occurs, the parser typically generates an error message or exception, indicating an error was detected.

3.3.3 Abstract Syntax Trees

The AST is a hierarchical structure used by almost all compilers to represent the constructs which are relevant for downstream processing. After parsing, the parser's primary role is to create an AST that represents the

syntactic structure of the source code. The AST captures the hierarchical relationships between code elements without necessarily producing executable code. It provides an abstract, organized representation of the code's syntax, enabling easier analysis, manipulation, and interpretation by compilers, interpreters, and other language processing tools.

The SlangForDotNet compiler[14] was originally written in .NET language and it is ported to Java. The original compiler hard coded the actions into the AST nodes. For this project, refactored the code to support the Gang Of Four (GOF) composite/ visitor pattern more on this discussed below section. In other words, AST data structure is organised as composite and processing nodes moved to the visitor implementations. The AST generation, type checking of the AST, transcompilation to C++ and Java bytecode generation were implemented as visitors.

Once the rules are generated from the decision tree, the compiler parses these rules to create an AST that corresponds to the rules. Subsequently, the subsequent stage involved performing type checking to identify any potential syntactic errors. In the absence of errors, the option becomes available to proceed with transcompilation to generate either C++ code or Java bytecode by leveraging Apache BCEL.

Writing modular back-ends

Traditionally, the AST are modelled as a composite (part/whole hierarchy) entity. The processing of the AST is modelled using visitors. The Composite Pattern is a structural design pattern that allows to the treatment of individual objects and compositions of objects uniformly. It's often used for building tree-like structures whereas the Visitor Pattern is a behavioral design pattern that allows adding further operations to objects without having to modify them. It is particularly useful when we have a complex object structure composed of different classes, and allows us to perform various operations on those objects without altering their code. In 1994 , Addison Wesley published a book titled, "The design Patterns- Elements of Reusable Object-Oriented Software" [9]. The book contains twenty three design patterns organised into structural, behavioral and creational Patterns. The builder and factory method falls under the creational Pattern. The composite and facade are a structural pattern. Visitor and interpreter are behavioral Patterns.

In effect structure is modelled as composites and behaviors are implemented using visitors. For a single AST modelled as pure data structure we can have multiple visitor implementations. The visitors are implemented for type checking, interpretation, transcompilation to C/C++ and compilation to Java bytecode. To understand this, one requires some familiarity with Gang Of Four design patterns [9]. This is mandatory because we leverage patterns like Builder, Composite, Visitor, Facade, Interpreter, Factory method etc while implementing the system.

In the AST of SLANG programming language, there exists a collection of twenty-four nodes that are organized in a hierarchical structure.

(a) Binary Expression

A Binary Expression is a type of node that represents an operation between two operands.

i. Binary Division

ii. Binary Multiplication

iii. Binary Plus

iv. Binary Minus

(b) Boolean Expression

A Boolean Expression is a type of expression that evaluates to a Boolean value (true or false). Boolean expressions are commonly used to model conditions, comparisons, and logical operations in programming.

i. Logical Expression

ii. Logical Not

iii. Relational Expression

(c) Constant

A Constant represents a fixed value that doesn't change during the execution of a program. Constants can include numerical values, string literals, boolean values, and other fixed data that is used in the code.

i. Boolean Constant

ii. Integer Constant

iii. Numeric Constant

iv. String Literal

(d) Unary Expression

A Unary Expression is an expression that involves a single operand and a unary operator. Unary operators are used to perform operations on a single value, such as negating a number or inverting a boolean value.

i. Unary Minus

ii. Unary Plus

(e) Call Expression

A Call Expression represents a function or method call. It indicates that we are invoking a specific function or method and passing arguments to it. Call Expressions are used to trigger the execution of a named routine or procedure.

(f) Statement

A Statement represents a single executable action or instruction within a program. Statements are used to describe the behavior and control flow of a program. Different types of statements perform different tasks, such as assignments, control flow, function calls, and more.

i. Assignment Statement

ii. If Statement

iii. PrintLine Statement

iv. Print Statement

v. Return Statement

vi. Variable Declaration Statement

vii. While Statement

(g) Proc

A Proc node contains a compilation unit which refers to a source code entity, a procedure which is a named code block, and a module which organizes related code elements. All three concepts contribute to well-structured and organized code in various programming languages.

i. Compilation Unit

ii. Procedure

iii. Module

The visitor of SLANG language contains a set of four nodes organised as a hierarchy.

(a) Semantic Visitor

Semantic Visitor allows us to integrate semantic checks into the process of traversing the AST such as type checking, symbol table population, scope resolution, and more. It allows us to associate semantics with the structure of the AST nodes and their interactions.

i. Type Check Visitor

A Type Check Visitor is a component used in software systems that implement the Visitor Pattern to perform type checking on elements of an AST. The Type Check Visitor focuses on ensuring the correct types are used within the AST. If type violations are found during traversal, the Type Check Visitor reports errors indicating type mismatches occurred.

(b) Expression Visitor

The Expression Visitor pattern is designed to separate the evaluation or analysis of expressions from the structure of the AST itself. This allows adding new operations or evaluations to expressions without altering the existing code for constructing or navigating the AST.

i. Bytecode Generator

A bytecode generator is a component in a programming language's compilation process that translates source code into Java bytecode, which is the intermediate representation that can be executed by the JVM.

ii. C Code Generator

A C code generator is a component that takes source code, such as program specifications or models, and generates corresponding C / C++ programming language code. The generated code can then be compiled and executed on a C compiler.

iii. Tree Evaluator (Interpreter)

Tree Evaluator or interpreter is a software component that reads and executes code directly, line by line, without the need for prior compilation. It translates source code instructions into machine-level instructions on-the-fly and executes them immediately.

3.3.4 Symbol Table

Throughout the compilation process, the compiler maintains a symbol table that stores information about symbols such as variables, functions, constants, and other language-specific constructs, encountered in the source code. The symbol table helps keep track of these symbol's attributes, such as their names, types, scopes and other relevant information.

3.3.5 Error Handling

The compiler performs error detection and reporting throughout the entire compilation process. If any syntax, semantic, or other errors are detected, appropriate error messages are generated to help the programmer identify and fix issues.

3.4 Generating Java Bytecode And C/C++ Code

3.4.1 Java Bytecode

The JVM is a crucial component of the Java platform. It's an abstract, machine-independent execution environment that provides the foundation for running Java bytecode. JVM is responsible for translating bytecode into machine code, managing memory, and executing Java applications. The JVM utilizes a stack-based architecture as part of its runtime environment. This architecture involves the use of a stack data structure to manage the execution of bytecode instructions. An overview of the stack-based virtual machine architecture of the JVM:

(a) Operand Stack

The heart of the stack-based architecture is the operand stack, which holds values that are used as operands for instructions. Each frame (corresponding to a method invocation) has its own operand stack. Instructions in bytecode often manipulate values on this stack.

(b) Frame

A frame represents the state of a method invocation, including local variables, operand stack, and other information required for execution. Frames are pushed onto the call stack when a method is called and popped off when the method returns.

(c) Local Variables

Each frame contains space for local variables used within the method. Local variables store values that are used within the method's scope, similar to variables in traditional programming languages.

(d) Method Area

The method area stores class-level information, including bytecode instructions, method information, constant pool, field details, and static variables. It is shared among all threads.

(e) Constant Pool

The constant pool is a table of constants used in a class. It includes literals, symbolic references, and other constants needed for the execution of the program.

(f) Instructions

JVM bytecode consists of a set of instructions that manipulate the operand stack, access local variables, and perform various operations. Instructions are designed to be compact and efficient for execution.

(g) Execution

During execution, the JVM retrieves bytecode instructions from the method area, interprets them, and performs operations using the operand stack and local variables. The stack-based nature of the architecture simplifies instruction decoding and helps manage memory usage efficiently.

(h) Method Invocation and Return

When a method is invoked, a new frame is created on the call stack to store the method's state. When the method returns, its frame is removed from the stack.

Apache BCEL serves as a Java library that equips developers with a comprehensive suite of tools and APIs to facilitate the analysis, manipulation, and generation of Java bytecode. This is useful for dynamically creating classes or modifying existing ones at runtime. BCEL finds specific applications in tasks involving direct engagement with bytecode. This pertains to scenarios such as creating custom JVM-based languages, building tools for code analysis or transformation, or developing bytecode manipulation frameworks. BCEL is one of several available bytecode engineering libraries for Java. Other notable libraries include ASM (ObjectWeb ASM) and Javassist. These libraries provide similar capabilities and are often used in different projects depending on specific requirements and preferences.

For this project we chose Apache BCEL because of its simplified programming model and we are only using a small subset of BCEL. We wrote a visitor for generating Java bytecode (code snippet given 6.1).

Creating .class file

The process of generating a .class file is as follows.

- Create a file stream as output

  This step involves creating a FileOutputStream to write data to a file. This stream will be used to write the generated bytecode to a .class file on the disk.

- Generate Code in the classGen buffer

  Use BCEL to programmatically create and define the bytecode structure of the class. This includes defining classes, methods, fields, and their instructions. The generated bytecode is stored in a byte array (referred to as the classGen buffer).

- Persist buffer into the disc file created above

  After generating the bytecode and storing it in the classGen buffer, write this buffer to the previously created .class file using the FileOutputStream. This process effectively saves the generated bytecode to a physical file on the disk.

The code snippet given 6.2 demonstrates the process.

Package .class file into JAR

To package .class files into a JAR programmatically using Java, we can use the java.util.jar package. Packaging .class files into a JAR (Java Archive) file is a common practice in Java development. A JAR file is a compressed archive format that can contain compiled Java class files, resources, and metadata. It's used to bundle Java classes and related files together for distribution, deployment, and reuse. Here's 6.4 a shows how to create a JAR file and add .class files to it using command line.

3.4.2 Transcompiler

A transcompiler, also known as a source-to-source compiler or transpiler, is a type of compiler that translates source code from one programming language to another. Unlike traditional compilers that translate source code into machine code or intermediate code for execution, a transpiler generates equivalent source code in a different programming language. They enable developers to take advantage of the strengths of different languages without rewriting entire applications from scratch. Transcompilers can be particularly useful when transitioning from one programming language to another while preserving existing code logic, or when targeting multiple platforms with a single codebase.

A visitor was successfully developed to generate ANSI C/C++ code from the AST generated by the composite structure. The resulting code was compiled into a .so (.dll for Windows Systems) file using the GNU Compiler Collection (GCC) compiler.

Create .cpp file

The process of generating .class file is as follows.

- Create a BufferedWriter character output stream

  Start by creating a BufferedWriter character output stream to write text data to a file.

- Generate C Code

  Generate C++ code as a string by replacing Java methods in the original code with their equivalent C++ counterparts. Since C++ has distinct standard libraries and APIs, this involves rewriting parts of the code to use C++ features.

- Persist buffer into the disc file created above

  After generating the C++ code string, write this string to the file using the BufferedWriter. This action effectively saves the generated C++ code to a physical file on the disk.

The code snippet given 6.3 demonstrates the process.

Package .cpp file into .so file

Packaging a .cpp file into a .so file involves several steps, including compiling the C++ code into object files and then linking them into a shared library. Here's a general outline of the process (reference 6.12):

- Compile to Object File

  Compile your C++ code into object files (.o) using a C++ compiler (such as g++ for GCC). Object files are intermediate binary files that contain the compiled code from the source files. This step generates machine code specific to your platform.

- Create Shared Library

  Using the same compiler, create the shared library (.so) by linking the object file. The generated shared library file contains the compiled and linked code from the .cpp file which can be used in other programs.

CHAPTER 4

Dataset

The data set [17] was gathered from the Iraqi society, with a specific acquisition from the laboratory of Medical City Hospital. The diabetes dataset consists of patient data including medical information and laboratory analysis. This dataset includes the following features.

- No_Pation: A numerical identifier for each patient in the dataset.
- Age: The age of the patient in years.
- Gender: The gender of the patient (e.g., Male or Female).
- Creatinine ratio (Cr): The ratio of creatinine, a waste product in the blood, typically used to assess kidney function.
- Body Mass Index (BMI): A measure of body fat based on height and weight.
- Urea: The level of urea, a waste product produced by the liver, in the patient's blood.
- Cholesterol (Chol): The total cholesterol level in the patient's blood.
- Fasting lipid profile: A set of measurements related to lipid levels in the blood, including LDL (low-density lipoprotein), VLDL (very low-density lipoprotein), Triglycerides (TG), and HDL (high-density lipoprotein) Cholesterol.
- HbA1c: The level of glycated hemoglobin, which provides information about average blood sugar levels over a few months.
- Class: The class or category of the patient's diabetes disease, which can be Diabetic, Non-Diabetic, or Predict-Diabetic.

These features serve as important variables for analyzing and predicting diabetes disease class based on the provided dataset. They provide insights into patients' health, medical measurements, and potential risk factors associated with diabetes. Thousand observations were collected and classified as Diabetic (Y), Non-Diabetic (N)

and Predicted-Diabetic (P). The table 4.1 provides an overview of the dataset's structure, while the 4.2 section displays the details of the first six rows within the dataset.

Table 4.1: Dataset Structure

ID	int
No_Pation	int
Gender	char
AGE	int
Urea	num
Cr	int
HbA1c	num
Chol	num
TG	num
HDL	num
LDL	num
VLDL	num
BMI	num
CLASS	char

Table 4.2: First 6 rows of the dataset

ID	No_Pation	Gender	AGE	Urea	Cr	HbA1c	Chol	TG	HDL	LDL	VLDL	BMI	CLASS
1	502	17975	F	50	4.7 46	4.9	4.2	0.9	2.4	1.4	0.5	24	N
2	735	34221	M	26	4.5 62	4.9	3.7	1.4	1.1	2.1	0.6	23	N
3	420	47975	F	50	4.7 46	4.9	4.2	0.9	2.4	1.4	0.5	24	N
4	680	87656	F	50	4.7 46	4.9	4.2	0.9	2.4	1.4	0.5	24	N
5	504	34223	M	33	7.1 46	4.9	4.9	1.0	0.8	2.0	0.4	21	N
6	634	34224	F	45	2.3 24	4.0	2.9	1.0	1.0	1.5	0.4	21	N

CHAPTER 5

Results And Discussion

The implementation of a tool that involves multiple technologies, including Java, Jython, Apache BCEL, and the g++ compiler was undertaken to empirically ground the research. Since the decision tree is a supervised algorithm, the total dataset was split into a training set and a test set in the ratio 70:30. The decision tree model was generated with a training set.

5.1 SLANG format generated from the decision tree model

The training set is provided through the command line. In Java, a preprocessor module parses the input line by line, creating a data structure compatible with Python. The data structure generated is injected into the Python script using Jython, which is a JVM implementation of Python. The injected data is transformed into Jython's intrinsic data structure using the Jython's AST module. Using the provided data structure as input, a decision tree is generated through the CART algorithm [19]. The decision tree is traversed recursively to create rules in SLANG format (reference 6.5).

5.2 Generation of C / C++ code and packaging as Shared Object

The SLANG script is parsed to generate an AST. The AST is traversed recursively using a transcompiler module, which generates cross-platform C/C++ code. The code can be compiled using an ANSI C /C++ compiler to generate object code for the desired target platform. To test the generated code, we compiled it on both GNU Linux (Ubuntu 20.04) and macOS (OSX). This demonstrates that the generated code (reference 6.6) is compatible with POSIX standards. A command-line program is developed to execute the .so module, allowing it to predict outcomes for individual data points (reference 6.8). We were able to generate consistent and identical outputs from both the decision tree module and the generated code (reference 6.11).

5.3 Generation of Java Bytecode and packaging as JAR file

Additionally, we developed a visitor implementation that utilizes the Apache BCEL library to generate Java bytecode. Upon running the code within the IntelliJ environment, it resulted in a .class file (reference 6.7). This generated .class file was then invoked from a separate calling program (reference 6.9). Following the compilation of the calling program, we bundled all the modules together into a Java executable archive. The calling program assessed the predicted outcomes for individual data points. We achieved consistent and identical outputs from both the decision tree module and the generated code (reference 6.10).

The primary concern was whether the generated rules would exhibit the predictive capacity of the decision trees. A sample program in Java is developed to assess the predictive capabilities of the generated code. Among the three hundred data points in the test set, only two were misclassified. This observation demonstrates that the decision tree and its corresponding rules are equivalent.

CHAPTER 6

Conclusion

6.1 Summary

Throughout this undertaking, a sophisticated tool was meticulously crafted to produce resilient and dependable outputs. This tool proficiently generates platform-specific binaries utilizing the GCC compiler and executable JAR files. It is designed for constructing machine learning models based on the CART algorithm. The models produced by this tool are versatile in deployment, suitable for various environments. They can be deployed in POSIX environments using .so, Windows environments using .dll, and enterprise settings through JAR files. Currently, this framework is customized for the Decision Tree algorithm, but its applicability extends to a broader spectrum. It holds the potential to be expanded to encompass other classification algorithms such as Apriori, Linear Regression, and beyond.

The entire undertaking has the potential to yield a collection of reusable artifacts, streamlining the process of deploying models in real-world machine learning projects. The identical approach can be employed for models generated by robust industrial packages like Scikit-Learn and Weka. By integrating this module into Weka and Scikit-Learn, the system can be elevated to a production-level status.

6.2 Future Roadmap

The diagram 6.1 illustrates the potential future enhancements of the tool through expansion and augmentation. Those are

- Extending the transcompiler logic involves enhancing the process that translates AST into equivalent Java or C# code. The integration of transcompilers into the Java source code or C# code, introduces a significant avenue for optimizing code execution. The existing tool does not perform any code optimization. Creating transcompilers for Java and C# enables the utilization of optimization functionalities present in well-established production-level compilers like OpenJDK (for Java) and Microsoft's .NET compilers (for C#). This approach has the potential to result in more efficient code generation, as it involves transferring the

responsibility for optimization to mature and extensively tested compiler toolchains.

- Utilizing the Low-Level Virtual Machine (LLVM) infrastructure instead of exclusively relying on C/C++ code enables the generation of direct binaries with enhanced performance. Furthermore, these binaries have the advantage of being adaptable to various platforms.

- Generating Intermediate Language (IL) code specifically tailored for the .NET platform involves producing code that conforms to the Common Intermediate Language (CIL), which is the low-level bytecode representation used by the .NET runtime. This bytecode is executed by the .NET .Common Language Runtime (CLR).

- Decision Trees can indeed be used as components within ensemble methods, which are techniques that combine multiple models to create a more powerful predictive model. The current implementation generates a rule set from a given tree. When working with ensembles, it is necessary to create distinct rule sets for the various trees generated by the algorithm. In the context of Random Forest, which employs Bagging (Bootstrap Aggregation), the predictor is applied to multiple rule sets, each corresponding to different trees, and a voting mechanism is used to make predictions.

- Another ensemble learning algorithm is Boosting. Boosting is a machine learning ensemble technique that aims to improve the performance of weak learners by combining their predictions. This tool can be extended to accommodate Boosting, which typically involves the creation of a single composite model through iterative refinement.

- Stacking is an ensemble machine learning technique which involves training a variety of base models, often using different algorithms or parameter settings. Extending this methodology to encompass Stacking is outside the scope of this project, even though certain algorithms, such as Linear Regression and the Apriori Algorithm, may be applicable in a Stacking context.

- Supporting both PMML and PICKLE encoded decision trees from libraries like Scikit-Learn and Weka which involves implementing the necessary code to convert decision trees generated by these libraries into the SLANG format. Also support rule generation by traversing Scikit-Learn and Weka decision tree models, which needs to implement code that can extract decision rules from these models.

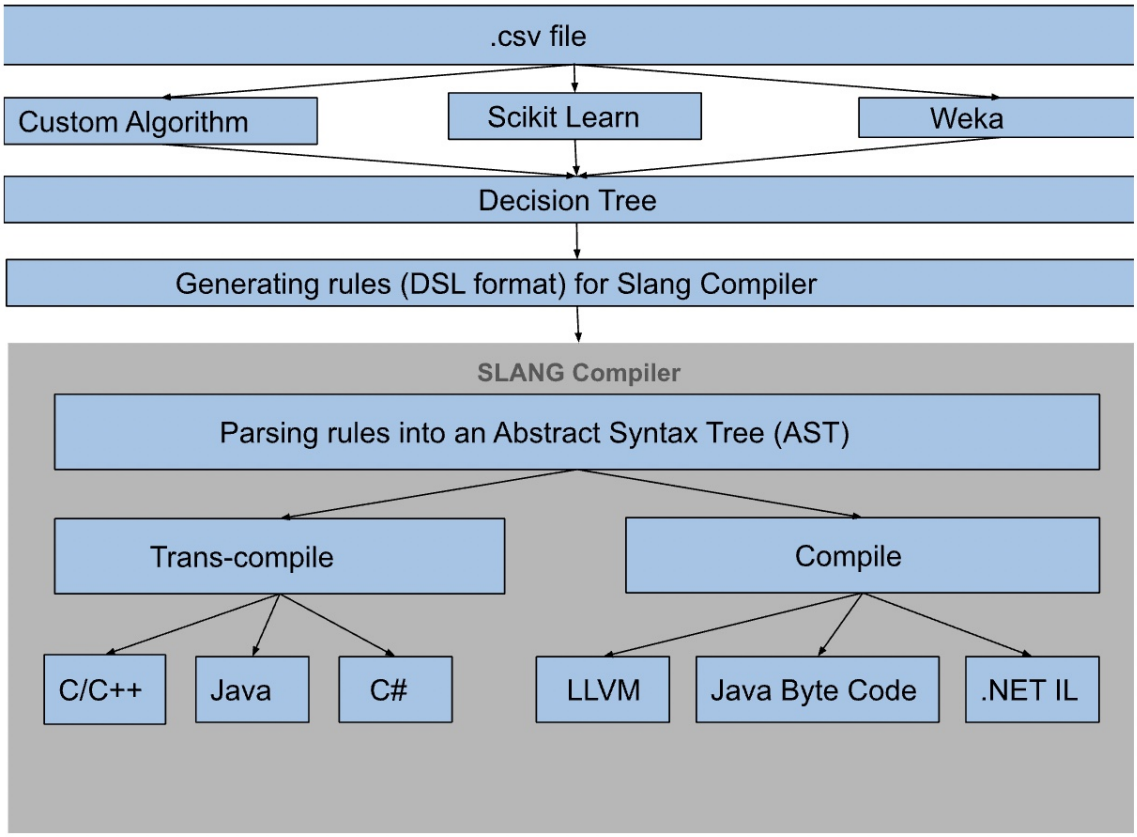

Figure 6.1: Future Roadmap. **Step 1:** Loading the tabular data as CSV file. **Step 2:** Generating Decision Tree Model from CSV file generated from Custom Algorithm, Scikit-Learn or Weka libraries. **Step 3:** Transforming decision tree to generate rules SLANG format **Step 4:** SLANG compiler parses rules into AST. **Step 5:** Tanscompilation of AST into C/C++ code or Java code or C# code and compilation of AST into Java Bytecode or LLVM or .NET Intermediate Language (IL).

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techiques and Tools*. Pearson Education, Inc, 2 edition, 2007.

[2] Rajendra Akerkar and Priti Srinivas Sajja. *Intelligent Techniques for Data Science*. Springer International, 2016.

[3] SÃ©rgio Branco, Carlos Ferreira, JoÃ£o Carvalho, Bruno Gaspar, and Jorge Cabral. Clara: Transpiler for cloud built machine learning models into resource-scarce embedded systems. In *IECON 2022 â 48th Annual Conference of the IEEE Industrial Electronics Society*, pages 1–6, 2022.

[4] Jason Brownlee. *Ensemble Learning Algorithms With Python*. v1.1 edition, 2021.

[5] Yingnong Dang, Qingwei Lin, and Peng Huang. Aiops: Real-world challenges and research innovations. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 4–5, 2019.

[6] Deepankar. Decision tree with cart algorithm. https://medium.com/geekculture/decision-trees-with-cart-algorithm-7e179acee8ff, 2021.

[7] Apache Software Foundation. Apche commons bcel. https://commons.apache.org/, 2022. Version 6.7.0.

[8] Martin Fowler and Rebecca Parsons. *Domain Specific Languages*. Addison-Wesley Professional, 1 edition, 2011.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 2017.

[10] Debashish Ghosh. *DSLs in Action*. Manning Publications, 2010.

[11] Roua Jabla, Maha Khemaja, FÃ©lix Buendia, and Sami Faiz. Automatic rule generation for decision-making in context-aware systems using machine learning. 2022(5202537):13, 2022.

[12] Tim Lindholm and Alex Buckley Frank Yellin Gilad Bracha. *The Java Virtual Machine Specification*. Taylor Francis Group, LLC, 7 edition, 2013.

[13] Andreas C. MuÌller and Sarah Guido. *Introduction to Machine Learning with Python*. O'Reilly Media, Inc., 1 edition, 2017.

[14] Praseed Pai. Compiler. https://github.com/praseedpai/SlangForDotNet/tree/master/STEP7, 2020.

[15] Praseed Pai and Shine Xavier. *.NET Design Patterns*. Packt Publishing Ltd., 1 edition, 2017.

[16] Paleyes, Andrei, Urma, Raoul-Gabriel, and Neil D. Lawrence. Challenges in deploying machine learning: A survey of case studies. *ACM Comput. Surv.*, 55(6), dec 2022.

[17] Ahlam Rashid. Diabetes dataset. https://data.mendeley.com/datasets/wj9rwkp9c2/1, 2020. Mendeley Data, V1.

[18] Anshul Saini. Decision tree algorithm â a complete guide. https://www.analyticsvidhya.com/blog/2021/08/decision-tree-algorithm/, 2021. Last Modified On July 12th, 2023.

[19] Toby Segaran. *Programming Collective Intelligence*. O'Reilly Media, Inc., 2 edition, 2007.

[20] Xindong Wu and Vipin Kumar. *The Top Ten Algorithms in Data Mining*. Taylor Francis Group, LLC, 1 edition, 2009.

Appendix

Table 6.1: The Backus Naur Form (BNF) Grammer for Slang Compiler

```
<Module> ::= {<Procedure>}+;
<Procedure>::= FUNCTION <type> func_name '(' arglist ')'
<stmts> END
<type> := NUMERIC | INTEGER | STRING | BOOLEAN
arglist ::= '(' {} ')' | '(' <type> arg_name [, arglist ] ')'
         <stmts> := { stmt }+
{stmt} := <vardeclstmt> | <printstmt>|<printlnstmt>
      <assignmentstmt>|<callstmt>|<ifstmt>| <whilestmt> | <returnstmt>
<vardeclstmt> ::= <type> var_name;
<printstmt> := PRINT <expr>;
<assignmentstmt>:= <variable> = value;
<ifstmt>::= IF <expr> THEN <stmts> [ ELSE <stmts> ]
      ENDIF <whilestmt>::= WHILE <expr> <stmts>
      WEND <returnstmt>:= Return <expr>
<expr> ::= <BExpr>
<BExpr> ::= <LExpr> LOGIC_OP <BExpr>
<LExpr> ::= <RExpr> REL_OP <LExpr>
<RExpr> ::= <Term> ADD_OP <RExpr>
<Term>::= <Factor> MUL_OP <Term>
<Factor> ::= <Numeric> | <String> | TRUE | FALSE | <variable> |
         '(' <expr> ')' | {+|-|!} <Factor> | <callexpr>
<callexpr> ::= funcname '(' actuals ')'
<LOGIC_OP> := '&&' | '||'
<REL_OP> := '>' |' < '|' >=' |' <=' |' <>' |' =='
<MUL_OP> := '*' |' /'
<ADD_OP> := '+' |' -'
```

Listing 6.1: Bytecode Generator Visitor Snippet

```
public class BCGeneratorVisitor implements IExpressionVisitor {
   @Override
   public SymbolInfo visit(Procedure procedure, RUNTIME_CONTEXT ctx,
                     ArrayList<Expression> actualParameterExpressions) throws
                        ↪ Exception {
      ...... logic goes here
   }
   @Override
   public SymbolInfo visit(NumericConstant num, RUNTIME_CONTEXT ctx) throws
      ↪ Exception {
      //// logic goes here
   }
   ........
   public SymbolInfo visit(ReturnStatement returnStatement, RUNTIME_CONTEXT ctx)
                     throws Exception {
      //// logic goes here
   }
}
```

Listing 6.2: Snippet that generates .class File

```
   @Override
   public SymbolInfo visit(TModule tmodule, RUNTIME_CONTEXT ctx) throws
      ↪ Exception {
      String className = tmodule.getName().toUpperCase() + ".class";
      FileOutputStream outputStream = new FileOutputStream(className);

      // Creates bytecode generation context for module
      BYTECODE_CONTEXT context = new BYTECODE_CONTEXT(
            tmodule, outputStream);
      context.SetModule(tmodule);

      // Iterates through the function and generates equivalent bytecode
      if (tmodule.getProcedures() != null) {
         for (Object p : tmodule.getProcedures())
         {
            Procedure procedure =(Procedure)p;
            // Visits the procedure node
```

```
            procedure.accept(this, context, null);
         }
      }


      // Generates bytecode and writes it file
      context.generate(outputStream);
      return null;
 }

//// rest of bytecode generation code omitted for breviety

public void generate(OutputStream out) throws IOException {
      classGen.getJavaClass().dump(out);
}
```

Listing 6.3: C Code Generator Visitor Snippet

```
public class CGeneratorVisitor implements IExpressionVisitor{
  @Override
   public SymbolInfo visit(TModule tmodule, RUNTIME_CONTEXT context) throws
      ↪ Exception {
      /// logic goes here
   }

   @Override
   public SymbolInfo visit( Procedure procedure, RUNTIME_CONTEXT context,
                       ArrayList<Expression> actualParameterExpressions) throws
                          ↪ Exception {
      /// logic goes here
   }

   ........

   @Override
   public SymbolInfo visit(ReturnStatement returnStatement, RUNTIME_CONTEXT
      ↪ context)
         throws Exception {

      /// logic goes here
```

```
    }
}
```

Listing 6.4: Generating jar file

```
javac Caller.java
jar cfe Model.jar Caller Caller.class DATASET.class Helper/Utils.class

Caller.java file contains the function thet calls the generated .class function.
    ↪ The .class file depends on the Util class and three classes are packaged
    ↪ into Model.jar file
```

Listing 6.5: Generated SLANG Format Generated From Decision Tree Model

```
FUNCTION STRING CMD_GET_S(NUMERIC OFFSET, STRING s )
    return s;
END
FUNCTION NUMERIC CMD_GET_I(NUMERIC OFFSET,STRING s )
    return 1;
END
FUNCTION NUMERIC CMD_GET_D(NUMERIC OFFSET, STRING s )
    return 0.0;
END
FUNCTION BOOLEAN CMD_GET_B(NUMERIC OFFSET,STRING s )
    return TRUE;
END
FUNCTION STRING PREDICT(STRING ARGS)

NUMERIC Var_9;
NUMERIC Var_8;
NUMERIC Var_1;
NUMERIC Var_0;
NUMERIC Var_3;
NUMERIC Var_12;
NUMERIC Var_4;
NUMERIC Var_7;
NUMERIC Var_6;
Var_9 = CMD_GET_D(9,ARGS);
Var_8 = CMD_GET_D(8,ARGS);
```

```
Var_1 = CMD_GET_D(1,ARGS);
Var_0 = CMD_GET_D(0,ARGS);
Var_3 = CMD_GET_D(3,ARGS);
Var_12 = CMD_GET_D(12,ARGS);
Var_4 = CMD_GET_D(4,ARGS);
Var_7 = CMD_GET_D(7,ARGS);
Var_6 = CMD_GET_D(6,ARGS);
IF ( Var_6 >= 6.5) THEN
 return "Y";
ELSE

 IF ( Var_6 >= 5.7) THEN
  IF ( Var_3 >= 51) THEN
   IF ( Var_8 >= 2.9) THEN
    IF ( Var_0 >= 363) THEN
     return "Y";
    ELSE

     return "P";
    ENDIF
   ELSE

    return "Y";
   ENDIF
  ELSE

   return "P";
  ENDIF
 ELSE

  IF ( Var_12 >= 25) THEN
   return "Y";
  ELSE

   IF ( Var_7 >= 5) THEN
    IF ( Var_8 >= 2) THEN
     return "Y";
    ELSE
```

```
       IF ( Var_4 >= 7.7) THEN
         return "Y";
       ELSE

         IF ( Var_9 >= 1.6) THEN
           IF ( Var_1 >= 45383) THEN
             return "N";
           ELSE

             return "Y";
           ENDIF
         ELSE

           return "N";
         ENDIF
       ENDIF
     ENDIF
    ELSE

      return "N";
    ENDIF
   ENDIF
 ENDIF
ENDIF
END
```

Listing 6.6: Generated C / C++ Code

```
#include <iostream>

#include <string>

#include <string.h>

using namespace std;

string predict(string ARGS);
void find_str(string s, string del, int offset, string & result) {
  // Use find function to find 1st position of delimiter.
```

```
  int end = s.find(del);
  int i = 0;
  while (end != -1) { // Loop until no delimiter is left in the string.
    string temp = s.substr(0, end);
    if (i++ == offset) {
      result = temp;
      return;
    }
    s.erase(s.begin(), s.begin() + end + 1);
    end = s.find(del);
  }
  result = s.substr(0, end);
}

string interpret(int offset, string args) {

  string ** arr = new string * [100];
  // for(int i=0;i<100;++i) { *arr[i] = (string)""; }

  int cnt = 0;
  string result = "";
  find_str(args, ",", offset, result);
  return result;

}

string cmd_get_s(int offset, string args) {

  return interpret(offset, args);
}

int cmd_get_i(int offset, string S) {
  string n = interpret(offset, S);
  return atoi(n.c_str());
}

double cmd_get_d(int offset, string S) {
  string n = interpret(offset, S);
```

```
  return atof(n.c_str());
}

bool cmd_get_b(int offset, string S) {
  string n = interpret(offset, S);
  return n == "TRUE" ? true : false;
}

extern "C"
bool predict_call(const char * str, char * str2) {
  string ret = predict(string(str));
  strcpy(str2, ret.c_str());
  return true;
}

int main(int argc, char ** argv) {
  if (argc <= 1) {
    return 0;
  }
  string strs = "";
  for (int i = 1; i < argc; ++i) {
    strs += string(argv[i]) + "|";
  }
  cout << strs << endl;
  string rs = predict(strs);
  cout << "Predicted␣....." << rs << "$$$" << endl;
  return 0;

}

string predict(string ARGS) {
  double VAR_9;
  double VAR_8;
  double VAR_1;
  double VAR_0;
  double VAR_3;
  double VAR_12;
  double VAR_4;
  double VAR_7;
```

```
double VAR_6;
VAR_9 = cmd_get_d(9.0, ARGS);
VAR_8 = cmd_get_d(8.0, ARGS);
VAR_1 = cmd_get_d(1.0, ARGS);
VAR_0 = cmd_get_d(0.0, ARGS);
VAR_3 = cmd_get_d(3.0, ARGS);
VAR_12 = cmd_get_d(12.0, ARGS);
VAR_4 = cmd_get_d(4.0, ARGS);
VAR_7 = cmd_get_d(7.0, ARGS);
VAR_6 = cmd_get_d(6.0, ARGS);
if (VAR_6 >= 6.5) {
  return "Y";

} else {
  if (VAR_6 >= 5.7) {
    if (VAR_3 >= 51.0) {
      if (VAR_8 >= 2.9) {
        if (VAR_0 >= 363.0) {
          return "Y";

        } else {
          return "P";
        }

      } else {
        return "Y";
      }

    } else {
      return "P";
    }

  } else {
    if (VAR_12 >= 25.0) {
      return "Y";

    } else {
      if (VAR_7 >= 5.0) {
        if (VAR_8 >= 2.0) {
```

```
          return "Y";

        } else {
          if (VAR_4 >= 7.7) {
            return "Y";

          } else {
            if (VAR_9 >= 1.6) {
              if (VAR_1 >= 45383.0) {
                return "N";

              } else {
                return "Y";
              }

            } else {
              return "N";
            }

          }

        }

      } else {
        return "N";
      }

    }

  }

}
```

Listing 6.7: Generated Java Bytecode

```
import Helper.Utils;
```

```
public class DATASET {
   public static String predict(String arg0) {
      double var1 = 0.0;
      double var3 = 0.0;
      double var5 = 0.0;
      double var7 = 0.0;
      double var9 = 0.0;
      double var11 = 0.0;
      double var13 = 0.0;
      double var15 = 0.0;
      double var17 = 0.0;
      var1 = Utils.cmd_get_d(9.0, arg0);
      var3 = Utils.cmd_get_d(8.0, arg0);
      var5 = Utils.cmd_get_d(1.0, arg0);
      var7 = Utils.cmd_get_d(0.0, arg0);
      var9 = Utils.cmd_get_d(3.0, arg0);
      var11 = Utils.cmd_get_d(12.0, arg0);
      var13 = Utils.cmd_get_d(4.0, arg0);
      var15 = Utils.cmd_get_d(7.0, arg0);
      var17 = Utils.cmd_get_d(6.0, arg0);
      double var19;
      if ((double)((var19 = var17 - 6.5) == 0.0 ? 0 : (var19 < 0.0 ? -1 : 1)) !=
         ↪ -1.0) {
         return "Y";
      } else {
         boolean var10000 = true;
         double var20;
         if ((double)((var20 = var17 - 5.7) == 0.0 ? 0 : (var20 < 0.0 ? -1 : 1))
            ↪ != -1.0) {
            double var21;
            if ((double)((var21 = var9 - 51.0) == 0.0 ? 0 : (var21 < 0.0 ? -1 :
               ↪ 1)) != -1.0) {
               double var22;
               if ((double)((var22 = var3 - 2.9) == 0.0 ? 0 : (var22 < 0.0 ? -1 :
                  ↪ 1)) != -1.0) {
                  double var23;
                  if ((double)((var23 = var7 - 363.0) == 0.0 ? 0 : (var23 < 0.0
                     ↪ ? -1 : 1)) != -1.0) {
                     return "Y";
```

```
                } else {
                    var10000 = true;
                    return "P";
                }
            } else {
                var10000 = true;
                return "Y";
            }
        } else {
            var10000 = true;
            return "P";
        }
    } else {
        var10000 = true;
        double var24;
        if ((double)((var24 = var11 - 25.0) == 0.0 ? 0 : (var24 < 0.0 ? -1 :
            ↪ 1)) != -1.0) {
            return "Y";
        } else {
            var10000 = true;
            double var25;
            if ((double)((var25 = var15 - 5.0) == 0.0 ? 0 : (var25 < 0.0 ? -1
                ↪ : 1)) != -1.0) {
                double var26;
                if ((double)((var26 = var3 - 2.0) == 0.0 ? 0 : (var26 < 0.0 ?
                    ↪ -1 : 1)) != -1.0) {
                    return "Y";
                } else {
                    var10000 = true;
                    double var27;
                    if ((double)((var27 = var13 - 7.7) == 0.0 ? 0 : (var27 <
                        ↪ 0.0 ? -1 : 1)) != -1.0) {
                        return "Y";
                    } else {
                        var10000 = true;
                        double var28;
                        if ((double)((var28 = var1 - 1.6) == 0.0 ? 0 : (var28 <
                            ↪ 0.0 ? -1 : 1)) != -1.0) {
                            double var29;
```

```
                        if ((double)((var29 = var5 - 45383.0) == 0.0 ? 0 : (
                            ↪ var29 < 0.0 ? -1 : 1)) != -1.0) {
                            return "N";
                        } else {
                            var10000 = true;
                            return "Y";
                        }
                    } else {
                        var10000 = true;
                        return "N";
                    }
                }
            }
        } else {
            var10000 = true;
            return "N";
        }
    }
}
```

Listing 6.8: C Test Function

```
#include <stdio.h>
#include <string.h>

extern "C"
bool predict_call(const char *ptr, char *);
int main(int argc, char **argv)
{
      char Temp[1024];
      if (argc <= 1)
      {
            printf("We need more command line arguments\n");
            return -1;
      }
```

```
      memset(Temp, 0, 1024);
      for (int i = 1; i < argc; i++)
      {
            strcat(Temp, argv[i]);
            strcat(Temp, ",");
      }

      char output[255];
      predict_call(Temp, output);
      printf("%s\n", output);
}
```

Listing 6.9: Java Test Function

```
import java.lang.*;
import Helper.*;
public class Caller
{
      public static void main(String [] args ){
            String rs = "";
            for(int c = 0; c < args.length; ++c )
                  rs += args[c]+",";
            System.out.println("Command␣Line␣:" + rs );
            String rst = DATASET.predict(rs);
            System.out.println("Predicted:␣" + rst);
      }


}
```

Listing 6.10: Packaging And Testing JAR File

```
(base) praseedpai@Praseeds-MBP STEP11FUNCTION_Latest % javac Caller.java
(base) praseedpai@Praseeds-MBP STEP11FUNCTION_Latest % jar cfe Model.jar Caller
   ↪ Caller.class DATASET.class Helper/Utils.class
(base) praseedpai@Praseeds-MBP STEP11FUNCTION_Latest % java -jar Model.jar 634
   ↪ 34224 'F' 45 2.3 24 4 2.9 1 1 1.5 0.4 21
Predicted: N
(base) praseedpai@Praseeds-MBP STEP11FUNCTION_Latest % java -jar Model.jar 191
   ↪ 454316 'M' 55 5.4 62 6.8 5.3 2 1 3.5 0.9 30.1
Predicted: Y
```

```
(base) praseedpai@Praseeds-MBP STEP11FUNCTION_Latest % java -jar Model.jar 466
    ↪ 26665 'F' 30 5.7 53 6 5.4 1.7 1.4 3.3 0.7 22
Predicted: P
```

Listing 6.11: Packaging And Testing Shared Object

```
\begin{lstlisting}
(base) praseedpai@Praseeds-MBP STEP11FUNCTION_Latest % g++ -shared -fPIC DATASET.
    ↪ cpp -o DATASET.so
\begin{lstlisting}
(base) praseedpai@Praseeds-MBP STEP11FUNCTION_Latest % g++ Caller.cpp ./DATASET.
    ↪ so
(base) praseedpai@Praseeds-MBP STEP11FUNCTION_Latest % ./a.out 700 87666 'M' 42
    ↪ 5.4 53 5.8 5.9 3.7 1.3 3.1 1.7 23
P
(base) praseedpai@Praseeds-MBP STEP11FUNCTION_Latest % ./a.out 634 34224 'F' 45
    ↪ 2.3 24 4 2.9 1 1 1.5 0.4 21 'N'
N
(base) praseedpai@Praseeds-MBP STEP11FUNCTION_Latest % ./a.out 510 34290 'M' 73
    ↪ 4.3 79 6.9 5.3 1.4 1.5 3.2 0.6 28
Y
```

Listing 6.12: Packaging C/C++ Code As Shared Object

```
g++ -shared -fPIC DATASET.cpp -o DATASET.o

where -c: Compile only, don't link.
-fPIC: Generate Position Independent Code, required for shared libraries.

g++ -shared -o DATASETLIB.so -o DATASET.o

where -shared: Create a shared library.
-o DATASETLIB.so: Output filename for the shared library.
```