Exercises | Refresh

Readability Analysis

Code

Points	Submission due	Status
30 / 30	in 2 days	100%

Recent results

80.01% (4 hours ago)	90.01% (4 hours ago)	90.01% (2 hours ago)	90.01% (an hour ago)	100% (3 minutes ago)
GRADED	GRADED	GRADED	GRADED	GRADED

Show all results

Tasks:

Readability Analysis

In this assignment, you will train a binary logistic regression model to predict whether a given Java method is more or less readable. Software readability is a property that influences how easily some piece of code can be read and understood. Since readability can affect maintainability and the general software quality, developers should aim to write code that is easy to read. Thus, deploying a model that automatically deems whether some piece of code is easy to read or not may be a helpful tool for programmers to maintain an easy-to-read code base that is less prone to faulty program behaviour.

Task Description

The aim of this task is to predict whether a Java method given as a `.jsnp` Java snippet file is readable or not. To this end, we train and evaluate a binary logistic regression classifier by employing a two-step process. First, in the `preprocess` stage, we establish a training and evaluation dataset for our classifier in the form of a CSV file. Each row of this CSV file maps a code snippet to a set of three code readability features and to the ground truth value that deems whether the code is readable or not as determined by human raters. Using the training dataset generated in phase one, we continue in the second step with training and evaluating our binary classifier on the generated dataset using the WEKA machine learning library.

You can invoke the preprocessing routine by executing the `preprocess` subcommand using the following command line parameter:

- `-s, --source` for specifying the path to the directory hosting `.jsnp` files.
- `-g, --ground-truth` for specifying the ground truth `.csv` file hosting human ratings of the specified code snippets.
- `-t, --target` for specifying the target `.csv` file to save the preprocessed data in.
- A parameter list of `FeatureMetrics` selected from `[LINES, TOKEN_ENTROPY, H_VOLUME]`.

We will provide you with a dataset of 200 `.jsnp` files and the corresponding CSV file that hosts the ground-truth data in the form of human readability ratings. Your task will be to implement the three readability features `LINES`, `TOKEN_ENTROPY`, `H_VOLUME` and apply them to the code snippets in order to generate the training dataset. The first feature, called `LINES`, computes the number of lines of the method in the given Java snippet. Since we are interested in determining the readability of some piece of code as experienced by a human, the feature also counts non-executable lines such as comments and empty lines.

`TOKEN_ENTROPY` measures the Shannon Entropy of the token stream extracted from a given code snippet s. This token stream may be extracted from the code snippet using the getTokenRange() method of the JavaParser library after parsing the snippet string using the provided `parseJavaSnippet(String codeSnippet)` method in the `Parser` utility class. The entropy is defined as $H(s) = -\sum_{i=1}^{n} p(x_i) * log_2(p(x_i))$, with $p(x_i)$ representing the probability of encountering token x_i in snippet s. In order to compute the probability $p(x_i)$, we can use the formula $p(x_i) = \frac{count(x_i)}{\sum_{j=1}^{n} count(x_j)}$ by counting the number of occurrences $count(x)$ of each token x in the provided code snippet.

The feature `H_VOLUME` computes the Halstead Volume of the given Java method. To this end, we first have to count the total number of operators N_1 and operands N_2 as well as the unique number of operators n_1 and operands n_2 in the provided code snippet. Next, we can then derive the program length $N = N_1 + N_2$ and the program vocabulary $n = n_1 + n_2$ by summarising the total and unique number of operators and operands, respectively. Based on the program length and the program vocabulary, we can then compute the Halstead Volume $V(s)$ of a given code snippet using the formula $V(s) = N * log_2(n)$. In order to determine the total and unique number of operators and operands, you have to use the VoidVisitorAdapter of the JavaParser library to implement the `OperatorVisitor` and `OperandVisitor` classes. This adapter comes with numerous visitors, which you'll have to use in order to visit the parts of the code that constitute as operators and operands. Generally, operators correspond to special symbols that perform specific actions on operands. For instance, the statement `x = y + z` consists of a binary addition operation (`+`) and an assignment operation (`=`), which involve the operands `x`, `y` and `z`. We restrict the set of operators according to the given `OperatorType` enum, which includes assignment, binary, unary, conditional and type comparison operations. Besides traditional assignment operators (`=`), we also count variable declarations as operations that assign a type to a variable. The set of operands is restricted to literals (boolean, char, integer, double, long, string, null) and names consisting of a single identifier. We do not distinguish between a String having a value of `NULL` and the `null` literal. Furthermore, we do not distinguish the type of the operand, e.g. `boolean a = true` and `String b = "true"` are both counted as the same operand.

Finally, the feature `CYCLOMATIC_COMPLEXITY` determines McCabe's Cyclomatic Complexity by computing the number of independent control flow paths of a given Java snippet. Hence, you can compute the metric by simply counting the number of branching statements within a method. To this end, you have to flesh out the implementation of the `CyclomaticComplexityVisitor` class, which extends the `VoidVisitorAdapter`. Besides common branching statements, keep in mind to include `catch` clauses, conditional statements (`?:`) and logical short circuit evaluations (`&&, ||`). The following instructions are not included: `else`, `switch` without explicit `case` labels, `default` labels of `switch` statements, `try` and `finally` blocks.

The second phase, which is responsible for training and evaluating a readability classifier on our preprocessed dataset, can be invoked using the `classify` subcommand. This subcommand expects the command line input `-d, --data` that defines the path to the training dataset we generated in the preprocessing step. Your task will be to implement the training and evaluation routine for a logistic regression model using the WEKA library. To this end, you will have to implement the two methods `loadDataset()` and `trainAndEvaluate(Instances dataset)`. The former method is responsible for mapping our CSV training dataset into an `Instances` object, which is then consumed by the latter method in order to train and evaluate the readability classifier. For the training and evaluation process, please instruct the WEKA library to apply 10-fold cross-validation using a starting seed of 1.

Additional Remarks:

- Make sure to add the suffix `Test` to all your unit tests and that you place them in the appropriate `test` directory.
- If you, **in any way**, make use of LLMs such as ChatGPT, upload the prompts you send to the LLM together with the answers you obtained in a folder called LLM. Furthermore, ensure to annotate every piece of code you write with the help of LLMs.
- You can trigger the mutation analysis on your machine by executing `mvn clean test pitest:mutationCoverage`, which outputs an `XML` file in `target/pit-reports` that tells you exactly which mutants have survived.

Evaluation Criteria

Unit Tests

1. **Correct implementation of NumberLinesFeature class** 3 of 3 tests passing
2. **Correct implementation of TokenEntropyFeature class** 3 of 3 tests passing
3. **Correct implementation of OperandVisitor class** 1 of 1 tests passing
4. **Correct implementation of OperatorVisitor class** 1 of 1 tests passing
5. **Correct implementation of HalsteadVolumeFeature class** 3 of 3 tests passing
6. **Correct implementation of CyclomaticComplexityFeature class** 3 of 3 tests passing

Functional Tests

4. **Correct implementation of preprocessing phase** 2 of 2 tests passing
   To pass this test, you have to implement the `collectCSVBody()` method of the `Preprocess` utility class such that each row corresponds to one code snippet. To this end, please follow the CSV column format `File,NumberLines,TokenEntropy,HalsteadVolume,CyclomaticComplexity,Truth`, with `File` representing the file name of the code snippet, `NumberLines,TokenEntropy,HalsteadVolume,CyclomaticComplexity` the respective feature values rounded two decimal places and `Truth` representing the binary value `Y` or `N` based on whether the average rater score of a given snippet is greater or equal (=`Y`) than the `TRUTH_THRESHOLD`. Make sure that the entries of the resulting CSV file are ordered according to the filename (1.jsnp -> 2.jsnp -> 3.jsnp -> ...).

5. **Correct implementation of classification phase** 3 of 3 tests passing
   To pass this test, you have to implement the `loadDataset()` and `trainAndEvaluate(Instances dataset)` methods of the `Classify` class. The two methods should be able to handle any CSV format in which the class truth value (e.g. `Truth`), is located at the final position of the CSV file. Please make sure that you use the WEKA machine library together with a logistic classifier that is evaluated using 10-fold cross-validation with a starting seed value of 1.

6. **Working preprocess-classify pipeline** 1 of 1 tests passing
   This test serves as an integration test that first generates a training dataset using the `preprocess` subcommand and the trains the logistic regression model on the generated dataset using the `classify` subcommand.

Coverage Values

7. **90% Line Coverage** 1 of 1 tests passing
8. **90% Branch Coverage** 1 of 1 tests passing
9. **80% Mutation Score** 1 of 1 tests passing

Reset Repository

Exercise details	
Release date	May 21, 2025 10:00
Submission due	Jun 10, 2025 17:00
Complaint possible	No