Decision Tree Planning and Unit Testing

PU Machine learning mit Java, Weka und UIMA

Nils Reiter,
nils.reiter@uni-koeln.de

December 8, 2020
Winter term 2020/21

Section 1

Looking Back

Decision Trees

Summary

- Classification algorithm
- Built around trees, recursive learning and prediction
- Pros
  - Highly transparent
  - Reasonably fast
  - Dependencies between features can be incorporated into the model
- Cons
  - No pairwise dependencies
  - May lead to overfitting – but pruning and bagging help
- Variants exist (e.g., CART)
- Random forest

Exercise 5

- Noteworthy
  - `target/classes`: Binary files – do not commit those
  - Different strategies for storing parameters
    - Setting parameter values to the classifier object, and keeping only the one with the highest score
    - Collect all scores in a map, then pick the highest
    - Additional object/array to store parameter values
  - Can't be done exhaustively
  - Parameters include filtering (e.g. ~~smoothing~~ sampling)

Section 2

Implementing Decision Trees

Introduction

- So far: User perspective – not anymore
- Until Christmas: We implement ID3 ourselves
- Three steps
  - Architecture: Modularization in classes and their relations, implementing unit tests
  - Foundations: Implementing utility and helper functions/classes, data structures
  - Putting it all together, implementing training routine

Introduction

- So far: User perspective – not anymore
- Until Christmas: We implement ID3 ourselves
- Three steps
  - Architecture: Modularization in classes and their relations, implementing unit tests
  - Foundations: Implementing utility and helper functions/classes, data structures
  - Putting it all together, implementing training routine
- Instances / data management from Weka

Introduction

- So far: User perspective – not anymore
- Until Christmas: We implement ID3 ourselves
- Three steps
  - Architecture: Modularization in classes and their relations, implementing unit tests
  - Foundations: Implementing utility and helper functions/classes, data structures
  - Putting it all together, implementing training routine
- Instances / data management from Weka
- Today
  - General software architecture for a decision tree
  - Unit testing

Java Fundamentals

- OOP basics: Classes and instances

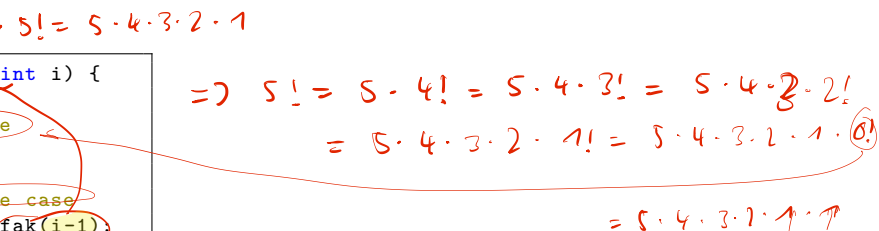
https://github.com/DH-Cologne/java-wegweiser

Java Fundamentals

- OOP basics: Classes and instances
- Recursion

https://github.com/DH-Cologne/java-wegweiser

```
1 public int fak(int i) {
2   if (i == 0) {
3     // base case
4     return 1;
5   } else {
6     // recursive case
7     return i * fak(i-1);
8   }
9 }
```



Java Fundamentals

- OOP basics: Classes and instances
- Recursion

https://github.com/DH-Cologne/java-wegweiser

```java
public int fak(int i) {
  if (i == 0) {
    // base case
    return 1;
  } else {
    // recursive case
    return i * fak(i-1);
  }
}
```

- Generics
  - Often used in container classes: `List<String>`
  - Used to keep the code flexible and more generic
  - Allows to check for type safety

```java
public class Tree<T> {
  T value;

  List<Tree<T>> children;
}
```

Software Architecture

Which classes/functions do we need?

Baum

- Baum {}
- Feature / Schwellenwert
- lokale Vorhersage für 1 Instanz
- save

Classifier

- Inputformat
- train
  - entropy auf 1 Attribut
  - information gain

Introduction

- Automatic verification, that the code works properly
- Why?
  - Large, complex projects: Changes at one place might break something else
  - Humans make mistakes

Introduction

- Automatic verification, that the code works properly
- Why?
  - Large, complex projects: Changes at one place might break something else
  - Humans make mistakes
- Unit tests
  - Collection of expected output for a given input
    - E.g., for `add(2,3)`, we expect 5
  - Include edge cases
    - E.g., illegal input

JUnit

- Java library to support unit testing
- Uses code annotations and static methods
- Version 4 vs. 5
  - User guide: `https://junit.org/junit5/docs/current/user-guide/`

```
1  import static org.junit.jupiter.api.Assertions.assertEquals;
2
3  import org.junit.jupiter.api.Test;
4
5  class TestCalculator {
6    Calculator calculator = new Calculator();
7
8    @Test
9    void addition() {
10     assertEquals(2, calculator.add(1, 1));
11   }
12 }
```

JUnit

Maven Integration: Adding the Dependency

```
<properties>
  <junit.version>5.7.0</junit.version>
</properties>
<!-- ... -->
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-api</artifactId>
  <version>${junit.version}</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <version>${junit.version}</version>
  <scope>test</scope>
</dependency>
```

Maven Integration: Code Location

- Test code in a maven project goes in `src/test/java`, resources used for testing in `src/test/resources`
- It's good practice to use the same package names within testing
- Prefix test classes with 'Test'

Example (Classes and Packages)

`src/main/java`

- `com.example.package.MyClass`
  - `public int calculcateStuff(double x, double y)`
  - `public String generateString(Object o)`

`src/test/java`

- `com.example.package.TestMyClass`
  - `public void testCalculateStuff()`
  - `public void testGenerateString()`

Example

- src
  - main
    - java
    - resources
  - test
    - java
    - resources
- target
- pom.xml

JUnit

Assertions

- Core of unit-testing
- Static methods of `org.junit.jupiter.api.Assertions` class

JUnit

Assertions

- Core of unit-testing
- Static methods of `org.junit.jupiter.api.Assertions` class
  - `assertTrue(boolean b)` / `assertFalse(boolean b)`
  - `assertNull(Object o)` / `assertNotNull(Object o)`
  - `assertEquals(int expected, int actual)` / `assertEquals(Object expected, Object actual)` / `assertEquals(char expected, char actual)` / …

JUnit

Assertions

- Core of unit-testing
- Static methods of `org.junit.jupiter.api.Assertions` class
  - `assertTrue(boolean b)` / `assertFalse(boolean b)`
  - `assertNull(Object o)` / `assertNotNull(Object o)`
  - `assertEquals(int expected, int actual)` / `assertEquals(Object expected, Object actual)` / `assertEquals(char expected, char actual)` / …
- Optional third argument
  - `assertEquals(int expected, int actual, String message)` – `message` is given if test fails

JUnit

Setup

- One class can contain multiple test methods
  - Each method will be run individually @Test
- If test setup is more than a single line, it's useful to encapsulate it in separate function
- Annotations
  - `@BeforeEach` – run before each test function
  - `@AfterEach` – run after each test function
  - `@BeforeAll` – run once before the first test function
  - `@AfterAll` – run once after the last test function

Mocking

- Unit tests for real programs require complex input objects
- Supplying them on the fly increases test run time
- Some combinations may be difficult to encode

Mocking

- Unit tests for real programs require complex input objects
- Supplying them on the fly increases test run time
- Some combinations may be difficult to encode
- A mock object pretends to be of a given class
- We can specify how it reacts to its public interface
- Java: Mockito – `http://mockito.org`

Mocking

- Unit tests for real programs require complex input objects
- Supplying them on the fly increases test run time
- Some combinations may be difficult to encode
- A mock object pretends to be of a given class
- We can specify how it reacts to its public interface
- Java: Mockito – `http://mockito.org`

```
1 @Test
2 void testTreeDepth() {
3   Tree tree = mock(Tree.class);
4   when(tree.getChildren()).thenReturn(new ArrayList<Tree>());
5   assertEquals(1, TreeUtility.depth(tree));
6 }
```

Best Practices

- Unit testing works best with well structured code
- 'Small functions' for specific purposes
- Each function/method should be tested with multiple unit tests
  - Different inputs
  - Unexpected, but technically possible inputs (e.g., non-sensical strings from user input, null, negative values, closed file handles, ...)
    - It's also possible to test for thrown exceptions
- Don't test trivial functions (e.g. getters/setters)

Section 3

Next Exercise

Exercise 06