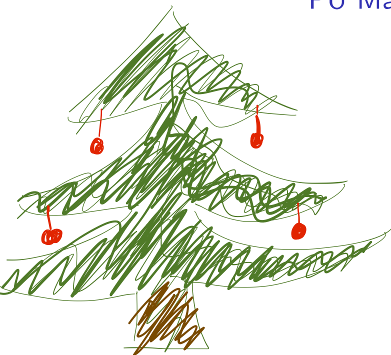
Tree Implementation, part 3

PU Machine learning mit Java, Weka und UIMA



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

December 22, 2020
Winter term 2020/21

Section 1

Looking Back

Last Week

Continuous Integration

- Run tests on every commit
- GitHub actions: CI implementation
  - Controlled with YAML files
  - Executed steps
  - Pre-defined actions or shell commands

action/checkout

run: mvn test

Tree Prediction

- Recursive function
- Base case: Make a prediction
- Recursion: Go into sub tree based on feature value

Exercise 7

- Set up the GitHub action
- Implement functions

https://github.com/idh-cologne-machine-learning-mit-java/exercise-07/tree/reference

Decision Tree

- `double predict(Instance)` ✔
- `double entropy(Instances)` ✔
- `double informationGain(Instances, int)` ✔
- `boolean tree.isLeaf()` ✔

Decision Tree

- `double predict(Instance)` ✔
- `double entropy(Instances)` ✔
- `double informationGain(Instances, int)` ✔
- `boolean tree.isLeaf()` ✔
- `Tree classifier.train(Instances)` ← Today

Recap: Training a Decision Tree

- Core idea: The tree represents splits of the training data
  1. Start with the full data set as D
  2. If D only contains members of a single class:
     - Done, this node predicts the class
  3. Else if there are no more features to select:
     - Done, this node predicts the majority class of D
  4. Else if D is empty
     - Done, the node predicts the majority class of its parent
  5. Else:
     - Select the feature f_i with highest information gain
     - Extract feature values of f_i of all instances in D
     - Split the data set according to f_i: $D = D_v \cup D_w \cup D_u \ldots$
     - For each subset as D, go back to 2

Recap: Training a Decision Tree

- Core idea: The tree represents splits of the training data
  1. Start with the full data set as D
  2. If D only contains members of a single class:
     - Done, this node predicts the class
  3. Else if there are no more features to select:
     - Done, this node predicts the majority class of D
  4. Else if D is empty
     - Done, the node predicts the majority class of its parent
  5. Else:
     - Select the feature f_i with highest information gain
     - Extract feature values of f_i of all instances in D
     - Split the data set according to f_i: $D = D_v \cup D_w \cup D_u \ldots$
     - For each subset as D, go back to 2

Training Function

- `Tree classifier.train(Instances)`
- New utility functions
  - get the majority class of a (partial) data set
  - split a data set into subsets according to an attribute

Subsets

- There is no built-in method in the `Instances` class

Training Function

- `Tree classifier.train(Instances)`
- New utility functions
  - get the majority class of a (partial) data set
  - split a data set into subsets according to an attribute

Subsets

- There is no built-in method in the `Instances` class
- How many subsets to we need?
  - As many as there are attribute values
- How do we label/enumerate the sub sets?
  - With the numeric representation of the attribute values (i.e., the `double` values)

Instances[] subsets(Instances, int)

```
1  /**
2   * For a given data set, create subsets based on one attribute. For each value
3   * of the attribute, one subset is created to hold all instances that have this
4   * attribute value.
5   */
6  protected static Instances[] subsets(Instances instances, int attributeIndex) {
7    // we need as many sub sets as this attribute has values
8    Instances[] ret = new Instances[instances.attribute(attributeIndex).numValues()];
9
10   // we initialize all subsets to be (shallow) copies of the original
11   // data set with capacity 0
12   for (int i = 0; i < instances.attribute(attributeIndex).numValues(); i++)
13     ret[i] = new Instances(instances, 0);
14
15   // depending on the attribute value, we add an instance to a different subset
16   for (Instance instance : instances)
17     ret[(int) instance.value(attributeIndex)].add(new DenseInstance(instance));
18
19   return ret;
20 }
```

Majority

- In leaf nodes, we need to find the majority class in a given data set
- There is no built-in function in an `Instances` object

Majority

- In leaf nodes, we need to find the majority class in a given data set
- There is no built-in function in an `Instances` object
- We need to count how many instances of which class are there
  - This was also needed for calculating entropy!
  - A good opportunity for a third utility function that is used multiple times:

    ```
    int[] countClasses(Instances)
    ```
- Once we have an `int[]`, it's pretty straightforward

int[] countClasses(Instances)

```
1  /**
2   * Utility function to extract the number of instances for
3   * target class.
4   */
5  protected static int[] countClasses(Instances instances) {
6    // we need as many positions as there are classes
7    int[] instanceNumbers = new int[instances.numClasses()];
8
9    // we count how many instances of each class are there
10   for (Instance instance : instances) {
11     instanceNumbers[(int) instance.classValue()] += 1;
12   }
13   return instanceNumbers;
14 }
```

int getMajority(int[])

```java
/**
 * For a given int array, return the index with the highest
 * value.
 */
protected static int getMajority(int[] instances) {
  // stores the highest number so far
  int majority = 0;
  // stores the index to the highest number so far
  int majorityIndex = 0;
  for (int i = 0; i < instances.length; i++) {
    if (instances[i] > majority) {
      majority = instances[i];
      majorityIndex = i;
    }
  }
  return majorityIndex;
}
```

Section 3

Next Exercise

Exercise 07