Department of Computer Science
Computer Science 6
(Data Management)

Knowledge Discovery in Databases with Exercises
Summer Semester 2025

Submission 2: Classification

About this Assignment

In this assignment, your task is to implement the algorithms for Decision Tree Induction and Naïve Bayes Classification. For this purpose, you have access to a basic code skeleton, some helper classes, and several test cases.

Key Data

- **Max. Group Size:** 3
- **Max. Points:** 50
- **Estimated Workload:** 5 - 7.5 hours

How to Work on the Assignment

To start working on the assignment, you'll need to accept the assignment via GitHub Classroom by clicking the provided link. This will set up a new GitHub repository for your group, packed with all the necessary files for the assignment. If you're joining an existing group, it'll add you to that group's repository.[1]

Once that's done, you have two main options for working on your assignment. You can clone the repository[2] to your local machine by navigating to `Code` → `Local`, which allows you to work directly from your computer. Alternatively, you might prefer using GitHub Codespaces by selecting `Code` → `Codespaces` for a virtual online environment, complete with the ability to run Python through the `Terminal` provided.

Whichever method you choose, it's crucial to commit and push your changes back to the repository to submit your solution[2]. After your submission, GitHub Actions takes over to automatically grade your solution and provide feedback. You'll find this feedback in the `Actions` tab of your repository. If you didn't receive full points, you can improve your solution and push the changes back to the repository to trigger a reevaluation.

[1] Each student must join individually. You can join groups while accepting an assignment.
[2] If you're unfamiliar with Git or GitHub, check out this helpful guide: `https://github.com/git-guides/`

How to Prepare the Transfer the Points to StudOn

In addition to joining the GitHub Classroom, you also need to register your GitHub username on StudOn. This is necessary to transfer the points you've earned on GitHub to StudOn. To do this, enter your GitHub username in `Submission 2 - GitHub Username`. Make sure to enter your username correctly, as otherwise, the points cannot be transferred.

After submission deadline, the points you've earned on GitHub will be transferred to StudOn. This process is not immediate and may take a few days. If you have any questions or issues, please contact us via the StudOn forum.

Restrictions

Within the scope of your implementation, you are not permitted to modify the helper classes, the test cases, or the provided GitHub Actions.

This will be checked on a random basis, and any attempt to do so will result in zero points for the involved group, similar to the consequences of plagiarism.

Task 1: Decision Tree Induction

Decision tree induction is a commonly used method for classifying datasets. While the fundamental approach to decision tree induction is not very variable, using different attribute selection methods can produce very different decision trees.

Important Note: Categorical and Continuous Attributes

In decision tree induction, a distinction is made between categorical and continuous attributes. To simplify the distinction, you can assume all attributes containing strings to be categorical, while numerical attributes are considered continuous. The target attributes are always categorical.

Task 1.1: Attribute Selection Methods

Since attribute selection methods play a crucial role in decision tree induction, it is reasonable to implement these first. In this submission, we limit ourselves to two methods: Information Gain and Gini Index.

Task 1.1.1: Information Gain

The Information Gain is a measure of the difference in entropy before and after splitting a dataset based on an attribute.

Task 1.1.1.1 **(1 Points)**

At the beginning of Apriori, the identification of 1-itemsets is paramount.

Open `information_gain.py` and implement the `calculate_entropy`, which calculates the entropy of a dataset with regard to a target attribute:

calculate_entropy

```
def calculate_entropy(dataset: pd.DataFrame, target_attribute: str) -> float:
```

Description:
- Calculate the entropy for a given target attribute in a dataset.

Parameters:
- `dataset` (pd.DataFrame): The dataset to calculate the entropy for.
- `target_attribute` (str): The target attribute used as the class label.

Returns:
- `float`: The calculated entropy (= expected information).

Make sure that you expect a pandas DataFrame as the dataset and a string as the target attribute. Make sure to return the calculated entropy as a `float`.

You can test whether your implementation is correct by executing the following command in the console:

```
pytest tests/information_gain/test_calculate_entropy.py
```

Task 1.1.1.2 (1 Points)

The next step is to calculate the entropy after the split.

Implement `calculate_information_partitioned`, which calculates the entropy of a dataset after splitting it based on a specific attribute:

calculate_information_partitioned

```
def calculate_information_partitioned(
    dataset: pd.DataFrame,
    target_attribute: str,
    partition_attribute: str,
    split_value: int | float = None,
) -> float:
```

Description:

- Calculate the information for a given target attribute in a dataset if the dataset is partitioned by a given attribute.

Parameters:

- `dataset` (pd.DataFrame): The dataset to calculate the information for.
- `target_attribute` (str): The target attribute used as the class label.
- `partition_attribute` (str): The attribute that is used to partition the dataset.
- `split_value` (int|float), default None: The value to split the partition attribute on. If set to None, the function will calculate the information for a discrete-valued partition attribute. If set to a value, the function will calculate the information for a continuous-valued partition attribute.

Returns:

- `float`: The calculated entropy.

Like `calculate_entropy`, `calculate_information_partitioned` also requires a dataset and a target attribute. Additionally, the function requires a string that specifies which attribute is used for partitioning. If the partitioning attribute is a continuous attribute, an optional numeric value can be provided, indicating where the partitioning into two partitions should occur.

The function should return the calculated entropy as a `float`.

You can test whether your implementation is correct by executing the following command in the console:

```
pytest tests/information_gain/test_calculate_information_partitioned.py
```

Task 1.1.1.3 (1 Points)

Both entropies can be used to calculate the information gain.

Implement `calculate_information_gain`, which calculates the information gain for a dataset based on a specific attribute:

calculate_information_gain

```
def calculate_information_gain(
    dataset: pd.DataFrame,
    target_attribute: str,
    partition_attribute: str,
    split_value: int | float = None,
) -> float:
```

Description:

- Calculate the information gain for a given target attribute in a dataset if the dataset is partitioned by a given attribute.

Parameters:

- `dataset` (pd.DataFrame): The dataset to calculate the information gain for.
- `target_attribute` (str): The target attribute used as the class label.
- `partition_attribute` (str): The attribute that is used to partition the dataset.
- `split_value` (int|float), default None: The value to split the partition attribute on. If set to None, the function will calculate the information gain for a discrete-valued partition attribute. If set to a value, the function will calculate the information gain for a continuous-valued partition attribute.

Returns:

- `float`: The calculated information gain.

The function expects a dataset, a target attribute, and a partitioning attribute. If the partitioning attribute is continuous, a split value can be provided. The function should return the calculated information gain as a `float`.

You can test whether your implementation is correct by executing the following command in the console:

```
pytest tests/information_gain/test_calculate_information_gain.py
```

Task 1.1.2: Gini Index

The Gini Index is another attribute selection method. It measures the impurity of a dataset.

Task 1.1.2.1 **(1 Points)**

To calculate the Gini Index, the impurity of the dataset has to be computed.

Open `gini_index.py`. Implement `calculate_impurity`, which calculates the impurity of a dataset with regard to a target attribute:

calculate_impurity

```
def calculate_impurity(dataset: pd.DataFrame, target_attribute: str) -> float:
```

Description:
- Calculate the impurity for a given target attribute in a dataset.

Parameters:
- `dataset` (pd.DataFrame): The dataset to calculate the impurity for.
- `target_attribute` (str): The target attribute used as the class label.

Returns:
- `float`: The calculated impurity.

The function expects a dataset and a target attribute. Make sure to return the calculated impurity as a `float`.

You can test whether your implementation is correct by executing the following command in the console:

```
pytest tests/gini_index/test_calculate_impurity.py
```

Task 1.1.2.2 **(1 Points)**

The next step is to calculate the impurity after the split.

Implement `calculate_impurity_partitioned`, which calculates the impurity of a dataset after splitting it based on a specific attribute:

calculate_impurity_partitioned

```
def calculate_impurity_partitioned(
    dataset: pd.DataFrame,
    target_attribute: str,
    partition_attribute: str,
    split: int | float | Set[str],
) -> float:
```

Description:

- Calculate the impurity for a given target attribute in a dataset if the dataset is partitioned by a given attribute and split.

Parameters:

- `dataset` (pd.DataFrame): The dataset to calculate the impurity for.
- `target_attribute` (str): The target attribute used as the class label.
- `partition_attribute` (str): The attribute that is used to partition the dataset.
- `split` (int|float|Set[str]): The split used to partition the partition attribute. If the partition attribute is discrete-valued, the split is a set of strings (Set[str]). If the partition attribute is continuous-valued, the split is a single value (int or float).

Returns:

- `float`: The calculated impurity.

The function expects a dataset, a target attribute, and a partitioning attribute. If the partitioning attribute is continuous, a single split value can be provided. If the partitioning attribute is discrete, a set of strings can be provided. The function should return the calculated impurity as a `float`.

You can test whether your implementation is correct by executing the following command in the console:

```
pytest tests/gini_index/test_calculate_impurity_partitioned.py
```

Task 1.1.2.3 (1 Points)

Both impurities can be used to calculate the gini index.

Implement `calculate_gini_index`, which calculates the gini index for a dataset based on a specific attribute:

calculate_gini_index

```
def calculate_gini_index(
    dataset: pd.DataFrame,
    target_attribute: str,
    partition_attribute: str,
    split: int | float | Set[str],
) -> float:
```

Description:

- Calculate the Gini index (= reduction of impurity) for a given target attribute in a dataset if the dataset is partitioned by a given attribute and split.

Parameters:

- `dataset` (pd.DataFrame): The dataset to calculate the Gini index for.
- `target_attribute` (str): The target attribute used as the class label.
- `partition_attribute` (str): The attribute that is used to partition the dataset.
- `split` (int|float|Set[str]): The split used to partition the partition attribute. If the partition attribute is discrete-valued, the split is a set of strings (Set[str]). If the partition attribute is continuous-valued, the split is a single value (int or float).

Returns:

- `float`: The calculated Gini index.

The function expects a dataset, a target attribute, and a partitioning attribute. If the partitioning attribute is continuous, a single split value can be provided. If the partitioning attribute is discrete, a set of strings can be provided. The function should return the calculated gini index as a `float`.

You can test whether your implementation is correct by executing the following command in the console:

```
pytest tests/gini_index/test_calculate_gini_index.py
```

Task 1.2: Training

After implementing the attribute selection methods, the next step is to implement the decision tree induction itself.

Task 1.2.1 **(3 Points)**

One important step in decision tree induction is to determine the best attribute to split the dataset on. For this purpose, the Information Gain or the Gini Index have to be calculated for each attribute. Since there might be multiple splits for the same attribute and therefore multiple information gains or gini indices, it is best to implement a separate function for this purpose.

Open `decision_tree.py` and implement `_calculate_information_gain`, which calculates the best possible information gain for a specific attribute:

_calculate_information_gain

```
def _calculate_information_gain(
    self,
    data: pd.DataFrame,
    attribute: str
) -> Tuple[float, List[DecisionTreeDecisionOutcome]]:
```

Description:
- Calculate the (best) information gain for a given attribute in a dataset.

Parameters:
- `data` (pd.DataFrame): The dataset to calculate the information gain for.
- `attribute` (str): The attribute to calculate the information gain for.

Returns:
- `float`: The calculated information gain.
- `List[DecisionTreeDecisionOutcome]`: The outcomes the best split of this attribute has.

The function expects the dataset and the attribute for which the information gain is to be calculated. The target classification attribute is already set in `self.target_attribute` when the function is called.

The function should return the calculated information gain as a `float` and a list of outcomes. The `DecisionTreeDecisionOutcome` objects represent the outcomes of the best split of the attribute (e.g. if the attribute is `Age`, the outcomes might be ≤ 25 and > 25).

You can test whether your implementation is correct by executing the following command in the console:

```
pytest tests/decision_tree/test_calculate_information_gain.py
```

Task 1.2.2 **(3 Points)**

The same has to be done for the Gini Index.

Implement `_calculate_gini_index`, which calculates the best possible gini index for a specific attribute:

_calculate_gini_index

```
def _calculate_gini_index(
    self,
    data: pd.DataFrame,
    attribute: str
) -> Tuple[float, List[DecisionTreeDecisionOutcome]]:
```

Description:

- Calculate the (best) gini index for a given attribute in a dataset.

Parameters:

- `data` (pd.DataFrame): The dataset to calculate the gini index for.
- `attribute` (str): The attribute to calculate the gini index for.

Returns:

- `float`: The calculated gini index (reduction of impurity).
- `List[DecisionTreeDecisionOutcome]`: The outcomes the best split of this attribute has.

The function expects the dataset and the attribute for which the gini index is to be calculated. The target classification attribute is already set in `self.target_attribute` when the function is called.

The function should return the calculated gini index as a `float` and a list of outcomes. The `DecisionTreeDecisionOutcome` objects represent the outcomes of the best split of the attribute (e.g. if the attribute is `Participation`, the outcomes might be {`High`, `Medium`} and {`Low`}).

You can test whether your implementation is correct by executing the following command in the console:

```
pytest tests/decision_tree/test_calculate_gini_index.py
```

Task 1.2.3 **(6 Points)**

These functions can now be used to find the best attribute to split the dataset on.

Implement `_find_best_split`, which finds the best split for a given dataset:

_find_best_split

```
def _find_best_split(
    self,
    data: pd.DataFrame,
    attribute_list: List[str],
    attribute_selection_method: str,
) -> Tuple[str, List[DecisionTreeDecisionOutcome]]:
```

Description:

- Find the best split for a given dataset and attribute list. Finding the best split includes finding the best attribute to split on and also (depending on the attribute selection method) the best set of outcomes to split on this attribute.

Parameters:

- `data` (pd.DataFrame): The dataset to find the best splitting attribute for.
- `attribute_list` (List[str]): The list of attributes to consider.
- `attribute_selection_method` (str): The attribute selection method to use.

Returns:

- `str`: The attribute to split on.
- `List[DecisionTreeDecisionOutcome]`: The outcomes a split on this attribute should have.

The function expects the dataset, a list of all attributes that might become the splitting attribute, and the attribute selection method. The attribute selection method can be either `information_gain` or `gini_index`. The function should return the best attribute to split on and a list of `DecisionTreeDecisionOutcomes`.

You can test whether your implementation is correct by executing the following command in the console:

```
pytest tests/decision_tree/test_find_best_split.py
```

Task 1.2.4 **(6 Points)**

The next step is to implement the recursive creation of the decision tree.

Implement `_build_tree`, which recursively builds the decision tree:

__build__tree

```
def _build_tree(
    self,
    data: pd.DataFrame,
    attribute_list: List[str],
    attribute_selection_method: str,
) -> DecisionTreeNode:
```

Description:

- Recursively build the decision tree.

Parameters:

- `data` (pd.DataFrame): The (partial) dataset to build the decision tree with.
- `attribute_list` (List[str]): The list of attributes to consider.
- `attribute_selection_method` (str): The attribute selection method to use.

Returns:

- `DecisionTreeNode`: The root node of the decision tree.

The function expects the dataset, a list of all attributes that might become the splitting attribute, and the attribute selection method. The attribute selection method can be either `information_gain` or `gini_index`. The function should return the `DecisionTreeNode` that represents the root node of the part of the decision tree that was built within the call of the function.

You can test whether your implementation is correct by executing the following command in the console:

```
pytest tests/decision_tree/test_build_tree.py
```

Task 1.2.5 **(4 Points)**

The last step is to implement the method to train the decision tree on a specific dataset.

Implement `fit`, which fits the decision tree to the dataset:

fit

```
def fit(
    self,
    dataset: pd.DataFrame,
    target_attribute: str,
    attribute_selection_method: str,
):
```

Description:

- Fit decision tree on a given dataset and target attribute, using a specified attribute selection method.

Parameters:

- `dataset` (pd.DataFrame): The dataset to fit the decision tree on.
- `target_attribute` (str): The target attribute to predict.
- `attribute_selection_method` (str): The attribute selection method to use.

Returns:

- None - The method saves the trained model in `self.target_attribute` and `self.tree`.

The function expects the dataset, the target attribute, and the attribute selection method that should be used to build the decision tree. The function doesn't return anything, but sets both members `self.target_attribute` and `self.tree`. The former is the target attribute, and the latter is the root node of the decision tree.

You can test whether your implementation is correct by executing the following command in the console:

```
pytest tests/decision_tree/test_fit.py
```

Task 1.3: Prediction

With a trained decision tree, the classes of new tuples can be predicted.

Task 1.3.1 **(2 Points)**

The first step is to implement the method to predict the class of a single tuple.

Within `decision_tree.py` implement `_predict_tuple`, which predicts the class of a single tuple:

_predict_tuple

```
def _predict_tuple(
    self,
    tuple: pd.Series,
    node: DecisionTreeNode
) -> str | int | float:
```

Description:
- Predict the target attribute for a given row in the dataset. This is a recursive function that traverses the decision tree until a leaf node is reached.

Parameters:
- `tuple` (pd.Series): The row to predict the target attribute for.
- `node` (DecisionTreeNode): The current node in the decision tree.

Returns:
- `str | int | float`: The predicted class label.

The function expects a single tuple as a pandas Series and the current node of the decision tree. The function should return the predicted class label.

You can test whether your implementation is correct by executing the following command in the console:

```
pytest tests/decision_tree/test_predict_tuple.py
```

Task 1.3.2 **(2 Points)**

The last step is to implement the method to predict the classes of a complete dataset.

Implement `predict`, which predicts the classes of a dataset:

predict

```
def predict(self, dataset: pd.DataFrame) -> List[str | int | float]:
```

Description:

- Predict the target attribute for a given dataset.

Parameters:

- `dataset` (pd.DataFrame): The dataset to predict the target attribute for.

Returns:

- `List[str | int | float]`: A list of predicted class labels.

The function expects a dataset and should return a list of predicted class labels.

You can test whether your implementation is correct by executing the following command in the console:

```
pytest tests/decision_tree/test_predict.py
```

Task 2: Naïve Bayes Classification

Naïve Bayes is a simple classification algorithm based on Bayes' Theorem. It is called "naïve" because it assumes that the attributes are conditionally independent given the class label.

Important Note: Categorical and Continuous Attributes

In naïve Bayes classification, a distinction is made between categorical and continuous attributes. To simplify the distinction, you can assume all attributes containing strings to be categorical, while numerical attributes are considered continuous. The target attributes are always categorical.

Task 2.1: Training

To be able to classify new tuples, the algorithm has to be trained on a dataset.

Task 2.1.1 (6 Points)

For the training, the algorithm has to calculate the prior probabilities for each of the classes.

Open `naive_bayes.py` and implement `_calculate_prior_probabilities`, which calculates the prior probabilities for each class:

_calculate_prior_probabilities

```
def _calculate_prior_probabilities(
    self,
    dataset: pd.DataFrame
) -> NaiveBayesPriorProbabilities:
```

Description:
- Calculate the prior probability for each class. (The target attribute has to be set before calling this method.)

Parameters:
- `dataset` (pd.DataFrame): The training dataset.

Returns:
- `NaiveBayesPriorProbabilities`: The prior probabilities for each class.

The function expects a dataset and should return an instance of `NaiveBayesPriorProbabilities`. This object contains the prior probabilities for each class. The target attribute is already set in `self.target_attribute` when the function is called.

You can test whether your implementation is correct by executing the following command in the console:

```
pytest tests/naive_bayes/test_calculate_prior_probabilities.py
```

Task 2.1.2 **(5 Points)**

The next step is to calculate the likelihoods for each attribute given the class label.

Implement `_calculate_likelihoods`, which calculates the likelihoods for each attribute given the class label:

_calculate_likelihoods

```
def _calculate_likelihoods(self, dataset: pd.DataFrame) -> NaiveBayesLikelihoods:
```

Description:
- Calculate the likelihoods for each attribute and class. (The target attribute has to be set before calling this method.)

Parameters:
- `dataset` (pd.DataFrame): The training dataset.

Returns:
- `NaiveBayesLikelihoods`: The likelihoods for each attribute and class.

The function expects a dataset and should return an instance of `NaiveBayesLikelihoods`. This object contains the likelihoods for each attribute given the class label. The target attribute is already set in `self.target_attribute` when the function is called.

You can test whether your implementation is correct by executing the following command in the console:

```
pytest tests/naive_bayes/test_calculate_likelihoods.py
```

Task 2.1.3 (3 Points)

The last step is to implement the method to train the naïve Bayes classifier on a specific dataset.

Implement `fit`, which fits the naïve Bayes classifier to the dataset:

fit

```
def fit(self, dataset: pd.DataFrame, target_attribute: str):
```

Description:

- Fit the Naive Bayes classifier to the training dataset. Sets the target attribute and the class labels. Calculates the prior probabilities, and the likelihoods.

Parameters:

- `dataset` (pd.DataFrame): The training dataset.
- `target_attribute` (str): The target attribute to predict.

Returns:

- None - The method saves the trained model in `self.target_attribute`, `self.class_labels`, `self.prior_probabilities`, and `self.likelihoods`.

The function expects the dataset and the target attribute. The function doesn't return anything, but sets the members `self.target_attribute`, `self.class_labels`, `self.prior_probabilities`, and `self.likelihoods`. The former is the target attribute, the second is a list of all possible class labels, the third is an instance of `NaiveBayesPriorProbabilities`, and the last is an instance of `NaiveBayesLikelihoods`.

You can test whether your implementation is correct by executing the following command in the console:

```
pytest tests/naive_bayes/test_fit.py
```

Task 2.2: Prediction

With a trained naïve Bayes classifier, the classes of new tuples can be predicted.

Task 2.2.1 **(2 Points)**

The first step is to implement the method to predict the class of a single tuple.

Within `naive_bayes.py` implement `_predict_tuple`, which predicts the class of a single tuple:

_predict_tuple

```
def _predict_tuple(self, tuple: pd.Series) -> str | int | float:
```

Description:
- Predict the target attribute for a given row in the dataset.

Parameters:
- `tuple` (pd.Series): The row in the dataset to predict the target attribute for.

Returns:
- `str | int | float`: The predicted class label.

The function expects a single tuple as a pandas Series. The function should return the predicted class label.

You can test whether your implementation is correct by executing the following command in the console:

```
pytest tests/naive_bayes/test_predict_tuple.py
```

Task 2.2.2 **(2 Points)**

The last step is to implement the method to predict the classes of a complete dataset.

Implement `predict`, which predicts the classes of a dataset:

predict

```
def predict(self, dataset: pd.DataFrame) -> List[str | int | float]:
```

Description:
- Predict the target attribute for a given dataset.

Parameters:
- `dataset` (pd.DataFrame): The dataset to predict the target attribute for.

Returns:
- `List[str | int | float]`: A list of predicted class labels.

The function expects a dataset and should return a list of predicted class labels.

You can test whether your implementation is correct by executing the following command in the console:

```
pytest tests/naive_bayes/test_predict.py
```

Appendices

In Task 1 and Task 2 test cases are provided and used to grade the submission.

Dataset(s)

The most test cases are based on the following data sets:

Small Student Dataset

All test cases starting with the prefix `test_with_small_student_dataset` are based on the small student dataset known from Exercise Sheet 4 - Task 1.

The dataset is structured as follows:

Age	Major	Participation	Passed
23	CS	High	Yes
23	DS	Low	No
26	DS	High	Yes
24	DS	Medium	Yes
26	DS	Medium	No
26	DS	Low	No

Table 1: Small Student Dataset

Small Submission Dataset

All test cases starting with the prefix `test_with_small_student_dataset` are based on the small submission dataset known from Exercise Sheet 4 - Task 2.

The dataset is structured as follows:

Topic	Knowledge	Hours	Passed
Classification	High	1,0	No
Clustering	Low	4,0	No
Frequent Patterns	High	5,0	Yes
Clustering	Medium	5,0	Yes
Frequent Patterns	High	2,0	No
Frequent Patterns	Medium	3,0	Yes
Classification	Low	6,0	Yes
Clustering	Low	5,0	Yes
Clustering	High	3,0	Yes
Classification	Medium	4,0	Yes

Table 2: Small Submission Dataset

Helper Classes

The following helper classes are provided in the `classes/` folder to support your implementation. Each class serves a specific purpose in the classification algorithms.

Decision Tree Data Structures

DecisionTreeNode (`classes/decision_tree_node.py`)

Abstract superclass for all decision tree node types.

- **Usage:** Base class for DecisionTreeInternalNode and DecisionTreeLeafNode
- **Note:** You typically don't create instances of this class directly

DecisionTreeInternalNode (`classes/decision_tree_internal_node.py`)

Represents an internal node in a decision tree that contains a decision attribute and branches.

- **Attributes:**
  - `attribute_label` (str) - The attribute this node makes decisions on
  - `branches` (List[DecisionTreeBranch]) - The branches starting from this node
- **Key Methods:**
  - `get_label()` - Returns the attribute label
  - `get_branches()` - Returns list of branches
- **Usage:** Created when building decision tree for non-leaf nodes
- **Example:**

```
internal_node = DecisionTreeInternalNode("Age", branches_list)
attribute = internal_node.get_label()  # Returns "Age"
```

DecisionTreeLeafNode (`classes/decision_tree_leaf_node.py`)

Represents a leaf node in a decision tree that contains a final class prediction.

- **Attributes:**
  - `class_label` (str|int|float) - The predicted class for this leaf
- **Key Methods:**
  - `get_label()` - Returns the class label
- **Usage:** Created when building decision tree for terminal nodes
- **Example:**

```
leaf_node = DecisionTreeLeafNode("Yes")
prediction = leaf_node.get_label()  # Returns "Yes"
```

DecisionTreeBranch (`classes/decision_tree_branch.py`)

Represents a branch connecting nodes in a decision tree.

- **Attributes:**
  - `label` (DecisionTreeDecisionOutcome) - The condition for this branch
  - `branch_node` (DecisionTreeNode) - The node this branch leads to
- **Key Methods:**
  - `get_label()` - Returns the branch condition
  - `get_branch_node()` - Returns the destination node
  - `value_matches(value)` - Checks if a value satisfies the branch condition
- **Usage:** Connects internal nodes to their child nodes with conditions
- **Example:**

```python
outcome = DecisionTreeDecisionOutcomeAbove(25)
branch = DecisionTreeBranch(outcome, child_node)
if branch.value_matches(30):  # True, since 30 > 25
    next_node = branch.get_branch_node()
```

Decision Outcome Classes

DecisionTreeDecisionOutcome (`classes/decision_tree_decision_outcome.py`)

Abstract base class for all decision outcomes in decision trees.

- **Key Methods:**
  - `value_matches(value)` - Checks if a value matches this outcome
- **Usage:** Base class for specific outcome types

DecisionTreeDecisionOutcomeEquals (`classes/decision_tree_decision_outcome_equals.py`)

Represents an outcome where values must exactly equal a specific value.

- **Attributes:**
  - `value` (str|int|float) - The exact value required for this outcome
- **Usage:** Used for categorical attributes or exact numeric matches
- **Example:**

```python
outcome = DecisionTreeDecisionOutcomeEquals("High")
print(outcome.value_matches("High"))   # True
print(outcome.value_matches("Medium")) # False
```

DecisionTreeDecisionOutcomeAbove (`classes/decision_tree_decision_outcome_above.py`)

Represents an outcome where values must be above a threshold.

- **Attributes:**
  - `value` (str|int|float) - The threshold value
- **Usage:** Used for continuous attributes with upper splits
- **Example:**

```python
outcome = DecisionTreeDecisionOutcomeAbove(25)
print(outcome.value_matches(30))  # True, since 30 > 25
print(outcome.value_matches(20))  # False, since 20 <= 25
print(str(outcome))               # ">25"
```

DecisionTreeDecisionOutcomeBelowEqual
(`classes/decision_tree_decision_outcome_below_equal.py`)

Represents an outcome where values must be below or equal to a threshold.

- **Attributes:**
  - `value` (str|int|float) - The threshold value
- **Usage:** Used for continuous attributes with lower splits
- **Example:**

```python
outcome = DecisionTreeDecisionOutcomeBelowEqual(25)
print(outcome.value_matches(20))  # True, since 20 <= 25
print(outcome.value_matches(30))  # False, since 30 > 25
print(str(outcome))               # "<=25"
```

DecisionTreeDecisionOutcomeInList (`classes/decision_tree_decision_outcome_in_list.py`)

Represents an outcome where values must be in a specific list of allowed values.

- **Attributes:**
  - `value` (List[str|int|float]) - List of allowed values for this outcome
- **Usage:** Used for categorical attributes with multiple valid values
- **Example:**

```python
outcome = DecisionTreeDecisionOutcomeInList(["High", "Medium"])
print(outcome.value_matches("High"))    # True
print(outcome.value_matches("Medium")) # True
print(outcome.value_matches("Low"))     # False
print(str(outcome))                     # "{High, Medium}"
```

Naïve Bayes Data Structures

NaiveBayesPriorProbabilities (`classes/naive_bayes_prior_probabilities.py`)

Stores prior probabilities for each class in a Naïve Bayes classifier.

- **Attributes:**
  - `prior_probabilities` (dict) - Dictionary mapping class labels to probabilities
- **Key Methods:**
  - `add_prior_probability(class_label, probability)` - Adds a prior probability
  - `get_prior_probability(class_label)` - Retrieves a prior probability
- **Usage:** Stores P(Class) for each class label
- **Example:**

```
priors = NaiveBayesPriorProbabilities()
priors.add_prior_probability("Yes", 0.6)
priors.add_prior_probability("No", 0.4)
prob_yes = priors.get_prior_probability("Yes")  # Returns 0.6
```

NaiveBayesLikelihoods (`classes/naive_bayes_likelihoods.py`)

Stores likelihoods for attributes given class labels in a Naïve Bayes classifier.

- **Attributes:**
  - `likelihoods` (dict) - Nested dictionary storing likelihood information
- **Key Methods:**
  - `add_categorical_likelihood(attribute, value, class_label, likelihood)` - Adds likelihood for categorical attributes
  - `add_continuous_likelihood(attribute, class_label, mean, std)` - Adds parameters for continuous attributes
  - `get_likelihood(attribute, value, class_label)` - Retrieves likelihood for given parameters
- **Usage:** Stores P(Attribute|Class) for both categorical and continuous attributes
- **Example:**

```
likelihoods = NaiveBayesLikelihoods()

# For categorical attribute
likelihoods.add_categorical_likelihood("Major", "CS", "Yes", 0.8)

# For continuous attribute (stores mean and std for Gaussian)
likelihoods.add_continuous_likelihood("Age", "Yes", 24.5, 2.1)

# Retrieve likelihoods
prob_cs_given_yes = likelihoods.get_likelihood("Major", "CS", "Yes")     # 0.8
prob_age_given_yes = likelihoods.get_likelihood("Age", 25.0, "Yes")      #
                                                Calculated using Gaussian
```

Practical Tips

- **Decision Tree Construction:**
  Use DecisionTreeInternalNode for decision points and DecisionTreeLeafNode for final predictions
- **Outcome Matching:**
  Different outcome types handle different split conditions - choose the appropriate type based on your attribute and split
- **Type Hints:**
  Pay attention to the expected types in method signatures - many methods accept str|int|float for flexibility
- **Error Handling:**
  The classes include appropriate error checking and warnings for common mistakes
- **String Representations:**
  Most classes have useful ___str___ methods for debugging and visualization
- **Decision Tree Traversal:**
  Use the value_matches() method on branches to determine which path to follow during prediction