



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 repository² 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 solution². 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.

¹Each student must join individually. You can join groups while accepting an assignment.

²If 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

(4 Points)

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

To calculate the difference between the entropies before and after the split, the entropy of the dataset prior to the split has to be computed.

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

```
1 def calculate_entropy(dataset: pd.DataFrame, target_attribute: str) -> float:
2     """
3     Calculate the entropy for a given target attribute in a dataset.
4
5     Parameters:
6     dataset (pd.DataFrame): The dataset to calculate the entropy for
7     target_attribute (str): The target attribute used as the class label
8
9     Returns:
10    float: The calculated entropy (= expected information)
11    """
12    # TODO
```

The method expects 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:

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

Task 1.1.1.2

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:

```
1 def calculate_information_partitioned(  
2     dataset: pd.DataFrame, target_attribute: str,  
3     partition_attribute: str, split_value: int | float = None,  
4 ) -> float:  
5     """  
6     Calculate the information for a given target attribute in a dataset if the dataset is  
7     partitioned by a given attribute.  
8  
9     Parameters:  
10    dataset (pd.DataFrame): The dataset to calculate the information for  
11    target_attribute (str): The target attribute used as the class label  
12    partition_attribute (str): The attribute that is used to partition the dataset  
13    split_value (int | float), default None: The value to split the partition attribute  
14    on. If set to None, the function will calculate the information for a discrete-valued  
15    partition attribute. If set to a value, the function will calculate the information  
16    for a continuous-valued partition attribute.  
17    """  
18    # TODO
```

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:

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

Task 1.1.1.3

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:

```
1 def calculate_information_gain(  
2     dataset: pd.DataFrame, target_attribute: str,  
3     partition_attribute: str, split_value: int | float = None,  
4 ) -> float:  
5     """  
6     Calculate the information gain for a given target attribute in a dataset if the  
7     dataset is partitioned by a given attribute.  
8  
9     Parameters:  
10    dataset (pd.DataFrame): The dataset to calculate the information gain for  
11    target_attribute (str): The target attribute used as the class label  
12    partition_attribute (str): The attribute that is used to partition the dataset  
13    split_value (int | float), default None: The value to split the partition attribute on.  
14    If set to None, the function will calculate the information gain for a discrete-valued  
15    partition attribute. If set to a value, the function will calculate the information  
16    gain for a continuous-valued partition attribute.  
17  
18    Returns:  
19    float: The calculated information gain  
20    """  
21    # TODO
```

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:

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

Task 1.1.2: Gini Index

(4 Points)

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

Task 1.1.2.1

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:

```
1 def calculate_impurity(dataset: pd.DataFrame, target_attribute: str) -> float:
2     """
3     Calculate the impurity for a given target attribute in a dataset.
4
5     Parameters:
6     dataset (pd.DataFrame): The dataset to calculate the impurity for
7     target_attribute (str): The target attribute used as the class label
8
9     Returns:
10    float: The calculated impurity
11    """
12    # TODO
```

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:

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

Task 1.1.2.2

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:

```

1 def calculate_impurity_partitioned(
2     dataset: pd.DataFrame, target_attribute: str,
3     partition_attribute: str, split: int | float | Set[str],
4 ) -> float:
5     """
6     Calculate the impurity for a given target attribute in a dataset if the dataset
7     is partitioned by a given attribute and split.
8
9     Parameters:
10    dataset (pd.DataFrame): The dataset to calculate the impurity for
11    target_attribute (str): The target attribute used as the class label
12    partition_attribute (str): The attribute that is used to partition the dataset
13    split (int | float | Set[str]): The split used to partition the partition attribute.
14    If the partition attribute is discrete-valued, the split is a set of strings
15    (Set[str]). If the partition attribute is continuous-valued, the split is a
16    single value (int or float).
17    """
18    # TODO

```

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:

```

1 pytest tests/gini_index/test_calculate_impurity_partitioned.py

```

Task 1.1.2.3

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:

```

1 def calculate_gini_index(
2     dataset: pd.DataFrame, target_attribute: str,
3     partition_attribute: str, split: int | float | Set[str],
4 ) -> float:
5     """
6     Calculate the Gini index (= reduction of impurity) for a given target attribute in a
7     dataset if the dataset is partitioned by a given attribute and split.
8
9     Parameters:
10    dataset (pd.DataFrame): The dataset to calculate the Gini index for
11    target_attribute (str): The target attribute used as the class label
12    partition_attribute (str): The attribute that is used to partition the dataset
13    split (int | float | Set[str]): The split used to partition the partition attribute.
14    If the partition attribute is discrete-valued, the split is a set of strings
15    (Set[str]). If the partition attribute is continuous-valued, the split is a
16    single value (int or float).
17
18    Returns:
19    float: The calculated Gini index
20    """
21    # TODO

```

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:

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

Task 1.2: Training

(20 Points)

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

Task 1.2.1

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:

```
1 def _calculate_information_gain(  
2     self, data: pd.DataFrame, attribute: str  
3 ) -> Tuple[float, List[DecisionTreeDecisionOutcome]]:  
4     """  
5     Calculate the (best) information gain for a given attribute in a dataset.  
6  
7     Parameters:  
8     data (pd.DataFrame): The dataset to calculate the information gain for  
9     attribute (str): The attribute to calculate the information gain for  
10  
11     Returns:  
12     float: The calculated information gain  
13     List[DecisionTreeDecisionOutcome]: The outcomes the best split of this attribute has  
14     """  
15     # If self.target_attribute is not set, raise an error  
16     if self.target_attribute is None:  
17         raise ValueError("Target attribute not set.")  
18  
19     # If the attribute is not in the dataset, raise an error  
20     if attribute not in data.columns:  
21         raise ValueError(f"Attribute '{attribute}' not in dataset.")  
22  
23     # TODO
```

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:

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

Task 1.2.2

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:

```
1 def _calculate_gini_index(  
2     self, data: pd.DataFrame, attribute: str  
3 ) -> Tuple[float, List[DecisionTreeDecisionOutcome]]:  
4     """  
5     Calculate the (best) gini index for a given attribute in a dataset.  
6  
7     Parameters:  
8     data (pd.DataFrame): The dataset to calculate the gini index for  
9     attribute (str): The attribute to calculate the gini index for  
10  
11     Returns:  
12     float: The calculated gini index (reduction of impurity)  
13     List[DecisionTreeDecisionOutcome]: The outcomes the best split of this attribute has  
14     """  
15     # If self.target_attribute is not set, raise an error  
16     if self.target_attribute is None:  
17         raise ValueError("Target attribute not set.")  
18  
19     # If the attribute is not in the dataset, raise an error  
20     if attribute not in data.columns:  
21         raise ValueError(f"Attribute '{attribute}' not in dataset.")  
22  
23     # TODO
```

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:

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

Task 1.2.3

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:


```

1 def _find_best_split(
2     self, data: pd.DataFrame, attribute_list: List[str], attribute_selection_method: str,
3 ) -> Tuple[str, List[DecisionTreeDecisionOutcome]]:
4     """
5     Find the best split for a given dataset and attribute list. Finding the best split
6     includes finding the best attribute to split on and also (depending on the attribute
7     selection method) the best set of outcomes to split on this attribute.
8
9     Parameters:
10    data (pd.DataFrame): The dataset to find the best splitting attribute for
11    attribute_list (List[str]): The list of attributes to consider
12    attribute_selection_method (str): The attribute selection method to use
13
14    Returns:
15    str: The attribute to split on
16    List[DecisionTreeDecisionOutcome]: The outcomes a split on this attribute should have
17    """
18    # TODO

```

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 `DecisionTreeDecisionOutcome`s.

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

```

1 pytest tests/decision_tree/test_find_best_split.py

```

Task 1.2.4

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

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

```

1 def _build_tree(
2     self,
3     data: pd.DataFrame,
4     attribute_list: List[str],
5     attribute_selection_method: str,
6 ) -> DecisionTreeNode:
7     """
8     Recursively build the decision tree.
9
10    Parameters:
11    data (pd.DataFrame): The (partial) dataset to build the decision tree with
12    attribute_list (List[str]): The list of attributes to consider
13    attribute_selection_method (str): The attribute selection method to use
14
15    Returns:
16    DecisionTreeNode: The root node of the decision tree
17    """
18    # TODO

```

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:

```

1 pytest tests/decision_tree/test_build_tree.py

```

Task 1.2.5

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:

```
1 def fit(  
2     self, dataset: pd.DataFrame,  
3     target_attribute: str, attribute_selection_method: str,  
4 ):  
5     """  
6     Fit decision tree on a given dataset and target attribute, using a specified  
7     attribute selection method.  
8  
9     Parameters:  
10    dataset (pd.DataFrame): The dataset to fit the decision tree on  
11    target_attribute (str): The target attribute to predict  
12    attribute_selection_method (str): The attribute selection method to use  
13    """  
14    # Make sure that the target_attribute is in the dataset  
15    if target_attribute not in dataset.columns:  
16        raise ValueError(f"Target attribute '{target_attribute}' not in dataset.")  
17  
18    # Make sure that the attribute_selection_method is valid  
19    if attribute_selection_method not in ["information_gain", "gini_index", ]:  
20        raise ValueError(f"Attribute selection method '{attribute_selection_method}',  
21                           not valid (select either 'information_gain' or 'gini_index').")  
22  
23    # TODO
```

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:

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

Task 1.3: Prediction

(4 Points)

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

Task 1.3.1

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:

```

1 def _predict_tuple(
2     self, tuple: pd.Series, node: DecisionTreeNode
3 ) -> str | int | float:
4     """
5     Predict the target attribute for a given row in the dataset.
6     This is a recursive function that traverses the decision tree until a leaf node
7     is reached.
8
9     Parameters:
10    tuple (pd.Series): The row to predict the target attribute for
11    node (DecisionTreeNode): The current node in the decision tree
12
13    Returns:
14    str | int | float: The predicted class label
15    """
16    # TODO

```

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:

```

1 pytest tests/decision_tree/test_predict_tuple.py

```

Task 1.3.2

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:

```

1 def predict(self, dataset: pd.DataFrame) -> List[str | int | float]:
2     """
3     Predict the target attribute for a given dataset.
4
5     Parameters:
6     dataset (pd.DataFrame): The dataset to predict the target attribute for
7
8     Returns:
9     List[str | int | float]: A list of predicted class labels
10    """
11
12    # If the tree is not fitted, raise an error
13    if self.tree is None:
14        raise ValueError("Tree not fitted. Call fit method first.")
15
16    # TODO

```

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:

```

1 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

(14 Points)

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

Task 2.1.1

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:

```
1 def _calculate_prior_probabilities(  
2     self, dataset: pd.DataFrame  
3 ) -> NaiveBayesPriorProbabilities:  
4     """  
5     Calculate the prior probability for each class.  
6     (The target attribute has to be set before calling this method.)  
7  
8     Parameters:  
9     dataset (pd.DataFrame): The training dataset  
10  
11     Returns:  
12     NaiveBayesPriorProbabilities: The prior probabilities for each class  
13     """  
14     # Make sure that the target_attribute is set  
15     if self.target_attribute is None:  
16         raise ValueError("Target attribute not set.")  
17  
18     # TODO
```

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:

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

Task 2.1.2

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:

```
1 def _calculate_likelihoods(self, dataset: pd.DataFrame) -> NaiveBayesLikelihoods:
2     """
3     Calculate the likelihoods for each attribute and class.
4     (The target attribute has to be set before calling this method.)
5
6     Parameters:
7     dataset (pd.DataFrame): The training dataset
8
9     Returns:
10    NaiveBayesLikelihoods: The likelihoods for each attribute and class
11    """
12    # Make sure that the target_attribute is set
13    if self.target_attribute is None:
14        raise ValueError("Target attribute not set.")
15
16    # TODO
```

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:

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

Task 2.1.3

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:

```
1 def fit(self, dataset: pd.DataFrame, target_attribute: str):
2     """
3     Fit the Naive Bayes classifier to the training dataset.
4     Sets the target attribute and the class labels.
5     Calculates the prior probabilities and the likelihoods.
6
7     Parameters:
8     dataset (pd.DataFrame): The training dataset
9     target_attribute (str): The target attribute to predict
10    """
11    # Make sure that the target_attribute is in the dataset
12    if target_attribute not in dataset.columns:
13        raise ValueError(f"Target attribute '{target_attribute}' not in dataset.")
14
15    # TODO
```

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:

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

Task 2.2: Prediction

(4 Points)

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

Task 2.2.1

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:

```
1 def _predict_tuple(self, tuple: pd.Series) -> str | int | float:
2     """
3     Predict the target attribute for a given row in the dataset.
4
5     Parameters:
6     tuple (pd.Series): The row in the dataset to predict the target attribute for
7
8     Returns:
9     str | int | float: The predicted class label
10    """
11    # TODO
```

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:

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

Task 2.2.2

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:

```
1 def predict(self, dataset: pd.DataFrame) -> List[str | int | float]:
2     """
3     Predict the target attribute for a given dataset.
4
5     Parameters:
6     dataset (pd.DataFrame): The dataset to predict the target attribute for
7
8     Returns:
9     List[str | int | float]: A list of predicted class labels
10    """
11
12    # If the likelihoods or/and the prior probabilities are not calculated yet, raise
13    # an error
14    if self.likelihoods is None or self.prior_probabilities is None:
15        raise ValueError("Model not trained yet.")
16
17    # TODO
```

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:

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

Appendices

In [Task 1](#) and [Task 2](#) test cases are provided and used to grade the submission.

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_submission_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