INF264 – Project 1
Implementing Decision Trees and Random Forests

Deadline: September 22nd 2024, 23:59
Deliver your submission on MittUiB

General Information

Projects are a compulsory part of this course. You need to pass both projects to be eligible for the final exam. **The project can be done either alone or in pairs**. If you decide to work in pairs, add a paragraph to your report explaining the division of labor. Note that both students will get the same grade, regardless of the division of labor. Grading will mainly be based on the following three qualities:

- **Correctness.** Your answers are correct and your code works as expected.
- **Clarity of code.** Your code is easy to read and understand for someone who has not seen it before. Use meaningful variable names and comments where necessary.
- **Reporting.** Your report is well-structured and clearly written.

Deliverables

You should deliver **exactly two files**[1]:

1. A **PDF report** containing an explanation of your approach, design choices and results to help us understand how your particular implementation works. Describe your approach and design choices in detail. This include choice of model selection procedure, hyperparameters, scoring metric, etc. You can include figures and snippets of code in the PDF to elaborate any point you are trying to make.
2. A **ZIP file** containing your code. We may want to run your code if we think it is necessary to confirm that it works the way it should. Please include a `README.txt` file in your ZIP file that briefly explains how we should run your code. If you have multiple files in your code directory, you must mention in the `README.txt` file which file is the main file that we need to run to execute your entire algorithm. Note that all the numbers that you report should be reproducible from the code that you submit.

[1]The reason we ask you **not** to put the report inside the ZIP file is that in this way, the graders can use MittUiB"s Speedgrader-functionality. So please make graders" work easier and follow the instructions.

Suggested structure of your code directory. You are encouraged to have three files in your code directory (in addition to the `README.txt` file): `decision_tree.py`, `random_forest.py`, and `run_experiments.ipynb` (notebook) or `run_experiments.py` (script). The first two files should contain the implementation of the decision tree and random forest algorithms, respectively. The third file should contain the code that loads the dataset, perform model selection, and evaluate the performance of your algorithms, i.e., the code we need to run to reproduce your results.

Programming languages. Please submit your code in Python. Notebooks, standalone scripts, or a combination of both are all acceptable. If you want to use another programming language, please ask us first.

Code of Conduct

The goal of the project is learning to implement machine learning algorithms. Therefore, you must write the code yourself. You are not allowed to use existing libraries for the decision tree and random forest implementations, except for basic packages like `numpy` and native Python libraries. You can use existing libraries such as `sklearn` and `matplotlib` for model selection, evaluation and visualization. You are not allowed to copy-paste code from online tutorials or similar. In other words, **submitting code that is written by someone else is considered cheating**. If you are unsure whether something is allowed or not, please ask us first.

Late Submission Policy

All late submissions will get a deduction of 2 points. In addition, there is a 2-point deduction for every starting 12-hour period. That is, a project submitted at 00:01 on September 23rd will get a 4-point deduction and a project submitted at 12:01 on the same day will get a 6-point deduction (and so on). All projects submitted on September 25th or later are automatically failed. (Executive summary: Submit your project on time. The late penalties are designed to be harsh enough so that nobody should benefit by returning their work late. Also note that late penalties can easily drag a good project under the acceptance threshold.) There will be no possibility to resubmit failed projects.

Based on experiences from the previous years, this is a time-consuming project, so **start working early!** Good luck, and make sure to use our Discord channel for questions and discussions.

Project Outline

This project consists of three main parts:

- **Section 1:** Implement a decision tree learning algorithm from scratch.
- **Section 2:** Implement a random forest algorithm using the decision tree implementation from the first part.
- **Section 3:** Evaluate the performance of your algorithms on a real-world dataset and compare them to existing implementations.

1 Decision Trees

In this section, you will implement a decision tree classifier from scratch. You will implement the Iterative Dichotomiser 3 (ID3) learning algorithm using entropy or Gini index as the impurity measure. We recommend that you implement a class called `DecisionTree` that represents the decision tree. The class should (at least) have the methods `fit(X, y)` and `predict(X)` to train the decision tree on a dataset and predict the class labels of new data points, respectively. The constructor of the class, i.e., `__init__`, should support the following arguments:

- `criterion`: The impurity measure to use. It should be either `"entropy"` or `"gini"`. The default value should be `"entropy"`.
- `max_depth`: The maximum depth of the tree. The default value should be `None`, which means that the tree will grow until all leaves are pure.

Some Tips

- It can be helpful to create a method that prints the decision tree in a human-readable format for debugging and visualization purposes.
- It is often good to split the implementation into multiple methods and/or functions to make the code more readable and easier to debug.
- Use `numpy` to calculate the entropy and information gain efficiently. Using `pandas` is fine for loading and manipulating datasets, but you should convert the data to `numpy` arrays before training the decision tree.
- Test your implementation regularly on a simple dataset to verify that it works as expected. You can use the `make_classification()` function from `sklearn.datasets` to generate a synthetic dataset for testing with a known number of features and classes.
- Do not move on to the random forest part until you are confident that your decision tree implementation works as expected.

1.1 The ID3 Algorithm

The ID3 algorithm is a greedy algorithm that builds a decision tree by recursively selecting the feature that maximizes the information gain at each node. The algorithm works as follows starting with all the training data at the root node:

- If all data points have the same label, return a leaf node with that label.
- If all data points have identical feature values, return a leaf node with the most common label.
- Otherwise, choose a feature that maximizes the information gain, split[2] the data based on the value of the feature, and add a branch for each subset of data. For each branch, call the algorithm recursively for the data points belonging to the particular branch.

[2]There are many ways one can choose the the possible values to split on. These include the mean, median, quantiles or all the unique midpoints between the feature values. For speed and simplicity, you are encouraged to use the mean or median, but you can experiment with other methods if you want.

You should implement the ID3 algorithm in the `fit()` method of your decision tree class using entropy as the impurity measure. The method should take a dataset `X` and labels `y` as input and build a decision tree that can be used to predict the class labels of new data points.

The `fit()` method should work for any number of features and classes. But, you can assume that all features in the dataset are continuous and that the labels are categorical/discrete. You can also assume that the dataset does not contain missing values.

To implement the `predict()` method, you should traverse the decision tree to predict the class label of a new data point. The method should take a dataset `X` as input and return the predicted class labels.

1.2 The Gini Index

The Gini index is another impurity measure that can be used to build decision trees. Implement the Gini index as an alternative impurity measure and make it possible to choose between the two impurity measures when creating the decision tree by setting the `criterion` argument in the constructor.

1.3 Maximum Depth

The maximum depth of the tree is a hyperparameter that controls how deep the tree can grow. If the maximum depth is reached, the tree will not split the data further and will return a leaf node with the most common label. Implement the maximum depth hyperparameter in your decision tree class' `fit()` method. The value should be specified in the constructor as an argument `max_depth`.

2 Random Forests

Random forest is an ensemble learning method that combines multiple decision trees to create a more robust and accurate model. Ensemble methods work by training multiple models and combining their predictions to make a final prediction, and will be covered in more detail later in the course.

In this section, you will implement a random forest class that uses the decision tree implementation from the previous section. Your class should be called `RandomForest` and should have a constructor that supports the following arguments:

- `n_estimators`: The number of trees in the forest.
- `max_depth`: The maximum depth of the trees in the forest.
- `criterion`: The impurity measure to use. It should be either `"entropy"` or `"gini"`.
- `max_features`: The number of features to consider when looking for the best split. Possible values are `"sqrt"`, `"log2"`, or `None`.

The method `fit(X, y)` should train the random forest on the dataset `X` and labels `y`. The method `predict(X)` should predict the class labels of the data points in `X`.
Fitting a random forest classifier involves training multiple decision trees on random subsets of the data (bootstrap samples) and combining their predictions via majority voting (aggregating). This

process is known as bagging (bootstrap aggregating) and can be used for any base classifier, not just decision trees. We will also implement random feature selection, where the number of features considered for each split is limited to a random subset of the total number of features.

2.1 Modifying the Decision Tree

For the random feature selection, you need to modify the decision tree implementation to consider only a subset of the features when looking for the best split[3]. You can implement this by adding a new argument `max_features` to the constructor of the decision tree class. The value of `max_features` should be stored in the decision tree object and used when looking for the best split. The default value should be `None`, which means that all features should be considered. If the value is `"sqrt"`, you should consider the square root of the total number of features, and if the value is `"log2"`, you should consider the log base 2 of the total number of features. These numbers should be rounded down to the nearest integer.

You can use the `numpy.random.choice()` function to sample a random subset of the features without replacement by specifying the `replace=False` argument.

2.2 Building the Random Forest

Implement the `fit(X, y)` method of the random forest class by training `n_estimators` decision trees on random subsets of the data. You can use the `numpy.random.choice()` function to sample a random subset of the data with replacement by specifying the `replace=True` argument. The size of the subset should be the same as the size of the original dataset.
Remember to pass along the values of the `max_features`, `max_depth`, and `criterion` arguments when creating the decision trees. Store the trained decision trees in a list to use for prediction later.

2.3 Making Predictions

Implement the `predict(X)` method of the random forest class by aggregating the predictions of the individual decision trees. That is, for each data point in `X`, make a prediction using each decision tree and combine the predictions via majority voting. The predicted class label should be the one that occurs most frequently among the decision trees.

3 Evaluating your Implementations

You are at the anual wine tasting party at the laboratory and the lights go out. To identify which of the wines are red and which are white, you decide to use your newly implemented decision tree and random forest algorithms to save the day. Your colleagues have analyzed the wines and provided you with a dataset to train and evaluate your models.

Download the provided dataset `wine_dataset_small.csv` from MittUiB. The dataset contains 5 numerical features and a binary class label indicating whether the wine is red or white. The dataset

[3]It is also possible to sample a random subset of the features on a per-tree basis. We will however implement node-level feature selection as this is more common in practice.

is based on the Wine Quality dataset [Cor+09] and contains the following features: `citric acid`, `residual sugar`, `pH`, `sulphates`, and `alcohol`. The last column `type` is the categorical (binary) class label where 0 indicates a white wine and 1 indicates a red wine. There are 500 rows (different wines) in the dataset. You can use the following code to load the dataset using only `numpy`:

```
import numpy as np

data = np.genfromtxt("wine_dataset_small.csv", delimiter=",", dtype=float, names=True)
feature_names = list(data.dtype.names[:-1])
target_name = data.dtype.names[-1]

X = np.array([data[feature] for feature in feature_names]).T
y = data[target_name].astype(int)

print(f"Feature columns names: {feature_names}")
print(f"Target column name: {target_name}")
print(f"X shape: {X.shape}")
print(f"y shape: {y.shape}")
```

3.1 Model Selection

Choose the best settings for the decision tree and random forest algorithms by performing model selection. This is an open-ended task, and you should consider the following questions when designing your model selection process:

- Which hyperparameters should you tune?
- Which values should you test for each hyperparameter?
- Which model selection method should you use (e.g., hold-out validation, k-fold cross-validation)?
- Which performance measure should you use for model selection (e.g., accuracy, F1-score)?
- How do you ensure that your model selection process is fair and unbiased?
- How can you ensure reproducibility of your results?

In your report, describe your model selection process in detail and explain your choices. You should also include the results of your model selection process, such as the best hyperparameters and the performance of the selected models on the validation set.

3.2 Model Evaluation and Comparison to Existing Implementations

Evaluate the performance of the selected decision tree and random forest models on the test set. Report the performance of the models using an appropriate performance measure and compare the results of the two models. You should also compare your implementation to an existing decision tree and random forest implementation, such as the `DecisionTreeClassifier` and `RandomForestClassifier` classes from `sklearn`.

Remember to also perform model selection for the existing implementations to ensure a fair comparison. Report the performance of the existing implementations on the test set and compare the results to your own implementations.

3.3 Another Dataset

In this section, you will test your decision tree and/or random forest implementation on a different real-world dataset. You can download the dataset `coffee_data.csv`[4].

The task here is to predict the country of origin of a coffee bean based on 8 numerical cupping scores. Cupping is a standardized way of evaluating coffee beans and is used by fancy coffee people to assess the quality of coffee beans. The following features, all taking values in the interval $[0, 10]$, are provided for each coffee bean: `Aroma`, `Flavor`, `Aftertaste`, `Acidity`, `Body`, `Balance`, `Uniformity` and `Sweetness`. In addition to these 8 columns, the last column named `Country.of.Origin` is the categorical (binary) class label. The country of origin is encoded as 0 or 1 (see the column **Label** in table 1).

Origin	**Label**	**Samples**
Colombia	0	183
Mexico	1	236

Table 1: Summary of the provided dataset showing the different countries of origin, their corresponding integer label, and the number of samples belonging to each class.

Redo your model selection process and evaluation on this dataset. Do you get the same hyperparameters as for the wine dataset? How do the performance of your models compare on the two datasets? Discuss the results in your report and include any additional insights you may have gained from working with the coffee dataset.

References

[Cor+09] Paulo Cortez et al. *Wine Quality*. UCI Machine Learning Repository. 2009. DOI: `https://doi.org/10.24432/C56S3T`.

[4]The dataset is based on the one from `https://github.com/jldbc/coffee-quality-database`.