Programming assignment 2: Random forests

In this assignment, we will continue the investigations of tree-based models for tabular data that we started in Programming assignment 1.

You should structure your solution as a **notebook**. You have two choices: either you work on your own machine using a Jupyter notebook. (You can work with Jupyter directly or use the notebook support of popular IDEs such as PyCharm or VS Code.) Alternatively, you use the Colab service, which works in a similar fashion. (Here is a document about Colab we prepared for another course. You can ignore the GPU-related parts for now.) In case you use Colab, you can submit your solution either by downloading the notebook (`.ipynb`) or by providing a link to the notebook. (In that case, make sure you allow access to the notebook.)

When you work with the notebook, you will enter your solution in the code cells. In addition, make sure that you add text cells that explain briefly what you are doing, and answer any questions you find in the assignment instructions. You should think of the notebook as a whole as a technical report.

This is a **group assignment** and your work should be carried out in a group of 2 or 3 people. If you don't have a group partner, please ask around in the Discussion tool in Canvas. Please include **the names of the group members** in your submission.

Submit your solution through the Canvas website.

If you need help, there will be assistance sessions where you can talk to teaching assistants on January 27. Please use this spreadsheet and enter a Zoom link so that the TA can reach you.

Deadline: February 1

Didactic purpose of this assignment:

- encoding categorical data for use with machine learning algorithms;
- understanding random forest learning algorithms for classification;
- continuing the investigations of overfitting from Programming assignment 1;
- learning to interpret a model by looking at feature importance scores.

References

- Lecture 2 on ensembles and random forests.
- Lecture 3 on preprocessing.

Task 1: Working with a dataset with categorical features

In Assignment 1, we didn't have to do much preprocessing, because all the features in the two datasets were *numerical*. (Actually, in the second dataset, we removed all non-numerical features.) In this assignment, we'll instead consider how to deal with non-numerical features.

We'll use the famous Adult dataset. This is a binary classification task, where our task is to predict whether an American individual earns more than $50,000 a year, given a number of numerical and categorical features. (The dataset was extracted from a 1994 census database.)

Step 1. Reading the data

Please download the two CSV files, the training set and the test set, and save them into your working directory This is the official train/test split defined by the people who created the dataset. It's the same data as in the the public distribution, except that we converted the format into a standard CSV format.

Write code to read the CSV file, for instance by using Pandas as in Assignment 1. Then split the data into an input part `X` and an output part `Y`. The output variable, which the classifier will predict, is called `target`.

Step 2: Encoding the features as numbers.

If you look at the data, you will note that it contains several features with categorical values, such as `workclass`, `education` etc. All scikit-learn models work with numerical data internally; this means that the categorical features need to be converted to numbers. The most straightforward way to carry out such a conversion is to use *one-hot encoding* of the features, also known as *dummy variables* in statistics. In this approach, we define one new column for each observed value of the feature.

Scikit-learn includes a number of tools that can do one-hot encoding of categorical features and we'll see how to use one of them, the `DictVectorizer`. An alternative approach that is a bit more Pandas-friendly and gives more low-level control is to use the recently introduced `ColumnTransformer`; if you're interested, you can read an introduction to this approach here. We won't use a `ColumnTransformer` here because it will make Task 3 in this assignment a bit too annoying to solve.

The `DictVectorizer` is used when we store our features as *named attributes* in dictionaries. For instance, we could represent one individual from the Adult dataset as follows:

```
{'age': 44,
 'workclass': 'Private',
 'education': 'Some-college',
 'education-num': 10,
 'marital-status': 'Married-civ-spouse',
 'occupation': 'Machine-op-inspct',
 'relationship': 'Husband',
 'race': 'Black',
 'sex': 'Male',
 'capital-gain': 7688,
 'capital-loss': 0,
 'hours-per-week': 40,
 'native-country': 'United-States'}
```

Pandas includes a utility to convert a `DataFrame` into a list of dictionaries:

```
dicts_for_my_training_data = my_training_data.to_dict('records')
```

Then make a `DictVectorizer` and apply it, writing something like the following:

```
dv = DictVectorizer()
X_train_encoded = dv.fit_transform(dicts_for_my_training_data)
```

The method `fit_transform` will first call `fit`, which as usual is the "training" method. For a `DictVectorizer`, "training" consists of building the mapping from categories to column positions. Then, the `transform` method will be called, which converts the data into a matrix.

Now that you have a numerical representation of the data, you can compute a cross-validation accuracy for the training set using one of the classifiers you explored in Programming Assignment 1.

To handle the test data, you just call `transform`, because this time the vectorizer does not need to be "trained." Use this to compute the accuracy on the test set.

```
X_test_encoded = dv.transform(dicts_for_my_test_data)
```

Step 3. Combining the steps.

In the example above, we first transformed the list of dictionaries into a numerical matrix, and then we used this matrix when training the classifier. A separate preprocessing step was carried out for the test set.

In machine learning setups, we often use long chains of preprocessing steps. The one-hot encoding is one example, and other such steps might be scaling, feature selection, imputation of missing values, etc. As you can imagine, keeping track of the preprocessing steps can be tedious and error-prone, so it makes sense to handle such preprocessing chains automatically.

A `Pipeline` consists of a sequence of scikit-learn modules. The most convenient way to build a `Pipeline` is to use the utility function `make_pipeline`. For instance, to build a pipeline consisting of a vectorization step and then a decision tree classifier, we could write

```
from sklearn.pipeline import make_pipeline

pipeline = make_pipeline(
  DictVectorizer(),
  DecisionTreeClassifier()
)
```

The `Pipeline` can be treated as any classifier: we can call `fit` and `predict` as usual. Concretely, when we call `fit` on a `Pipeline`, it will in turn call `fit_transform` on all intermediate steps and then `fit` on the final step. When we call `predict`, `transform` will be called on the intermediate steps and then `predict` on the final step.

Build a pipeline that includes the classifier that you selected previously, and make sure that it works.

Task 2: Decision trees and random forests

In the previous assignment, in one of the optional tasks (Task 4, step 4) we investigated the performance of a regression model as a function of the depth of the decision trees.

Underfitting and overfitting in decision tree classifiers.

As the first step, please reproduce this experiment for this dataset, but now using scikit-learn's `DecisionTreeClassifier` instead of your own regression model. Of course, you should use an evaluation metric for classification, not the mean squared error. Do you see a similar effect now?

Underfitting and overfitting in random forest classifiers.

Replace the `DecisionTreeClassifier` with a `RandomForestClassifier`.

The hyperparameter `n_estimators` defines the number of decision trees used in ensemble.

Investigate how the underfitting/overfitting curve is affected by the number of trees. You can investigate ensemble sizes ranging from 1 up until a few hundred. (**Clarification**: you will produce one such curve for each ensemble size. The *x* axis still represents the maximal tree depth.)

Hint. These experiments can take some time to run. When `n_estimators` is large, you can reduce the training time quite a bit by adjusting the hyperparameter `n_jobs`, which will train several trees in parallel. By default, only one CPU core is used, but if you set `n_jobs=-1`, all cores will be used.

Some things that you might want to discuss in your report:

- What's the difference between the curve for a decision tree and for a random forest with an ensemble size of 1, and why do we see this difference?
- What happens with the curve for random forests as the ensemble size grows?
- What happens with the best observed test set accuracy as the ensemble size grows?
- What happens with the training time as the ensemble size grows?

Task 3: Feature importances in random forest classifiers

Decision trees and random forests are trained by computing *importance scores* for individual features in different ways: information gain, Gini impurity, variance reduction, etc.

As a way to make our classifiers more interpretable, we can print the importance scores. In scikit-learn, decision trees and ensemble classifiers such as random forests all define an attribute called `feature_importances_` (note the final underscore in this name). This is a NumPy array that stores the importance scores for each feature column in the training data matrix. For random forests and other tree ensembles, these importance scores are computed by averaging the scores when training all the different trees in the ensemble.

To make these importance scores easier to understand, we can use the attribute `feature_names_` (note the underscore again) in the `DictVectorizer`.

Sort the features by importance scores in reverse order (so that the most important feature comes first), inspect the first few of these features, and try to reason about why you got this result.

Hint. If you used a `Pipeline`, you can access the parts of the sequence via the list `pipeline.steps`. For instance, `pipeline.steps[0][1]` will be the first step, `pipeline.steps[1][1]` will be the second step, etc.

Hint. This way of computing feature importance scores just tells us whether a feature is good for discriminating between the classes: it does *not* tell us what the relationship between the feature and an output class is: whether the feature makes it *more* or *less* likely that the person is a high earner.

For your report, please also mention an *alternative* way to compute some sort of importance score of individual features. (You don't need to implement it.) Here, you can either use your common sense, or optionally read the discussion by Parr et al. (2018) that gives some criticism of decision tree-based feature importance scores and discusses some alternatives.