Solution

December 11, 2019

0.1 Exercise 1

Exercise 1.1 For this exercise, we first need to load the Wine dataset into memory. The text of the lab already explains how to do that.

```
In [1]: from sklearn.datasets import load_wine

        dataset = load_wine()
        X = dataset["data"]
        y = dataset["target"]
        feature_names = dataset["feature_names"]

        X.shape, y.shape
```

```
Out[1]: ((178, 13), (178,))
```

```
In [2]: feature_names
```

```
Out[2]: ['alcohol',
         'malic_acid',
         'ash',
         'alcalinity_of_ash',
         'magnesium',
         'total_phenols',
         'flavanoids',
         'nonflavanoid_phenols',
         'proanthocyanins',
         'color_intensity',
         'hue',
         'od280/od315_of_diluted_wines',
         'proline']
```

We now know that we will be using the previously printed 13 features, and that we have a total of 178 points in our dataset. To find out whether any value is missing, we can check using NumPy's `isnan()` function, which returns a mask selecting onlyl "Not a Number" (NaN) values.

```
In [3]: import numpy as np

        X[np.isnan(X)]
```

```
Out[3]: array([], dtype=float64)
```

`isnan()` returns an empty result, meaning that there are no missing values. Please note that, in this case, the dataset has already been prepared so as to have meaningful values (i.e. nothing is missing, and if anything was missing, it would have been flagged as a NaN). This is typically not the case with "dirty" data: in that case, you may want to run more thorough checks (e.g. making sure that everything is cast to a number correctly, that string fields contain meaningful data, and so on).

Finally, we need to count the number of elements available for each of the three classes. For this purpose, we can use a useful class that lets us count objects in a list (or iterable). This is called `Counter` and is contained in the `collections` module. Given an interable, `Counter` returns a dictionary: the keys are the elements contained in the list, the values are the number of times those elements appear in the list. For example,

```
In [4]: from collections import Counter

        Counter(['a','b','a','a','a','c','c','d'])

Out[4]: Counter({'a': 4, 'b': 1, 'c': 2, 'd': 1})
```

We can use `Counter` on our `y` to get the count of each of the three classes.

```
In [5]: Counter(y)

Out[5]: Counter({0: 59, 1: 71, 2: 48})
```

This means that the problem is not perfectly balanced (approximately 33%/40%/27%).

Exercise 1.2 For this exercise, we simply need to build a decision tree based on our `X` and `y`. We will be using a default configuration for the tree, so with no additional arguments passed when created.

```
In [6]: from sklearn.tree import DecisionTreeClassifier

        clf = DecisionTreeClassifier()
        clf.fit(X, y)

Out[6]: DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
                               max_features=None, max_leaf_nodes=None,
                               min_impurity_decrease=0.0, min_impurity_split=None,
                               min_samples_leaf=1, min_samples_split=2,
                               min_weight_fraction_leaf=0.0, presort=False,
                               random_state=None, splitter='best')
```

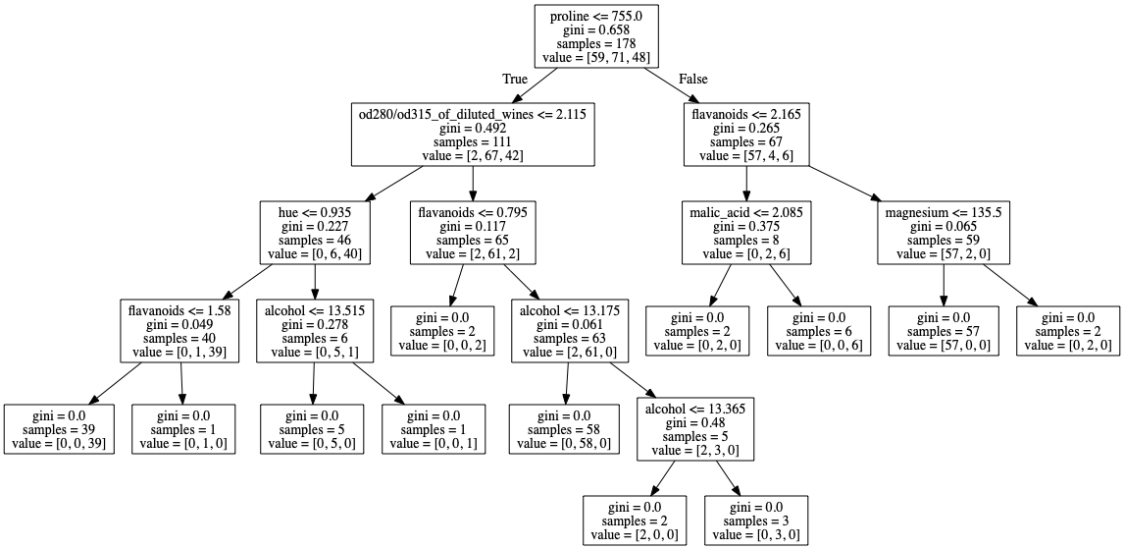
Notice how we do not need to normalize the data before training the classifier. You may have learned that, in some cases, data normalization is a necessary step when preprocessing data. For decision trees, though, this step is not required: considering that the tree is built by considering one feature at a time, different features will never be compared against one another (nor will linear combinations be ever used). Because of this, you will typically not need to normalize your data with trees.

Exercise 1.3 After having installed the required libraries, we can use the provided code snippet to display the obtained decision tree.

```
In [7]: import pydot
        from IPython.display import Image
        from sklearn.tree import export_graphviz

        dot_code = export_graphviz(clf, feature_names=feature_names)
        graph = pydot.graph_from_dot_data(dot_code)
        Image(graph[0].create_png())
```

Out[7]:



For, notice how each split generates two branches: as can be seen from the root split, the convention used is that the left-most branch is the one selected when the comparison yields a "true" value, the right-most one if the comparison is false.

We can see that each of the nodes of the tree contain some information: - the attribute used for the split, and the threshold value used (e.g. `proline <= 755.0`), This is true for all nodes except the leaves, which obviously do not need to split anything - the GINI index for each node. We are using GINI as an indicator of how pure each node is (see `DecisionTreeClassifier`'s documentation to learn what the default values for the decision tree are). Because of this, the visualization will include the GINI index for each node. You may have learned that, for binary problems, the GINI index ranges between 0 and 0.5. In this case, where we have 3 classes, the GINI index can be higher than that (indeed, the root node has an index of 0.658). Based on the definition of GINI index, we can work out that, for an N-classes problem, the GINI index will range from 0 to 1-1/N. In this case, the maximum GINI index we can get is of 0.667 (for a node that has the same number of elements for each of the three classes). This is almost the case for the root node but, considering that the problem is not perfectly balanced (as already discussed), the GINI value is slightly lower than that - the number of samples that can be found at that node. It starts from 178 for the root node, and defines, for each node, how many of the points of the dataset reach that

node - the number of samples that reach the node, divided by class. This is useful for computing the GINI index at each node. For the root node, for example, the GINI will be `1 - (59/178)**2 - (71/178)**2 - (48/178)**2`. Indeed,

```
In [8]: 1 - (59/178)**2 - (71/178)**2 - (48/178)**2

Out[8]: 0.6583133442747129
```

With all this information, we can analyze the tree in more detail. In particular, we can immediately notice that some of the nodes are "taking decisions" (i.e. they are splitting) based on very small portions of the training dataset. For example, the split for `flavanoids <= 1.58` separates 39 elements belonging to class "2" from a single element belonging to class "1". Now, this might be a meaningful split. It is more likely, though, that the tree is only adding a split there because "we asked it to" (i.e. the tree only stops when it reaches pure nodes). Most likely, the split added there is only useful to separate that very specific point from class "1" to those specific 39 points in class "2", and can unlikely be generalized to new data. This is a good example of what overfitting is. If we were to stop the construction of the tree one step earlier (i.e. at the node with [0,1,39]) and assign the majority label ("2") to all points reaching that node, we would probably do a better job at generalizing new points.

We can also observe, from this decision tree, that many features are used multiple times. `flavanoids`, for example, is used in 3 separate occasions. This is fine, since the decision tree does not "discard" any of the features it has already used. Additionally, this feature is used in three different root-to-leaf paths, so there is no overlapping at all. Instead, the `alcohol` feature is used twice in the same root-to-leaf path (`alcohol <= 13.175` and `alcohol <= 13.365`). Notice that, in these cases, the different thresholds will be defined so as to be compatible with one another: since the `alcohol <= 13.365` split is at the right of the previous node, it means that `alcohol` is already > 13.175 and, indeed, the next check is for `alcohol` and the value 13.365. This implies that elements in the left leaf will have a value for alcohol that is between 13.175 and 13.365.

Exercise 1.4 Let's now make some predictions. We will use `accuracy_score()` to compute the accuracy of our classifier on the same data we used for the training.

```
In [9]: from sklearn.metrics import accuracy_score

        accuracy_score(y, clf.predict(X))

Out[9]: 1.0
```

The result, unsurprisingly is of 1 (i.e. 100% accuracy). This is a particularly high value. You are unlikely to find any non-trivial problem where you can achieve a 100% accuracy. The reason why we are getting such a high accuracy is that we are testing our model on the same data that we used for the training. Typically, the model will learn the training data particularly well. As such, it is not particularly useful to compute the accuracy on the same data that we used for the training, as that will not tell us anything about how good our classifier is at generalizing to new data.

Exercise 1.5 Instead of training our dataset with all the data, we will now split the dataset into two parts: a training set for the actual training of the model, and a test set to assess how well our model performs on new data (the model will only "see" the test data after its training, so it will not be able to learn anything new from it).

We can use the `train_test_split()` function from sklearn to split any number of arrays (in our case two, `X` and `y`) into two non-overlapping arrays. We can specify how large we want our test (or training) data to be, as a fraction (in our case, we will use 80% for the training data, and 20% for the test). This function also takes care of shuffling the data for us:

```
In [10]: from sklearn.model_selection import train_test_split

         X_train, X_test, \
         y_train, y_test = train_test_split(X, y, test_size=.2)
```

Notice how useful it is for us to have our data shuffled. The following is how the original labels are distributed:

```
In [11]: y

Out[11]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1,
                1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2,
                2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
                2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
                2, 2])
```

The labels are sorted by value. If we were not to shuffle the data, we would get something like the following:

```
In [12]: X_train, X_test,\
         y_train, y_test = train_test_split(X, y, test_size=.2, shuffle=False)

In [13]: y_train, y_test

Out[13]: (array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1,
                 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2,
                 2, 2, 2, 2, 2, 2, 2, 2, 2]),
          array([2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
                 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]))
```

Our test data is comprised only of 2's, and our training data contains only a very limited number of 2's. This is probably not a very well partitioned dataset.

If, instead, we shuffle our data, we expect to find a proportionate number of points for each of the classes. However, if the problem is unbalanced, we might end up not retaining the same data distributions in the two samples.

For example, in our original problem, we had the following fractions of data in each of the three classes:

```
In [14]: c = Counter(y)
         for i in [0,1,2]:
             print(i, c[i]/len(y))
```

```
0 0.33146067415730335
1 0.398876404494382
2 0.2696629213483146
```

```
In [15]: X_train, X_test,\
         y_train, y_test = train_test_split(X, y, test_size=.2)
```

With a random split such as the one above, we might end up with the following partitions:

```
In [16]: c = Counter(y_train)
         print("train")
         for i in [0,1,2]:
             print(i, c[i]/len(y_train))

         c = Counter(y_test)
         print("test")
         for i in [0,1,2]:
             print(i, c[i]/len(y_test))
```

```
train
0 0.33098591549295775
1 0.3873239436619718
2 0.28169014084507044
test
0 0.3333333333333333
1 0.4444444444444444
2 0.2222222222222222
```

This is not a faithful representation of how the data is distributed in the original dataset. This problem can be exacerbated in more unbalanced problems. We can use stratification to overcome this problem: with it, we can preserve the original label distribution. The `stratify` parameter requires the array that defines, for each individual, the subpopulation to which it belongs (in our case, our label).

```
In [17]: X_train, X_test,\
         y_train, y_test = train_test_split(X, y, test_size=.2, stratify=y)
```

```
In [18]: c = Counter(y_train)
         print("train")
         for i in [0,1,2]:
             print(i, c[i]/len(y_train))

         c = Counter(y_test)
```

```
        print("test")
        for i in [0,1,2]:
            print(i, c[i]/len(y_test))
```

```
train
0 0.33098591549295775
1 0.4014084507042254
2 0.2676056338028169
test
0 0.3333333333333333
1 0.3888888888888889
2 0.2777777777777778
```

Exercise 1.6 Now that we have a training and a test set, we can train a new classifier with the first part of the data, and validate it on the second one.

```
In [19]: clf = DecisionTreeClassifier()
         clf.fit(X_train, y_train)
         accuracy_score(y_test, clf.predict(X_test))
```

```
Out[19]: 0.9444444444444444
```

Now, this result is much more meaningful than the previous one. It tells us, approximately, what accuracy we should expect for our model for new data. You may obtain a different value for this accuracy when re-running the code presented above. This is because we are using different training/test splits (remember the shuffling step!). However, you should obtain a value comparable to this one (approximately between 85 and 95%).

Now, the accuracy tells us about the "overall" classifier performance. If we want to know more detailed information (e.g. which classes we are getting right most of the time, or what kind of errors we are typically doing), we can resort to other metrics.

```
In [20]: from sklearn.metrics import confusion_matrix,\
                                    precision_score,\
                                    recall_score,\
                                    f1_score

         y_pred = clf.predict(X_test)
         print(confusion_matrix(y_test, y_pred))
         print(precision_score(y_test, y_pred, average=None))
         print(recall_score(y_test, y_pred, average=None))
         print(f1_score(y_test, y_pred, average=None))
```

```
[[11  1  0]
 [ 1 13  0]
 [ 0  0 10]]
[0.91666667 0.92857143 1.        ]
[0.91666667 0.92857143 1.        ]
[0.91666667 0.92857143 1.        ]
```

From the confusion matrix, there is no particular concern about the validity of our classifier (if the classifier were to consistently mispredict a class for another one, that would have required some additional attention). The f1 score tells us that the classifier has no particular problems predicting the three classes (the f1 score for the first class is slightly higher than the others, but that is it).

We can also use `classification_report()` to get all of the previous information into a single report (useful for an overview of the classifier performance).

```
In [21]: from sklearn.metrics import classification_report

         print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.92      0.92      0.92        12
           1       0.93      0.93      0.93        14
           2       1.00      1.00      1.00        10

    accuracy                           0.94        36
   macro avg       0.95      0.95      0.95        36
weighted avg       0.94      0.94      0.94        36
```

Exercise 1.7 In Exercise 1.3 we noticed how the tree we are training sometimes overfits the data. We can use some of the parameters of the decision tree to lessen the entity of this overfitting. For example, if we limit the maximum depth (`max_depth`) that can be reached by the tree, we might prevent the tree from growing too deep. Since the overfitting typically occurs at deeper levels of the three, where decisions are taken only on small fractions of data, this might be an effective way of "pruning" the tree. Another way of limiting this overfitting is through `min_impurity_decrease`. You can read more about this parameter in the documentation of the decision tree (it is also discussed in Exercise 1.8). In short, this parameter defines what the minimum impurity decrease achieved by a split should be. If all the splits that can be made at a certain point are below this minimum value, no split is made and the node becomes a leaf.

Now, through the "grid search" process we can test various configurations and identify the best one for our problem, but how do we define which sets of values we should try? This requires some additional understanding of the problem and of the parameters themselves. In particular, - for `max_depth`: we already know that, if we let the tree grow indefinitely (as was done in Exercise 1.2), the maximum depth reached is of 5. So, it probably does not make sense to explore values of `max_depth` larger than this number - for `min_impurity_decrease`, we can consider what the impurity decrease is for some of the nodes. For example, the one we have already studied (`flavanoids <= 1.58`). Here, The impurity decrease is (according to the definition used in the documentation): 40/178*0.049 - 39/178*0 - 1/178*0 = 0.011. This is an example of impurity decrease where we do *not* want to perform a split. Instead, for example, the split `od280/od315_of_diluted_wines <= 2.115` is a split we actually want to happen. For this one, the impurity decrease is 111/178*0.492 - 46/178*0.227 - 65/178*0.117 = 0.205. This, as such, approximately defines meaningful values for this split (approximately from 0 to 0.1)

```
In [22]: from sklearn.model_selection import ParameterGrid

         params = {
             "max_depth": [None, 2, 3, 4, 5],
             "min_impurity_decrease": [0, .01, .03, .07, .09, .11]
         }

         accuracies = []
         for config in ParameterGrid(params):
             clf = DecisionTreeClassifier(**config)
             clf.fit(X_train, y_train)
             accuracies.append(accuracy_score(y_test, clf.predict(X_test)))
         max(accuracies)

Out[22]: 0.9722222222222222
```

This is the result we obtain with the previous grid search. Once again, your result may vary based on your train/test data. It is now a good moment to discuss – or rather, to not discuss – the selected configuration.

We are basing this exercise on a toy dataset, with three well-separated classes. This means that most "meaningful" configurations of the decision tree can reach a meaningful solution.

Exercise 1.8 With the previous exercise, we have defined what the best hyperparameters configuration should be. We also have an estimate of how good the best performing configuration is (the one we obtained on the test set). The problem, is that this estimate is, in a way, biased: we have selected the configuration that performs best on our test set. Much like there is a risk of overfitting on the training data because we learn it "too well", there is also a risk of overfitting on the data we use for the validation, if we keep tweaking and adjusting our model to optimize the performance on that dataset.

For this reason, we typically use three separate datasets: the training, the validation and the test set. We use the first one for training the model, the second one to select which model to use (of the many we can create) and the last one to get a final estimate of how good our classifier is.

The problem with adding an additional set is that we have a limited amount of data available for all these operations. Removing an additional portion of data means that we will need to reduce either the number of points used for the training (thus training worse classifiers), or the test set size (thus producing less accurate estimates of our model's performance).

This is why, in cases where data is limited, we typically use the so-called k-fold cross validation. You can learn more about it on the course slides. In short, we use the training set for both training and validation, by rotating the data we use for either operation.

Scikit-learn's `KFold` class helps us do just that. It divides the provided dataset into k splits and returns a generator with the indices we can use (through fancy indexing) for the training and validation.

We can now split our original dataset (X, y) into a test set (`X_test`, which we will set aside and use only for the final evaluation) and a set (`X_train_valid`) that we will use for both training and validation (through k-fold cross validation).

At each iteration a different fold is used for the validation, while the rest of `X_train_valid` will be used for the training. This means that, for each classifier, we will get k accuracies, which we then need to aggregate to obtain a single, "overall" accuracy.

We will do this k-fold validation for each of the possible configurations in our grid search. When this is done we will have, for each configuration, an estimate of its accuracy. We can then select the best classifier based on this. Notice how we have never touch `X_test` to select the classifier. Finally, we can use `X_test` to get an estimate of how good the selected model is on new data.

```
In [23]: from sklearn.model_selection import KFold

         X_train_valid, X_test,\
         y_train_valid, y_test = train_test_split(X, y, stratify=y)
         kf = KFold(5)

         accuracies = []
         for config in ParameterGrid(params):
             clf_accuracies = []
             counts = []
             for train_indices, valid_indices in kf.split(X_train_valid):
                 X_train = X_train_valid[train_indices]
                 y_train = y_train_valid[train_indices]
                 X_valid = X_train_valid[valid_indices]
                 y_valid = y_train_valid[valid_indices]

                 # keep track of the number of elements in each split
                 counts.append(len(train_indices))

                 clf = DecisionTreeClassifier(**config)
                 clf.fit(X_train, y_train)
                 acc = accuracy_score(y_valid, clf.predict(X_valid))
                 clf_accuracies.append(acc)
             accuracies.append(np.average(clf_accuracies, weights=counts))
```

Notice how we are computing the final accuracy not as the mean of the accuracies on each fold, but rather as a weighted average (using `np.average` with the `weights` parameter), where the weight is the number of elements in each partition. This is the correct way of computing the average accuracy when each fold might have a different number of elements. Intuitively, this is because we want larger folds to have a larger contribution to the final results. More practically, one can prove that computing the weighted average is the same as counting the overall number of correct predictions and dividing by the total number of elements (since each point is only used once for the validation, we can talk about the "overall accuracy" on `X_train_valid`).

So now we can get the best performing configuration (through argmax()), train a classifier on all `X_train_valid` (since the cross-validation has already occurred, we can now use all the data available for the new training). Finally, we can get our performance estimate on new data to actually know how good our classifier is.

```
In [24]: best_config = list(ParameterGrid(params))[np.argmax(accuracies)]
         clf = DecisionTreeClassifier(**best_config)
         clf.fit(X_train_valid, y_train_valid)
         accuracy_score(y_test, clf.predict(X_test))
```

```
Out[24]: 0.9333333333333333
```

So, our final classifier has an accuracy of approximately 90%. Once again, your results may vary, since our datasets are so small.

Exercise 1.9

> Even if you are not interested in the solution to this exercise, please jump to the end of the explanation, where you will find useful information on scikit-learn's decision trees.

For this exercise, we need to compute the feature importance of each feature used by the decision tree. You can see the feature importance as an indication of how useful each feature was for the creation of the decision tree. We will be using the same classifier trained in Exercise 1.2, which is defined as:

```
In [25]: clf = DecisionTreeClassifier()
         clf.fit(X, y)
```

```
Out[25]: DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
                                max_features=None, max_leaf_nodes=None,
                                min_impurity_decrease=0.0, min_impurity_split=None,
                                min_samples_leaf=1, min_samples_split=2,
                                min_weight_fraction_leaf=0.0, presort=False,
                                random_state=None, splitter='best')
```

Scikit-learn's feature importance is defined as the total impurity decrease that each feature introduces. Every time a split occurs, a decrease in impurity is occurring. For the computation of the feature importance, each feature is "attributed" all the impurity decreases of the splits that make use of that feature.

Because of this, to compute the feature importance, we first need to compute the impurity decrease of each node of the tree.

Given a scikit-learn's decision, tree, we can access the following information from its tree structure: - `clf.tree_.feature`: a list of the features used at each node. A value of -2 indicates a leaf node (where no split occurs) - `clf.tree_.impurity`: a list of impurity decreases for each node. These are computed with the following formula (the following is taken from scikit-learn's documentation for `DecisionTreeClassifier`: `N_t / N * (imp - N_t_R / N_t * right_imp - N_t_L / N_t * left_imp)` (where `N` is the total number of samples, `N_t` is the number of samples at the current node, `N_t_L` is the number of samples in the left child, and `N_t_R` is the number of samples in the right child) - `clf.tree_.value`: a list of cardinalities of the classes for each node. For example, for the root node, the corresponding `value` would be `[59, 71, 48]`

The lists of `feature`, `impurity` and `value` are the pre-order visits of the tree. Now, to compute the impurity decrease we need to know, for each node, who its children are. This information is not readily available from the pre-order visit. So, with the information we have (and some understanding of the tree data structure) we can reconstruct the tree and compute the impurity decrease at each node.

Notice how those three lists are all we need for our purposes: - `impurity` tells us what the impurity at a single node is (the `imp`, `right_imp` and `left_imp` in the previous formula) - `value` tells us the number of elements in each node. We do not need to know the information at a "single class" level, so we will be aggregating this data to get the "overall" number of elements at each

node (for the root, that would be 178). This information tells us the `N_t`, `N_t_R` and `N_t_L` of the previous formula) - `feature` tells us which feature we should "credit" for each impurity decrease

The first step is to define what a node of the tree will look like for us. For this, we create the `Node` class, which contains all the information we need.

```
In [26]: class Node:
             def __init__(self, impurity=0, feature=-2, value=0):
                 self.right = None
                 self.left = None
                 self.impurity = impurity
                 self.feature = feature
                 self.value = value

             def __repr__(self):
                 return "leaf" if self.feature == -2 else feature_names[self.feature]
```

By default, a new `Node` object will be a leaf (no children, impurity=0 and no feature). We can modify the object's attribute by directly accessing them.

Notice that we also define the `__repr__` method. This method defines a representation of our object (used, for example, when printing it). In our case, we will use the "leaf" string for leaves, and the feature name of the feature used for the split for split nodes.

We can now build a list of nodes in the same pre-order order as `feature`, `impurity` and `value`. Then, we can print them.

```
In [27]: # as already mentioned, we only need
         # the overall count of elements in each node,
         # not the information for each class.
         # The following line computes that list
         values = clf.tree_.value.sum(axis=2).flatten()

         nodes = [ Node(impurity, feature, value) \
                  for impurity, feature, value in \
                  zip(clf.tree_.impurity, clf.tree_.feature, values) ]
         nodes

Out[27]: [proline,
          od280/od315_of_diluted_wines,
          hue,
          flavanoids,
          leaf,
          leaf,
          flavanoids,
          leaf,
          leaf,
          flavanoids,
          leaf,
          alcohol,
          leaf,
          color_intensity,
```

```
         leaf,
         leaf,
         flavanoids,
         flavanoids,
         leaf,
         leaf,
         color_intensity,
         leaf,
         leaf]
```

Then, we can traverse the tree (which we know has a root in `nodes[0]`) to populate the `.right` and `.left` values of each node.

```
In [28]: def traverse_tree(nodes, i):
             # we use `feature==-2` to stop
             # the traversal, as we just reached a leaf
             if nodes[i].feature == -2:
                 # return the offset to sum to
                 # the caller's "i" to get
                 # the next element (1 = "skip this element")
                 return 1
             # next node is the one
             # right next to the current one
             nodes[i].right = nodes[i+1]
             offset_right = traverse_tree(nodes, i+1)
             nodes[i].left = nodes[i+offset_right+1]
             offset_left = traverse_tree(nodes, i+offset_right+1)
             return offset_right + offset_left + 1

         traverse_tree(nodes, 0)
```

```
Out[28]: 23
```

We can now traverse the tree in pre-order and, if we did a good job, the resulting visit should be the same as our previously defined `nodes`.

```
In [29]: def pre_order(node, visited):
             visited.append(node)
             if node.feature==-2:
                 return
             pre_order(node.right, visited)
             pre_order(node.left, visited)

         po_visit = []
         pre_order(nodes[0], po_visit)
         print(po_visit == nodes)
```

```
True
```

Nice :). Now, we can once again traverse the tree in pre order and compute, for each node we visit, its feature importance.

```
In [30]: def pre_order_id(node, tot_size, visited):
             left, right = node.left, node.right
             if node.feature==-2:
                 visited.append(0)
                 return
             decr = (node.impurity * node.value \
                     - left.impurity * left.value \
                     - right.impurity * right.value)/tot_size
             visited.append(decr)
             pre_order_id(node.right, tot_size, visited)
             pre_order_id(node.left, tot_size, visited)

         impurity_decreases = []
         pre_order_id(nodes[0], nodes[0].value, impurity_decreases)
         impurity_decreases
```

```
Out[30]: [0.2517854009364391,
          0.20542179096160534,
          0.03830402214623025,
          0.010955056179775272,
          0,
          0,
          0.009363295880149813,
          0,
          0,
          0.021110973919962688,
          0,
          0.008275370073122866,
          0,
          0.01348314606741573,
          0,
          0,
          0.06105020507820828,
          0.016853932584269662,
          0,
          0,
          0.02171015044753381,
          0,
          0]
```

The list `impurity_decreases` contains the impurity decrease for each node (0 for leaves). We now need to: - aggregate the information at a "feature" level (by summing contributions from different splits, and - normalize the date so that all feature importances sum to 1.

```
In [31]: feature_importances = np.zeros(X.shape[1])
         for feature, decr in zip(clf.tree_.feature, impurity_decreases):
```

```
            feature_importances[feature] += decr
        feature_importances /= feature_importances.sum()
        dict(zip(feature_names, feature_importances))
```

```
Out[31]: {'alcohol': 0.012570564071187309,
          'malic_acid': 0.0,
          'ash': 0.0,
          'alcalinity_of_ash': 0.0,
          'magnesium': 0.0,
          'total_phenols': 0.0,
          'flavanoids': 0.18127152469290994,
          'nonflavanoid_phenols': 0.0,
          'proanthocyanins': 0.0,
          'color_intensity': 0.0534597951279922,
          'hue': 0.05818509146040653,
          'od280/od315_of_diluted_wines': 0.31204257478317693,
          'proline': 0.38247044986432716}
```

Now, with these feature importances, we can observe what the really meaningful features are. `proline` has the highest feature importance and, indeed, it is used for the first split in our tree. The features that have never been used hava an importance of 0 (clearly). Notice how `od280/od315_of_diluted_wines` has a high feature importance: if you check the decision tree we plotted in Exercise 1.3, you will see that this feature is used pretty soon (so, with a lot of data) to make a useful split. Notice, instead, how features that are used multiple times (e.g. `alcohol`, `color_intensity` have a low importance. Despite being used a lot, you can see that these features have been used for small splits, without significant impurity decreases.

Instead of implementing the entire tree as we did in this solution, you may want to try a simpler way of computing the feature importance: scikit-learn's trees offer `clf.tree_.children_right` and `clf.tree_.children_left`, which can tell you where the right and left children of each node are (once again, the lists are pre-ordered).

Finally, while this exercise was an entertaining way of computing feature importances, you will not need to go through all of this every time you build a new tree. `DecisionTreeClassifiers` offer the `feature_importances_` attribute that contains exactly the values we just computed. Indeed, we can see that the computed feature importances match.

```
In [32]: np.allclose(feature_importances, clf.feature_importances_)
```

```
Out[32]: True
```

(We are using `np.allclose()`, a function that compares to arrays for equality within a tolerance (by default, of 10^-5))

0.2 Exercise 2

Exercise 2.1 As usual, we first need to load the data into memory. The dataset is contained in a csv file, so we can use pandas to load it.

```
In [33]: import pandas as pd

         df = pd.read_csv("synth.csv")
         df.values.shape
```

```
Out[33]: (500, 3)
```

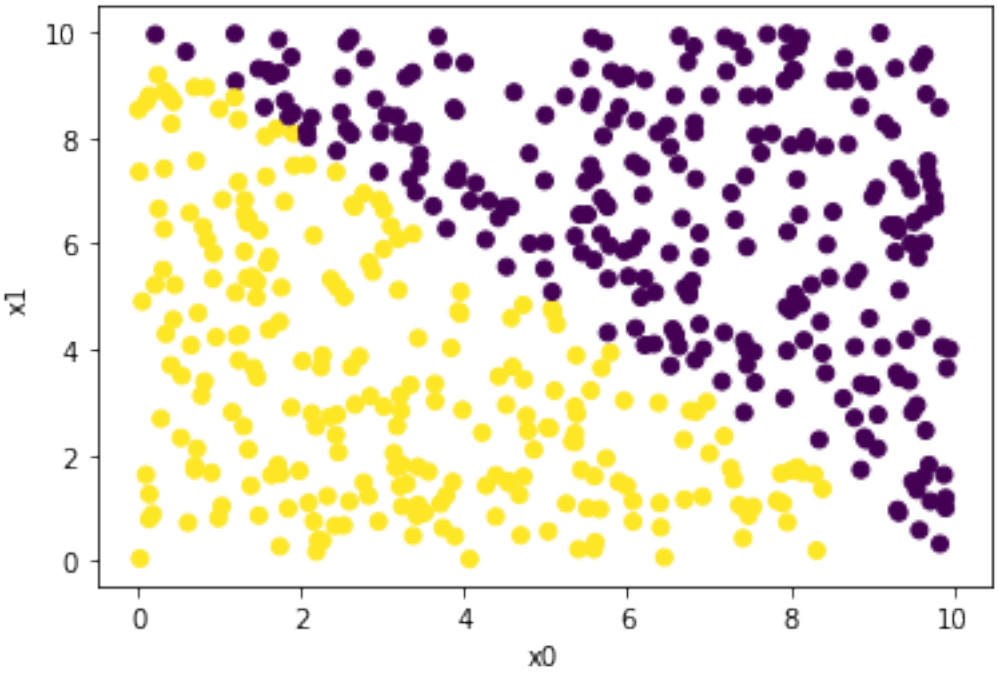
For a pandas DataFrame, we can access to the underlying NumPy matrix through the `values` attribute. In this case, our dataset has 500 rows and 3 columns: 2 features and a class label.

Being the problem 2-dimensional, we can use a scatter plot to visualize it.

```
In [34]: import matplotlib.pyplot as plt
         %matplotlib inline

         X = df.values[:, :2]
         y = df.values[:, 2]
         plt.scatter(X[:, 0], X[:, 1], c=y)
         plt.xlabel("x0")
         plt.ylabel("x1")

Out[34]: Text(0, 0.5, 'x1')
```



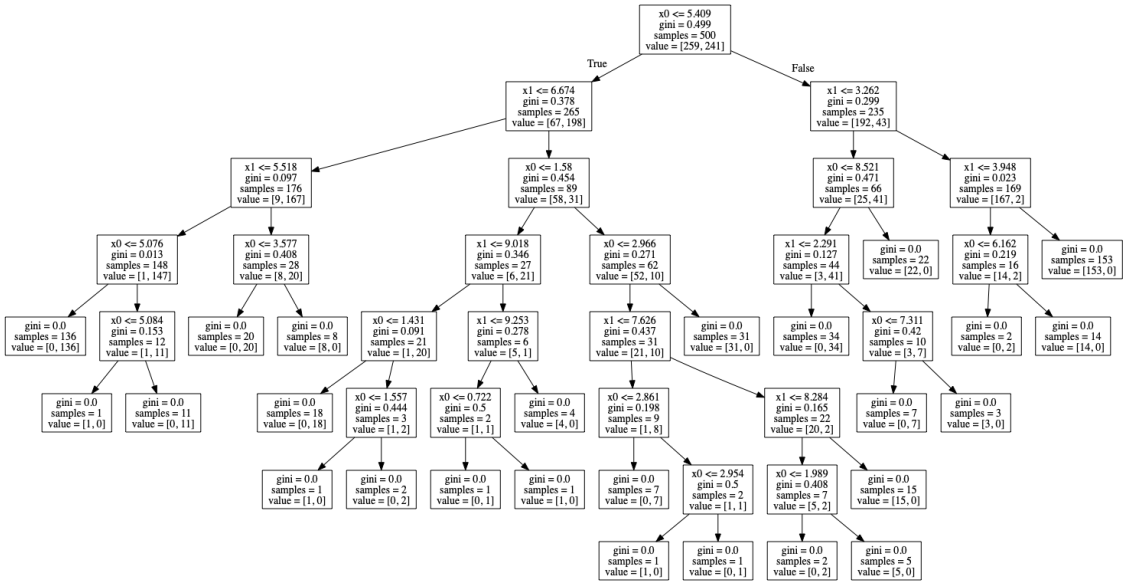
The problem, as we can see, is really simple. A slanted line can separate the problem easily. Unfortunately, decision trees can only take decisions that are "perpendicular" to the axes. This means that we may have the split `x0 < 5`, which is perpendicular to the x0 axis, but we will never have a `x0 + 2*x1 < 3` kind of split: this makes it obvious that a decision tree will not be able to separate the classes "cleanly". Instead, it will probably iteratively try to isolate rectangular "chunks" of the space.

Exercise 2.2 We can now build our decision tree and visualize it.

```
In [35]: clf = DecisionTreeClassifier()
         clf.fit(X, y)

         dot_code = export_graphviz(clf, feature_names=["x0", "x1"])
         graph = pydot.graph_from_dot_data(dot_code)
         Image(graph[0].create_png())
```

Out[35]:



As expected, the decision tree is not capable of separating this linearly separable problem with a "simple" solution. We can say that the tree is not "learning" the underlying rationale behind the data. Instead, it tries to approximate the data as best it can with what it can do.

Exercise 2.3 We already know that decision trees can only work on a single feature at a time, and that it can only compute splits based on `feature <= threshold`. We also know (by observing our data), that the problem can be separated by a line in the form of `x0 + x1 + K = 0` (where `K` needs to be defined).

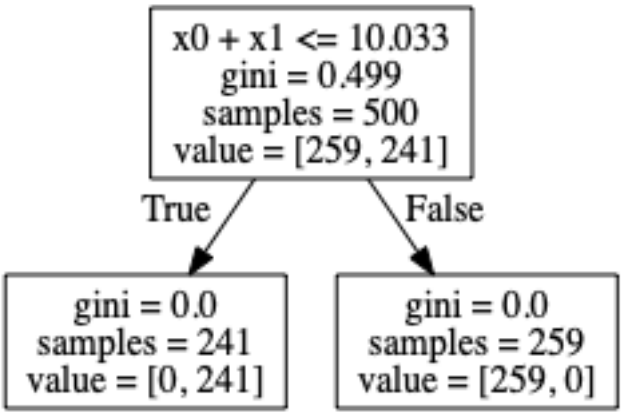
With these two pieces of information, we can build a dataset `X_` more suitable for our decision tree: this new dataset will contain the sum of `x0` and `x1`, so that the decision tree will be able to compute the right `K` for the split.

```
In [36]: X_ = (X[:, 0] + X[:, 1]).reshape(500,1)

         clf_ = DecisionTreeClassifier()
         clf_.fit(X_, y)

         dot_code = export_graphviz(clf_, feature_names=["x0 + x1"])
         graph = pydot.graph_from_dot_data(dot_code)
         Image(graph[0].create_png())
```

Out[36]:



Indeed, now that the decision tree has the means to learn the correct rationale behind the classes, it learns a much simpler rule for classifying the data.

Exercise 2.4 For this exercise, we need to once again consider `clf.tree_`'s attributes. In this case, we will make use of `feature` and `threshold`. While we know the former from a previous exercise, the latter simply contains the threshold value used for each split.

Both lists are in pre-order. So, we will need to once again visit the tree. This time, though, we will also need to keep track, at each step of the visit, what the "bounding box" for the current node is. This can be defined in terms of the tuple `(x0_min, x0_max, x1_min, x1_max)`. We need this information because, every time we draw a split, we want its line to be contained within the area of the 2d space that is relevant for the node under study.

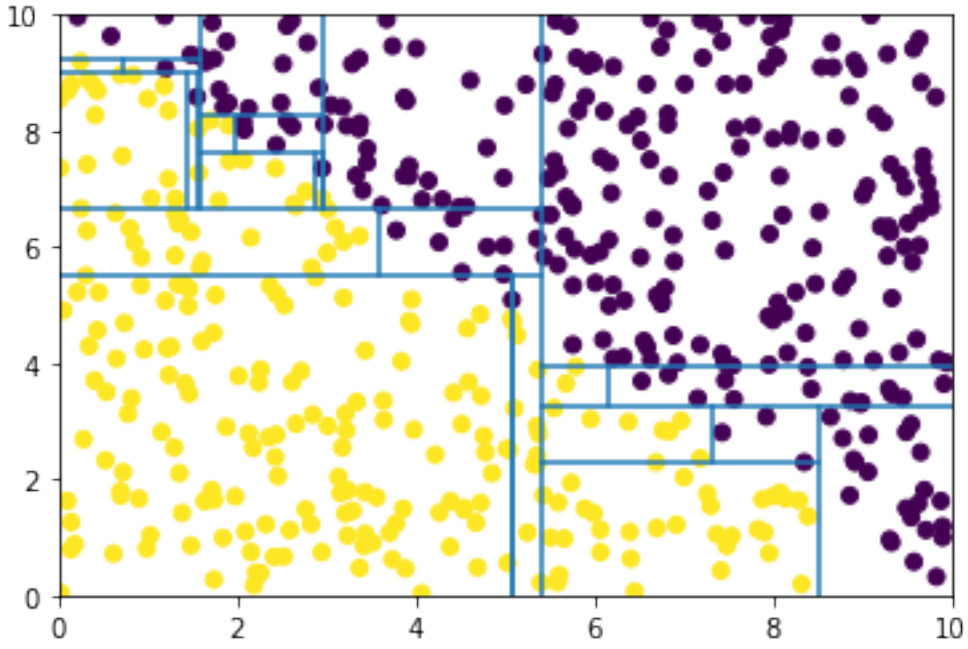
```
In [37]: def draw_split(features, thresholds, i, x0_min, x0_max, x1_min, x1_max):
             f, t = features[i], thresholds[i]
             if f == -2:
                 # leaf node, return offset to be added
                 # for the parent to visit the next child
                 return 1
             # draw the current split
             if f == 0: # vertical split (x0)
                 plt.axvline(t, x1_min*.1, x1_max*.1)
                 # prepare the new bounding boxes

                 # from x0_min to threshold
                 bb_left = (x0_min, t, x1_min, x1_max)
                 # from threshold to x0_max
                 bb_right = (t, x0_max, x1_min, x1_max)
             else: # horizontal split (x1)
                 plt.axhline(t, x0_min*.1, x0_max*.1)

                 bb_left = (x0_min, x0_max, x1_min, t)
```

```
            bb_right = (x0_min, x0_max, t, x1_max)
        off_left = draw_split(features, thresholds, i+1, *bb_left)
        off_right = draw_split(features, thresholds, i+off_left+1, *bb_right)
        return off_right + off_left + 1
    plt.scatter(X[:, 0], X[:, 1], c=y)
    plt.xlim(0,10)
    plt.ylim(0,10)
    draw_split(clf.tree_.feature, clf.tree_.threshold, 0, 0, 10, 0, 10)
```

Out[37]: 49



Here, we can see exactly what we were expecting: not being able to use an oblique line, the decision tree creates rectangular areas that are smaller and smaller as the tree grows. Note how each of the final rectangles represents a leaf in the decision tree. We are using a default configuration for the decision tree: this configuration grows until all leaves are pure: indeed, you will see that there are no rectangles containing nodes from both classes.

As a final consideration, notice how we limit the x and y axis to the 0-10 range. This is because, when plotting the lines, what we pass to the function are two numbers which, according to the documentation: > Should be between 0 and 1, 0 being the bottom of the plot, 1 the top of the plot

So, since the values of the dataset range between 0 and 10, we multiply the actual coordinates of the lines by 0.1. But, since matplotlib adds some "margin" by default on both axes, we need to be strict on the limits we use.

0.3 Exercise 3

Exercise 3.1 For this exercise, we need to load the MNIST dataset and split it into a training and a test set.

```
In [38]: from sklearn.datasets import fetch_openml

         dataset = fetch_openml("mnist_784")
         X = dataset["data"]
         y = dataset["target"]
         X_train, X_test, y_train, y_test = \
             train_test_split(X, y, test_size=1/7, shuffle=True, stratify=y)
```

Exercise 3.2 In this exercise, much like in previous exercises, we will train a default decision tree on our training data, and measure its accuracy on the test set.

```
In [39]: clf = DecisionTreeClassifier()
         clf.fit(X_train, y_train)
         accuracy_score(y_test, clf.predict(X_test))

Out[39]: 0.8741
```

Being this a balanced 10 classes problem (where random guess gets it right 10% of the time), we can say that our decision tree is definitely up to something. You may notice, if you run the code multiple times, that the accuracy is now much more stable than it was for Exercise 1. We now have a larger dataset: the training and test set sizes are now much larger, making our problem more robust to the selection on the test set.

Exercise 3.3 We will now implement our very own version of a random forest. We will base this solution on scikit-learn's decision trees.

For the random forest, we will instantiate a specific number of trees and train each one with a different subset of data. To select the subsets of data, we can use NumPy's `choice()` function. This function takes an iterable as first parameter, the number of elements to extract (`size`) and whether or not there should be replacement (`replace`). In our case, we want to extract N points (where N is the size of the dataset) **with** replacement.

For the predictions, we then get the predictions of all trees. Then, we use majority voting on the predictions to select the "final" label (for each point). Doing majority voting of a collection of "votes" (labels) means getting the mode (i.e. the most frequently occurring) of the list. In SciPy, the function `mode()` does just that.

```
In [40]: from scipy.stats import mode

         class MyRandomForestClassifier:
             def __init__(self, n_estimators=10, max_features='sqrt'):
                 self.trees = [ DecisionTreeClassifier(max_features=max_features)\
                                for _ in range(n_estimators) ]

             def fit(self, X, y):
                 for tree in self.trees:
                     subset = np.random.choice(range(X.shape[0]),\
                                               size=X.shape[0],\
                                               replace=True)
                     tree.fit(X[subset], y[subset])
```

```
            def predict(self, X):
                # get predictions of all trees for X
                predictions = [ tree.predict(X) for tree in self.trees ]
                return mode(predictions, axis=0)[0][0]
```

Notice that we are letting `DecisionTreeClassifier` handle the number of features used. This is because the decision tree will, when performing each split, select a random subset of features to use.

A common misconception about random forests is that each tree is trained on a specific subset of features (instead of having all features available and selecting a random subset at each split). Just to see how such a classifier would perform, we will also implement `PseudoRandomForestClassifier`, which selects a subset of features for each tree to train on.

```
In [41]: class PseudoRandomForestClassifier:
            def __init__(self, n_estimators, max_features):
                self.trees = [ DecisionTreeClassifier()\
                               for _ in range(n_estimators) ]
                self.max_features = max_features

            def fit(self, X, y):
                self.features = []
                for tree in self.trees:
                    f_sub = np.random.choice(range(X.shape[1]),\
                                             size=self.max_features,\
                                             replace=False)
                    subset = np.random.choice(range(X.shape[0]),\
                                              size=X.shape[0],\
                                              replace=True)
                    tree.fit(X[subset][:, f_sub], y[subset])
                    self.features.append(f_sub)

            def predict(self, X):
                predictions = [ tree.predict(X[:, f_sub])\
                                for tree, f_sub in zip(self.trees, self.features) ]
                return mode(predictions, axis=0)[0][0]
```

We will use this second random forest in the next exercise, to compare it with `MyRandomForestClassifier`.

Exercise 3.4 Now we can proceed with the fitting and testing of our decision tree, as follows:

```
In [42]: # using a subset that is sqrt(784)
         clf = MyRandomForestClassifier(10, 28)
         clf.fit(X_train, y_train)
         accuracy_score(y_test, clf.predict(X_test))

Out[42]: 0.9495
```

With 10 trees, we already get significantly better performance than a single decision tree. Let's see, instead, how `PseudoRandomForestClassifier` performs:

```
In [43]: # using a subset that is sqrt(784)
         clf = PseudoRandomForestClassifier(10, 28)
         clf.fit(X_train, y_train)
         accuracy_score(y_test, clf.predict(X_test))
```

```
Out[43]: 0.8417
```

This random forest not only performs worse than our "correct" one, but it also fares worse than a single decision tree! This is because we are setting a "tree-level" constraint on the features that can be used. This means that, if a subset of features selected for a tree does not contain any "good" features, the tree will necessarily perform poorly. On the other hand, if our constraint is "softer" (i.e. we select feature subsets at each split of each tree), a tree will have a better chance of receiving, during its creation, "good" features (here, with "good" features we are referring to features that have a strong predictive capability, i.e. a high feature importance).

Now, back to our good random forest. Let's see how it behaves with an increasing number of trees.

```
In [44]: for n_estimators in range(10, 101, 10): # 101 so that "100" is included
             clf = MyRandomForestClassifier(n_estimators, 28)
             clf.fit(X_train, y_train)
             print(n_estimators, accuracy_score(y_test, clf.predict(X_test)))
```

```
10 0.9476
20 0.9599
30 0.9655
40 0.9648
50 0.9678
60 0.9679
70 0.9681
80 0.9701
90 0.97
100 0.9688
```

While there is an improvement in performance, the random forest seems to hardly get past the 97% accuracy mark. This is approximately the kind of performance we can expect with this random forest on this dataset. This is approximately a 10% improvement in accuracy with regard to a decision tree. Considering that these results are obtained with a bunch of decision trees, you can see how powerful ensamble approaches are.

Exercise 3.5 For the next exercise, we will simply run the same code as before, but using scikit-learn's `RandomForestClassifier`.

```
In [45]: from sklearn.ensemble import RandomForestClassifier

         for n_estimators in range(10, 101, 10):
```

```
            clf = RandomForestClassifier(n_estimators, max_features='sqrt')
            clf.fit(X_train, y_train)
            print(n_estimators, accuracy_score(y_test, clf.predict(X_test)))
```

```
10 0.9477
20 0.962
30 0.962
40 0.9687
50 0.9662
60 0.9666
70 0.9688
80 0.9686
90 0.9672
100 0.9689
```

Thankfully, we are getting results similar to the ones we got with our very own version. Good job :)! You may be interested in testing the wrongly implemented version of the random forest, to see how it performs. Before doing that, how would you expect it to behave?

Finally, as an extra exercise, let's take a look at the training times of a single decision tree and a random forest (with 10 estimators).

```
In [46]: from timeit import timeit

         clf_tree = DecisionTreeClassifier()
         print("Decision tree", timeit(lambda: clf_tree.fit(X_train, y_train),\
                                        number=1))
         clf_rf = RandomForestClassifier(10)
         print("Random forest", timeit(lambda: clf_rf.fit(X_train, y_train),\
                                        number=1))
```

```
Decision tree 19.105313292999995
Random forest 3.7599132019998933
```

Interestingly, the random forest is faster than a single decision tree, even though it trains 10x the amount of trees! This is not black magic, but rather a result of the reduction in the number of features used at each split. Now, for each split, the tree has a much lower number of features to check, resulting in a significantly lower execution time.

Exercise 3.6 Much like for decision trees, we now need to compute the feature importance for the random forest. We have already implemented our own version of the feature importance of the decision tree and verified that it matches the one returned by scikit-learn. Therefore, we will now use the feature importance of the single decision trees to compute the feature importance of a random forest.

For practical reasons, instead of using our version of the random forest, we will instead be using scikit-learn's one. This version stores the list of trees in the `estimators_` attribute, instead of `trees`. Apart from this, you can compute the feature importance for your version of the random forest with the code below.

```
In [47]: def rf_feature_importance(rf):
             fi_trees = np.array([ tree.feature_importances_\
                                  for tree in rf.estimators_ ])
             return fi_trees.sum(axis=0) / fi_trees.sum()
```

What we do here is simply extracting the feature importance of each tree, sum it for each feature (one feature for each column) and normalize it so that is sums to 1. Scikit-learn's random forests also compute the feature importance this way, so we can use their version to make sure we are doing everything right.

```
In [48]: clf = RandomForestClassifier(10)
         clf.fit(X_train, y_train)
         np.allclose(rf_feature_importance(clf), clf.feature_importances_)

Out[48]: True
```

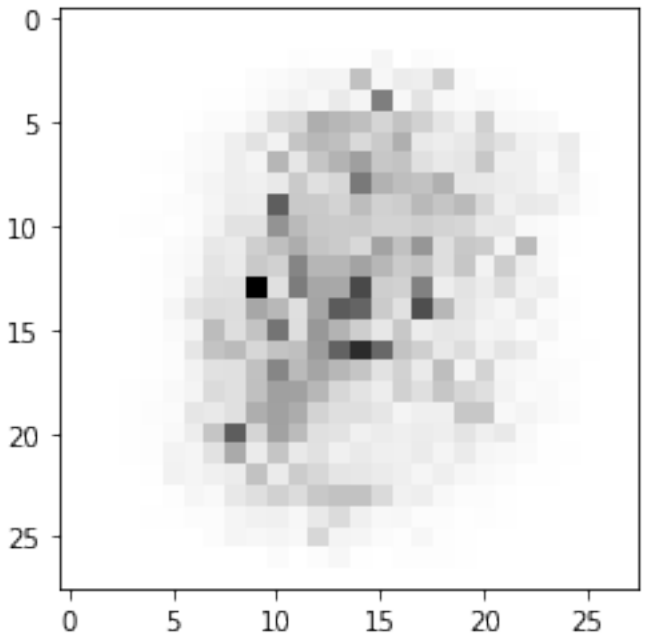
Once again, the results match to within a tolerance of 10^-5.

Exercise 3.7 We have obtained the importance of each feature, and we know that the 784 features represent a 28x28 image. So now we can plot the importances as a 28x28 grid of values, to understand which parts of the images the random forest is taking into account.

Since we have already imported matplotlib, let's go ahead and use it to visualize the image (or, you can use seaborn, as recommended by the exercise – the result will be similar, with the exception that seaborn's heatmap also adds a legend).

```
In [49]: plt.imshow(clf.feature_importances_.reshape(28,28), cmap='binary')

Out[49]: <matplotlib.image.AxesImage at 0x12bac9410>
```

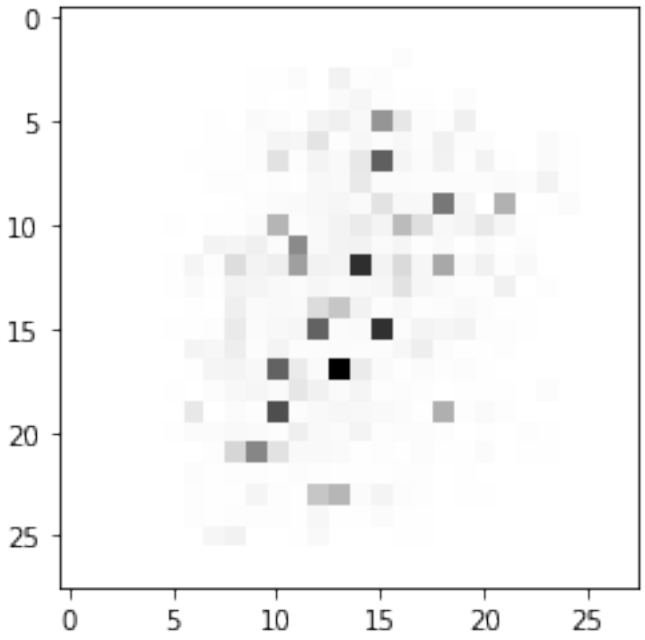


This makes sense. The classifier is checking the pixels in the central part of the image, since that is where there will typically be the main differences among figures. Pixels on the edge, on the other hand, will unlikely be considered, as they will typically be white.

Finally, let's have a look at what a single tree's feature importance map looks like.

```
In [50]: clf = DecisionTreeClassifier()
         clf.fit(X_train, y_train)
         plt.imshow(clf.feature_importances_.reshape(28,28), cmap='binary')

Out[50]: <matplotlib.image.AxesImage at 0x12baac0d0>
```



As you can see, the tree checks scattered portions of the image and assigns high importance to a few pixels. This means that the decision tree will take its decisions mostly based on just a few pixels of the image. This, as we have seen from the results on the test set, will be less likely to generalize well to new data.