Run all Kernel Compute not connected

Exercise: Random forests and model architecture

```
import pandas
!wget https://raw.githubusercontent.com/MicrosoftDocs/mslearn-introduction-to-machine-learning/
!wget https://raw.githubusercontent.com/MicrosoftDocs/mslearn-introduction-to-machine-learning/
import numpy as np
from sklearn.model_selection import train_test_split
import graphing # custom graphing code. See our GitHub repo for details

# Import the data from the .csv file
dataset = pandas.read_csv('san_fran_crime.csv', delimiter="\t")

# Remember to one-hot encode our crime and PdDistrict variables
categorical_features = ["Category", "PdDistrict"]
dataset = pandas.get_dummies(dataset, columns=categorical_features, drop_first=False)
```

```
# Split the dataset in an 90/10 train/test ratio.
# Recall that our dataset is very large so we can afford to do this
# with only 10% entering the test set
train, test = train_test_split(dataset, test_size=0.1, random_state=2, shuffle=True)

# Let's have a look at the data and the relationship we are going to model
print(dataset.head())
print("train shape:", train.shape)
print("test shape:", test.shape)
```

Hopefully this looks familiar to you! If not, jump back and go through the previous exercise on decision trees.

```
from sklearn.metrics import balanced_accuracy_score

# Make a utility method that we can re-use throughout this exercise
# To easily fit and test out model

features = [c for c in dataset.columns if c != "Resolution"]

def fit_and_test_model(model):
    '''
    Trains a model and tests it against both train and test sets
    '''
    global features

    # Train the model
    model.fit(train[features], train.Resolution)

    # Assess its performance
    # -- Train
```

```
    predictions = model.predict(train[features])
    train_accuracy = balanced_accuracy_score(train.Resolution, predictions)

    # -- Test
    predictions = model.predict(test[features])
    test_accuracy = balanced_accuracy_score(test.Resolution, predictions)

    return train_accuracy, test_accuracy


print("Ready to go!")
```

Decision tree

Let's quickly train a reasonably well-tuned decision tree to remind ourselves of its performance:

```
import sklearn.tree
# re-fit our last decision tree to print out its performance
model = sklearn.tree.DecisionTreeClassifier(random_state=1, max_depth=10)

dt_train_accuracy, dt_test_accuracy = fit_and_test_model(model)

print("Decision Tree Performance:")
print("Train accuracy", dt_train_accuracy)
print("Test accuracy", dt_test_accuracy)
```

Random Forest

A random forest is a collection of decision trees that work together to calculate the label for a sample.

Trees in a random forest are trained independently, on different partitions of data, and thus develop different biases; but when combined, they're less likely to overfit the data.

Let's build a very simple forest with two trees and the *default* parameters:

```
from sklearn.ensemble import RandomForestClassifier

# Create a random forest model with two trees
random_forest = RandomForestClassifier( n_estimators=2,
                                        random_state=2,
                                        verbose=False)

# Train and test the model
train_accuracy, test_accuracy = fit_and_test_model(random_forest)
print("Random Forest Performance:")
print("Train accuracy", train_accuracy)
print("Test accuracy", test_accuracy)
```

Our two-tree forest has done more poorly than the single tree on the test set, though it's done a better job on the train set.

To some extent, this should be expected. Random forests usually work with many more trees. Simply having two allowed it to overfit the training data much better than the original decision tree.

Altering the number of trees

Let's then build several forest models, each with a different number of trees, and see how they perform:

```
import graphing

# n_estimators states how many trees to put in the model
# We will make one model for every entry in this list
# and see how well each model performs
```

```
n_estimators = [2, 5, 10, 20, 50]

# Train our models and report their performance
train_accuracies = []
test_accuracies = []

for n_estimator in n_estimators:
    print("Preparing a model with", n_estimator, "trees...")

    # Prepare the model
    rf = RandomForestClassifier(n_estimators=n_estimator,
                                random_state=2,
                                verbose=False)

    # Train and test the result
    train_accuracy, test_accuracy = fit_and_test_model(rf)

    # Save the results
    test_accuracies.append(test_accuracy)
    train_accuracies.append(train_accuracy)
```

```
# Plot results
graphing.line_2D(dict(Train=train_accuracies, Test=test_accuracies),
                    n_estimators,
                    label_x="Numer of trees (n_estimators)",
                    label_y="Accuracy",
                    title="Performance X number of trees", show=True)
```

The metrics look great for the *training* set, but not so much for the *test* set. More trees tended to help both, but only up to a point.

We might have expected the number of trees to resolve our overfitting problem, but this was not the case! Chances are that the model is simply too complex relative to the data, allowing it to overfit the training set.

Altering the minimum number of samples for split parameter

parameter

Recall that decision trees have a *root node*, *internal nodes*, and *leaf nodes*, and that the first two can be split into newer nodes with subsets of data.

If we let our model split and create too many nodes, it can become increasingly complex and start to overfit.

One way to limit that complexity is to tell the model that each node needs to have **at least** a certain number of samples, otherwise it can't split into subnodes.

In other words, we can set the model's `min_samples_split` parameter to the least number of samples required so that a node can be split.

Our default value for `min_samples_split` is only `2`, so models will quickly become too complex if that parameter is left untouched.

We'll now use the best-performing model we generated, then try it with different `min_samples_split` values and compare the results:

```
# Shrink the training set temporarily to explore this
# setting with a more normal sample size
full_trainset = train
train = full_trainset[:1000] # limit to 1000 samples

min_samples_split = [2, 10, 20, 50, 100, 500]

# Train our models and report their performance
train_accuracies = []
test_accuracies = []

for min_samples in min_samples_split:
    print("Preparing a model with min_samples_split = ", min_samples)

    # Prepare the model
    rf = RandomForestClassifier(n_estimators=20,
                                min_samples_split=min_samples,
                                random_state=2,
                                verbose=False)
```

```
        # Train and test the result
        train_accuracy, test_accuracy = fit_and_test_model(rf)

        # Save the results
        test_accuracies.append(test_accuracy)
        train_accuracies.append(train_accuracy)


    # Plot results
    graphing.line_2D(dict(Train=train_accuracies, Test=test_accuracies),
                        min_samples_split,
                        label_x="Minimum samples split (min_samples_split)",
                        label_y="Accuracy",
                        title="Performance", show=True)

    # Rol back the trainset to the full set
    train = full_trainset
```

As you'll notice in the preceding data, small restrictions on the model's complexity by limiting its ability to split nodes reduce the gap between training and test performance. If this is subtle, it does so without damaging test performance at all.

By limiting the model complexity, we address `overfitting`, thus improving its ability to generalize and make accurate predictions on *unseen* data.

Notice that using `min_samples_split=20` gave us the best result for the *test* set, and that higher values worsened outcomes.

Altering the model depth

A related method to limit the trees is restricting `max_depth`. This is equivalent to `max_depth` we used for our decision tree, earlier. Its default value is `None`, which means nodes can be expanded until all leaves are *pure* (all samples in it have the same label) or have less samples than the value set for `min_samples_split`.

Whether `max_depth`, or `min_samples_split` is more appropriate depends on the nature of your dataset, including its size. Usually we need to experiment to find the best settings. Let's investigate `max_depth` as though we only had 500 crime samples available for our training set.

```
# Shrink the training set temporarily to explore this
# setting with a more normal sample size
full_trainset = train
train = full_trainset[:500] # limit to 500 samples

max_depths = [2, 4, 6, 8, 10, 15, 20, 50, 100]

# Train our models and report their performance
train_accuracies = []
test_accuracies = []

for max_depth in max_depths:
    print("Preparing a model with max_depth = ", max_depth)
    # Prepare the model
    rf = RandomForestClassifier(n_estimators=20,
                                max_depth=max_depth,
                                random_state=2,
                                verbose=False)

    # Train and test the result
    train_accuracy, test_accuracy = fit_and_test_model(rf)

    # Save the results
    test_accuracies.append(test_accuracy)
    train_accuracies.append(train_accuracy)


# Plot results
graphing.line_2D(dict(Train=train_accuracies, Test=test_accuracies),
                    max_depths,
                    label_x="Maximum depth (max_depths)",
                    label_y="Accuracy",
                    title="Performance", show=True)
```

```
# Rol back the trainset to the full set
train = full_trainset
```

The preceding plot tells us that our model actually **benefits** from a higher value for `max_depth` , up to the limit of `15` .

Increasing depth beyond this point begins to harm test performance, because it constrains the model too much for it to generalize.

As usual, it's important to evaluate different values when setting model parameters and defining its architecture.

```
# Prepare the model
rf = RandomForestClassifier(n_estimators=200,
                            max_depth=128,
                            max_features=25,
                            min_samples_split=2,
                            random_state=2,
                            verbose=False)

# Train and test the result
print("Training model. This may take 1 - 2 minutes")
train_accuracy, test_accuracy = fit_and_test_model(rf)

# Print out results, compared to the decision tree
data = {"Model": ["Decision tree","Final random forest"],
        "Train sensitivity": [dt_train_accuracy, train_accuracy],
        "Test sensitivity": [dt_test_accuracy, test_accuracy]
```

```
            "Test sensitivity": [dt_test_accuracy, test_accuracy]
        }

pandas.DataFrame(data, columns = ["Model", "Train sensitivity", "Test sensitivity"])
```

As you can see, fine-tuning the model's parameters resulted in a significant improvement in the *test* set results.

No compute Compute not connected Viewing Kernel not connected

Next unit: Hyperparameters in classification

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆