⌄ In Depth: Decision Trees and Random Forests

Previously we have looked in depth at a simple generative classifier (naive Bayes; see In Depth: Naive Bayes Classification) and a powerful discriminative classifier (support vector machines; see In-Depth: Support Vector Machines). Here we'll take a look at another powerful algorithm: a nonparametric algorithm called *random forests*. Random forests are an example of an *ensemble* method, meaning one that relies on aggregating the results of a set of simpler estimators. The somewhat surprising result with such ensemble methods is that the sum can be greater than the parts: that is, the predictive accuracy of a majority vote among a number of estimators can end up being better than that of any of the individual estimators doing the voting! We will see examples of this in the following sections.

We begin with the standard imports:

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
#plt.style.use('seaborn-whitegrid') # This part gave me an error
```

Changed.

```
# Checked available styles
print(plt.style.available)
```

⇥ ['Solarize_Light2', '_classic_test_patch', '_mpl-gallery', '_mpl-gallery-nogrid', 'bmh', 'classic', 'dark_background', 'fast', 'five

```
#Changed the style
plt.style.use('seaborn-v0_8-whitegrid')
```

```
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import RandomForestClassifier
from ipywidgets import interact
from sklearn.ensemble import RandomForestRegressor
from sklearn.datasets import load_digits, load_iris
from sklearn import metrics
import seaborn as sns
import pandas as pd
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
```

⌄ Motivating Random Forests: Decision Trees

Random forests are an example of an ensemble learner built on decision trees. For this reason, we'll start by discussing decision trees themselves.

Decision trees are extremely intuitive ways to classify or label objects: you simply ask a series of questions designed to zero in on the classification. For example, if you wanted to build a decision tree to classify animals you come across while on a hike, you might construct the one shown in the following figure.

The binary splitting makes this extremely efficient: in a well-constructed tree, each question will cut the number of options by approximately half, very quickly narrowing the options even among a large number of classes. The trick, of course, comes in deciding which questions to ask at each step. In machine learning implementations of decision trees, the questions generally take the form of axis-aligned splits in the data: that is, each node in the tree splits the data into two groups using a cutoff value within one of the features. Let's now look at an example of this.

⌄ Creating a Decision Tree

Consider the following two-dimensional data, which has one of four class labels (see the following figure):

```
#Generates and visualise 300 points belongs to 4 clusters
from sklearn.datasets import make_blobs

X, y = make_blobs(n_samples=300, centers=4,
                  random_state=0, cluster_std=1.0)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='rainbow');
```
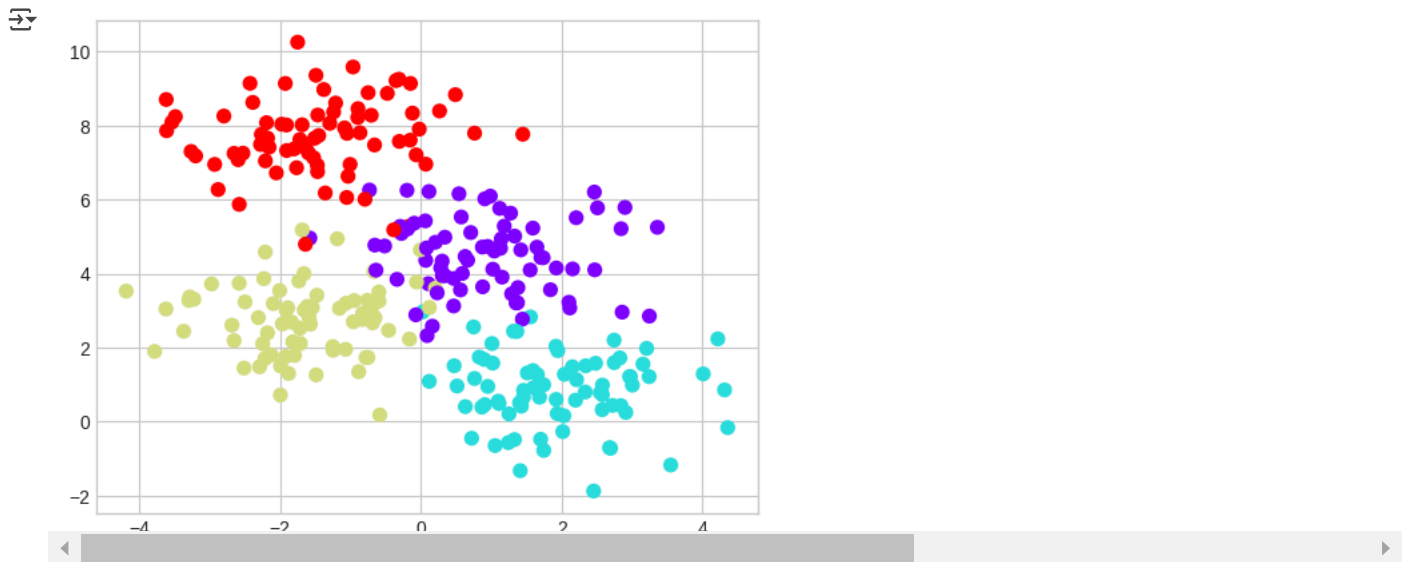
A simple decision tree built on this data will iteratively split the data along one or the other axis according to some quantitative criterion, and at each level assign the label of the new region according to a majority vote of points within it. The following figure presents a visualization of the first four levels of a decision tree classifier for this data.

Notice that after the first split, every point in the upper branch remains unchanged, so there is no need to further subdivide this branch. Except for nodes that contain all of one color, at each level *every* region is again split along one of the two features.

Changed.

```
# Splitting Data into 3 Parts

# from sklearn.model_selection import train_test_split

X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.4, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)

# Print the sizes
print(f"Training set: {len(X_train)} samples")
print(f"Validation set: {len(X_val)} samples")
print(f"Testing set: {len(X_test)} samples")
```

```
Training set: 180 samples
Validation set: 60 samples
Testing set: 60 samples
```

This process of fitting a decision tree to our data can be done in Scikit-Learn with the `DecisionTreeClassifier` estimator:

Changed.

```
# from sklearn.tree import DecisionTreeClassifier
#tree = DecisionTreeClassifier().fit(X, y)

# Training Decision Tree using training set
tree = DecisionTreeClassifier().fit(X_train, y_train)
```

Changed.

```
# Evaluate on validation set
val_score = tree.score(X_val, y_val)
print("Validation Accuracy:", val_score)
```

```
Validation Accuracy: 0.95
```

Changed.

```
# Evaluate on Test Set
test_score = tree.score(X_test, y_test)
print("Test Accuracy:", test_score)
```

```
Test Accuracy: 0.8833333333333333
```

Let's write a utility function to help us visualize the output of the classifier:

```python
def visualize_classifier(model, X, y, ax=None, cmap='rainbow'):
    ax = ax or plt.gca()

    # Plot the training points
    ax.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap=cmap,
               clim=(y.min(), y.max()), zorder=3)
    ax.axis('tight')
    ax.axis('off')
    xlim = ax.get_xlim()
    ylim = ax.get_ylim()

    # Remove model fitting from the function
    #model.fit(X, y)

    xx, yy = np.meshgrid(np.linspace(*xlim, num=200),
                         np.linspace(*ylim, num=200))
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

    # Create a color plot with the results
    n_classes = len(np.unique(y))
    contours = ax.contourf(xx, yy, Z, alpha=0.3,
                           levels=np.arange(n_classes + 1) - 0.5,
                           cmap=cmap, zorder=1)

    ax.set(xlim=xlim, ylim=ylim)
```
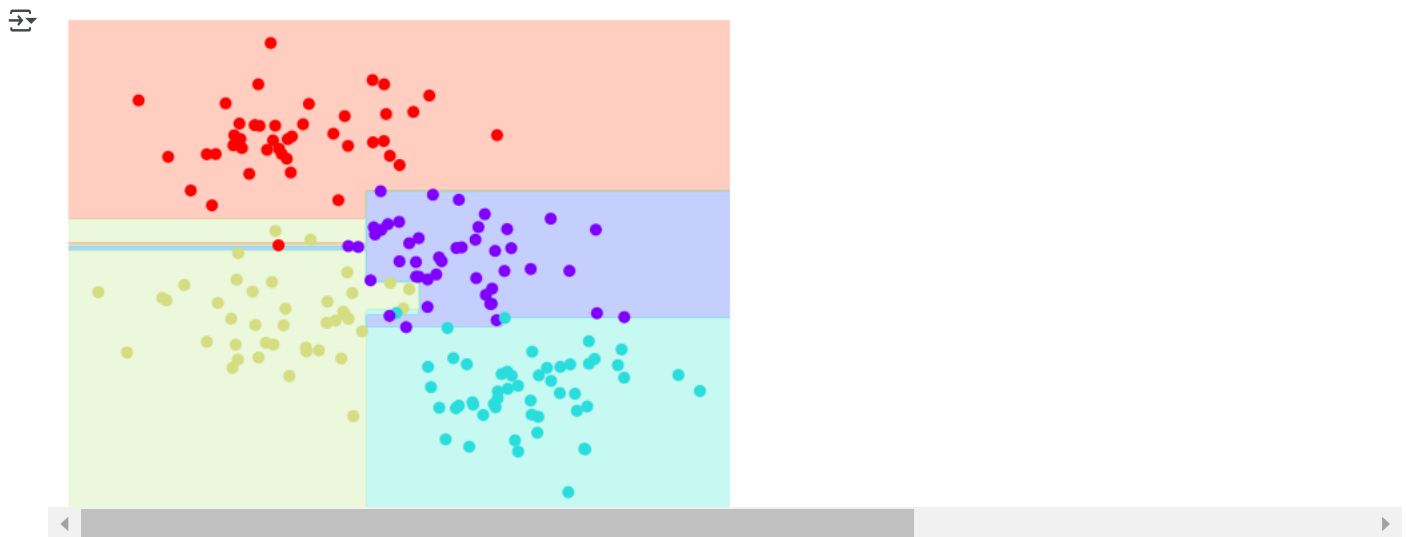
Now we can examine what the decision tree classification looks like (see the following figure):

Changed.

```python
# visualize_classifier only using training set

#visualize_classifier(DecisionTreeClassifier(), X, y)
visualize_classifier(tree, X_train, y_train)
```


If you're running this notebook live, you can use the helper script included in the online appendix to bring up an interactive visualization of the decision tree building process:

Changed.

```python
# helpers_05_08 is found in the online appendix
#import helpers_05_08
#helpers_05_08.plot_tree_interactive(X, y);
```

Notice that as the depth increases, we tend to get very strangely shaped classification regions; for example, at a depth of five, there is a tall and skinny purple region between the yellow and blue regions. It's clear that this is less a result of the true, intrinsic data distribution, and more a result of the particular sampling or noise properties of the data. That is, this decision tree, even at only five levels deep, is clearly overfitting our data.

Changed.

```python
# Instead of helpers_05_08

# from ipywidgets import interact

def visualize_tree(estimator, X, y, boundaries=True, xlim=None, ylim=None, ax=None):
    ax = ax or plt.gca()

    # Plot training points
    ax.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap='viridis',
               clim=(y.min(), y.max()), zorder=3)
    ax.axis('tight')
    ax.axis('off')
    if xlim is None:
        xlim = ax.get_xlim()
    if ylim is None:
        ylim = ax.get_ylim()

    # Create a grid for visualization
    xx, yy = np.meshgrid(np.linspace(*xlim, num=200),
                         np.linspace(*ylim, num=200))
    Z = estimator.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

    # Plot decision regions
    n_classes = len(np.unique(y))
    contours = ax.contourf(xx, yy, Z, alpha=0.3,
                           levels=np.arange(n_classes + 1) - 0.5,
                           cmap='viridis', zorder=1)

    ax.set(xlim=xlim, ylim=ylim)

    # Plot decision boundaries
    def plot_boundaries(i, xlim, ylim):
        if i >= 0:
            tree = estimator.tree_

            if tree.feature[i] == 0:
                ax.plot([tree.threshold[i], tree.threshold[i]], ylim, '-k', zorder=2)
                plot_boundaries(tree.children_left[i],
                                [xlim[0], tree.threshold[i]], ylim)
                plot_boundaries(tree.children_right[i],
                                [tree.threshold[i], xlim[1]], ylim)

            elif tree.feature[i] == 1:
                ax.plot(xlim, [tree.threshold[i], tree.threshold[i]], '-k', zorder=2)
                plot_boundaries(tree.children_left[i], xlim,
                                [ylim[0], tree.threshold[i]])
                plot_boundaries(tree.children_right[i], xlim,
                                [tree.threshold[i], ylim[1]])

    if boundaries:
        plot_boundaries(0, xlim, ylim)


# Interactive tree visualization (now only using training data)
def plot_tree_interactive(X, y):
    def interactive_tree(depth=5):
        clf = DecisionTreeClassifier(max_depth=depth, random_state=0)
        clf.fit(X_train, y_train)  # Train on training set only
        visualize_tree(clf, X_train, y_train)

    return interact(interactive_tree, depth=(1, 5))


def randomized_tree_interactive(X_train, y_train):
    N = int(0.75 * X_train.shape[0])  # Use 75% of training data

    xlim = (X_train[:, 0].min(), X_train[:, 0].max())
    ylim = (X_train[:, 1].min(), X_train[:, 1].max())

    def fit_randomized_tree(random_state=0):
        clf = DecisionTreeClassifier(max_depth=15)  # Initialize the tree
        i = np.arange(len(y_train))
        rng = np.random.RandomState(random_state)
        rng.shuffle(i)  # Shuffle dataset indices

        X_subset = X_train[i[:N]]  # Select random 75% subset
        y_subset = y_train[i[:N]]

        clf.fit(X_subset, y_subset)  # Fit the model before predicting

        visualize_tree(clf, X_subset, y_subset, boundaries=False,
```
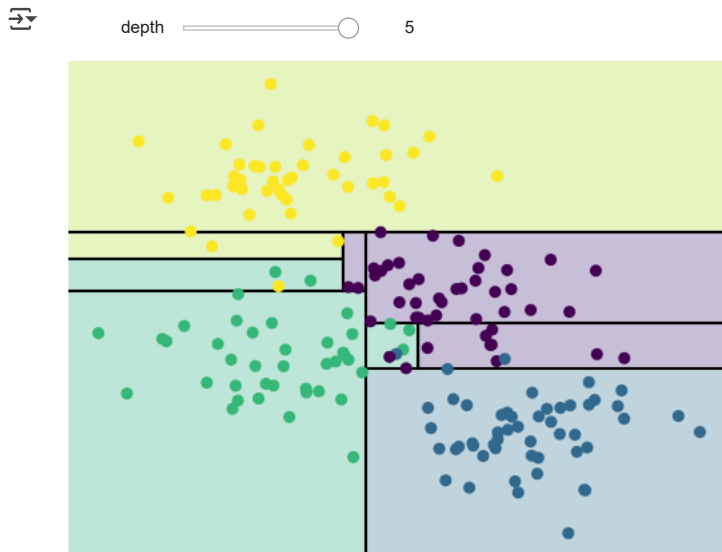
```
                    xlim=xlim, ylim=ylim)

    interact(fit_randomized_tree, random_state=(0, 100))
```

```
plot_tree_interactive(X_train, y_train)
```



```
plot_tree_interactive.<locals>.interactive_tree
def interactive_tree(depth=5)
```
<no docstring>

∨ Decision Trees and Overfitting

Such overfitting turns out to be a general property of decision trees: it is very easy to go too deep in the tree, and thus to fit details of the particular data rather than the overall properties of the distributions it is drawn from. Another way to see this overfitting is to look at models trained on different subsets of the data—for example, in this figure we train two different trees, each on half of the original data.

It is clear that in some places the two trees produce consistent results (e.g., in the four corners), while in other places the two trees give very different classifications (e.g., in the regions between any two clusters). The key observation is that the inconsistencies tend to happen where the classification is less certain, and thus by using information from *both* of these trees, we might come up with a better result!
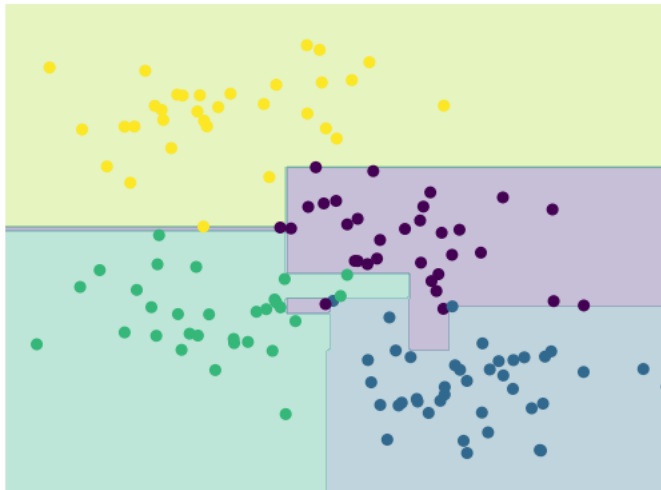
If you are running this notebook live, the following function will allow you to interactively display the fits of trees trained on a random subset of the data:

Changed.

```
# helpers_05_08 is found in the online appendix
#import helpers_05_08
#helpers_05_08.randomized_tree_interactive(X, y)
```

```
randomized_tree_interactive(X_train, y_train)
```

random_state ◯━━━━━━━━   0

Just as using information from two trees improves our results, we might expect that using information from many trees would improve our results even further.

Ensembles of Estimators: Random Forests

This notion—that multiple overfitting estimators can be combined to reduce the effect of this overfitting—is what underlies an ensemble method called *bagging*. Bagging makes use of an ensemble (a grab bag, perhaps) of parallel estimators, each of which overfits the data, and averages the results to find a better classification. An ensemble of randomized decision trees is known as a *random forest*.

This type of bagging classification can be done manually using Scikit-Learn's `BaggingClassifier` meta-estimator, as shown here (see the following figure):

```
# from sklearn.tree import DecisionTreeClassifier
# from sklearn.ensemble import BaggingClassifier

tree = DecisionTreeClassifier()
bag = BaggingClassifier(tree, n_estimators=100, max_samples=0.8,
                        random_state=1)

#bag.fit(X, y)
bag.fit(X_train, y_train)

#visualize_classifier(bag, X, y)
visualize_classifier(bag, X_train, y_train)
```
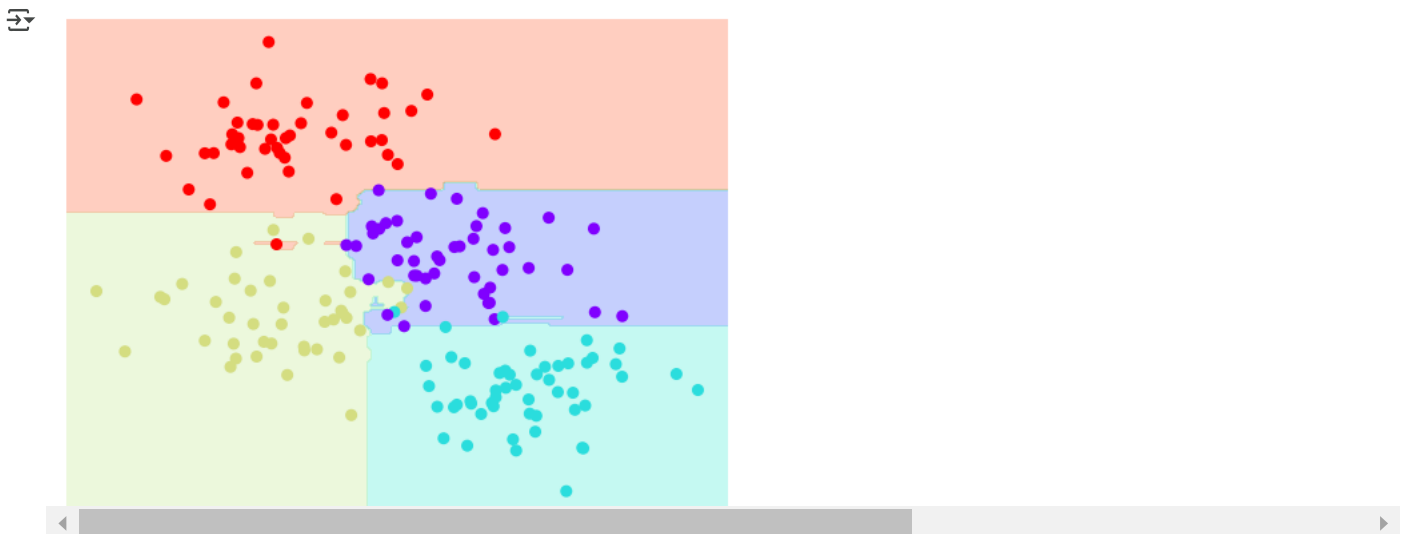

In this example, we have randomized the data by fitting each estimator with a random subset of 80% of the training points. In practice, decision trees are more effectively randomized by injecting some stochasticity in how the splits are chosen: this way all the data contributes to the fit each time, but the results of the fit still have the desired randomness. For example, when determining which feature to split on, the randomized tree might select from among the top several features. You can read more technical details about these randomization strategies in the Scikit-Learn documentation and references within.

In Scikit-Learn, such an optimized ensemble of randomized decision trees is implemented in the `RandomForestClassifier` estimator, which takes care of all the randomization automatically. All you need to do is select a number of estimators, and it will very quickly—in parallel, if desired—fit the ensemble of trees (see the following figure):

```python
# from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier(n_estimators=100, random_state=0)
model.fit(X_train, y_train)

visualize_classifier(model, X_train, y_train);
```
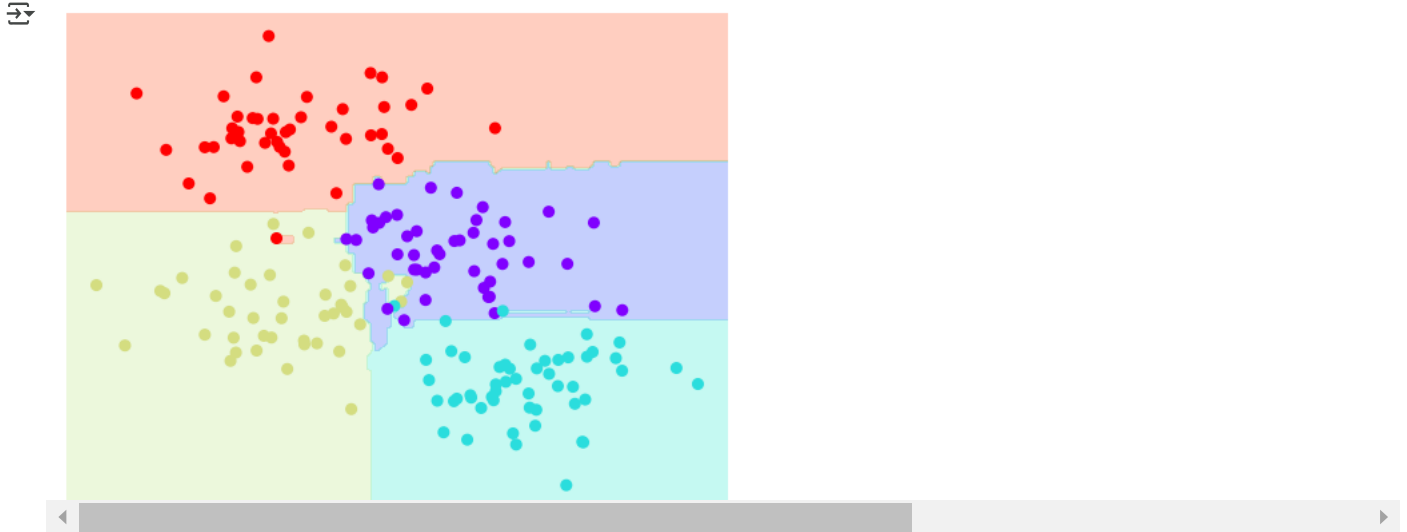

We see that by averaging over 100 randomly perturbed models, we end up with an overall model that is much closer to our intuition about how the parameter space should be split.

Random Forest Regression

In the previous section we considered random forests within the context of classification. Random forests can also be made to work in the case of regression (that is, with continuous rather than categorical variables). The estimator to use for this is the `RandomForestRegressor`, and the syntax is very similar to what we saw earlier.
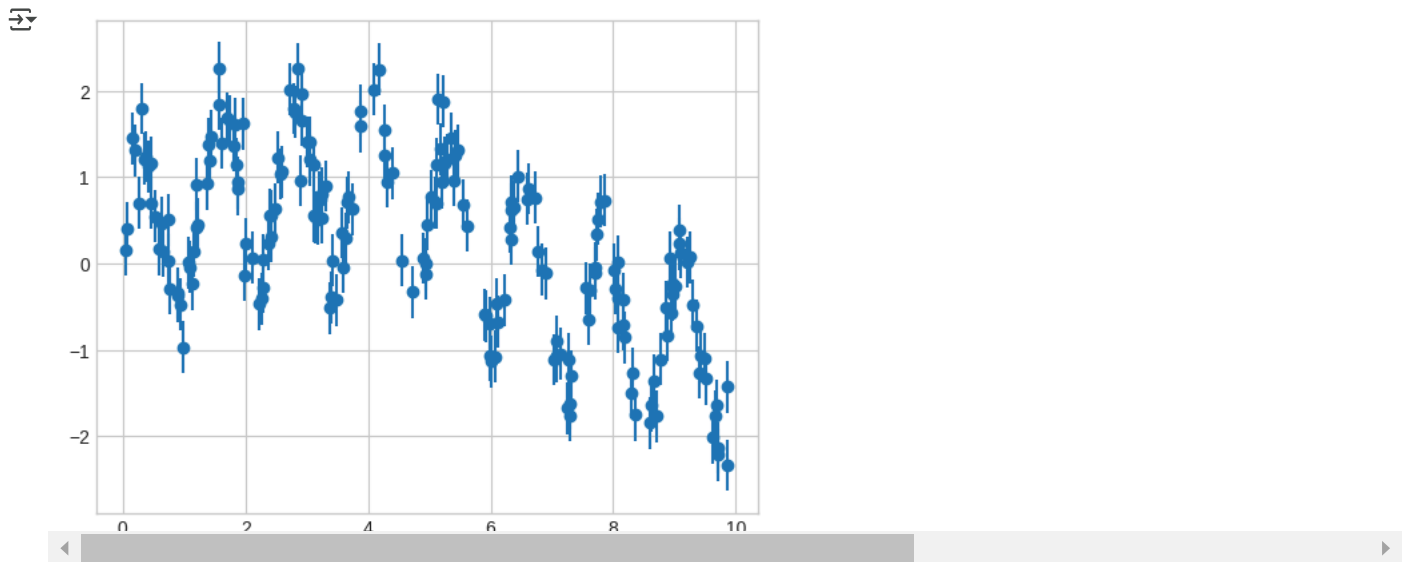
Consider the following data, drawn from the combination of a fast and slow oscillation (see the following figure):

```python
rng = np.random.RandomState(42)
x = 10 * rng.rand(200)

def model(x, sigma=0.3):
    fast_oscillation = np.sin(5 * x)
    slow_oscillation = np.sin(0.5 * x)
    noise = sigma * rng.randn(len(x))

    return slow_oscillation + fast_oscillation + noise

y = model(x)
plt.errorbar(x, y, 0.3, fmt='o');
```

Using the random forest regressor, we can find the best-fit curve as follows (see the following figure):

```python
# from sklearn.ensemble import RandomForestRegressor
forest = RandomForestRegressor(200)
forest.fit(x[:, None], y)

xfit = np.linspace(0, 10, 1000)
yfit = forest.predict(xfit[:, None])
ytrue = model(xfit, sigma=0)

plt.errorbar(x, y, 0.3, fmt='o', alpha=0.5)
plt.plot(xfit, yfit, '-r');
plt.plot(xfit, ytrue, '-k', alpha=0.5);
```
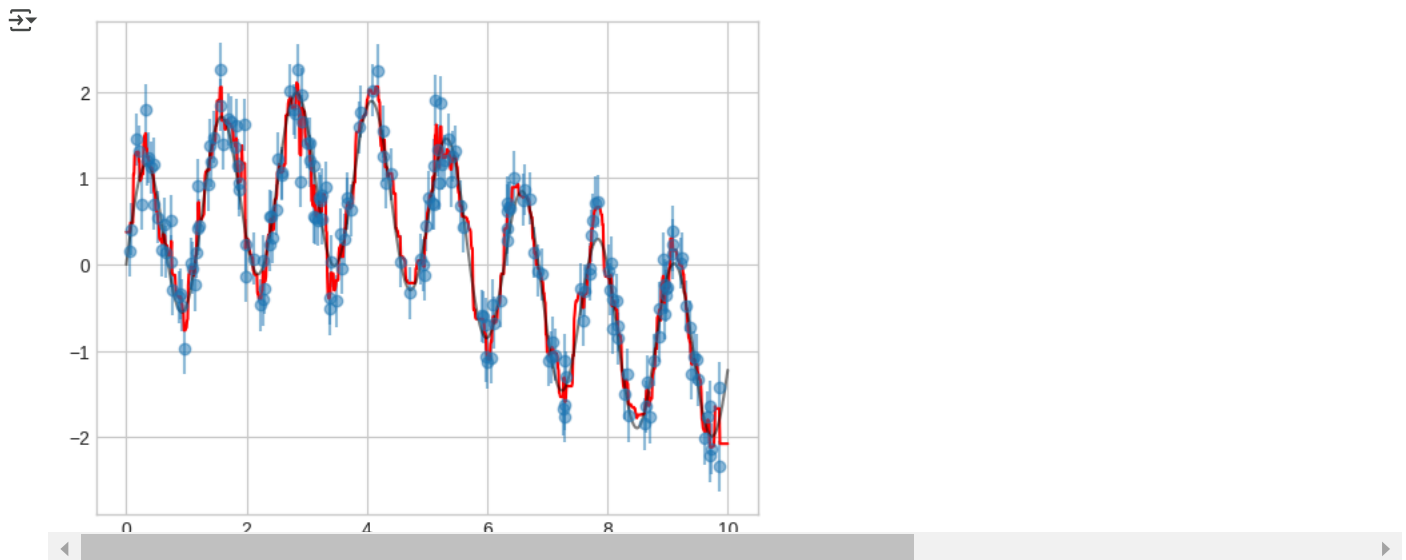

Here the true model is shown in the smooth gray curve, while the random forest model is shown by the jagged red curve. The nonparametric random forest model is flexible enough to fit the multiperiod data, without us needing to specifying a multi-period model!

⌄ Example: Random Forest for Classifying Digits

In Chapter 38 we worked through an example using the digits dataset included with Scikit-Learn. Let's use that again here to see how the random forest classifier can be applied in this context:

```python
# from sklearn.datasets import load_digits
digits = load_digits()
digits.keys()
```

```
dict_keys(['data', 'target', 'frame', 'feature_names', 'target_names', 'images', 'DESCR'])
```

To remind us what we're looking at, we'll visualize the first few data points (see the following figure):

```python
# set up the figure
fig = plt.figure(figsize=(6, 6))
fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.05, wspace=0.05)

# plot the digits
for i in range(64):
    ax = fig.add_subplot(8, 8, i + 1, xticks=[], yticks=[])
    ax.imshow(digits.images[i], cmap=plt.cm.binary, interpolation='nearest')

    # label the image with the target value
    ax.text(0, 7, str(digits.target[i]))
```
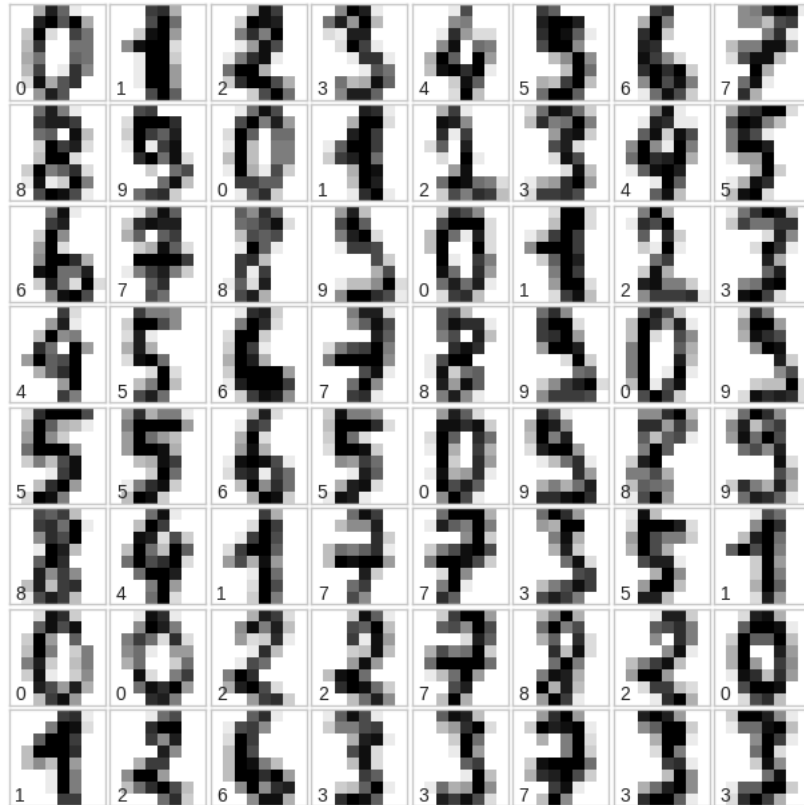

Changed.

```python
print(X.shape, y.shape)
```

(300, 2) (200,)

Changed.

```python
X, y = digits.data, digits.target
print(X.shape, y.shape)
```

(1797, 64) (1797,)

Changed.

We can classify the digits using a random forest as follows:

```python
#from sklearn.model_selection import train_test_split

#Xtrain, Xtest, ytrain, ytest = train_test_split(digits.data, digits.target, random_state=0)
#model = RandomForestClassifier(n_estimators=1000)
#model.fit(Xtrain, ytrain)
#ypred = model.predict(Xtest)

X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42, stratify=y)
X_valid, X_test, y_valid, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42, stratify=y_temp)


model = RandomForestClassifier(n_estimators=200, max_depth=10, random_state=42)
model.fit(X_train, y_train)
```

```
y_valid_pred = model.predict(X_valid)
```

Changed.

Let's look at the classification report for this classifier:

```
# from sklearn import metrics
#print(metrics.classification_report(ypred, ytest))
print(metrics.classification_report(y_valid, y_valid_pred))
```

	precision	recall	f1-score	support
0	1.00	0.96	0.98	27
1	0.93	0.96	0.95	27
2	1.00	1.00	1.00	27
3	0.96	0.96	0.96	27
4	0.96	0.96	0.96	27
5	0.96	0.96	0.96	28
6	1.00	1.00	1.00	27
7	0.93	1.00	0.96	27
8	0.92	0.88	0.90	26
9	0.96	0.93	0.94	27
accuracy			0.96	270
macro avg	0.96	0.96	0.96	270
weighted avg	0.96	0.96	0.96	270

Changed.

```
#Evaluate on Test Set
y_test_pred = model.predict(X_test)
print(metrics.classification_report(y_test, y_test_pred))
```

	precision	recall	f1-score	support
0	1.00	0.96	0.98	27
1	0.93	1.00	0.97	28
2	1.00	1.00	1.00	26
3	0.96	0.93	0.95	28
4	0.96	1.00	0.98	27
5	1.00	1.00	1.00	27
6	1.00	0.96	0.98	27
7	0.90	1.00	0.95	27
8	0.92	0.85	0.88	26
9	0.96	0.93	0.94	27
accuracy			0.96	270
macro avg	0.96	0.96	0.96	270
weighted avg	0.96	0.96	0.96	270

Changed.

And for good measure, plot the confusion matrix (see the following figure):

```
# from sklearn.metrics import confusion_matrix
# import seaborn as sns

#mat = confusion_matrix(ytest, ypred)
mat = confusion_matrix(y_test, y_test_pred)
sns.heatmap(mat.T, square=True, annot=True, fmt='d',
            cbar=False, cmap='Blues')
plt.xlabel('true label')
plt.ylabel('predicted label');
```
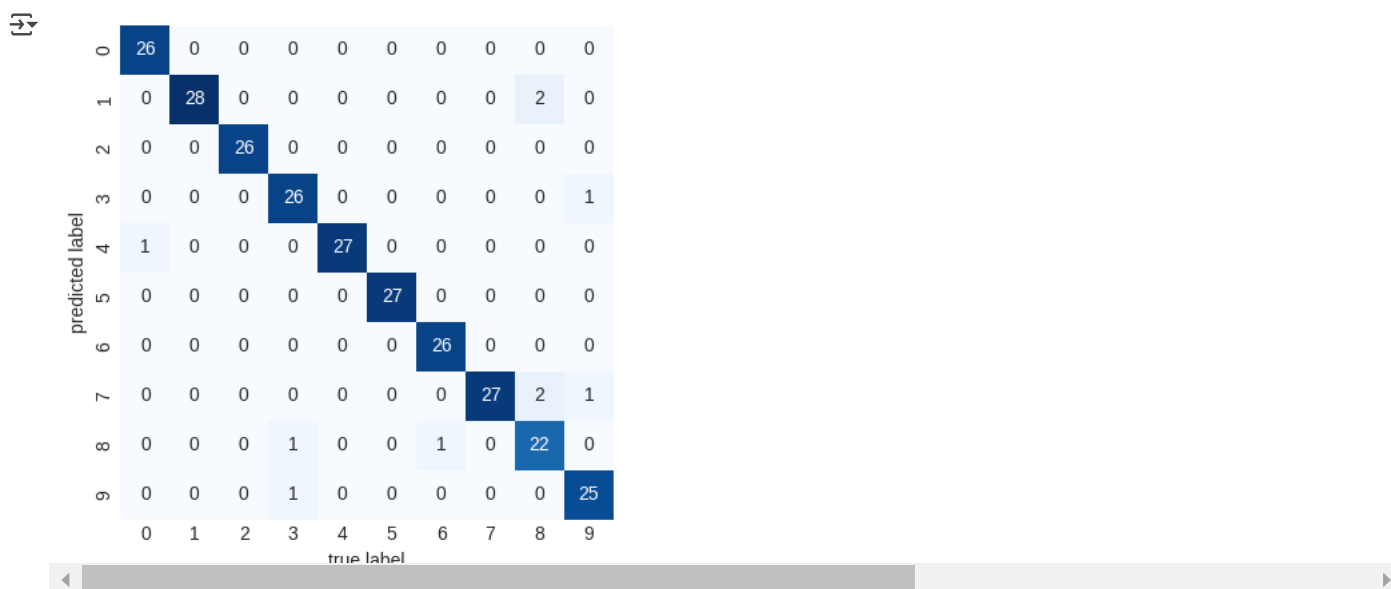
We find that a simple, untuned random forest results in a quite accurate classification of the digits data.

Summary

This chapter provided a brief introduction to the concept of ensemble estimators, and in particular the random forest, an ensemble of randomized decision trees. Random forests are a powerful method with several advantages:

- Both training and prediction are very fast, because of the simplicity of the underlying decision trees. In addition, both tasks can be straightforwardly parallelized, because the individual trees are entirely independent entities.
- The multiple trees allow for a probabilistic classification: a majority vote among estimators gives an estimate of the probability (accessed in Scikit-Learn with the `predict_proba` method).
- The nonparametric model is extremely flexible and can thus perform well on tasks that are underfit by other estimators.

A primary disadvantage of random forests is that the results are not easily interpretable: that is, if you would like to draw conclusions about the *meaning* of the classification model, random forests may not be the best choice.

Changed.

Application of Decision Tree and Random Forest - Iris dataset

Unsupported cell type. Double-click to inspect/edit the content.

Unsupported cell type. Double-click to inspect/edit the content.

```
# Load the Iris dataset
iris = load_iris()

df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
df['target'] = iris.target
df.head()
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

Next steps: Generate code with `df`   👁 View recommended plots   New interactive sheet

```
# Split into features (X) and target (y)
X = df.drop(columns=['target'])
y = df['target']
```

```
# Split dataset
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42, stratify=y)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42, stratify=y_temp)
```

⌄ Comparrison between Decision Tree and Random Forest

```
# Train and predict the Decision Tree Classifier
dt_model = DecisionTreeClassifier(random_state=42)
dt_model.fit(X_train, y_train)

y_pred_dt = dt_model.predict(X_val)


# Train and predict the Random Forest Classifier
rf_model = RandomForestClassifier(n_estimators=500, random_state=42)
rf_model.fit(X_train, y_train)

y_pred_rf = rf_model.predict(X_val)


# Compare accuracy
dt_accuracy = accuracy_score(y_val, y_pred_dt)
rf_accuracy = accuracy_score(y_val, y_pred_rf)

print(f"Decision Tree Accuracy (Validation Set): {dt_accuracy:.4f}")
print(f"Random Forest Accuracy (Validation Set): {rf_accuracy:.4f}")
```

```
→  Decision Tree Accuracy (Validation Set): 0.9545
   Random Forest Accuracy (Validation Set): 0.8636
```
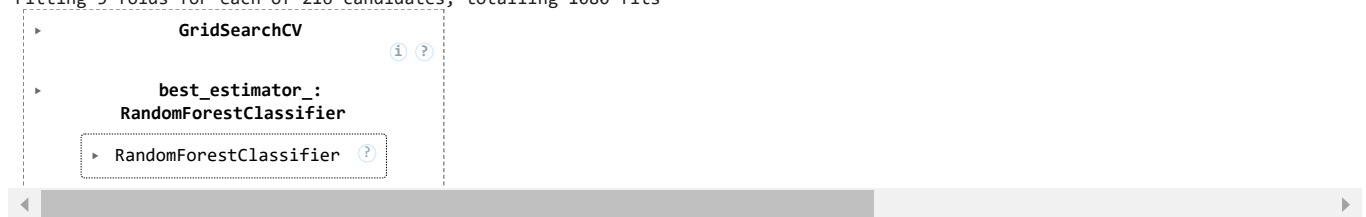
⌄ Hyperparameter Tuning

```
# Define the parameter grid
param_grid = {
    'n_estimators': [100, 300, 500],  # Number of trees
    'max_depth': [None, 10, 20, 30],  # Depth of each tree
    'min_samples_split': [2, 5, 10],  # Minimum samples to split a node
    'min_samples_leaf': [1, 2, 4],  # Minimum samples per leaf
    'max_features': ['sqrt', 'log2']  # Number of features per split
}


# Initialize Random Forest model
rf_model = RandomForestClassifier(random_state=42)

# Perform Grid Search with 5-fold cross-validation
grid_search = GridSearchCV(estimator=rf_model, param_grid=param_grid,
                           scoring='accuracy', cv=5, n_jobs=-1, verbose=2)

# Fit GridSearchCV to the training data
grid_search.fit(X_train, y_train)
```

```
→  Fitting 5 folds for each of 216 candidates, totalling 1080 fits
```

```
          ▸          GridSearchCV
                                         ⓘ ?

          ▸      best_estimator_:
               RandomForestClassifier

             ▸ RandomForestClassifier  ?
```

```
best_params = grid_search.best_params_
print("Best Hyperparameters:", best_params)
```

```
→  Best Hyperparameters: {'max_depth': None, 'max_features': 'sqrt', 'min_samples_leaf': 4, 'min_samples_split': 2, 'n_estimators': 10
```

```
# Train the best Random Forest model with optimized parameters
best_rf = RandomForestClassifier(
    n_estimators=100,        # Best number of trees
    max_depth=None,          # No depth limit (trees grow fully
    max_features='sqrt',     # Best number of features per spli
    min_samples_split=2,     # Minimum samples to split a node
    min_samples_leaf=4,      # Minimum samples per leaf
    random_state=42
)
```

```
# Fit the model on training data
```

```
                    RandomForestClassifier          ⓘ ?
RandomForestClassifier(min samples leaf=4. random state=42)
```

```
# Predict on validation set
y_pred_best_rf = best_rf.predict(X_val)
```

```
best_rf_accuracy = accuracy_score(y_val, y_pred_best_rf)
print(f"Optimized Random Forest Accuracy (Validation Set): {best_rf_accuracy:.4f}")
```

```
Optimized Random Forest Accuracy (Validation Set): 0.8636
```

```
# Compute accuracy
# Evaluate Random Forest on Test Set
y_test_best_rf = best_rf.predict(X_test)
best_rf_accuracy = accuracy_score(y_test, y_test_best_rf)
print(f"Optimized Random Forest Accuracy (Validation Set): {best_rf_accuracy:.4f}")
```

```
# Evaluate Decision Tree on Test Set
dt_test_acc = accuracy_score(y_test, dt_model.predict(X_test))
print(f"Decision Tree Accuracy (Test Set): {dt_test_acc:.4f}")
```

```
Optimized Random Forest Accuracy (Validation Set): 0.9565
Decision Tree Accuracy (Test Set): 0.9130
```

∨ Train Random Forest on the Full Dataset (Train + Validation)

```
# Merge Train & Validation sets for final model training
X_final_train = pd.concat([X_train, X_val])
y_final_train = pd.concat([y_train, y_val])
```

```
# Train the final Random Forest model on the full training set
final_rf = RandomForestClassifier(
    n_estimators=100,
    max_depth=None,
    max_features='sqrt',
    min_samples_split=2,
    min_samples_leaf=4,
    random_state=42
)
final_rf.fit(X_final_train, y_final_train)
```

```
                    RandomForestClassifier          ⓘ ?
RandomForestClassifier(min samples leaf=4. random state=42)
```

```
# Predict on Test Set
y test pred = final rf predict(X test)
```