# Chapter 7. Ensemble Learning and Random Forests

Suppose you pose a complex question to thousands of random people, then aggregate their answers. In many cases you will find that this aggregated answer is better than an expert's answer. This is called the *wisdom of the crowd*. Similarly, if you aggregate the predictions of a group of predictors (such as classifiers or regressors), you will often get better predictions than with the best individual predictor. A group of predictors is called an *ensemble*; thus, this technique is called *Ensemble Learning*, and an Ensemble Learning algorithm is called an *Ensemble method*.

As an example of an Ensemble method, you can train a group of Decision Tree classifiers, each on a different random subset of the training set. To make predictions, you obtain the predictions of all the individual trees, then predict the class that gets the most votes (see the last exercise in Chapter 6). Such an ensemble of Decision Trees is called a *Random Forest*, and despite its simplicity, this is one of the most powerful Machine Learning algorithms available today.

As discussed in Chapter 2, you will often use Ensemble methods near the end of a project, once you have already built a few good predictors, to combine them into an even better predictor. In fact, the winning solutions in Machine Learning competitions often involve several Ensemble methods (most famously in the Netflix Prize competition).

In this chapter we will discuss the most popular Ensemble methods, including *bagging*, *boosting*, and *stacking*. We will also explore Random Forests.

## Voting Classifiers

Suppose you have trained a few classifiers, each one achieving about 80% accuracy. You may have a Logistic Regression classifier, an SVM classifier, a Random Forest classifier, a K-Nearest Neighbors classifier, and perhaps a few more (see Figure 7-1).

*Figure 7-1. Training diverse classifiers*

A very simple way to create an even better classifier is to aggregate the predictions of each classifier and predict the class that gets the most votes. This majority-vote classifier is called a *hard voting* classifier (see Figure 7-2).

*Figure 7-2. Hard voting classifier predictions*

Somewhat surprisingly, this voting classifier often achieves a higher accuracy than the best classifier in the ensemble. In fact, even if each classifier is a *weak learner* (meaning it does only slightly better than random guessing), the ensemble can still be a *strong learner* (achieving high accuracy), provided there are a sufficient number of weak learners and they are sufficiently diverse.

How is this possible? The following analogy can help shed some light on this mystery. Suppose you have a slightly biased coin that has a 51% chance of coming up heads and 49% chance of coming up tails. If you toss it 1,000 times, you will generally get more or less 510 heads and 490 tails, and hence a majority of heads. If you do the math, you will find that the probability of obtaining a majority of heads after 1,000 tosses is close to 75%. The more you toss the coin, the higher the probability (e.g., with 10,000 tosses, the probability climbs over 97%). This is due to the *law of large numbers*: as you keep tossing the coin, the ratio of heads gets closer and closer to the probability of heads (51%). Figure 7-3 shows 10 series of biased coin tosses. You can see that as the number of tosses increases, the ratio of heads approaches 51%. Eventually all 10 series end up so close to 51% that they are consistently above 50%.

*Figure 7-3. The law of large numbers*

Similarly, suppose you build an ensemble containing 1,000 classifiers that are individually correct only 51% of the time (barely better than random guessing). If you predict the majority voted class, you can hope for up to 75% accuracy! However, this is only true if all classifiers are perfectly independent, making uncorrelated errors, which is clearly not the case because they are trained on the same data. They are likely to make the same types of errors, so there will be many majority votes for the wrong class, reducing the ensemble's accuracy.

**TIP**

Ensemble methods work best when the predictors are as independent from one another as possible. One way to get diverse classifiers is to train them using very different algorithms. This increases the chance that they will make very different types of errors, improving the ensemble's accuracy.

The following code creates and trains a voting classifier in Scikit-Learn, composed of three diverse classifiers (the training set is the moons dataset, introduced in Chapter 5):

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

log_clf = LogisticRegression()
rnd_clf = RandomForestClassifier()
svm_clf = SVC()

voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='hard')
voting_clf.fit(X_train, y_train)
```

Let's look at each classifier's accuracy on the test set:

```
>>> from sklearn.metrics import accuracy_score
>>> for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
...     clf.fit(X_train, y_train)
...     y_pred = clf.predict(X_test)
...     print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
...
LogisticRegression 0.864
RandomForestClassifier 0.896
SVC 0.888
VotingClassifier 0.904
```

There you have it! The voting classifier slightly outperforms all the individual classifiers.

If all classifiers are able to estimate class probabilities (i.e., they all have a `predict_proba()` method), then you can tell Scikit-Learn to predict the class with the highest class probability, averaged over all the individual classifiers. This is called *soft voting*. It often achieves higher performance than hard voting because it gives more weight to highly confident votes. All you need to do is replace `voting="hard"` with `voting="soft"` and ensure that all classifiers can estimate class probabilities. This is not the case for the `SVC` class by default, so you need to set its `probability` hyperparameter to `True` (this will make the `SVC` class use cross-validation to estimate class probabilities, slowing down training, and it will add a `predict_proba()` method). If you modify the preceding code to use soft voting, you will find that the voting classifier achieves over 91.2% accuracy!

## Bagging and Pasting

One way to get a diverse set of classifiers is to use very different training algorithms, as just discussed. Another approach is to use the same training algorithm for every predictor and train them on different random subsets of the training set. When sampling is performed *with* replacement, this method is called *bagging*[1] (short for *bootstrap aggregating*[2]). When sampling is performed *without* replacement, it is called *pasting*.[3]

In other words, both bagging and pasting allow training instances to be sampled several times across multiple predictors, but only bagging allows training instances to be sampled several times for the same predictor. This sampling and training process is represented in Figure 7-4.

*Figure 7-4. Bagging and pasting involves training several predictors on different random samples of the training set*

Once all predictors are trained, the ensemble can make a prediction for a new instance by simply aggregating the predictions of all predictors. The aggregation function is typically the *statistical mode* (i.e., the most frequent prediction, just like a hard voting classifier) for classification, or the average for regression. Each individual predictor has a higher bias than if it were trained on the original training set, but aggregation reduces both bias and variance.[4] Generally, the net result is that the ensemble has a similar bias but a lower variance than a single predictor trained on the original training set.

As you can see in Figure 7-4, predictors can all be trained in parallel, via different CPU cores or even different servers. Similarly, predictions can be made in parallel. This is one of the reasons bagging and pasting are such popular methods: they scale very well.

## Bagging and Pasting in Scikit-Learn

Scikit-Learn offers a simple API for both bagging and pasting with the `BaggingClassifier` class (or `BaggingRegressor` for regression). The following code trains an ensemble of 500 Decision Tree classifiers:[5] each is trained on 100 training instances randomly sampled from the training set with replacement (this is an example of bagging, but if you want to use pasting instead, just set `bootstrap=False`). The `n_jobs` parameter tells Scikit-Learn the number of CPU cores to use for training and predictions (`-1` tells Scikit-Learn to use all available cores):

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

bag_clf = BaggingClassifier(
    DecisionTreeClassifier(), n_estimators=500,
    max_samples=100, bootstrap=True, n_jobs=-1)
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
```

> **NOTE**
>
> The `BaggingClassifier` automatically performs soft voting instead of hard voting if the base classifier can estimate class probabilities (i.e., if it has a `predict_proba()` method), which is the case with Decision Tree classifiers.

Figure 7-5 compares the decision boundary of a single Decision Tree with the decision boundary of a bagging ensemble of 500 trees (from the preceding code), both trained on the moons dataset. As you can see, the ensemble's predictions will likely generalize much better than the single Decision Tree's predictions: the ensemble has a comparable bias but a smaller variance (it makes roughly the same number of errors on the training set, but the decision boundary is less irregular).

*Figure 7-5. A single Decision Tree (left) versus a bagging ensemble of 500 trees (right)*

Bootstrapping introduces a bit more diversity in the subsets that each predictor is trained on, so bagging ends up with a slightly higher bias than pasting; but the extra diversity also means that the predictors end up being less correlated, so the ensemble's variance is reduced. Overall, bagging often results in better models, which explains why it is generally preferred. However, if you have spare time and CPU power, you can use cross-validation to evaluate both bagging and pasting and select the one that works best.

## Out-of-Bag Evaluation

With bagging, some instances may be sampled several times for any given predictor, while others may not be sampled at all. By default a `BaggingClassifier` samples *m* training instances with replacement (`bootstrap=True`), where *m* is the size of the training set. This means that only about 63% of the training instances are sampled on average for each predictor.[6] The remaining 37% of the training instances that are not sampled are called *out-of-bag* (oob) instances. Note that they are not the same 37% for all predictors.

Since a predictor never sees the oob instances during training, it can be evaluated on these instances, without the need for a separate validation set. You can evaluate the ensemble itself by averaging out the oob evaluations of each predictor.

In Scikit-Learn, you can set `oob_score=True` when creating a `BaggingClassifier` to request an automatic oob evaluation after training. The following code demonstrates this. The resulting evaluation score is available through the `oob_score_` variable:

```
>>> bag_clf = BaggingClassifier(
...     DecisionTreeClassifier(), n_estimators=500,
...     bootstrap=True, n_jobs=-1, oob_score=True)
...
>>> bag_clf.fit(X_train, y_train)
```

```
>>> bag_clf.oob_score_
0.90133333333333332
```

According to this oob evaluation, this `BaggingClassifier` is likely to achieve about 90.1% accuracy on the test set. Let's verify this:

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = bag_clf.predict(X_test)
>>> accuracy_score(y_test, y_pred)
0.91200000000000003
```

We get 91.2% accuracy on the test set—close enough!

The oob decision function for each training instance is also available through the `oob_decision_function_` variable. In this case (since the base estimator has a `predict_proba()` method), the decision function returns the class probabilities for each training instance. For example, the oob evaluation estimates that the first training instance has a 68.25% probability of belonging to the positive class (and 31.75% of belonging to the negative class):

```
>>> bag_clf.oob_decision_function_
array([[0.31746032, 0.68253968],
       [0.34117647, 0.65882353],
       [1.        , 0.        ],
       ...
       [1.        , 0.        ],
       [0.03108808, 0.96891192],
       [0.57291667, 0.42708333]])
```

## Random Patches and Random Subspaces

The `BaggingClassifier` class supports sampling the features as well. Sampling is controlled by two hyperparameters: `max_features` and `bootstrap_features`. They work the same way as `max_samples` and `bootstrap`, but for feature sampling instead of instance sampling. Thus, each predictor will be trained on a random subset of the input features.

This technique is particularly useful when you are dealing with high-dimensional inputs (such as images). Sampling both training instances and features is called the *Random Patches* method.[7] Keeping all training instances (by setting `bootstrap=False` and `max_samples=1.0`) but sampling features (by setting `bootstrap_features` to `True` and/or `max_features` to a value smaller than `1.0`) is called the *Random Subspaces* method.[8]

in other words, a tree composed of a single decision node plus two leaf nodes. This is the default base estimator for the `AdaBoostClassifier` class:

```
from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=200,
    algorithm="SAMME.R", learning_rate=0.5)
ada_clf.fit(X_train, y_train)
```

> **TIP**
>
> If your AdaBoost ensemble is overfitting the training set, you can try reducing the number of estimators or more strongly regularizing the base estimator.

## Gradient Boosting

Another very popular boosting algorithm is *Gradient Boosting*.[17] Just like AdaBoost, Gradient Boosting works by sequentially adding predictors to an ensemble, each one correcting its predecessor. However, instead of tweaking the instance weights at every iteration like AdaBoost does, this method tries to fit the new predictor to the *residual errors* made by the previous predictor.

Let's go through a simple regression example, using Decision Trees as the base predictors (of course, Gradient Boosting also works great with regression tasks). This is called *Gradient Tree Boosting*, or *Gradient Boosted Regression Trees* (GBRT). First, let's fit a `DecisionTreeRegressor` to the training set (for example, a noisy quadratic training set):

```
from sklearn.tree import DecisionTreeRegressor

tree_reg1 = DecisionTreeRegressor(max_depth=2)
tree_reg1.fit(X, y)
```

Next, we'll train a second `DecisionTreeRegressor` on the residual errors made by the first predictor:

```
y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2)
tree_reg2.fit(X, y2)
```

Then we train a third regressor on the residual errors made by the second predictor:

```
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2)
tree_reg3.fit(X, y3)
```

Now we have an ensemble containing three trees. It can make predictions on a new instance simply by adding up the predictions of all the trees:

```
y_pred = sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3))
```

Figure 7-9 represents the predictions of these three trees in the left column, and the ensemble's predictions in the right column. In the first row, the ensemble has just one tree, so its predictions are exactly the same as the first tree's predictions. In the second row, a new tree is trained on the residual errors of the first tree. On the right you can see that the ensemble's predictions are equal to the sum of the predictions of the first two trees. Similarly, in the third row another tree is trained on the residual errors of the second tree. You can see that the ensemble's predictions gradually get better as trees are added to the ensemble.

A simpler way to train GBRT ensembles is to use Scikit-Learn's `GradientBoostingRegressor` class. Much like the `RandomForestRegressor` class, it has hyperparameters to control the growth of Decision Trees (e.g., `max_depth`, `min_samples_leaf`), as well as hyperparameters to control the ensemble training, such as the number of trees (`n_estimators`). The following code creates the same ensemble as the previous one:

```
from sklearn.ensemble import GradientBoostingRegressor

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=3, learning_rate=1.0)
gbrt.fit(X, y)
```

*Figure 7-9. In this depiction of Gradient Boosting, the first predictor (top left) is trained normally, then each consecutive predictor (middle left and lower left) is trained on the previous predictor's residuals; the right column shows the resulting ensemble's predictions*

The `learning_rate` hyperparameter scales the contribution of each tree. If you set it to a low value, such as `0.1`, you will need more trees in the ensemble to fit the training set, but the predictions will usually generalize better. This is a regularization technique called *shrinkage*. Figure 7-10 shows two GBRT ensembles trained with a low learning rate: the one on the left does not have enough trees to fit the training set, while the one on the right has too many trees and overfits the training set.

*Figure 7-10. GBRT ensembles with not enough predictors (left) and too many (right)*

In order to find the optimal number of trees, you can use early stopping (see Chapter 4). A simple way to implement this is to use the `staged_predict()` method: it returns an iterator over the predictions made by the ensemble at each stage of training (with one tree, two trees, etc.). The following code trains a GBRT ensemble with 120 trees, then measures the validation error at each stage of training to find the optimal number of trees, and finally trains another GBRT ensemble using the optimal number of trees:

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

X_train, X_val, y_train, y_val = train_test_split(X, y)

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=120)
gbrt.fit(X_train, y_train)

errors = [mean_squared_error(y_val, y_pred)
          for y_pred in gbrt.staged_predict(X_val)]
bst_n_estimators = np.argmin(errors) + 1

gbrt_best = GradientBoostingRegressor(max_depth=2,n_estimators=bst_n_estimators)
gbrt_best.fit(X_train, y_train)
```

The validation errors are represented on the left of Figure 7-11, and the best model's predictions are represented on the right.

*Figure 7-11. Tuning the number of trees using early stopping*

It is also possible to implement early stopping by actually stopping training early (instead of training a large number of trees first and then looking back to find the optimal number). You can do so by setting `warm_start=True`, which makes Scikit-Learn keep existing trees when the `fit()` method is called, allowing incremental training. The following code stops training when the validation error does not improve for five iterations in a row:

```
gbrt = GradientBoostingRegressor(max_depth=2, warm_start=True)

min_val_error = float("inf")
error_going_up = 0
for n_estimators in range(1, 120):
    gbrt.n_estimators = n_estimators
    gbrt.fit(X_train, y_train)
    y_pred = gbrt.predict(X_val)
    val_error = mean_squared_error(y_val, y_pred)
    if val_error < min_val_error:
        min_val_error = val_error
        error_going_up = 0
    else:
        error_going_up += 1
        if error_going_up == 5:
            break  # early stopping
```

The `GradientBoostingRegressor` class also supports a `subsample` hyperparameter, which specifies the fraction of training instances to be used for training each tree. For example, if `subsample=0.25`, then each tree is trained on 25% of the training instances, selected randomly. As you can probably guess by now, this technique trades a higher bias for a lower variance. It also speeds up training considerably. This is called *Stochastic Gradient Boosting*.

> **NOTE**
>
> It is possible to use Gradient Boosting with other cost functions. This is controlled by the `loss` hyperparameter (see Scikit-Learn's documentation for more details).

It is worth noting that an optimized implementation of Gradient Boosting is available in the popular Python library XGBoost, which stands for Extreme Gradient Boosting. This package was initially developed by Tianqi Chen as part of the Distributed (Deep) Machine Learning Community (DMLC), and it aims to be extremely fast, scalable, and portable. In fact, XGBoost is often an important component of the winning entries in ML competitions. XGBoost's API is quite similar to Scikit-Learn's:

```
import xgboost

xgb_reg = xgboost.XGBRegressor()
xgb_reg.fit(X_train, y_train)
y_pred = xgb_reg.predict(X_val)
```

XGBoost also offers several nice features, such as automatically taking care of early stopping:

```
xgb_reg.fit(X_train, y_train,
            eval_set=[(X_val, y_val)], early_stopping_rounds=2)
y_pred = xgb_reg.predict(X_val)
```

You should definitely check it out!

## Stacking

The last Ensemble method we will discuss in this chapter is called *stacking* (short for *stacked generalization*).[18] It is based on a simple idea: instead of using trivial functions (such as hard voting) to aggregate the predictions of all predictors in an ensemble, why don't we train a model to perform this aggregation? Figure 7-12 shows such an ensemble performing a regression task on a new instance. Each of the bottom three predictors predicts a different value (3.1, 2.7, and 2.9), and then the final predictor (called a *blender*, or a *meta learner*) takes these predictions as inputs and makes the final prediction (3.0).

*Figure 7-12. Aggregating predictions using a blending predictor*

To train the blender, a common approach is to use a hold-out set.[19] Let's see how it works. First, the training set is split into two subsets. The first subset is used to train the predictors in the first layer (see Figure 7-13).

*Figure 7-13. Training the first layer*

Next, the first layer's predictors are used to make predictions on the second (held-out) set (see Figure 7-14). This ensures that the predictions are "clean," since the predictors never saw these instances during training. For each instance in the hold-out set, there are three predicted values. We can create a new training set using these predicted values as input features (which makes this new training set 3D), and keeping the target values. The blender is trained on this new training set, so it learns to predict the target value, given the first layer's predictions.

*Figure 7-14. Training the blender*

It is actually possible to train several different blenders this way (e.g., one using Linear Regression, another using Random Forest Regression), to get a whole layer of blenders. The trick is to split the training set into three subsets: the first one is used to train the first layer, the second one is used to create the training set used to train the second layer (using predictions made by the predictors of the first layer), and the third one is used to create the training set to train the third layer (using predictions made by the predictors of the second layer). Once this is done, we can make a prediction for a new instance by going through each layer sequentially, as shown in Figure 7-15.

*Figure 7-15. Predictions in a multilayer stacking ensemble*

Unfortunately, Scikit-Learn does not support stacking directly, but it is not too hard to roll out your own implementation (see the following exercises). Alternatively, you can use an open source implementation such as `brew`.

## Exercises

1. If you have trained five different models on the exact same training data, and they all achieve 95% precision, is there any chance that you can combine these models to get better results? If so, how? If not, why?

2. What is the difference between hard and soft voting classifiers?
3. Is it possible to speed up training of a bagging ensemble by distributing it across multiple servers? What about pasting ensembles, boosting ensembles, Random Forests, or stacking ensembles?
4. What is the benefit of out-of-bag evaluation?
5. What makes Extra-Trees more random than regular Random Forests? How can this extra randomness help? Are Extra-Trees slower or faster than regular Random Forests?
6. If your AdaBoost ensemble underfits the training data, which hyperparameters should you tweak and how?
7. If your Gradient Boosting ensemble overfits the training set, should you increase or decrease the learning rate?
8. Load the MNIST data (introduced in Chapter 3), and split it into a training set, a validation set, and a test set (e.g., use 50,000 instances for training, 10,000 for validation, and 10,000 for testing). Then train various classifiers, such as a Random Forest classifier, an Extra-Trees classifier, and an SVM classifier. Next, try to combine them into an ensemble that outperforms each individual classifier on the validation set, using soft or hard voting. Once you have found one, try it on the test set. How much better does it perform compared to the individual classifiers?
9. Run the individual classifiers from the previous exercise to make predictions on the validation set, and create a new training set with the resulting predictions: each training instance is a vector containing the set of predictions from all your classifiers for an image, and the target is the image's class. Train a classifier on this new training set. Congratulations, you have just trained a blender, and together with the classifiers it forms a stacking ensemble! Now evaluate the ensemble on the test set. For each image in the test set, make predictions with all your classifiers, then feed the predictions to the blender to get the ensemble's predictions. How does it compare to the voting classifier you trained earlier?

Solutions to these exercises are available in Appendix A.

---

1 Leo Breiman, "Bagging Predictors," *Machine Learning* 24, no. 2 (1996): 123–140.

2 In statistics, resampling with replacement is called *bootstrapping*.

3 Leo Breiman, “Pasting Small Votes for Classification in Large Databases and On-Line,” *Machine Learning* 36, no. 1–2 (1999): 85–103.

4 Bias and variance were introduced in Chapter 4.

5 `max_samples` can alternatively be set to a float between 0.0 and 1.0, in which case the max number of instances to sample is equal to the size of the training set times `max_samples`.

6 As $m$ grows, this ratio approaches $1 - \exp(-1) \approx 63.212\%$.

7 Gilles Louppe and Pierre Geurts, “Ensembles on Random Patches,” *Lecture Notes in Computer Science* 7523 (2012): 346–361.

8 Tin Kam Ho, “The Random Subspace Method for Constructing Decision Forests,” *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20, no. 8 (1998): 832–844.

9 Tin Kam Ho, “Random Decision Forests,” *Proceedings of the Third International Conference on Document Analysis and Recognition* 1 (1995): 278.

10 The `BaggingClassifier` class remains useful if you want a bag of something other than Decision Trees.

11 There are a few notable exceptions: `splitter` is absent (forced to `"random"`), `presort` is absent (forced to `False`), `max_samples` is absent (forced to `1.0`), and `base_estimator` is absent (forced to `DecisionTreeClassifier` with the provided hyperparameters).

12 Pierre Geurts et al., “Extremely Randomized Trees,” *Machine Learning* 63, no. 1 (2006): 3–42.

13 Yoav Freund and Robert E. Schapire, “A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting,” *Journal of Computer and System Sciences* 55, no. 1 (1997): 119–139.

14 This is just for illustrative purposes. SVMs are generally not good base predictors for AdaBoost; they are slow and tend to be unstable with it.

15 The original AdaBoost algorithm does not use a learning rate hyperparameter.

16 For more details, see Ji Zhu et al., “Multi-Class AdaBoost,” *Statistics and Its Interface* 2, no. 3 (2009): 349–360.

17 Gradient Boosting was first introduced in Leo Breiman’s 1997 paper “Arcing the Edge” and was further developed in the 1999 paper “Greedy Function Approximation: A Gradient Boosting Machine” by Jerome H. Friedman.

18 David H. Wolpert, “Stacked Generalization,” *Neural Networks* 5, no. 2 (1992): 241–259.

19 Alternatively, it is possible to use out-of-fold predictions. In some contexts this is called *stacking*, while using a hold-out set is called *blending*. For many people these terms are synonymous.

In Figure 6.17, *K* and *N*, respectively, are number of features and records in the original set, while *k* and *n*, respectively, are the number of features and records in the sampled set (bootstrapped samples). The process of bootstrapping samples from original set to build multiple models and aggregating their results for final prediction is called *Bagging*. The term "bagging" comes from bootstrapping plus aggregating. One of the most widely used bagging technique is *Random Forest*.

### 6.5.5 | Random Forest

Random forest is one of the most popular ensemble techniques used in the industry due to its performance and scalability. A random forest is an ensemble of decision trees (classification and regression tree), where each decision tree is built from bootstrap samples (sampling with replacement) and randomly selected subset of features without replacement. The decision trees are normally grown deep (without pruning).

The number of estimators or models to be used in Random Forest can be tuned to increase the model accuracy. The hyperparameters in a Random Forest model are

1. Number of decision trees.
2. Number of records and features to be sampled.
3. Depth and search criteria (Gini impurity index or entropy).

*RandomForestClassifier* is available in *sklearn.ensemble* and takes the following parameters:

1. *n_estimators*: integer – The number of trees in the forest.
2. *criterion*: string – The function to measure the quality of a split, Gini or entropy. Default criteria used is Gini.
3. *max_feature*: int, float – The number of features to be used for each tree.
   - If int, then consider max_features for each tree.
   - If float, then max_features is a percentage and int(max_features * n_features) features are considered for each tree.
   - If "auto", then max_features = sqrt(n_features).
   - If "sqrt", then max_features = sqrt(n_features) (same as "auto").
   - If "log2", then max_features = log2(n_features).
4. *max_depth*: integer – The maximum depth of the tree.

#### *6.5.5.1 Building Random Forest Model*

Random forest model can be developed using the following commands. In this case, *max_depth* is 10 and *n_estimators* is 10.

```
## Importing Random Forest Classifier from the sklearn.ensemble
from sklearn.ensemble import RandomForestClassifier

## Initializing the Random Forest Classifier with max_dept and
## n_estimators
radm_clf = RandomForestClassifier(max_depth=10, n_estimators=10)
radm_clf.fit( train_X, train_y )
```

```
RandomForestClassifier(bootstrap=True, class_weight=None,
            criterion='gini',
            max_depth=15, max_features='auto', 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, n_estimators=20, n_jobs=1,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False)
```

Now we find the ROC AUC score and draw the ROC curve using utility method *draw_roc_curve()* (Figure 6.18).

```
_, _, _, _ = draw_roc_curve( radm_clf, test_X, test_y );
```

**FIGURE 6.18** ROC AUC curve of random forest model with 10 decision trees.

AUC for the random forest model is 0.86 (Figure 6.18) and better compared to the KNN model. But we can still improve the accuracy by using grid search by fine-tuning the hyperparameters.

### *6.5.5.2 Grid Search for Optimal Parameters*

We are not sure whether the hyperparameter used in the previous section for developing the random forest is optimal. We can use grid computing to find the best values for hyperparameters; one disadvantage of grid search is that it can be time-consuming. We will tune only the parameters *max_depth, n_estimators* and *max_features* and use a small set of values for search.

The commands for grid computing are provided below:

1. max_depth: 10 or 15
2. n_estimators: 10 or 20
3. max_features: sqrt or 0.2

```
## Configuring parameters and values for searched
tuned_parameters = [{'max_depth': [10, 15],
                     'n_estimators': [10,20],
                     'max_features': ['sqrt', 0.2]}]

## Initializing the RF classifier
radm_clf = RandomForestClassifier()

## Configuring search with the tunable parameters
clf = GridSearchCV(radm_clf,
                   tuned_parameters,
                   cv=5,
                   scoring='roc_auc')

## Fitting the training set
clf.fit(train_X, train_y)
```

```
GridSearchCV(cv=5, error_score='raise',
       estimator=RandomForestClassifier(bootstrap=True, class_
           weight=None, criterion='gini',
           max_depth=None, max_features='auto', 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, n_estimators=10, n_jobs=1,
           oob_score=False, random_state=None, verbose=0,
           warm_start=False),
       fit_params=None, iid=True, n_jobs=1,
       param_grid=[{'n_estimators': [10, 20], 'max_features: ['sqrt',
       'auto'], 'max_depth': [10, 15]}],
       pre_dispatch='2*n_jobs, refit=True, return_train_score= 'warn',
       scorings='roc_auc', verbose=0)
```

Printing the best score and best parameters.

```
clf.best_score_
```

```
0.9413
```

AUC score has reached 0.94 with the following optimal values for the hyperparameters. This may change slightly for the readers because the folds are created randomly by the cross-validation strategy.

```
clf.best_params_
```

```
{'max_depth': 15, 'max_features': 'auto', 'n_estimators': 20}
```

The best accuracy of 0.9413 AUC score is given by a random forest model with 20 decision trees (estimators), a maximum number of features as auto (square root of number of total features) and max depth as 15.

#### *6.5.5.3 Building the Final Model with Optimal Parameter Values*

Following commands can be used for developing a new random forest model with optimal parameter values.

```
## Initializing the Random Forest Model with the optimal values
radm_clf = RandomForestClassifier(max_depth=15,
                                  n_estimators=20,
                                  max_features = 'auto')

## Fitting the model with the training set
radm_clf.fit( train_X, train_y )
```

```
RandomForestClassifier(bootstrap=True, class_weight=None,
            criterion='gini',
            max_depth=15, max_features= 'auto', 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, n_estimators=20, n_jobs=1,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False)
```

#### *6.5.5.4 ROC AUC Score*

Following commands can be used for finding ROC AUC score and drawing the ROC curve using our utility method draw_roc_curve (Figure 6.19).

```
_, _, _, _ = draw_roc_curve( clf, test_X, test_y )
```

As show in the figure, the AUC value for this random forest is 0.94.

#### *6.5.5.5 Drawing the Confusion Matrix*

Use the following commands to drawing the confusion matrix using the utility method *draw_cm()* (Figure 6.20).

```
pred_y = radm_clf.predict( test_X )
draw_cm( test_y, pred_y )
```

**FIGURE 6.19** ROC AUC curve of random forest model with 10 decision trees.

**FIGURE 6.20** Confusion matrix of random forest model.

The model can detect 464 out of 575 subscribed cases and there are only 75 false positives cases. There could be slight variation in these values since the trees are developed using random sampling with replacement.

Let us print the classification report as shown in Table 6.3.

```
print( metrics.classification_report( test_y, pred_y ) )
```

TABLE 6.3 Classification report

| | Precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.91 | 0.94 | 0.93 | 1225 |
| 1 | 0.86 | 0.81 | 0.83 | 575 |
| avg/total | 0.90 | 0.90 | 0.90 | 1800 |

As shown in Table 6.3, the precision and recall for positive cases are 0.86 and 0.81, respectively, which are far better than what we obtained using logistic regression (Section 6.5.2) and KNN model (Section 6.5.3).

### *6.5.5.6 Finding Important Features*

Random forest algorithm reports feature importance by considering feature usage over all the trees in the forest. This gives good insight into which features have important information with respect to the outcome variable. It uses "Gini impurity reduction" or "mean decrease impurity" for calculating the importance.

Feature importance is calculated for a feature by multiplying error reduction at any node by the feature with the proportion of samples reaching that node. Then the values are averaged over all the trees to find final feature importance.

In *sklearn,* the classifiers return a parameter called *featureimportances*, which holds the feature importance values. We can store these values along with the column names in a DataFrame and plot them in descending order of importance for better visualization and interpretation (Figure 6.21).

```
import numpy as np

# Create a dataframe to store the features and their corresponding
# importances
feature_rank = pd.DataFrame( { 'feature': train_X.columns,
                               'importance': radm_clf.feature_
                               importances_ } )

## Sorting the features based on their importances with most
## important feature at top.
feature_rank = feature_rank.sort_values('importance', ascending =
                                                          False)

plt.figure(figsize=(8, 6))
# plot the values
sn.barplot( y = 'feature', x = 'importance', data = feature_rank );
```

**FIGURE 6.21** Features sorted by their importance values in the random forest model.

The top 5 features (Figure 6.21) are *balance, age, previous-campaign, current-campaign* and *house-loan_yes*. The importance score is normalized and shows the relative importance of features. The cumulative sum of features importance can show the amount of variance explained by the top five features.

Pandas' *cumsum()* method can be used to compute the cumulative sum, which is shown in Table 6.4.

```
feature_rank['cumsum'] = feature_rank.importance.cumsum() * 100
feature_rank.head(10)
```

**TABLE 6.4** Feature importance from random forest model

| | feature | importance | cumsum |
|---|---|---|---|
| 1 | balance | 0.256938 | 25.693789 |
| 0 | age | 0.214292 | 47.122946 |
| 3 | previous-campaign | 0.115596 | 58.682579 |
| 2 | current-campaign | 0.109440 | 69.626538 |
| 21 | housing-loan_yes | 0.045655 | 74.192006 |
| 17 | education_secondary | 0.026044 | 76.796379 |
| 15 | marital_married | 0.025683 | 79.364678 |

TABLE 6.4 Continued

| | feature | importance | cumsum |
|---|---|---|---|
| 22 | personal-loan_yes | 0.024546 | 81.819316 |
| 4 | job_blue-collar | 0.021823 | 84.001666 |
| 16 | marital_single | 0.019638 | 85.965419 |

The top five features provide 75% of the information in the data with respect to the outcome variable. This technique can also be used for feature selection. Random forest being a black box model, cannot be interpreted. But it can be used to select a subset of features using feature importance criteria and build simpler models for interpretation.

## 6.5.6 | Boosting

Boosting is another popular ensemble technique which combines multiple *weak classifiers* into a single *strong classifier*. A weak classifier is one which is slightly better than random guessing. That is, the error is less than 50%. Any classification algorithm can be used for boosting and is called base classifier. Boosting builds multiple classifiers in a sequential manner as opposed to bagging, which can build classifiers in parallel. Boosting builds initial classifier by giving equal weights to each sample and then focuses on correctly classifying misclassified examples in subsequent classifiers. Two most widely used boosting algorithms are *AdaBoost* and *Gradient Boosting*.

### *6.5.6.1 AdaBoost*

AdaBoost assigns weight to each record in a training set, which determines the probability that a specific record is selected for training a classifier. For first classifier, the weights of all examples will be equal (random sampling), but after training the first classifier, AdaBoost increases the weight for the misclassified records so that these records will have a higher probability of selection in the training set in anticipation that the next classifier will learn to classify them correctly.

After each classifier (usually a decision tree) is trained, the classifier's weight is computed based on its accuracy. More accurate classifiers are given more weight. The final prediction is based on the weighted sum of all classifiers and given by the following formula (Schapire, 1999):

$$F(X_i) = \text{Sign}\left(\sum_{k=1}^{K} \alpha_k f_k(X_i)\right) \tag{6.15}$$

where $K$ is the total number of classifiers, $f_k(x_i)$ is the output of weak classifier $k$ for input feature set $X_i$, $\alpha_k$ is the weight applied to classifier $k$ and is given by

$$\alpha_k = \frac{1}{2}\ln\left(\frac{1-\varepsilon_k}{\varepsilon_k}\right) \tag{6.16}$$

Here $\alpha_k$ is based on the classifier's error rate $\varepsilon_k$, which is the number of misclassified records over the training set divided by the total number of records of the training set. The final output $F(X_i)$ is just a linear combination of all of the weak classifiers, and final decision depends on the sign of this sum.

*sklearn.ensemble.AdaBoostClassifier* takes the following parameters to build an AdaBoost model:

1. *base_estimator*: object – The base estimator from which the boosted ensemble is built. The estimator should have the capability to utilize differential weights for samples (for example, decision tree or logistic regression).
2. *n_estimators*: integer – The maximum number of estimators at which boosting algorithm is terminated. In the case of perfect fit, the learning procedure is stopped early. Default is 50.
3. *learning_rate*: float – Learning rate shrinks the contribution of each classifier specified by learning_rate. There is a trade-off between learning_rate and n_estimators. Default value for learning_rate is 1.

The commands for building an AdaBoost model using Logistic Regression as base classifier and using 50 classifiers are as follows:

```
## Importing AdaBoost classifier
from sklearn.ensemble import AdaBoostClassifier

## Initializing logistic regression to use as base classifier
logreg_clf = LogisticRegression()

## Initializing AdaBoost classifier with 50 classifers
ada_clf = AdaBoostClassifier(logreg_clf, n_estimators=50)

## Fitting AdaBoost model to training set
ada_clf.fit(train_X, train_y)
```

```
AdaBoostClassifier(algorithm= 'SAMME.R',
          base_estimator=LogisticRegression(C=1.0, class_
          weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr',
          n_jobs=1,
          penalty='12', random_state=None, solver='liblinear',
          tol=0.0001,
          verbose=0, warm_start=False),
          learning_rate=1.0, n_estimators=50, random_state=None)
```

Use the following command to find ROC AUC score. We then draw the ROC curve for the AdaBoost classifier using our custom method *draw_roc_curve()* (Figure 6.22).

```
_, _, _, _ = draw_roc_curve( ada_clf, test_X, test_y )
```

As shown in Figure 6.22, the AUC for the AdaBoost is 0.71.

### *6.5.6.2 Gradient Boosting*

AdaBoost focusses on the misclassified examples in subsequent classifiers, whereas Gradient Boosting focusses on residuals from previous classifiers and fits a model to the residuals. Gradient boosting algorithm repetitively leverages the patterns in residuals and strengthens the model with weak predictions and makes it better. Once it reaches a stage in which residuals do not have any pattern that could be modeled, it stops modeling residuals. It uses gradient descent algorithm in each stage to minimize the error. Gradient boosting typically uses decision tree as base classifier.

#### 7.2.1.2 Other Distance Metrics

Some of the other widely used distances are as follows:

1. **Minkowski Distance:** It is the generalized distance measure between two observations.
2. **Jaccard Similarity Coefficient:** It is a measure used when the data is qualitative, especially when attributes can be represented in binary form.
3. **Cosine Similarity:** In this, $X_1$ and $X_2$ are two $n$-dimensional vectors. It measures the angle between two vectors (thus called as vector space model).
4. **Gower's Similarity Coefficient:** The distance and similarity measures that we have discussed so far are valid either for quantitative data or qualitative data. *Gower's similarity coefficient* can be used when both quantitative and qualitative features are present.

## 7.3 | *K*-MEANS CLUSTERING

*K*-means clustering is one of the frequently used clustering algorithms. It is a non-hierarchical clustering method in which the number of clusters ($K$) is decided *a priori*. The observations in the sample are assigned to one of the clusters (say $C_1, C_2, \ldots, C_K$) based on the distance between the observation and the centroid of the clusters.

The following steps are used in *K*-means clustering algorithm:

1. Decide the value of $K$ (which can be fine-tuned later).
2. Choose $K$ observations from the data that are likely to be in different clusters. There are many ways of choosing these initial $K$ values; the easiest approach is to choose observations that are farthest (in one of the parameters of the data).
3. The $K$ observations selected in step 2 are the centroids of those clusters.
4. For remaining observations, find the cluster closest to the centroid. Add the new observation (say observation $j$) to the cluster with the closest centroid. Adjust the centroid after adding a new observation to the cluster. The closest centroid is chosen based upon an appropriate distance measure.
5. Repeat step 4 until all observations are assigned to a cluster.

For the data described in Section 7.2 (*Income Data.csv*), we can create three clusters, as we know there are three segments from Figure 7.1.

*sklearn* library has *KMeans* algorithm. Initialize *KMeans* with the number of clusters ($k$) as an argument and call *fit()* method with the DataFrame, which contains entities and their features that need to be clustered.

```
from sklearn.cluster import KMeans
clusters = KMeans(3)
clusters.fit(customers_df)
```

```
KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
      n_clusters=3, n_init=10, n_jobs=1, precompute_distances='auto',
      random_state=None, tol=0.0001, verbose=0)
```

The output variable *clusters.labels_* contains labels that identify the cluster to which an observation belongs. We can concatenate it with the customers' data and verify.

```
customers_df["clusterid"] = clusters.labels_
```

Let us print first five customers with their cluster centers.

```
customers_df[0:5]
```

| | Income | Age | Clusterid |
|---|---|---|---|
| 0 | 41100.0 | 48.75 | 1 |
| 1 | 54100.0 | 28.10 | 2 |
| 2 | 47800.0 | 46.75 | 1 |
| 3 | 19100.0 | 40.25 | 0 |
| 4 | 18200.0 | 35.80 | 0 |

The three segments are numbered as 0, 1, and 2. The first customer belongs to cluster 1 whereas the second customer belongs to cluster 2 and so on.

### 7.3.1 | Plotting Customers with Their Segments

We will plot and depict each customer segment with different markings (see Figure 7.2). Plotting the age and income of customers from different segments with different markers. The markers [+, ^, .] will depict the clusters differently.

```
markers = ['+','^','.']
sn.lmplot("age", "income",
          data = customers_df,
          hue = "clusterid",
          fit_reg = False,
          markers = markers,
          size = 4);
```

**FIGURE 7.2** Three customer segments created by clustering.

The above clusters are mostly segmented based on income. This is because the salary is on a larger scale compared to the age. The scale of age is 0 to 60, while salary ranges from 0 to 50000. For example, difference in age between two customers, with age 20 and 70, is significant, but numerical difference is only 50. Similarly, difference in income between two customers, with income of 10000 and 11000, is not significant; the numerical difference is 1000. So, the distance will always be determined by the difference in salary and not in age. Hence before creating clusters, all features need to be normalized and brought to normalized scale. *StandardScaler in skleardn.preprocessing* normalizes all values by subtracting all values from its mean and dividing by standard deviation. Hence

$$X_{normalized} = \frac{X_i - \overline{X}}{\sigma_X}$$

where $\overline{X}$ is the mean of $X$ and $\sigma_X$ is the corresponding standard deviation of $X$ feature.

### 7.3.2 | Normalizing Features

*StandardScaler* is available under *sklearn.preprocessing* module. Import and initialize the *StandardScaler* and pass DataFrame to *fit()* method to transform the column values.

```
from sklearn.preprocessing import StandardScaler
```

```
scaler = StandardScaler()
scaled_customers_df = scaler.fit_transform(customers_df[["age",
                      "income"]])
scaled_customers_df[0:5]
```

```
array([[ 1.3701637,   0.09718548],
      [-1.3791283,    0.90602749],
      [ 1.10388844,   0.51405021],
      [ 0.23849387,   -1.27162408],
      [-0.35396857,   -1.32762083]])
```

We can create the clusters again using normalized feature set.

```
from sklearn.cluster import KMeans
clusters_new = KMeans(3, random_state=42)
clusters_new.fit(scaled_customers_df)
customers_df["clusterid_new"] = clusters_new.labels_
```

The new segments created can be plotted using the scatter plot and marking each segment differently (Figure 7.3).

```
markers = ['+','^','.']
```

```
sn.lmplot("age", "income",
         data = customers_df,
         hue = "clusterid_new",
         fit_reg = False,
         markers = markers,
         size = 4);
```

**FIGURE 7.3** Three customer segments created by clustering after normalizing features.

Now the clusters seem to have been created properly. The cluster centers are given by *clusters.cluster_centers_*.

```
clusters.cluster_centers_
```

```
array([[ 0.5361335,  0.96264295,  0.51632566, -0.61618888],
       [-0.92065895, -0.6352664, -0.86290041, -0.33288365],
       [ 0.7690509,  -0.6547531,  0.69314951, 1.89814505]])
```

But these are standardized values. Hence, we can calculate the cluster centers from the original DataFrame using the information "which customers have been allocated to which segment".

### 7.3.3 | Cluster Centers and Interpreting the Clusters

Each cluster is defined by its cluster center. The cluster center explains the characteristic of the cluster and helps us to interpret the cluster. For example, in this example of customer segmentation, the cluster center provides the average age and average income of each cluster. Of course, the customer's age and income will vary from the cluster centers and is called the cluster variance.

```
customers_df.groupby('clusterid')['age',
                                  'income'].agg(["mean",
                                                 'std']).reset_index()
```

| Clusterid | Age | | Income | |
|---|---|---|---|---|
| | Mean | Std | Mean | Std |
| 0 | 31.700435 | 6.122122 | 54675.652174 | 2362.224320 |
| 1 | 39.174479 | 3.626068 | 18144.791667 | 6745.241906 |
| 2 | 46.419101 | 2.289620 | 43053.932584 | 3613.769632 |

CHAPTER 5

# Support Vector Machines

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as he or she writes—so you can take advantage of these technologies long before the official release of these titles. The following will be Chapter 5 in the final release of the book.

A *Support Vector Machine* (SVM) is a very powerful and versatile Machine Learning model, capable of performing linear or nonlinear classification, regression, and even outlier detection. It is one of the most popular models in Machine Learning, and anyone interested in Machine Learning should have it in their toolbox. SVMs are particularly well suited for classification of complex but small- or medium-sized datasets.

This chapter will explain the core concepts of SVMs, how to use them, and how they work.

## Linear SVM Classification

The fundamental idea behind SVMs is best explained with some pictures. Figure 5-1 shows part of the iris dataset that was introduced at the end of Chapter 4. The two classes can clearly be separated easily with a straight line (they are *linearly separable*). The left plot shows the decision boundaries of three possible linear classifiers. The model whose decision boundary is represented by the dashed line is so bad that it does not even separate the classes properly. The other two models work perfectly on this training set, but their decision boundaries come so close to the instances that these models will probably not perform as well on new instances. In contrast, the solid line in the plot on the right represents the decision boundary of an SVM classifier; this line not only separates the two classes but also stays as far away from the closest training instances as possible. You can think of an SVM classifier as fitting the

widest possible street (represented by the parallel dashed lines) between the classes. This is called *large margin classification*.

*Figure 5-1. Large margin classification*

Notice that adding more training instances "off the street" will not affect the decision boundary at all: it is fully determined (or "supported") by the instances located on the edge of the street. These instances are called the *support vectors* (they are circled in Figure 5-1).

SVMs are sensitive to the feature scales, as you can see in Figure 5-2: on the left plot, the vertical scale is much larger than the horizontal scale, so the widest possible street is close to horizontal. After feature scaling (e.g., using Scikit-Learn's `StandardScaler`), the decision boundary looks much better (on the right plot).

*Figure 5-2. Sensitivity to feature scales*

## Soft Margin Classification

If we strictly impose that all instances be off the street and on the right side, this is called *hard margin classification*. There are two main issues with hard margin classification. First, it only works if the data is linearly separable, and second it is quite sensitive to outliers. Figure 5-3 shows the iris dataset with just one additional outlier: on the left, it is impossible to find a hard margin, and on the right the decision boundary ends up very different from the one we saw in Figure 5-1 without the outlier, and it will probably not generalize as well.

*Figure 5-3. Hard margin sensitivity to outliers*

To avoid these issues it is preferable to use a more flexible model. The objective is to find a good balance between keeping the street as large as possible and limiting the *margin violations* (i.e., instances that end up in the middle of the street or even on the wrong side). This is called *soft margin classification*.

In Scikit-Learn's SVM classes, you can control this balance using the `C` hyperparameter: a smaller `C` value leads to a wider street but more margin violations. Figure 5-4 shows the decision boundaries and margins of two soft margin SVM classifiers on a nonlinearly separable dataset. On the left, using a low `C` value the margin is quite large, but many instances end up on the street. On the right, using a high `C` value the classifier makes fewer margin violations but ends up with a smaller margin. However, it seems likely that the first classifier will generalize better: in fact even on this training set it makes fewer prediction errors, since most of the margin violations are actually on the correct side of the decision boundary.

*Figure 5-4. Large margin (left) versus fewer margin violations (right)*

If your SVM model is overfitting, you can try regularizing it by reducing `C`.

The following Scikit-Learn code loads the iris dataset, scales the features, and then trains a linear SVM model (using the `LinearSVC` class with $C = 1$ and the *hinge loss* function, described shortly) to detect Iris-Virginica flowers. The resulting model is represented on the left of Figure 5-4.

```
import numpy as np
from sklearn import datasets
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC

iris = datasets.load_iris()
X = iris["data"][:, (2, 3)]  # petal length, petal width
y = (iris["target"] == 2).astype(np.float64)  # Iris-Virginica

svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("linear_svc", LinearSVC(C=1, loss="hinge")),
    ])

svm_clf.fit(X, y)
```

Then, as usual, you can use the model to make predictions:

```
>>> svm_clf.predict([[5.5, 1.7]])
array([1.])
```

Unlike Logistic Regression classifiers, SVM classifiers do not output probabilities for each class.

Alternatively, you could use the `SVC` class, using `SVC(kernel="linear", C=1)`, but it is much slower, especially with large training sets, so it is not recommended. Another option is to use the `SGDClassifier` class, with `SGDClassifier(loss="hinge", alpha=1/(m*C))`. This applies regular Stochastic Gradient Descent (see Chapter 4) to train a linear SVM classifier. It does not converge as fast as the `LinearSVC` class, but it can be useful to handle huge datasets that do not fit in memory (out-of-core training), or to handle online classification tasks.

The `LinearSVC` class regularizes the bias term, so you should center the training set first by subtracting its mean. This is automatic if you scale the data using the `StandardScaler`. Moreover, make sure you set the `loss` hyperparameter to `"hinge"`, as it is not the default value. Finally, for better performance you should set the `dual` hyperparameter to `False`, unless there are more features than training instances (we will discuss duality later in the chapter).

# Nonlinear SVM Classification

Although linear SVM classifiers are efficient and work surprisingly well in many cases, many datasets are not even close to being linearly separable. One approach to handling nonlinear datasets is to add more features, such as polynomial features (as you did in Chapter 4); in some cases this can result in a linearly separable dataset. Consider the left plot in Figure 5-5: it represents a simple dataset with just one feature $x_1$. This dataset is not linearly separable, as you can see. But if you add a second feature $x_2 = (x_1)^2$, the resulting 2D dataset is perfectly linearly separable.

*Figure 5-5. Adding features to make a dataset linearly separable*

To implement this idea using Scikit-Learn, you can create a `Pipeline` containing a `PolynomialFeatures` transformer (discussed in "Polynomial Regression" on page 130), followed by a `StandardScaler` and a `LinearSVC`. Let's test this on the moons dataset: this is a toy dataset for binary classification in which the data points are shaped as two interleaving half circles (see Figure 5-6). You can generate this dataset using the `make_moons()` function:

```
from sklearn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures

polynomial_svm_clf = Pipeline([
        ("poly_features", PolynomialFeatures(degree=3)),
        ("scaler", StandardScaler()),
        ("svm_clf", LinearSVC(C=10, loss="hinge"))
    ])

polynomial_svm_clf.fit(X, y)
```

*Figure 5-6. Linear SVM classifier using polynomial features*

## Polynomial Kernel

Adding polynomial features is simple to implement and can work great with all sorts of Machine Learning algorithms (not just SVMs), but at a low polynomial degree it cannot deal with very complex datasets, and with a high polynomial degree it creates a huge number of features, making the model too slow.

Fortunately, when using SVMs you can apply an almost miraculous mathematical technique called the *kernel trick* (it is explained in a moment). It makes it possible to get the same result as if you added many polynomial features, even with very high-degree polynomials, without actually having to add them. So there is no combinatorial explosion of the number of features since you don't actually add any features. This trick is implemented by the `SVC` class. Let's test it on the moons dataset:

```
from sklearn.svm import SVC
poly_kernel_svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))
    ])
poly_kernel_svm_clf.fit(X, y)
```

This code trains an SVM classifier using a 3rd-degree polynomial kernel. It is represented on the left of Figure 5-7. On the right is another SVM classifier using a 10th-degree polynomial kernel. Obviously, if your model is overfitting, you might want to

reduce the polynomial degree. Conversely, if it is underfitting, you can try increasing it. The hyperparameter `coef0` controls how much the model is influenced by high-degree polynomials versus low-degree polynomials.

*Figure 5-7. SVM classifiers with a polynomial kernel*

A common approach to find the right hyperparameter values is to use grid search (see Chapter 2). It is often faster to first do a very coarse grid search, then a finer grid search around the best values found. Having a good sense of what each hyperparameter actually does can also help you search in the right part of the hyperparameter space.

## Adding Similarity Features

Another technique to tackle nonlinear problems is to add features computed using a *similarity function* that measures how much each instance resembles a particular *landmark*. For example, let's take the one-dimensional dataset discussed earlier and add two landmarks to it at $x_1 = -2$ and $x_1 = 1$ (see the left plot in Figure 5-8). Next, let's define the similarity function to be the Gaussian *Radial Basis Function* (*RBF*) with $\gamma = 0.3$ (see Equation 5-1).

*Equation 5-1. Gaussian RBF*

$$\phi_\gamma(\mathbf{x}, \ell) = \exp\left(-\gamma \| \mathbf{x} - \ell \|^2\right)$$

It is a bell-shaped function varying from 0 (very far away from the landmark) to 1 (at the landmark). Now we are ready to compute the new features. For example, let's look at the instance $x_1 = -1$: it is located at a distance of 1 from the first landmark, and 2 from the second landmark. Therefore its new features are $x_2 = \exp(-0.3 \times 1^2) \approx 0.74$ and $x_3 = \exp(-0.3 \times 2^2) \approx 0.30$. The plot on the right of Figure 5-8 shows the transformed dataset (dropping the original features). As you can see, it is now linearly separable.

*Figure 5-8. Similarity features using the Gaussian RBF*

You may wonder how to select the landmarks. The simplest approach is to create a landmark at the location of each and every instance in the dataset. This creates many dimensions and thus increases the chances that the transformed training set will be linearly separable. The downside is that a training set with $m$ instances and $n$ features gets transformed into a training set with $m$ instances and $m$ features (assuming you drop the original features). If your training set is very large, you end up with an equally large number of features.

## Gaussian RBF Kernel

Just like the polynomial features method, the similarity features method can be useful with any Machine Learning algorithm, but it may be computationally expensive to compute all the additional features, especially on large training sets. However, once again the kernel trick does its SVM magic: it makes it possible to obtain a similar result as if you had added many similarity features, without actually having to add them. Let's try the Gaussian RBF kernel using the `SVC` class:

```
rbf_kernel_svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="rbf", gamma=5, C=0.001))
    ])
rbf_kernel_svm_clf.fit(X, y)
```

This model is represented on the bottom left of Figure 5-9. The other plots show models trained with different values of hyperparameters `gamma` ($\gamma$) and $C$. Increasing `gamma` makes the bell-shape curve narrower (see the left plot of Figure 5-8), and as a result each instance's range of influence is smaller: the decision boundary ends up being more irregular, wiggling around individual instances. Conversely, a small `gamma` value makes the bell-shaped curve wider, so instances have a larger range of influence, and the decision boundary ends up smoother. So $\gamma$ acts like a regularization hyperparameter: if your model is overfitting, you should reduce it, and if it is underfitting, you should increase it (similar to the `C` hyperparameter).

*Figure 5-9. SVM classifiers using an RBF kernel*

Other kernels exist but are used much more rarely. For example, some kernels are specialized for specific data structures. *String kernels* are sometimes used when classifying text documents or DNA sequences (e.g., using the *string subsequence kernel* or kernels based on the *Levenshtein distance*).

With so many kernels to choose from, how can you decide which one to use? As a rule of thumb, you should always try the linear kernel first (remember that `LinearSVC` is much faster than `SVC(kernel="linear")`), especially if the training set is very large or if it has plenty of features. If the training set is not too large, you should try the Gaussian RBF kernel as well; it works well in most cases. Then if you have spare time and computing power, you can also experiment with a few other kernels using cross-validation and grid search, especially if there are kernels specialized for your training set's data structure.

## Computational Complexity

The `LinearSVC` class is based on the *liblinear* library, which implements an optimized algorithm for linear SVMs.[1] It does not support the kernel trick, but it scales almost

1 "A Dual Coordinate Descent Method for Large-scale Linear SVM," Lin et al. (2008).

linearly with the number of training instances and the number of features: its training time complexity is roughly $O(m \times n)$.

The algorithm takes longer if you require a very high precision. This is controlled by the tolerance hyperparameter $\epsilon$ (called `tol` in Scikit-Learn). In most classification tasks, the default tolerance is fine.

The `SVC` class is based on the *libsvm* library, which implements an algorithm that supports the kernel trick.[2] The training time complexity is usually between $O(m^2 \times n)$ and $O(m^3 \times n)$. Unfortunately, this means that it gets dreadfully slow when the number of training instances gets large (e.g., hundreds of thousands of instances). This algorithm is perfect for complex but small or medium training sets. However, it scales well with the number of features, especially with *sparse features* (i.e., when each instance has few nonzero features). In this case, the algorithm scales roughly with the average number of nonzero features per instance. Table 5-1 compares Scikit-Learn's SVM classification classes.

*Table 5-1. Comparison of Scikit-Learn classes for SVM classification*

| Class | Time complexity | Out-of-core support | Scaling required | Kernel trick |
|---|---|---|---|---|
| `LinearSVC` | $O(m \times n)$ | No | Yes | No |
| `SGDClassifier` | $O(m \times n)$ | Yes | Yes | No |
| `SVC` | $O(m^2 \times n)$ to $O(m^3 \times n)$ | No | Yes | Yes |

# SVM Regression

As we mentioned earlier, the SVM algorithm is quite versatile: not only does it support linear and nonlinear classification, but it also supports linear and nonlinear regression. The trick is to reverse the objective: instead of trying to fit the largest possible street between two classes while limiting margin violations, SVM Regression tries to fit as many instances as possible *on* the street while limiting margin violations (i.e., instances *off* the street). The width of the street is controlled by a hyperparameter $\epsilon$. Figure 5-10 shows two linear SVM Regression models trained on some random linear data, one with a large margin ($\epsilon = 1.5$) and the other with a small margin ($\epsilon = 0.5$).

2 "Sequential Minimal Optimization (SMO)," J. Platt (1998).

*Figure 5-10. SVM Regression*

Adding more training instances within the margin does not affect the model's predictions; thus, the model is said to be *ϵ-insensitive*.

You can use Scikit-Learn's `LinearSVR` class to perform linear SVM Regression. The following code produces the model represented on the left of Figure 5-10 (the training data should be scaled and centered first):

```
from sklearn.svm import LinearSVR

svm_reg = LinearSVR(epsilon=1.5)
svm_reg.fit(X, y)
```

To tackle nonlinear regression tasks, you can use a kernelized SVM model. For example, Figure 5-11 shows SVM Regression on a random quadratic training set, using a 2nd-degree polynomial kernel. There is little regularization on the left plot (i.e., a large `C` value), and much more regularization on the right plot (i.e., a small `C` value).

*Figure 5-11. SVM regression using a 2nd-degree polynomial kernel*

The following code produces the model represented on the left of Figure 5-11 using Scikit-Learn's `SVR` class (which supports the kernel trick). The `SVR` class is the regression equivalent of the `SVC` class, and the `LinearSVR` class is the regression equivalent of the `LinearSVC` class. The `LinearSVR` class scales linearly with the size of the training set (just like the `LinearSVC` class), while the `SVR` class gets much too slow when the training set grows large (just like the `SVC` class).

```
from sklearn.svm import SVR

svm_poly_reg = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)
svm_poly_reg.fit(X, y)
```

SVMs can also be used for outlier detection; see Scikit-Learn's documentation for more details.

# Under the Hood

This section explains how SVMs make predictions and how their training algorithms work, starting with linear SVM classifiers. You can safely skip it and go straight to the exercises at the end of this chapter if you are just getting started with Machine Learning, and come back later when you want to get a deeper understanding of SVMs.

First, a word about notations: in Chapter 4 we used the convention of putting all the model parameters in one vector $\boldsymbol{\theta}$, including the bias term $\theta_0$ and the input feature weights $\theta_1$ to $\theta_n$, and adding a bias input $x_0 = 1$ to all instances. In this chapter, we will use a different convention, which is more convenient (and more common) when you are dealing with SVMs: the bias term will be called $b$ and the feature weights vector will be called $\mathbf{w}$. No bias feature will be added to the input feature vectors.

## Decision Function and Predictions

The linear SVM classifier model predicts the class of a new instance $\mathbf{x}$ by simply computing the decision function $\mathbf{w}^T \mathbf{x} + b = w_1 x_1 + \cdots + w_n x_n + b$: if the result is positive, the predicted class $\hat{y}$ is the positive class (1), or else it is the negative class (0); see Equation 5-2.

*Equation 5-2. Linear SVM classifier prediction*

$$\hat{y} = \begin{cases} 0 & \text{if } \mathbf{w}^T\mathbf{x} + b < 0, \\ 1 & \text{if } \mathbf{w}^T\mathbf{x} + b \geq 0 \end{cases}$$

Figure 5-12 shows the decision function that corresponds to the model on the left of Figure 5-4: it is a two-dimensional plane since this dataset has two features (petal width and petal length). The decision boundary is the set of points where the decision function is equal to 0: it is the intersection of two planes, which is a straight line (represented by the thick solid line).[3]

*Figure 5-12. Decision function for the iris dataset*

The dashed lines represent the points where the decision function is equal to 1 or –1: they are parallel and at equal distance to the decision boundary, forming a margin around it. Training a linear SVM classifier means finding the value of $\mathbf{w}$ and $b$ that make this margin as wide as possible while avoiding margin violations (hard margin) or limiting them (soft margin).

## Training Objective

Consider the slope of the decision function: it is equal to the norm of the weight vector, $\| \mathbf{w} \|$. If we divide this slope by 2, the points where the decision function is equal to ±1 are going to be twice as far away from the decision boundary. In other words, dividing the slope by 2 will multiply the margin by 2. Perhaps this is easier to visualize in 2D in Figure 5-13. The smaller the weight vector $\mathbf{w}$, the larger the margin.

3 More generally, when there are $n$ features, the decision function is an $n$-dimensional *hyperplane*, and the decision boundary is an $(n - 1)$-dimensional hyperplane.

*Figure 5-13. A smaller weight vector results in a larger margin*

So we want to minimize $\| \mathbf{w} \|$ to get a large margin. However, if we also want to avoid any margin violation (hard margin), then we need the decision function to be greater than 1 for all positive training instances, and lower than –1 for negative training instances. If we define $t^{(i)} = -1$ for negative instances (if $y^{(i)} = 0$) and $t^{(i)} = 1$ for positive instances (if $y^{(i)} = 1$), then we can express this constraint as $t^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + b) \geq 1$ for all instances.

We can therefore express the hard margin linear SVM classifier objective as the *constrained optimization* problem in Equation 5-3.

*Equation 5-3. Hard margin linear SVM classifier objective*

$$\underset{\mathbf{w}, b}{\text{minimize}} \quad \frac{1}{2}\mathbf{w}^T\mathbf{w}$$

$$\text{subject to} \quad t^{(i)}\left(\mathbf{w}^T\mathbf{x}^{(i)} + b\right) \geq 1 \quad \text{for } i = 1, 2, \cdots, m$$

We are minimizing $\frac{1}{2}\mathbf{w}^T \mathbf{w}$, which is equal to $\frac{1}{2}\| \mathbf{w} \|^2$, rather than minimizing $\| \mathbf{w} \|$. Indeed, $\frac{1}{2}\| \mathbf{w} \|^2$ has a nice and simple derivative (it is just $\mathbf{w}$) while $\| \mathbf{w} \|$ is not differentiable at $\mathbf{w} = \mathbf{0}$. Optimization algorithms work much better on differentiable functions.

To get the soft margin objective, we need to introduce a *slack variable* $\zeta^{(i)} \geq 0$ for each instance:[4] $\zeta^{(i)}$ measures how much the i$^{\text{th}}$ instance is allowed to violate the margin. We now have two conflicting objectives: making the slack variables as small as possible to reduce the margin violations, and making $\frac{1}{2}\mathbf{w}^T \mathbf{w}$ as small as possible to increase the margin. This is where the `C` hyperparameter comes in: it allows us to define the trade-

4 Zeta ($\zeta$) is the 6$^{\text{th}}$ letter of the Greek alphabet.

off between these two objectives. This gives us the constrained optimization problem in Equation 5-4.

*Equation 5-4. Soft margin linear SVM classifier objective*

$$\underset{\mathbf{w}, b, \zeta}{\text{minimize}} \quad \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{m}\zeta^{(i)}$$

$$\text{subject to} \quad t^{(i)}\left(\mathbf{w}^T\mathbf{x}^{(i)} + b\right) \geq 1 - \zeta^{(i)} \quad \text{and} \quad \zeta^{(i)} \geq 0 \quad \text{for } i = 1, 2, \cdots, m$$

## Quadratic Programming

The hard margin and soft margin problems are both convex quadratic optimization problems with linear constraints. Such problems are known as *Quadratic Programming* (QP) problems. Many off-the-shelf solvers are available to solve QP problems using a variety of techniques that are outside the scope of this book.[5] The general problem formulation is given by Equation 5-5.

*Equation 5-5. Quadratic Programming problem*

$$\begin{aligned}
&\underset{\mathbf{p}}{\text{Minimize}} \quad \frac{1}{2}\mathbf{p}^T\mathbf{H}\mathbf{p} \quad + \quad \mathbf{f}^T\mathbf{p} \\
&\text{subject to} \quad \mathbf{A}\mathbf{p} \leq \mathbf{b} \\
&\text{where} \quad \begin{cases}
\mathbf{p} & \text{is an } n_p\text{-dimensional vector } (n_p = \text{number of parameters}), \\
\mathbf{H} & \text{is an } n_p \times n_p \text{ matrix}, \\
\mathbf{f} & \text{is an } n_p\text{-dimensional vector}, \\
\mathbf{A} & \text{is an } n_c \times n_p \text{ matrix } (n_c = \text{number of constraints}), \\
\mathbf{b} & \text{is an } n_c\text{-dimensional vector}.
\end{cases}
\end{aligned}$$

Note that the expression $\mathbf{A}\,\mathbf{p} \leq \mathbf{b}$ actually defines $n_c$ constraints: $\mathbf{p}^T\,\mathbf{a}^{(i)} \leq b^{(i)}$ for $i = 1, 2, \cdots, n_c$, where $\mathbf{a}^{(i)}$ is the vector containing the elements of the i$^{\text{th}}$ row of $\mathbf{A}$ and $b^{(i)}$ is the i$^{\text{th}}$ element of $\mathbf{b}$.

You can easily verify that if you set the QP parameters in the following way, you get the hard margin linear SVM classifier objective:

- $n_p = n + 1$, where $n$ is the number of features (the +1 is for the bias term).

5 To learn more about Quadratic Programming, you can start by reading Stephen Boyd and Lieven Vandenberghe, *Convex Optimization* (Cambridge, UK: Cambridge University Press, 2004) or watch Richard Brown's series of video lectures.