Chapter 11
Classification: Assigning Observations to Known Categories

In Chap. 10, we learned about using clustering methods to identify structure within a dataset. We learned about how the most challenging part of clustering is not applying a model, but interpreting the output in a meaningful and useful way. In this chapter, we will explore *supervised learning* methods. Unlike with clustering, generally, the value of a supervised model output is inherent in the framing of the question. This makes interpretation easier, but it requires an outcome variable to have a strong relationship with its indicator variables, and it benefits from data that are well structured and clean. With statistical modeling, people often say "garbage in, garbage out," meaning that even a very sophisticated model will not be able to produce reliable results if the data are not high quality or there is no actual relationship between input and output variables.

11.1 Classification

Whereas clustering is the process of *discovering* group membership, classification is the *prediction* of membership. In this section we look at two examples of classification: predicting segment membership, and predicting who is likely to subscribe to a service.

Classification uses observations whose status is *known* to derive predictors, and then applies those predictors to new observations. When working with a single dataset it is typically split into a *training* set that is used to develop the classification model, and a *test* set that is used to determine performance. It is crucial not to assess performance on the same observations that were used to develop the model.

A classification project typically includes the following steps at a minimum:

- A dataset is collected in which group membership for each observation is known or assigned (e.g., assigned by behavioral observation, expert rating, or clustering procedures)
- The dataset is split into a training set and a test set. A common pattern is to select 50–80% of the observations for the training set (70% seems to be particularly common), and to assign the remaining observations to the test set.
- A prediction model is built, with a goal to predict membership in the training data as well as possible.
- The resulting model is then assessed for performance using the test data. Performance is assessed to see that it exceeds chance (base rate). Additionally one might assess whether the method performs better than a reasonable alternative (and simpler or better-known) model.

Classification is an even more complex area than clustering, with hundreds of methods, thousands of academic papers each year, and enormous interest with technology and data analytics firms. Our goal is not to cover all of that but to demonstrate the common patterns, in Python generally and scikit-learn specifically, using two of the best-known and most useful classification methods, the naive Bayes and random forest classifiers.

11.1.1 Naive Bayes Classification: `GaussianNB()`

A simple yet powerful classification method is the *Naive Bayes* (NB) classifier. Naive Bayes uses training data to learn the probability of class membership as a function of each predictor variable considered independently (hence "naive"). When applied to new data, class membership is assigned to the category considered to be most likely according to the joint

J. S. Schwarz et al., *Python for Marketing Research and Analytics*, https://doi.org/10.1007/978-3-030-49720-0_11

probabilities assigned by the combination of predictors. We use the `naive_bayes` library from scikit-learn (Pedregosa et al. 2011).

We will use the same data as in Chap. 10:

```
In [0]: import pandas as pd
        seg_df = pd.read_csv('http://bit.ly/PMR-ch5')
        seg_df['is_female'] = seg_df.gender == 'female'
        seg_sub = seg_df.drop(['Segment', 'gender'], axis=1)
        seg_sub.head()
```

The first step in training a classifier is to split the data into *training* and *test* data, which will allow one to check whether the model works on the test data (or is instead overfitted to the training data). We select 70% of the data to use for training and keep the unselected cases as holdout (test) data. Classification requires known segment assignments in order to learn how to assign new values, which we will store in `seg_labels`. The convention is that the independent variables are assigned to `X`, e.g. `X_train` and `X_test`, and the dependent variable (or *label*) to `y`, e.g. `y_train` and `y_test`.

```
In [1]: import numpy as np

        seg_labels = seg_df.Segment
        np.random.seed(537)
        rand_idx = np.random.rand(seg_labels.shape[0])
        train_idx = rand_idx <= 0.7
        test_idx = rand_idx > 0.7

        X_train = seg_sub.iloc[train_idx]
        X_test = seg_sub.iloc[test_idx]

        y_train = seg_labels.iloc[train_idx]
        y_test = seg_labels.iloc[test_idx]
```

Why do we hold out a subset of the data in `X_test` and `y_test`? We do so to assess *overfitting* of the model. The model might learn the training data incredibly well and be able to assign labels within the training dataset with 100% accuracy, but we want the model to be *generalizable*, to be effective for data it has not observed. By training the model on one subset of the data and then evaluating its performance on another subset, we can estimate its performance on unknown data.

We then use the training data to train a naive Bayes classifier to predict Segment membership from all other variables in the training data. This is a very simple command:

```
In [2]: from sklearn import naive_bayes

        nb = naive_bayes.GaussianNB()

        nb.fit(X_train, y_train)

        list(zip(nb.classes_, nb.class_prior_))
```

```
Out[2]: [('moving up', 0.27102803738317754),
         ('suburb_mix', 0.32242990654205606),
         ('travelers', 0.2523364485981308),
         ('urban_hip', 0.1542056074766355)]
```

Looking at the `class_prior_` values offers some insight into how the model works. First, the a priori likelihood of segment membership—i.e., the estimated odds of membership before any other information is added—is 27.1% for the Moving up segment, 32.2% for the Suburb mix segment, and so forth. The model uses probabilities conditional on each predictor.

The NB classifier starts with the observed probabilities of gender, age, etc., *conditional on segment* found in the training data. It then uses Bayes' Rules to compute the *probability of segment*, conditional on gender, age, etc. This can then be used to estimate segment membership (i.e., assign a label or make a prediction) in new observations such as the test data. You have likely seen a description of how Bayes' Rule works, and we will not repeat it here. For details, refer to a general text on Bayesian methods such as Kruschke (2016).

What does this look like in practice? We can generate predictions for the whole input dataset, including both training and test data, and look at the true and predicted labels for a few users:

```
In [3]: predictions = nb.predict(seg_sub)
        seg_sub_pred = seg_sub.copy()
        seg_sub_pred['prediction'] = predictions
        seg_sub_pred['true_segment'] = seg_df['Segment']
        seg_sub_pred.sample(5)
```

```
Out[3]:          age        income     kids  own_home  subscribe  is_female  \
        183  32.806946  60752.625106     5     False      False       True
        194  43.302666  71789.130948     1     False      False      False
        201  34.294615  62236.114534     5     False      False      False
        99   31.673893  75433.895743     3      True      False      False
        10   79.650722  32013.086824     0      True      False      False

             prediction  true_segment
        183   moving_up    moving_up
        194  suburb_mix    moving_up
        201   moving_up    moving_up
        99   suburb_mix   suburb_mix
        10    travelers    travelers
```

What do we see here? First, comparing the `prediction` and `true_segment` columns, the model appears to be performing well: only a single row does not match. The model has used those other fields, age, income, kids, etc., to generate a reasonable prediction.

We see it performed fairly well on these few users, but how well did the model perform overall? Using the test data, we can check the accuracy of the model using the `score()` method:

```
In [4]: nb.score(X_test, y_test)
```

```
Out[4]: 0.8488372093023255
```

This returns an accuracy score, the agreement between predicted and actual segment membership, which in this case is about 85%.

However, when the base rate of an outcome is high, then a high level of raw agreement is not meaningful on its own. For example, if 98% of consumers do not purchase a product, then a prediction accuracy of 95% (off by 3%) is worse than simply predicting 100% non-purchase (off by 2%). Instead of raw agreement, one should assess performance of the model in terms of predictive power. A common metric used is the F_1 *score*, which is the harmonic mean of precision and recall (which we introduce in detail below).

In this case, we see that NB was able to recover the segments in the test data imperfectly but substantially better than chance, the F_1 score is also about 85%:

```
In [5]: from sklearn import metrics

        y_pred = nb.predict(X_test)

        metrics.f1_score(y_true=y_test, y_pred=y_pred, average='weighted')
```

```
Out[5]: 0.8532809445929236
```

The `average` parameter specifies how the performance for the four different segments (called *classes* in `sklearn`) should be combined, in this case `weighted`. This means that the F_1 score will be calculated for each class and the average calculated, weighted by the class proportions in the population.

We compare performance for each category using what is known in machine learning as a *confusion matrix*:

```
In [6]: import seaborn as sns
        import matplotlib.pyplot as plt
```

```
def confusion_matrix(y_true, y_pred, model):
  conf_mat = metrics.confusion_matrix(y_true, y_pred)

  sns.heatmap(conf_mat.T,
              xticklabels=model.classes_, yticklabels=model.classes_,
              annot=True, fmt='d')
  plt.xlabel('true label')
  plt.ylabel('predicted label')
```

```
In [7]: confusion_matrix(y_test, y_pred, nb)
```

The output can be seen in Fig. 11.1

Correct predictions are indicated on the diagonal. The NB prediction (shown in the rows) was correct for the vast majority of observations in each segment, except moving_up. When we examine individual categories, we see that NB was correct for every proposed member of the Urban hip segment (17 correct out of 17 proposed), and for over 96% of the Traveler proposals (26 correct out of the proposed 27). However, it incorrectly classified 10 of the actual 31 Suburb mix respondents into other segments, and similarly failed to identify 9 of the true Moving up segment.

This demonstrates the asymmetry of prediction: the model needs to correctly identify both true positives *and* true negatives. There is tension between those requirements, which correspond to two important statistical concepts in machine learning.

The first is *precision*, which is the proportion of the sample identified with a particular label that truly has that label, i.e. the proportion of all labeled positives that are true positives:

$$precision = \frac{true\ positives}{true\ positives + false\ positives} \tag{11.1}$$

Precision can be read from the rows of the confusion matrix. In this case, the NB model demonstrated a precision of 100% for the Urban hip segment (17/17), 96% for the travelers segment (26/27), 87.5% for the Suburban mix segment (21/24), and only 50% for the Moving up segment (9/18).

The second important concept is *recall* or *sensitivity*, also called the *true positive rate*. It is the proportion of all positives that were correctly identified:

$$recall = \frac{true\ positives}{true\ positives + false\ negatives} \tag{11.2}$$

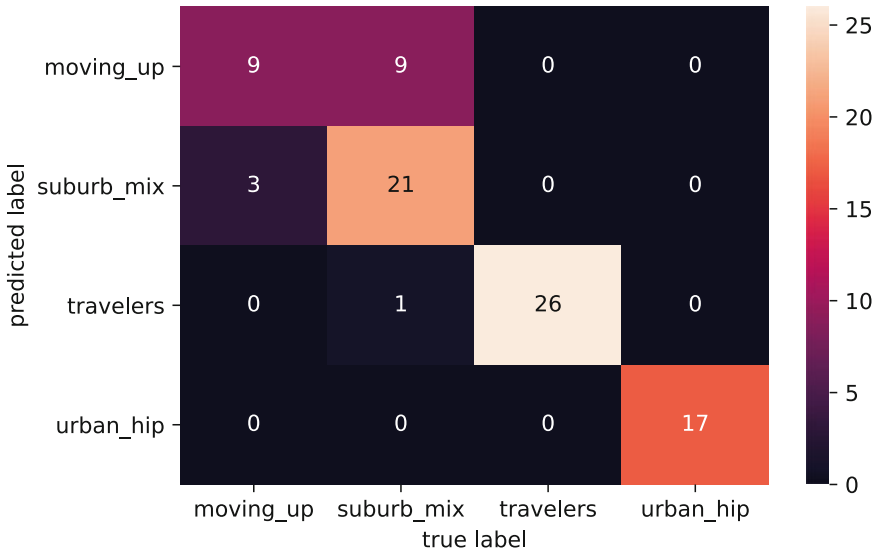

Fig. 11.1 A confusion matrix exposes class-specific performance of the model. The NB model performed well on identifying travelers and urban_hip, slightly less well on suburb_mix, and rather poorly on moving_up

Recall can be read from the columns of the confusion matrix. Here the NB model demonstrated a recall of 100% for the Urban hip (17/17) and Travelers (26/26) segments, but only 68% for the Suburban mix segment (21/31), and 75% for the Moving up segment (9/12).

If we return to our F_1 score from earlier, the formula for that is:

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{11.3}$$

We can write a function to calculate these values:

```
In [8]: def return_precision_recall(y_true, y_pred, model):
            conf_mat = metrics.confusion_matrix(y_true, y_pred)

            precision = pd.Series(metrics.precision_score(y_test,
                                                          y_pred,
                                                          average=None),
                                  index=model.classes_)
            recall = pd.Series(metrics.recall_score(y_test,
                                                    y_pred,
                                                    average=None),
                               index=model.classes_)
            f1 = pd.Series(2 * (precision * recall)/(precision + recall),
                           index=model.classes_)

            return pd.DataFrame([precision, recall, f1], index=['precision',
                                                                'recall', 'f1'])

In [9]: return_precision_recall(y_test, y_pred, nb)

Out[9]:            moving up  suburb_mix  travelers  urban_hip
        precision       0.50    0.875000   0.962963        1.0
        recall          0.75    0.677419   1.000000        1.0
        f1              0.60    0.763636   0.981132        1.0
```

There is likely to be a different business gain for identifying true positives and true negatives, versus the costs of false positives and false negatives. If you have estimates of these costs, you can use the confusion matrix to compute a custom metric for evaluating your classification results.

To better understand the model performance, we can visualize the decision boundaries in the PCA space, as show in Fig. 11.2:

```
In [10]: from sklearn import clone, decomposition

         def plot_decision_pca(model, X, y):
           width, height = 500, 500

           # Transform the X values using a PCA
           p = decomposition.PCA(random_state=132, svd_solver='full')
           X_transformed = p.fit_transform(X.iloc[:,:2])

           # Pull the first two dimensions
           x0 = X_transformed[:, 0]
           x1 = X_transformed[:, 1]

           # Get evenly spaced values between the min and max values
           x0_g = np.linspace(x0.min(), x0.max(), width)
           x1_g = np.linspace(x1.min(), x1.max(), height)

           # Create a "grid" of those evenly spaced values from each vector
```

```
    xx, yy = np.meshgrid(x0_g, x1_g)

    # Stack together all of the sampled values
    X_grid_transformed = np.vstack([xx.ravel(), yy.ravel()]).T

    # Do the inverse transform to get the non-PCA transformed values
    X_grid = p.inverse_transform(X_grid_transformed)

    # Fit a clone of the model using use inverse transformed columns
    # From the first two PCA dimensions.
    # Predict values on the sampled values
    model_c = clone(model)
    model_c.fit(p.inverse_transform(np.vstack([x0, x1]).T), y)
    X_grid_labels = model_c.predict(X_grid)

    # Create a class mapper to map from class string to an integer
    class_mapper = {class_:i for i,class_ in enumerate(model.classes_)}

    plt.figure(figsize=(6,6))
    # Plot the predicted values
    a = plt.scatter(x0, x1,
                    c=[class_mapper[label] for label in y],
                    cmap=plt.cm.rainbow, edgecolor='k', vmin=0, vmax=3)
    plt.contourf(xx, yy,
                 np.reshape([class_mapper[label]
                             for label in X_grid_labels],
                            (width, height)),
                 cmap=a.cmap, alpha=0.5, levels=3)
    cb = plt.colorbar(ticks=[0.5, 1.2, 2, 2.8])
    _ = cb.ax.set_yticklabels(model.classes_)
    plt.title('Decision boundaries with true values overlaid')
    plt.xlabel('First principal component')
    plt.ylabel('Second principal component')
```

```
In [11]: plot_decision_pca(nb, X_test, y_test)
```

The code to generate Fig. 11.2 is somewhat complex. Briefly, what it does is sample evenly in a grid from within the first two components of the PCA space, assessing the model prediction at each point. It then overlays the true values for the test set, and we can see where they disagree from the model prediction.

Looking at decision boundary plots can offer insight into the model's performance, or lack thereof. In this case, we can see that the Suburban mix and Moving up segments are interspersed, which is consistent with what we saw in the confusion matrix. The question of how to deal with that becomes a business decision. We might, for example, decide that we don't need to distinguish between these two segments and collapse them. Or we might try to collect another type of data for which these two segments do differ, perhaps education level or car ownership.

As we did for clustering, we check the predicted segments' summary values using the summary function we wrote in Chap. 10. However, because we now have labeled test data, we can also compare that to the summary values using the true membership:

```
In [12]: !pip install python_marketing_research
         from python_marketing_research_functions import chapter10
         chapter10.check_clusters(seg_sub, nb.predict(seg_sub))
```

```
[('moving up', 75), ('suburb_mix', 91), ('travelers', 84), ('urban_hip', 50)]
```

```
Out[12]:                    age        income  is_female      kids  own_home  \
         moving up    34.550570  49054.980474   0.760000  1.906667  0.400000
```

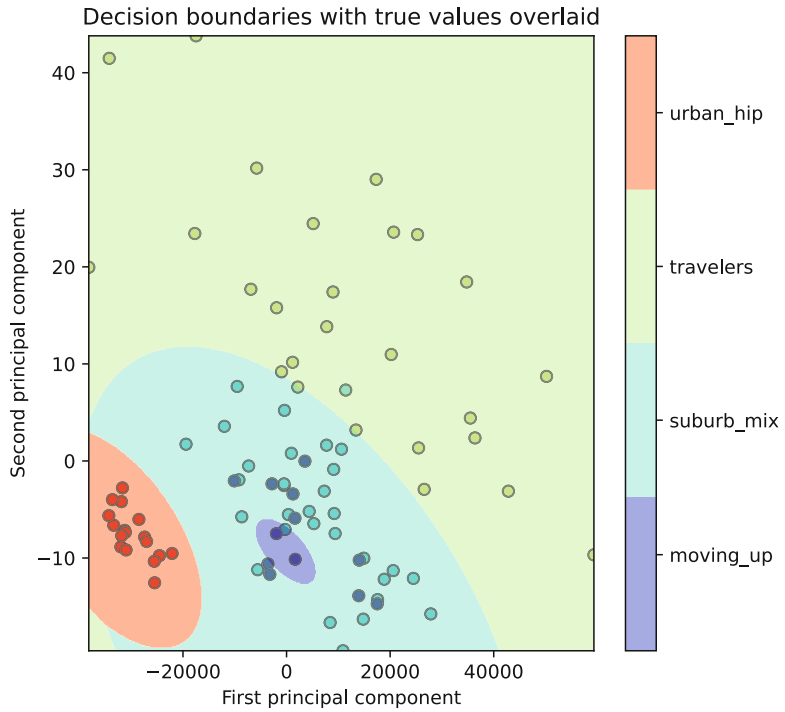

Fig. 11.2 Mapping the decision boundaries in PCA space exposes why the model discriminates poorly between the Suburban mix and Moving up segments: they are interspersed in the first two principal components

```
suburb_mix  40.251478  57644.538964   0.461538  1.978022  0.461538
travelers   57.489784  62650.866954   0.345238  0.023810  0.642857
urban_hip   23.873716  20267.737317   0.320000  1.140000  0.140000

            subscribe
moving up    0.213333
suburb_mix   0.054945
travelers    0.035714
urban_hip    0.220000
```

```
In [13]: chapter10.check_clusters(seg_sub, seg_labels)
```

```
[('moving up', 70), ('suburb_mix', 100), ('travelers', 80), ('urban_hip', 50)]
```

```
Out[13]:                  age        income  is_female      kids  own_home  \
Segment
moving up   36.216087  51763.552666      0.700  1.857143  0.357143
suburb_mix  39.284730  55552.282925      0.530  1.950000  0.480000
travelers   57.746500  62609.655328      0.325  0.000000  0.662500
urban_hip   23.873716  20267.737317      0.320  1.140000  0.140000

            subscribe
Segment
moving up    0.214286
suburb_mix   0.070000
travelers    0.025000
urban_hip    0.220000
```

The summary of demographics for the proposed segments (the first summary above) is very similar to the values in the true segments (the second summary). Thus, although NB assigned some observations to the wrong segments, its overall

model of the segment descriptive values—at least at the mean values—is similar for the proposed and true segments. By making such a comparison using the test data, we gain confidence that although assignment is not perfect on a case by case basis, the overall group definitions are quite similar.

For naive Bayes models, we can estimate not only the most likely segment but also the odds of membership in each segment, using the `predict_proba()` method:

```
In [14]: pd.DataFrame(nb.predict_proba(seg_sub),
                       columns=nb.classes_).sample(5).round(4)
```

Out[14]:

	moving_up	suburb_mix	travelers	urban_hip
26	0.0000	0.0065	0.9935	0.0
188	0.7116	0.2851	0.0033	0.0
263	0.0000	0.0000	0.0000	1.0
129	0.5240	0.4759	0.0001	0.0
192	0.5957	0.4043	0.0000	0.0

This tells us that Respondent 188 is estimated to be about 71% likely to be a member of Moving up, yet 29% likely to be in Suburban mix. Respondent 26 is estimated nearly 100% likely to be in Travelers. This kind of individual-level detail can suggest which individuals to target according to the difficulty of targeting and the degree of certainty. For high-cost campaigns, we might target only those most certain to be in a segment; whereas for low-cost campaigns, we might target people for second-best segment membership in addition to primary segment assignment. Because we are able to predict membership for new cases that have not been assigned, we can score new customers or others in a database, as long as we have the relevant predictor data used in the classification model.

We conclude that the naive Bayes model works well for the data analyzed here, with performance much better than chance, overall 85% accuracy in segment assignment, and demographics that are similar between the proposed and actual segments. It also provides interpretable individual-level estimation of membership likelihood.

Of course there are times when naive Bayes may not perform well, and it's always a good idea to try multiple methods. For an alternative, we next examine random forest models.

11.1.2 Random Forest Classification: `RandomForestClassifier()`

A random forest (RF) classifier does not attempt to fit a single model to data but instead builds an *ensemble* of models that jointly classify the data (Breiman 2001; Liaw and Wiener 2002). RF does this by fitting a large number of classification trees. In order to find an assortment of models, each tree is optimized to fit only *some* of the observations (in our case, customers) using only *some* of the predictors. The ensemble of all trees is the *forest*.

When a new case is predicted, it is predicted by every tree and the final decision is awarded to the *consensus* value that receives the most votes. In this way, a random forest avoids dependencies on precise model specification while remaining resilient in the face of difficult data conditions, such as data that are collinear or wide (more columns than rows). Random forest models perform well across a wide variety of datasets and problems (Fernández-Delgado et al. 2014).

In Python, a random forest may be created with code very similar to that for naive Bayes models. We use the same `X_train` training data as in Sect. 11.1.1, and call `RandomForestClassifier()` from the scikit-learn `ensemble` package to fit the classifier:

```
In [15]: from sklearn import ensemble

         np.random.seed(23432)
         rf = ensemble.RandomForestClassifier(n_estimators=50)

         rf.fit(X_train, y_train)
```

There are two things to note about the call to `RandomForestClassifier()`. First, random forests are random to some extent, as the name says. They select variables and subsets of data probabilistically. Thus, we use `set.seed()` before modeling, so that we get the same model if we re-run the code later. Second, we used the argument `n_estimators=50` to specify the number of trees to create in the forest.

Note that it's common to use many more trees for a RF model, as there is little risk of overfitting. Many people will start with 1000 trees, for example. In practice, this often does not improve the accuracy of the model relative to a model

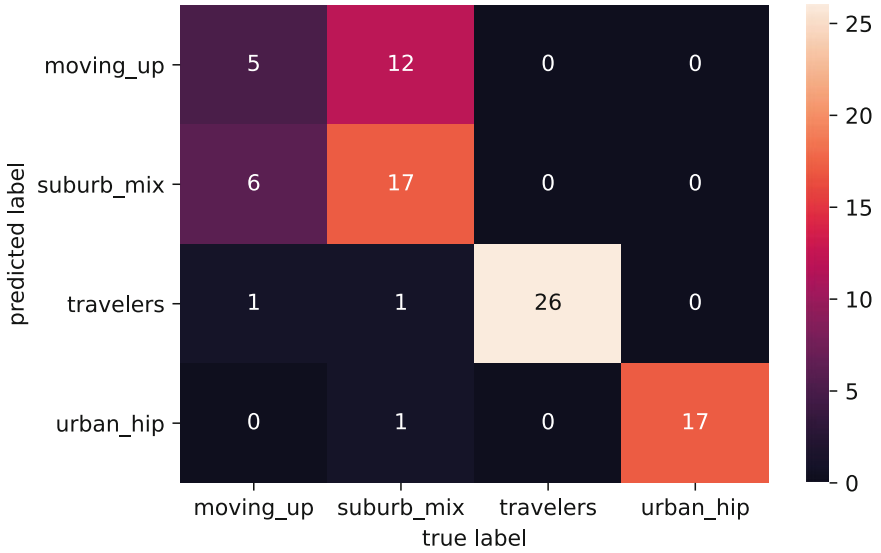

Fig. 11.3 Like the NB model, the RF model performed well on identifying travelers and urban_hip, but less well on suburb_mix, and moving_up

with fewer trees. However, it often improves the precision of the model, providing higher resolution values for the predicted class probabilities and variable importance (see Sect. 11.1.3). To determine the optimal number of trees, it's common to use hyperparameter tuning (see Sect. 11.2).

We can again check the F_1 score of the model:

```
In [16]: rf.score(X_test, y_test)

Out[16]: 0.7558139534883721

In [17]: y_pred = rf.predict(X_test)

         metrics.f1_score(y_test, y_pred, average='micro')

Out[17]: 0.7582299105153958
```

We see that the RF model performed a bit less well than the NB model, but is still a reasonably strong fit.

We can inspect the confusion matrix in Fig. 11.3 to better understand the class-level performance:

```
In [18]: confusion_matrix(y_test, y_pred, rf)
```

And look at class-specific precision and recall:

```
In [19]: return_precision_recall(y_test, y_pred, rf)

Out[19]:            moving up  suburb_mix  travelers  urban_hip
         precision   0.294118    0.739130   0.928571   0.944444
         recall      0.416667    0.548387   1.000000   1.000000
         f1          0.344828    0.629630   0.962963   0.971429
```

Overall, the RF model performed similarly to—albeit slightly worse than—the NB model. It showed poor discrimination of the Suburban mix and Moving up segments.

Inspecting the decision boundary visualization in Fig. 11.4 (code omitted) also gives us insight into the difference between the NB and RF models. Whereas the decision boundaries were smooth and convex for the NB model, the RF model gives jagged and discontinuous boundaries. This reflects the fact that the NB model boundaries are defined by gaussian estimates of feature distributions, whereas the RF model makes no assumptions about the underlying distribution of the features, but can find different association patterns in different parts of the space.

What does a random forest look like? Figure 11.5 shows one trees from among those we fit above. The complete forest comprises 50 such trees that differ in structure and the predictors used. When an observation is classified, it is assigned to the group that is predicted by the greatest number of trees within the ensemble. This ensemble of trees are what enable the "jagged" decision boundaries that are common in RF models. The tree in Fig. 11.5 is produced with this code:

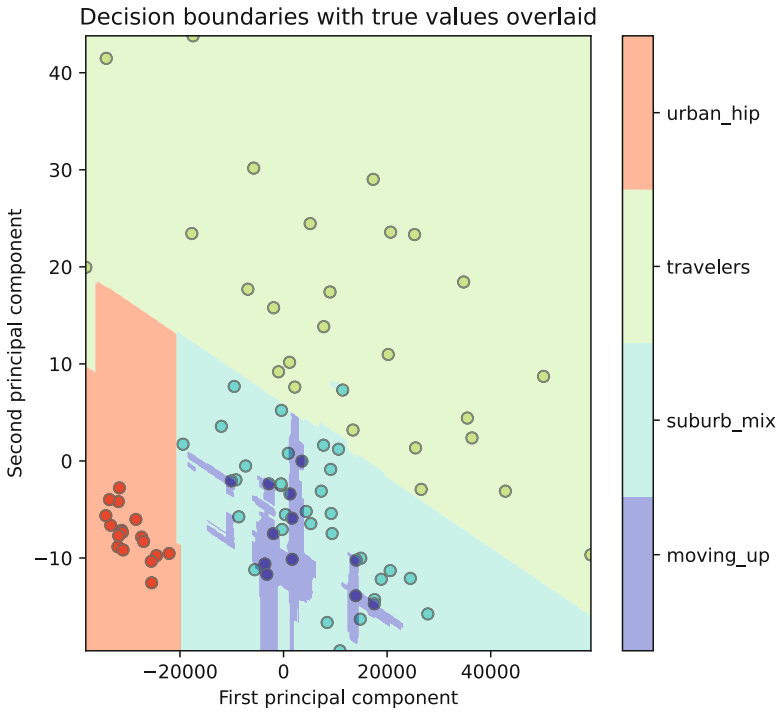

Fig. 11.4 The decision boundaries of the RF model are much more "jagged" and discontinuous than the NB model. This is because the RF model consists of decision trees that can learn disconnected boundaries, whereas the boundaries in the NB model are defined by smooth gaussians

```
In [20]: import graphviz
         from sklearn import tree
         from IPython.display import Image


         tree_0 = rf.estimators_[0]
         dot_data = tree.export_graphviz(tree_0, out_file=None,
                                         feature_names=X_train.columns,
                                         class_names=rf.classes_)
         tree_graph = graphviz.Source(dot_data, format='png')
         tree_graph.render('tmp', view=True)
         Image('tmp.png', width=1000, height=1000)
```

Inspecting the decision space in PCA space can help us understand areas of model weakness, but it does not offer much insight into what is driving model decisions based on actual features. We can also inspect the decision boundaries for pairs of features, such as age and income. We can write a similar function to do this:

```
In [21]: def pairwise_decision_boundary(model, X_train, y_train,
                                        X_test, y_test,
                                        first_column, second_column,
                                        jitter=False):
             width, height = 1000, 1000
             # Create a class mapper to map from class string to an integer
             class_mapper = {c:i for i,c in enumerate(model.classes_)}

             x0 = X_train[first_column]
             x1 = X_train[second_column]
             # Get evenly spaced values between the min and max values
             x0_g = np.linspace(x0.min(), x0.max(), width)
             x1_g = np.linspace(x1.min(), x1.max(), height)
```

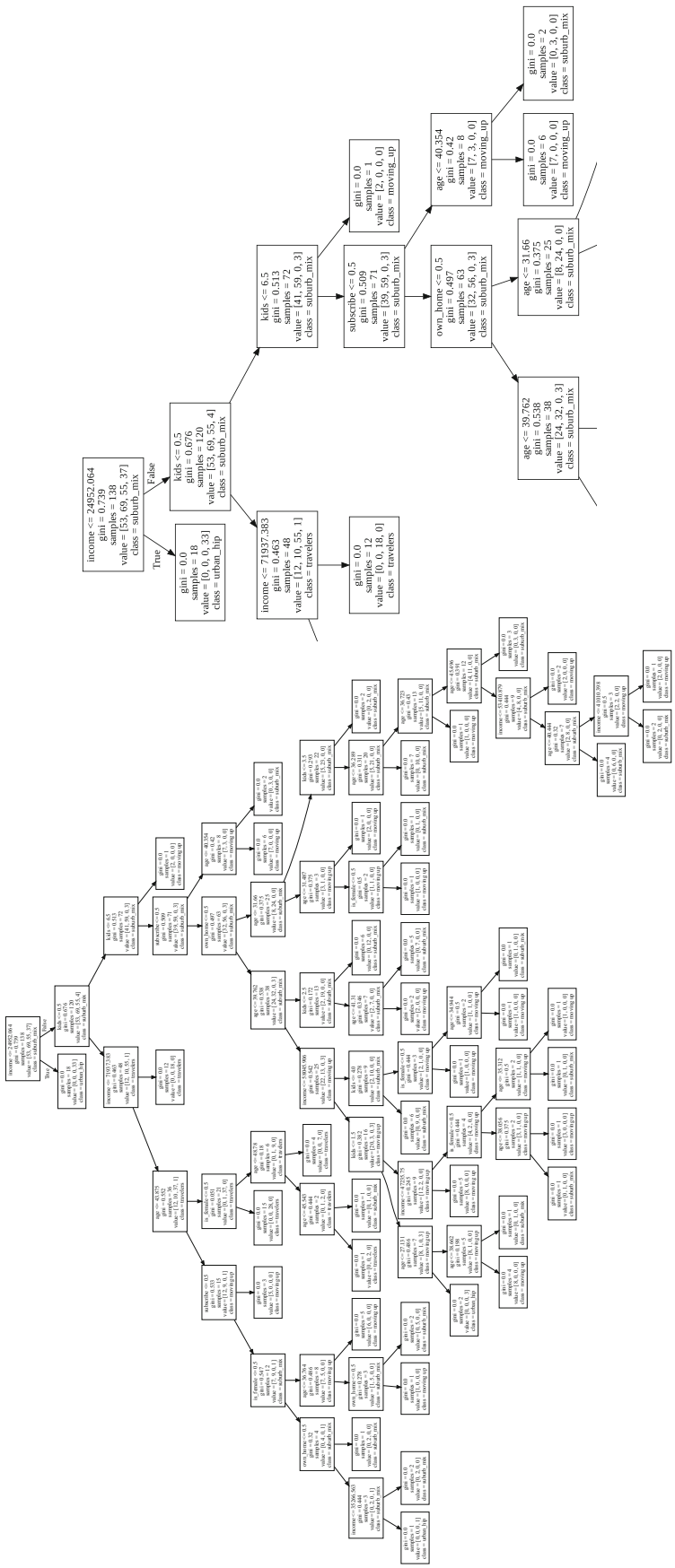

Fig. 11.5 One example among the 50 trees in the ensemble found by `RandomForestClassifier()` for segment prediction in `seg_sub`. The trees differ substantially in structure and variable usage. No single tree is expected to be a particularly good predictor in itself, yet the ensemble of all trees may predict well in aggregate by voting on the assignment of observations to outcome groups. The overall structure of the tree can be seen in the zoomed out view (upper panel) and the details of individual notes in a subset of the tree (lower panel)

```
    # Create a "grid" of those evenly spaced values from each vector
    xx, yy = np.meshgrid(x0_g, x1_g)
    # Stack together all of the sampled values
    X_grid = np.vstack([xx.ravel(), yy.ravel()]).T

    model_c = clone(model)
    model_c.fit(X_train.loc[:,[first_column, second_column]], y_train)
    X_grid_labels = model_c.predict(X_grid)
    # Plot the predicted values
    j_x0, j_x1 = 0, 0
    if jitter:
      j_x0 = (np.random.random(X_test.shape[0])-0.5)/10.
      j_x1 = (np.random.random(X_test.shape[0])-0.5)/10.
    a = plt.scatter(X_test[first_column] + j_x0,
                    X_test[second_column] + j_x1,
                    c=[class_mapper[l] for l in y_test],
                    cmap=plt.cm.rainbow,
                    edgecolor='k', vmin=0, vmax=3)
    plt.contourf(xx, yy,
                 np.reshape([class_mapper[l] for l in X_grid_labels],
                            (width, height)),
                cmap=a.cmap, alpha=0.5, levels=3)
    plt.title('Decision boundaries with true values overlaid')
    plt.xlabel(first_column)
    plt.ylabel(second_column)
    cb = plt.colorbar(ticks=[0.5, 1.2, 2, 2.8])
    cb.ax.set_yticklabels(model.classes_)
```

We can then look at the decision boundaries between age & income and between subscription status & number of children, shown in Fig. 11.6. We have an optional jitter argument for when we are looking at discrete or boolean values, such as subscription state (otherwise the points would all overlay each other).

```
In [22]: pairwise_decision_boundary(rf, X_train, y_train, X_test, y_test,
                                    'age', 'income')

In [23]: pairwise_decision_boundary(rf, X_train, y_train, X_test, y_test,
                                    'age', 'kids', jitter=True)
```

In Fig. 11.6, we see a few things of interest. From the left panel, we can see that the model has learned that individuals above about 50 years old are very likely to be Travelers and those below that age and below about $32,000 in income are likely to be from the Urban hip segment. Individuals who make more than Urban hip but are younger than Travelers belong to either Suburban mix or Moving up, without a really clear pattern, although Suburban mix tends to be a bit older or wealthier.

In the right panel of Fig. 11.6, we can see that in the model the number of kids only differentiates the Travelers segment.

It is possible to inspect the distribution of predictions for individual cases, again using the `predict_proba()` method:

```
In [24]: pd.DataFrame(rf.predict_proba(X_test), columns=rf.classes_).sample(5)
```

Out[24]:

	moving up	suburb_mix	travelers	urban_hip
76	0.00	0.00	0.00	1.0
19	0.02	0.06	0.92	0.0
39	0.24	0.76	0.00	0.0
73	0.00	0.00	0.00	1.0
56	0.10	0.76	0.14	0.0

These values reflect the number of trees the "voted" that a particular observation belonged in a particular class. As for the NB model, we can then understand the model's overall confidence in each assignment. For example, the model has assigned 100% probability that samples 73 and 76 are in the Urban hip segment. Samples 39 and 56 each have a 76% chance

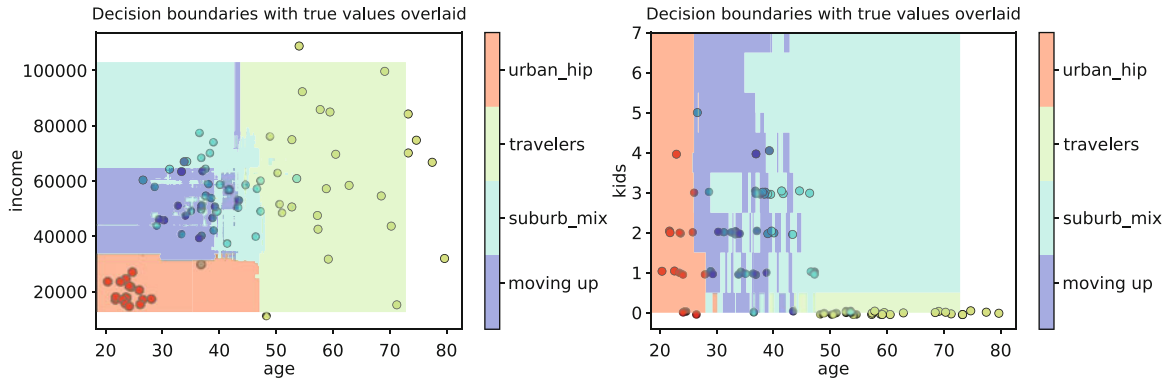

Fig. 11.6 Our `pairwise_decision_boundary()` function visualizes an estimation of the boundaries between classes that the model has learned. In the left panel, we see that the model associates high age with the Travelers group and low age and low income with the Urban hip group. The Suburb mix and Moving up groups are less well differentiated, but the Moving up tend to be either younger or have lower income than the Suburb mix group. In the right panel, we see that having few children and high age is associated with Travelers, but that the number of kids is not very predictive between the other groups

of being in the Suburban mix segment, but sample 39 is next most likely to be in Moving up, whereas for sample 56 its second-most-likely class is Traveler.

The proposed and actual segments are quite similar in the mean values of the variables in our summary function:

```
In [25]: chapter10.check_clusters(seg_sub, rf.predict(seg_sub))

[('moving up', 75), ('suburb_mix', 92), ('travelers', 82), ('urban_hip', 51)]
```

```
Out[25]:                    age        income  is_female      kids  own_home  \
         moving up    35.983633  51603.477437   0.640000  1.933333  0.386667
         suburb_mix   39.465911  56134.885424   0.554348  1.923913  0.467391
         travelers    57.522142  62472.064488   0.341463  0.000000  0.658537
         urban_hip    24.128490  20459.935615   0.333333  1.176471  0.137255

                     subscribe
         moving up    0.253333
         suburb_mix   0.032609
         travelers    0.024390
         urban_hip    0.215686
```

```
In [26]: chapter10.check_clusters(seg_sub, seg_labels)

[('moving up', 70), ('suburb_mix', 100), ('travelers', 80), ('urban_hip', 50)]
```

```
Out[26]:                    age        income  is_female      kids  own_home  \
         Segment
         moving up    36.216087  51763.552666       0.700  1.857143  0.357143
         suburb_mix   39.284730  55552.282925       0.530  1.950000  0.480000
         travelers    57.746500  62609.655328       0.325  0.000000  0.662500
         urban_hip    23.873716  20267.737317       0.320  1.140000  0.140000

                     subscribe
         Segment
         moving up    0.214286
         suburb_mix   0.070000
         travelers    0.025000
         urban_hip    0.220000
```

11.1.3 Random Forest Variable Importance

Random forest models are particularly good for one common marketing problem: estimating the importance of classification variables. Remember that RF fits many trees, where each tree is optimized for a portion of the data. It uses the remainder of the data—known as "out of bag" or OOB data—to assess the tree's performance more generally. Because each tree uses only a subset of variables, RF models are able to handle very *wide* data where there are more—even many, many more—predictor variables than there are observations.

An RF model assesses the importance of a variable in a simple yet powerful way: for one variable at a time, it randomly permutes (alters) the variable's values, computes the model accuracy in OOB data using the permuted values, and compares that to the accuracy with the real data. If the variable is important, then its performance will degrade when its observed values are randomly permuted. If, however, the model remains just as accurate as it is with real data, then the variable in question is not very important (Breiman 2001). As noted before, it is common to use more trees when the goal of the model is to determine variable importance, as the overall model will have more coverage of the variable space and variable importance values will have higher precision.

We can view the calculated importance of each feature in the `feature _importances_` parameter on the RF model:

```
In [27]: pd.Series(rf.feature_importances_,
                   index=seg_sub.columns).sort_values(ascending=False)

Out[27]: age           0.437028
         income        0.313560
         kids          0.150136
         is_female     0.035421
         own_home      0.032164
         subscribe     0.031691
         dtype: float64
```

Age and income are the most useful variables, which is consistent with the decision boundaries visualizations. Understanding variable importance can enable a deeper understanding of the features that define differences between classes, enabling more intelligent business decisions.

Variable or feature selection is often the first step for developing more advanced machine learning models, such as deep neural networks. Random forests unique properties make them invaluable tools in the process of feature engineering. In this case, we would expect to observe a similar degree of fit by removing the gender, home ownership, and subscription variables, which have minimal predictive power. Eliminating uninformative features is extremely important in other classifier models, many of which are very sensitive to the presence of noise from uninformative features.

11.2 Prediction: Identifying Potential Customers*

We now turn to another use for classification: to predict potential customers. An important business question—especially in high-churn categories such as mobile subscriptions—is how to reach new customers. If we have data on past prospects that includes potential predictors such as demographics, and an outcome such as purchase, we can develop a model to identify customers for whom the outcome is most likely among new prospects. In this section, we use a random forest model and attempt to predict subscription status from our dataset `seg_sub`.

As usual with classification problems, we split the data into a training sample and a test sample:

```
In [28]: subscribe_label = seg_sub.subscribe

         seg_sub_nosub = seg_sub.drop('subscribe', axis=1)

         np.random.seed(7885)
         rand_idx = np.random.rand(subscribe_label.shape[0])
         train_idx = rand_idx <= 0.65
         test_idx = rand_idx > 0.65

         X_train = seg_sub_nosub.iloc[train_idx]
         X_test = seg_sub_nosub.iloc[test_idx]
```

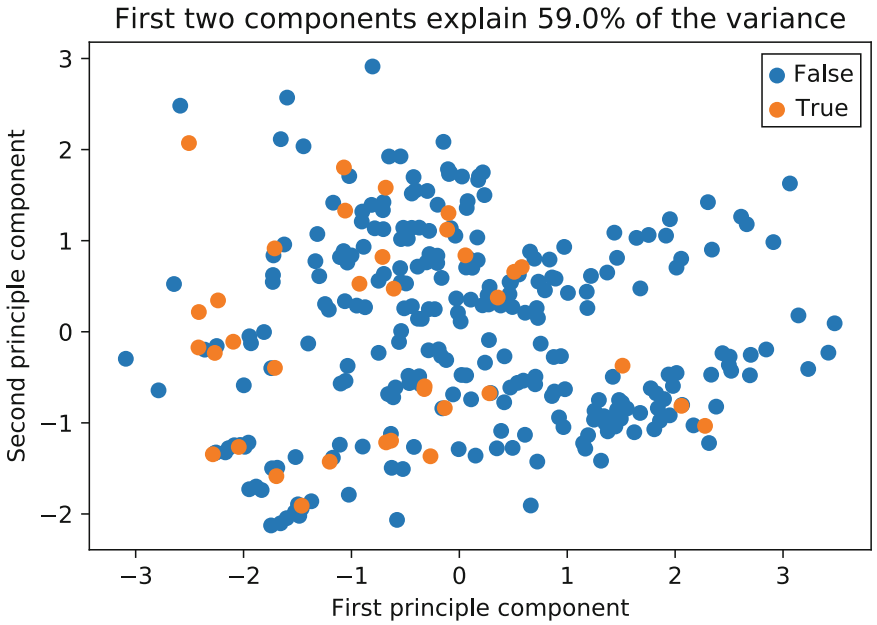

Fig. 11.7 Cluster plot for the subscribers and non-subscribers. The two groups show little differentiation on the principal components, which suggests that classifying respondents into the groups and predicting subscribers could be difficult

```
y_train = subscribe_label.iloc[train_idx]
y_test = subscribe_label.iloc[test_idx]
```

Next, we wonder how difficult it will be to identify potential subscribers. Are subscribers in the training set well-differentiated from non-subscribers? We use `cluster_plot()` from Chap. 10 to check the differentiation:

```
In [29]: chapter10.cluster_plot(seg_sub_nosub, subscribe_label)
```

The result in Fig. 11.7 shows that the subscribers and non-subscribers are not well differentiated when plotted against principal components (which reflect about 59% of the variance in the data). This suggests that the problem will be difficult!

We fit an initial RF model to predict `subscribe`:

```
In [30]: rf_sub = ensemble.RandomForestClassifier(n_estimators=100,
                                                  random_state=86,
                                                  class_weight=\
                                                    'balanced_subsample')

         rf_sub.fit(X_train, y_train)

         y_pred = rf_sub.predict(X_test)

In [31]: rf_sub.score(X_test, y_test)

Out[31]: 0.9072164948453608
```

An accuracy of 90% looks good. But lets check the confusion matrix:

```
In [32]: confusion_matrix(y_test, y_pred, rf_sub)
```

The results in Fig. 11.8 are not encouraging. Although the error rate might initially sound good at 90.3% overall, we have a recall of only 10%.

This is expected given the interspersion of the classes in the cluster plot. But it is also exacerbated by the *class imbalance* problem in machine learning. When one category dominates the data, it is very difficult to learn to predict other groups. This frequently arises with small-proportion problems, such as predicting the comparatively rare individuals who will purchase a product, who have a medical condition, who are security threats, and so forth.

Fig. 11.8 The random forest model did not perform well a predicting the subscription state, achieving a recall of only 10% (1/10)

A general solution is to balance the classes by sampling more from the small group. In RF models, this can be accomplished by telling the classifier to use a balanced group when it samples data to fit each tree, which we did with the `class_weight='balanced_subsample'`. However it was not sufficient to overcome the poor predictive power of the variables themselves.

We used default values for all the model parameters. An important concept in machine learning is *hyperparameter tuning*, where we explore the model parameter space to identify the parameters that lead to the best fit. We can perform a *grid search*, where we sample many combinations of parameters to find an optimum.

This requires a scoring function. The F_1 score is useful because it balances precision and recall. We can run the grid search easily:

```
In [33]: from sklearn import model_selection

         rf_sub_cv = ensemble.RandomForestClassifier(random_state=34,
                                                     class_weight=\
                                                       'balanced_subsample')
         parameters = {'n_estimators': [10, 100, 500],
                       'max_depth': [5, 10, 30],
                       'min_samples_split': [2,5],
                       'min_samples_leaf': [1,2,5]}
         clf = model_selection.GridSearchCV(rf_sub_cv, parameters,
                                            cv=5, scoring='f1_weighted')
         clf.fit(X_train, y_train)
```

We can inspect the best scoring parameters using the `best_params_` parameter:

```
In [34]: clf.best_params_

Out[34]: {'max_depth': 10,
          'min_samples_leaf': 1,
          'min_samples_split': 2,
          'n_estimators': 100}
```

Looking at the confusion matrix in Fig. 11.9, we see that this model did not perform any better:

Fig. 11.9 The hyperparameter tuned model did not perform any better

```
In [35]: y_pred_be = clf.best_estimator_.predict(X_test)

         confusion_matrix(y_test, y_pred_be, clf.best_estimator_)
```

What if we try a different scoring function? Let's imagine we want to optimize for recall, that is we want to find as many potential positives as possible, accepting that many will be false positives. We can use recall as the scoring function:

```
In [36]: rf_sub_cv = ensemble.RandomForestClassifier(random_state=34,
                                                    class_weight=\
                                                      'balanced_subsample')
         parameters = {'n_estimators': [10, 100, 500],
                       'max_depth': [5, 10, 30],
                       'min_samples_split': [2,5],
                       'min_samples_leaf': [1,2,5]}
         clf = model_selection.GridSearchCV(rf_sub_cv, parameters,
                                            cv=5, scoring='recall')
         clf.fit(X_train, y_train)

In [37]: clf.best_params_

Out[37]: {'max_depth': 5,
          'min_samples_leaf': 2,
          'min_samples_split': 2,
          'n_estimators': 10}

In [38]: y_pred_be = clf.best_estimator_.predict(X_test)

         confusion_matrix(y_test, y_pred_be, clf.best_estimator_)
```

The confusion matrix in Fig. 11.10 is still not great, but it is at least different! Our precision went from 100% (1/1) to 15% (2/13), but our recall went from 10% (1/10) to 20% (2/10).

Another knob we can turn is the sample weighting. We used a balanced subsample weighting, which accounts for some of the imbalance in the frequency of each class, but we can further oversample the rarer class with the `class_weight` parameter. This will push the model toward recall. One risk with this approach is overfitting: the model becomes so tuned to oversampled rare class that it is not generalizable.

Fig. 11.10 Optimizing the hyperparameter tuning for recall improved the recall at the expense of precision

```
In [39]:rf_sub = ensemble.RandomForestClassifier(n_estimators=10,
                                                 random_state=86,
                                                 max_depth=5,
                                                 min_samples_leaf=2,
                                                 min_samples_split=2,
                                                 class_weight=\
                                                   {False: 1, True:50})

         rf_sub.fit(X_train, y_train)

        y_pred = rf_sub.predict(X_test)
        confusion_matrix(y_test, y_pred, rf_sub)
```

We can see in Fig. 11.11 that this had a large effect. We have boosted our recall to 70% (7/10), but at a substantial cost to precision: only 17.5% (7/40) of the observations that the model labeled as `True` are true positives; 33 were false positives.

Despite the fact that the variables here have poor predictive power, we can clearly see that optimizing our hyperparameter tuning for a different score led to a differently tuned model with different tradeoffs. This is an important concept in machine learning. An analyst should consider the business outcome carefully and tune a model appropriately. This often differs from a generic or abstract "accuracy" metric.

It's also worth nothing that machine learning is not magic, it is statistics. If there is no strong signal in the data, you cannot build a good prediction. What should one do in that circumstance? The most obvious option would be to find other features that might be more predictive.

11.3 Learning More*

We covered the basics of classification in this chapter. Many of the resources we recommended in Sect. 10.4 for clustering are also relevant for understanding classification. A recommended introduction to the field of statistical learning is James et al., *An Introduction to Statistical Learning* (ISL) (James et al. 2013). A more advanced treatment of the topics in ISL is Hastie et al., *The Elements of Statistical Learning* (Hastie et al. 2016).

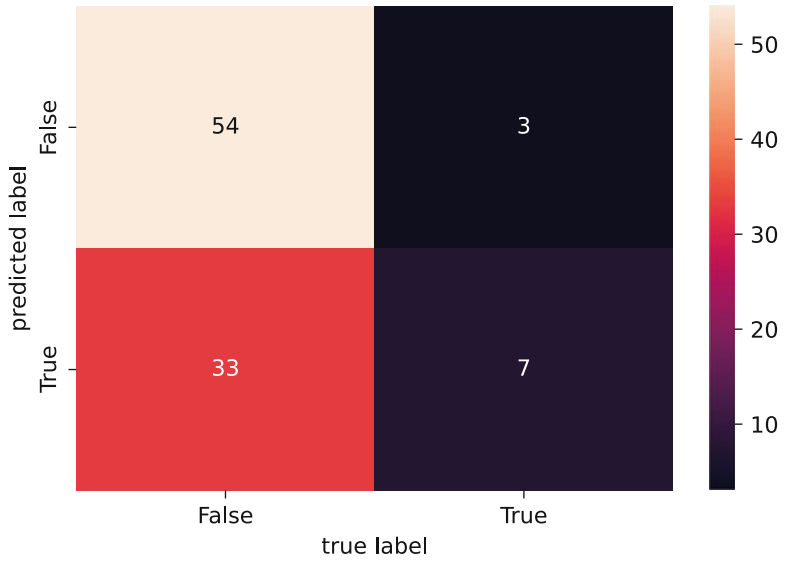

Fig. 11.11 Changing the sample weighting is another means to adjust precision/recall. Here by oversampling the `True` class, we improved the recall at the expense of precision

For classification and especially prediction, in addition to ISL noted above, an applied, practitioner-friendly text is Kuhn and Johnson's *Applied Predictive Modeling*. Scikit-learn has many supervised learning modules (scikit-learn developers 2019b), many of which include classification methods. Overall these packages represent many of the most common methods in machine learning.

11.4 Key Points

In this chapter, we examined the basic structure of classification methods that may be used to predict group membership for new observations. In addition, we saw how classifier visualization techniques may be used to understand model performance. Following are key points to keep in mind when working on classification and prediction models.

- With classification models, data should be split into training and test groups, and models validated on the test (holdout) data (Sect. 11.1).
- We examined naive Bayes models (`sklearn.naive_bayes.GaussianNB()`, Sect. 11.1.1) and random forest models (`sklearn.ensemble.RandomForest Classifier()`, Sect. 11.1.2). These—and many other classification methods—have quite similar syntax, making it easy to try and compare models.
- Precision, or the proportion of all predicted positives that are truly positive, and recall, the proportion of all true positives that were correctly labeled as positive are important concepts in tuning classifiers, are at odds with each other: improving one tends to worsen the other.
- We learned how to explore decision boundary visualizations to better understand the limitations of the model and of the underlying variables.
- A useful feature of random forest models is their ability to determine variable importance for prediction, even when there are a large number of variables (Sect. 11.1.3).
- A common problem in classification is class imbalance, where one group dominates the observations and makes it difficult to predict the other group. We saw how to correct this for random forest models with the `class_weight` argument, resulting in a more successful predictive model (Sect. 11.2).
- We saw how to run a grid search using `sklearn.model_selection. GridSearchCV()` for hyperparameter tuning of the model, enabling us to find an optimal model under different scoring metrics. We also saw that hyperparameter tuning cannot overcome uninformative variables!