5_Tune the area under the curve

October 13, 2021

1 Exercise: Tune the area under the curve

In this exercise, we will make and compare two models, using ROC curves, and tune one using the area under the curve (AUC).

The goal of our models is to identify whether each item detected on the mountain is a hiker (`true`) or a tree (`false`). We will work with our `motion` feature here. Let's take a look:

[ ]:
```
import numpy
import pandas
import graphings # custom graphing code. See our GitHub repo for details
import sklearn.model_selection

# Load our data from disk
df = pandas.read_csv("hiker_or_tree.csv", delimiter="\\t")

# Remove features we no longer want
del df["height"]
del df["texture"]

# Split into train and test
train, test =  sklearn.model_selection.train_test_split(df, test_size=0.5,␣
 ↪random_state=1)

# Graph our feature
graphings.multiple_histogram(test, label_x="motion", label_group="is_hiker",␣
 ↪nbins=12)
```

```
C:\Users\aduzo\Anaconda3\lib\site-packages\ipykernel_launcher.py:7:
ParserWarning: Falling back to the 'python' engine because the 'c' engine does
not support regex separators (separators > 1 char and different from '\s+' are
interpreted as regex); you can avoid this warning by specifying engine='python'.
  import sys
```

Motion seems associated with hikers more than trees, but not perfectly. Presumably this is because trees blow in the wind, and some hikers are found sitting down.

1.1 A logistic regression model and a random forest

Let's train the same logistic regression model we used in the previous exercise, as well as a random forest model. Both will try to predict which objects are hikers.

First the logistic regression:

[ ]:
```
import statsmodels.api
from sklearn.metrics import accuracy_score

# This is a helper method that reformats the data to be compatible
# with this particular logistic regression model
prep_data = lambda x:  numpy.column_stack((numpy.full(x.shape, 1), x))

# Train a logistic regression model to predict hiker based on motion
lr_model = statsmodels.api.Logit(train.is_hiker, prep_data(train.motion),␣
 ↪add_constant=True).fit()

# Assess its performance
# -- Train
predictions = lr_model.predict(prep_data(train.motion)) > 0.5
train_accuracy = accuracy_score(train.is_hiker, predictions)

# -- Test
predictions = lr_model.predict(prep_data(test.motion)) > 0.5
test_accuracy = accuracy_score(test.is_hiker, predictions)

print("Train accuracy", train_accuracy)
print("Test accuracy", test_accuracy)

# Plot the model
predict_with_logistic_regression = lambda x: lr_model.predict(prep_data(x))
graphings.scatter_2D(test, label_x="motion", label_y="is_hiker",␣
 ↪title="Logistic Regression", trendline=predict_with_logistic_regression)
```

```
Optimization terminated successfully.
         Current function value: 0.260202
         Iterations 8
Train accuracy 0.916
Test accuracy 0.888
```

Now our random forest model:

[ ]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

# Create a random forest model with 50 trees
random_forest = RandomForestClassifier(random_state=2,
                                       verbose=False)
```

```
# Train the model
random_forest.fit(train[["motion"]], train.is_hiker)

# Assess its performance
# -- Train
predictions = random_forest.predict(train[["motion"]])
train_accuracy = accuracy_score(train.is_hiker, predictions)

# -- Test
predictions = random_forest.predict(test[["motion"]])
test_accuracy = accuracy_score(test.is_hiker, predictions)


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

```
Random Forest Performance:
Train accuracy 1.0
Test accuracy 0.852
```

These models have similar, but not identical, performance on the test set in terms of accuracy.

1.2 Create ROC plots

Let's create ROC curves for these models. To do this, we will simply import code from the last exercises so that we can focus on what we would like to learn here. If you need a refresher on how these were made, re-read the last exercise.

Note that we've made a slight change. Now our method produces both a graph, and the table of numbers we used to create the graph.

First let's look at the logistic regression model:

[ ]:
```
from m2d_make_roc import create_roc_curve # import our previous ROC code

fig, thresholds_lr = create_roc_curve(predict_with_logistic_regression, test,␣
 ↪"motion")

# Uncomment the line below if you would like to see the area under the curve
#fig.update_traces(fill="tozeroy")

fig.show()

# Show the table of results
thresholds_lr
```

```
[ ]:      threshold       fpr       tpr
     0    -0.000001  1.000000  1.000000
     1     0.000000  1.000000  1.000000
     2     0.010101  1.000000  1.000000
     3     0.020202  1.000000  1.000000
     4     0.030303  0.895161  0.968254
     ..         …         …         …
     99    0.969697  0.000000  0.563492
     100   0.979798  0.000000  0.539683
     101   0.989899  0.000000  0.464286
     102   1.000000  0.000000  0.000000
     103   1.000100  0.000000  0.000000

     [104 rows x 3 columns]
```

We can see our model does better than chance (it is not a diagonal line). Our table shows the false positive rate (*fpr*) and true positive rate (*tpr*) for each threshold.

Let's repeat this for our random forest model:

```
[ ]: # Don't worry about this lambda function. It simply reorganizes
     # the data into the shape expected by the random forest model,
     # and calls predict_proba, which gives us predicted probabilities
     # that the label is 'hiker'
     predict_with_random_forest = lambda x: random_forest.predict_proba(numpy.
      ↪array(x).reshape(-1, 1))[:,1]

     # Create the ROC curve
     fig, thresholds_rf = create_roc_curve(predict_with_random_forest, test,␣
      ↪"motion")

     # Uncomment the line below if you would like to see the area under the curve
     #fig.update_traces(fill="tozeroy")

     fig.show()

     # Show the table of results
     thresholds_lr
```

```
[ ]:      threshold       fpr       tpr
     0    -0.000001  1.000000  1.000000
     1     0.000000  1.000000  1.000000
     2     0.010101  1.000000  1.000000
     3     0.020202  1.000000  1.000000
     4     0.030303  0.895161  0.968254
     ..         …         …         …
     99    0.969697  0.000000  0.563492
     100   0.979798  0.000000  0.539683
```

```
101    0.989899  0.000000  0.464286
102    1.000000  0.000000  0.000000
103    1.000100  0.000000  0.000000

[104 rows x 3 columns]
```

1.3 Area under the curve

Our models look quite similar. Which model do we think is best? Let's use *area under the curve* (AUC) to compare them. We should expect a number larger than 0.5, because these models are both better than chance, but smaller than 1, because they are not perfect.

[ ]:
```
from sklearn.metrics import roc_auc_score

# Logistic regression
print("Logistic Regression AUC:", roc_auc_score(test.is_hiker,␣
 ↪predict_with_logistic_regression(test.motion)))

# Random Forest
print("Random Forest AUC:", roc_auc_score(test.is_hiker,␣
 ↪predict_with_random_forest(test.motion)))
```

```
Logistic Regression AUC: 0.936907962109575
Random Forest AUC: 0.9306275601638505
```

By a very thin margin, the logistic regression model comes out on top.

Remember, this doesn't mean the logistic regression model will always do a better job than the random forest. It means that the logistic regression model is a slightly better choice for this kind of data, and probably is marginally less reliant on having the perfect decision thresholds chosen.

1.4 Decision Threshold Tuning

We can also use our ROC information to find the best thresholds to use. We'll just work with our random forest model for this part.

First, let's take a look at the rate of True and False positives with the default threshold of 0.5:

[ ]:
```
# Print out its expected performance at the default threshold of 0.5
# We previously obtained this information when we created our graphs
row_of_0point5 = thresholds_rf[thresholds_rf.threshold == 0.5]
print("TPR at threshold of 0.5:", row_of_0point5.tpr.values[0])
print("FPR at threshold of 0.5:", row_of_0point5.fpr.values[0])
```

```
TPR at threshold of 0.5: 0.8611111111111112
FPR at threshold of 0.5: 0.15725806451612903
```

We can expect that, when real hikers are seen, we have an 86% chance of identifying them. When trees or are seen, we have a 16% chance of identifying them as a hiker.

Let's say that for our particular situation, we consider obtaining true positive just as important as avoiding a false positive. We don't want to ignore hikers on the mountain, but we also don't want to send our team out into dangerous conditions for no reason.

We can find the best threshold by making our own scoring system, and seeing which threshold would get the best result:

[ ]:
```
thresholds_rf.tpr - thresholds_rf.fpr
```

[ ]:
```
0      0.000000
1      0.541283
2      0.589798
3      0.610023
4      0.626152
         …
99     0.769457
100    0.769457
101    0.761521
102    0.000000
103    0.000000
Length: 104, dtype: float64
```

[ ]:
```
# Calculate how good each threshold is from our TPR and FPR.
# Our criteria is that the TPR is as high as possible and
# the FPR is as low as possible. We consider them equally important
scores = thresholds_rf.tpr - thresholds_rf.fpr
numpy.argmax(scores)
```

[ ]:
```
76
```

[ ]:
```
# Find the entry with the highest score according to our criteria
index_of_best_score = numpy.argmax(scores)
```

[ ]:
```
best_threshold = thresholds_rf.threshold[index_of_best_score]
print("Best threshold:", best_threshold)

# Print out its expected performance
print("TPR at this threshold:", thresholds_rf.tpr[index_of_best_score])
print("FPR at this threshold:", thresholds_rf.fpr[index_of_best_score])
```

```
Best threshold: 0.7373737373737375
TPR at this threshold: 0.8333333333333334
FPR at this threshold: 0.036290322580645164
```

Our best threshold, with this criteria, is 0.74, not 0.5! This would have us still identify 83% of hikers properly - a slight decrease from 86% - but only mis-identify 3.6% of trees as hikers.

If you would like, play with how we are calculating our scores here, and see how the threshold is adjusted.

1.5 Summary

That's it! Here we've created ROC curves for two different models, using code we wrote in the previous exercise.

Visually, they were quite similar, and when we compared them using the area-under-the-curve metric we found that the logistic regression model was marginally better performing.

We then used the ROC curve to tune our random forest model, based on criteria specific to our circumstances. Our very simple criteria of `TPR - FPR` let us pick a threshold that was right for us.