7_Hyperparameter_tuning_with_random_forests

October 6, 2021

1 Exercise: Random forests and hyperparameters

The goal of this unit is to explore how hyperparameters change training, and thus model performance. The line between model architecture and hyperparameters is a bit blurry for random forests because training itself actually changes the architecture of the model by adding or removing branches.

We will again persue our goal of predicting which crimes in San Francisco will be resolved.

1.1 Data and Training Preparation

Let's load our data, split it, and prepare for training. This is the same code you've seen in the previous exercises. If you've not done those - go back and do them now!

[ ]:
```
# This code is exactly the same as what we have done in the previous exercises.␣
↪You do not need to read it again.

import pandas
from sklearn.model_selection import train_test_split
from sklearn.metrics import balanced_accuracy_score
import graphings # custom graphing code. See our GitHub repo for details

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

# One-hot encode features
dataset = pandas.get_dummies(dataset, columns=["Category", "PdDistrict"],␣
↪drop_first=False)

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

# Make a utility method that we can re-use throughout this exercise
# To easily fit and test out model
def fit_and_test_model(model):
    '''
    Trains a model and tests it against both train and test sets
    '''
    global features
    # Train the model
    model.fit(train[features], train.Resolution)

    # Assess its performance
    # -- Train
    predictions = model.predict(train[features])
    train_accuracy = balanced_accuracy_score(train.Resolution, predictions)

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

    return train_accuracy, test_accuracy


print("Ready!")
dataset.head()
```

```
Ready!
```

[ ]:

```
   DayOfWeek  Resolution           X          Y  day_of_year  time_in_hours  \
0          5        True -122.403405  37.775421           29      11.000000
1          5        True -122.403405  37.775421           29      11.000000
2          1        True -122.388856  37.729981          116      14.983333
3          2       False -122.412971  37.785788            5      23.833333
4          5       False -122.419672  37.765050            1       0.500000

   Category_ARSON  Category_ASSAULT  Category_BAD CHECKS  Category_BRIBERY  \
0               0                 0                    0                 0
1               0                 0                    0                 0
2               0                 0                    0                 0
3               0                 0                    0                 0
4               0                 0                    0                 0

   …  PdDistrict_BAYVIEW  PdDistrict_CENTRAL  PdDistrict_INGLESIDE  \
0  …                   0                   0                     0
1  …                   0                   0                     0
2  …                   1                   0                     0
3  …                   0                   0                     0
4  …                   0                   0                     0

   PdDistrict_MISSION  PdDistrict_NORTHERN  PdDistrict_PARK  \
0                   0                    0                0
1                   0                    0                0
2                   0                    0                0
3                   0                    0                0
```

```
4                    1                    0                 0

   PdDistrict_RICHMOND  PdDistrict_SOUTHERN  PdDistrict_TARAVAL  \
0                    0                    1                   0
1                    0                    1                   0
2                    0                    0                   0
3                    0                    0                   0
4                    0                    0                   0

   PdDistrict_TENDERLOIN
0                      0
1                      0
2                      0
3                      1
4                      0

[5 rows x 54 columns]
```

Let's not forget to split our data!

```
[ ]: # Split the dataset in an 90/10 train/test ratio.
     train, test = train_test_split(dataset, test_size=0.1, random_state=2,␣
      ↪shuffle=True)
```

```
[ ]: train.shape
```

```
[ ]: (135387, 54)
```

1.2 Criteria to split on

The first hyperparameter we will work with is the criterion. This is essentially a kind of cost function that is used to determine whether a node should be split or not. We have two options available in the package that we are using: `gini` and `entropy`. Let's try them both:

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

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

```
[ ]: (1000, 54)
```

1.2.1 Gini

[ ]:
```
# Prepare the model
rf = RandomForestClassifier(n_estimators=10,
                            # max_depth=12,
                            # max_features=cur_max_features,
                            random_state=2,
                            criterion="gini",
                            verbose=False)
# Train and test the result
train_accuracy, test_accuracy = fit_and_test_model(rf)
# Train and test the result
print(train_accuracy, test_accuracy)
```

```
0.9719390319921176 0.6842617724903477
```

1.2.2 Entropy

[ ]:
```
# Prepare the model
rf = RandomForestClassifier(n_estimators=10,
                            random_state=2,
                            criterion="entropy",
                            verbose=False)
# Train and test the result
train_accuracy, test_accuracy = fit_and_test_model(rf)
# Train and test the result
print(train_accuracy, test_accuracy)

# Roll back the train dataset to the full train set
train = full_trainset
```

```
0.9701142144738695 0.6912451859737102
```

Results are subtly different, and usually only subtly as both criterion use a similar way to assess performance. We suggest you try different sample sizes, such as 50 and 50000, to see how this changes with larger or smaller samples.

1.3 Minimum impurity decrease

The minimum impurity decrease is another criterion that is used to assess whether a node should be split. It is used by the `gini` or `entropy` algorithms we used, above. If minimu impurity decrease is high, then splitting a node must result in substantial performance improvement. If it is very low, then nodes can be split even if they offer very little to no performance improvements on the training dataset.

[ ]:
```
import numpy as np

np.linspace(0, 0.0005, num=100)
```

```
[ ]: array([0.00000000e+00, 5.05050505e-06, 1.01010101e-05, 1.51515152e-05,
            2.02020202e-05, 2.52525253e-05, 3.03030303e-05, 3.53535354e-05,
            4.04040404e-05, 4.54545455e-05, 5.05050505e-05, 5.55555556e-05,
            6.06060606e-05, 6.56565657e-05, 7.07070707e-05, 7.57575758e-05,
            8.08080808e-05, 8.58585859e-05, 9.09090909e-05, 9.59595960e-05,
            1.01010101e-04, 1.06060606e-04, 1.11111111e-04, 1.16161616e-04,
            1.21212121e-04, 1.26262626e-04, 1.31313131e-04, 1.36363636e-04,
            1.41414141e-04, 1.46464646e-04, 1.51515152e-04, 1.56565657e-04,
            1.61616162e-04, 1.66666667e-04, 1.71717172e-04, 1.76767677e-04,
            1.81818182e-04, 1.86868687e-04, 1.91919192e-04, 1.96969697e-04,
            2.02020202e-04, 2.07070707e-04, 2.12121212e-04, 2.17171717e-04,
            2.22222222e-04, 2.27272727e-04, 2.32323232e-04, 2.37373737e-04,
            2.42424242e-04, 2.47474747e-04, 2.52525253e-04, 2.57575758e-04,
            2.62626263e-04, 2.67676768e-04, 2.72727273e-04, 2.77777778e-04,
            2.82828283e-04, 2.87878788e-04, 2.92929293e-04, 2.97979798e-04,
            3.03030303e-04, 3.08080808e-04, 3.13131313e-04, 3.18181818e-04,
            3.23232323e-04, 3.28282828e-04, 3.33333333e-04, 3.38383838e-04,
            3.43434343e-04, 3.48484848e-04, 3.53535354e-04, 3.58585859e-04,
            3.63636364e-04, 3.68686869e-04, 3.73737374e-04, 3.78787879e-04,
            3.83838384e-04, 3.88888889e-04, 3.93939394e-04, 3.98989899e-04,
            4.04040404e-04, 4.09090909e-04, 4.14141414e-04, 4.19191919e-04,
            4.24242424e-04, 4.29292929e-04, 4.34343434e-04, 4.39393939e-04,
            4.44444444e-04, 4.49494949e-04, 4.54545455e-04, 4.59595960e-04,
            4.64646465e-04, 4.69696970e-04, 4.74747475e-04, 4.79797980e-04,
            4.84848485e-04, 4.89898990e-04, 4.94949495e-04, 5.00000000e-04])
```

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

     min_impurity_decreases = np.linspace(0, 0.0005, num=100)

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

     print("Working...")
     for min_impurity_decrease in min_impurity_decreases:

         # Prepare the model
         rf = RandomForestClassifier(n_estimators=10,
                                     min_impurity_decrease=min_impurity_decrease,
                                     random_state=2,
                                     verbose=False)

         # Train and test the result
```

```
    train_accuracy, test_accuracy = fit_and_test_model(rf)

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


# Plot results
graphings.line_2D(dict(Train=train_accuracies, Test=test_accuracies),
                    min_impurity_decreases,
                    label_x="Minimum impurity decreases␣
 ↪(min_impurity_decrease)",
                    label_y="Accuracy",
                    title="Performance", show=True)

# Roll back the train dataset to the full train set
train = full_trainset
```

```
Working…
```

Notice that *train* performance drastically reduces as we get more scrict about when a node can be split. This is because the higher the minimum impurity decrease, the more strict we are about growing our tree. The smaller the tree, the less overfitting we will see.

Changes in *test* performance are more subtle. A small increase above zero appears to increase test performance. Further increases begin to hurt test performance only subtly.

This is similar to what we saw in the previous exercise about model size - more complex models (those with more nodes) can fit the training data better, but once they exceed a certain complexity, they begin to overfit.

1.4 Maximum number of features

When trees are created, they are provided with a subset of the data. This not only means they see a certain collection of rows (samples), but also a certain collection of columns (features). The more features are provided, the more likely a given tree is going to overfit. Let's see what happens when we restrict the maximum number of features that can be provided to each tree in the forest:

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

max_features = range(10, len(features) +1)

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

```
print("Working...")
for cur_max_features in max_features:
    # Prepare the model
    rf = RandomForestClassifier(n_estimators=50,
                                max_depth=12,
                                max_features=cur_max_features,
                                random_state=2,
                                verbose=False)

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

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


# Plot results
graphings.line_2D(dict(Train=train_accuracies, Test=test_accuracies),
                    max_features,
                    label_x="Maximum number of features (max_features)",
                    label_y="Accuracy",
                    title="Performance", show=True)

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

```
Working…
```

1.5 Seeding

Finally, we come to seeding. When trees are initially made, there is a degree of randomness that is used to decide which features and samples will be provided to which trees. Changing the random state (seed) value changes this initial state.

The random seed is not a parameter to be tuned, but its effects on our models shouldn't be forgotten, particularly when there isn't much data to work with. Let's see how our model behaves with different random states.

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


seeds = range(0,101)
```

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

for seed in seeds:
    # Prepare the model
    rf = RandomForestClassifier(n_estimators=10,
                                random_state=seed,
                                verbose=False)

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

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


# Plot results
graphings.line_2D(dict(Train=train_accuracies, Test=test_accuracies),
                    seeds,
                    label_x="Minimum impurity decreases␣
 ↪(min_impurity_decrease)",
                    label_y="Accuracy",
                    title="Performance", show=True)

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

Performance, particularly on the test set, is variable and thus some part of performance is blind luck. This is not only because the initial state of the model can be different, but also that we split our training and test data differently. Whether this would apply to a holdout set is not easy to tell without trying it

There's no correlation between high or low seed values and performance: seed is not something to 'tune'. The seed is a random factor and it can help or hinder depending on the model at play. Generally speaking, when we work with small amounts of data, we are more likely to be affected by different seed values. More complex models can also be affected more by the seed, but not always.

Try changing the sample size and/or number of trees in the model above and watch how the variability in performance changes. Think about why this might be.

1.6 Summary

Complex models typically have associated hyperparameters that can affect training. The extent to which these matter, and how they affect the result, depends on the data at hand and complexity of the model being used. We usually need to experiment somewhat with these in order to achieve optimum performance for the data that we have.