Select and Train Model

Friday, January 17, 2020 2:29 PM

Note Written by: Zulfadli Zainal

Recap!

1. We defined our problems
2. We explore the data
3. We sampled training set + test set
4. We wrote transformation pipeline to cleanup and prepare data
5. Make sure all data is numerical

Now, lets train the model.

Training and Evaluation of Training Set

This stage is much more simple than previous step

First, train **Linear Regression Model**

```
from sklearn.linear_model import LinearRegression
```

```
lin_reg = LinearRegression()
lin_reg.fit(housing_prepared, housing_labels)
```

from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(housing_prepared, housing_labels)

Done!!

Now we can test our model.

```
################################Test your model################################

some_data = housing_prepared.iloc[:5]
some_labels = housing_labels.iloc[:5]

print('\n\nPredictions:', lin_reg.predict(some_data))
print('\nLabels:', list(some_labels))
```

some_data = housing_prepared.iloc[:5]
some_labels = housing_labels.iloc[:5]
print('\n\nPredictions:', lin_reg.predict(some_data))
print('\nLabels:', list(some_labels))

Result:

Predictions: [181313.23430337 286451.78145112 263328.07605752 140991.31402388 177337.39815452]
Labels: [103000.0, 382100.0, 172600.0, 93400.0, 96500.0]

Seems like the predictions is really bad. Lets calculate the error of this predictions.

```
##############################Measure Error#####################################

housing_predictions = lin_reg.predict(housing_prepared)
lin_mse = mean_squared_error(housing_labels, housing_predictions)
lin_rmse = np.sqrt(lin_mse)
print('\nMSE:\t', lin_mse)
print('\nRMSE:\t', lin_rmse)
```

housing_predictions = lin_reg.predict(housing_prepared)
lin_mse = mean_squared_error(housing_labels, housing_predictions)
lin_rmse = np.sqrt(lin_mse)
print('\nMSE:\t', lin_mse)
print('\nRMSE:\t', lin_rmse)

Result:
RMSE: 69362.3413523808

The RMSE is around $69,362..

Most districts median_housing_values range between $120,000 and $265,000, so a typical prediction error of $68,628 is not very satisfying. This is an example of a model underfitting the training data.

Why this problem happens?

Usually, because the features not providing enough information to predict.

or that the model is not powerful enough.

How to solve the problem?

1. Select more powerful model
2. Feed algorithm with better features
3. Reduce the constrains on the model (The model is not regularized so this is not an options)

Using Other Models to Train

Lets try to train the model using **Decision Tree Model**.

```
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error
```

```
##############################Decision Tree Regressor##############################

tree_reg = DecisionTreeRegressor()
tree_reg.fit(housing_prepared, housing_labels)

################################Measure Error#####################################

housing_predictions = tree_reg.predict(housing_prepared)
tree_mse = mean_squared_error(housing_labels, housing_predictions)
tree_rmse = np.sqrt(tree_mse)
print('\nMSE:\t', tree_mse)
print('\nRMSE:\t', tree_rmse)
```

from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error

##############################Decision Tree Regressor##############################

tree_reg = DecisionTreeRegressor()
tree_reg.fit(housing_prepared, housing_labels)

################################Measure Error#####################################

housing_predictions = tree_reg.predict(housing_prepared)
tree_mse = mean_squared_error(housing_labels, housing_predictions)
tree_rmse = np.sqrt(tree_mse)
print('\nMSE:\t', tree_mse)
print('\nRMSE:\t', tree_rmse)

Result:

MSE: 0.0 - Almost no error
RMSE: 0.0 - Almost no error

Wait, what!? No error at all? Could this model really be absolutely perfect? Of course, it is much more likely that the model has badly overfit the data. How can you be sure? As we saw earlier, you don't want to touch the test set until you are ready to launch a model you are confident about, so you need to use part of the training set for training, and part for model validation.

CROSS VALIDATION!

One way to evaluate the model -> split the training set to smaller training set -> then train the model again.
It's a troublesome work but there is a feature in scikit learn -> Cross Validation

Below code perfroms *K-Fold Cross Validation* ->

1. Randomly split training set into 10 subsets (called folds)
2. Train & evaluate the model 10 times
3. Picking a different fold for evaluation every time and training on the other 9 folds. The result is an array containing the 10 evaluation scores.

```
from sklearn.model_selection import cross_val_score
```

```
############################Cross Validation#################################

scores = cross_val_score(tree_reg, housing_prepared,
                         housing_labels, scoring="neg_mean_squared_error", cv=10)
tree_rmse_scores = np.sqrt(-scores)


def display_scores(scores):
    print('Scores: ', scores)
    print('Mean: ', scores.mean())
    print('Standard Deviation: ', scores.std())


display_scores(tree_rmse_scores)
```

from sklearn.model_selection import cross_val_score

############################Cross Validation#################################

scores = cross_val_score(tree_reg, housing_prepared,
housing_labels, scoring="neg_mean_squared_error", cv=10)
tree_rmse_scores = np.sqrt(-scores)

def display_scores(scores):
print('Scores: ', scores)
print('Mean: ', scores.mean())
print('Standard Deviation: ', scores.std())

display_scores(tree_rmse_scores)

Result:

Scores: [70191.64362507 71486.44357537 67194.47770702 69194.20002733 70626.18235611 67841.54196656 68604.35415927 71871.14214659 69418.91513128 70417.81304033]
Mean: 69684.67137349278
Standard Deviation: 1441.681737390377

Now the Decision Tree doesn't look as good as it did earlier. In fact, it seems to perform worse than the Linear Regression model!

Compare result with other model -> In this case, previous linear regression.

```
from sklearn.model_selection import cross_val_score
```

```
############################Cross Validation#################################

scores = cross_val_score(lin_reg, housing_prepared,
                         housing_labels, scoring="neg_mean_squared_error", cv=10)
lin_rmse_scores = np.sqrt(-scores)


def display_scores(scores):
    print('Scores: ', scores)
    print('Mean: ', scores.mean())
    print('Standard Deviation: ', scores.std())


display_scores(lin_rmse_scores)
```

from sklearn.model_selection import cross_val_score

##############################Cross Validation##

scores = cross_val_score(lin_reg, housing_prepared,
housing_labels, scoring="neg_mean_squared_error", cv=10)
lin_rmse_scores = np.sqrt(-scores)

def display_scores(scores):
print('Scores: ', scores)
print('Mean: ', scores.mean())
print('Standard Deviation: ', scores.std())

display_scores(lin_rmse_scores)

Result:

Scores: [66170.13482881 72783.99723185 68950.52987763 67640.0509114 70438.49542911 66523.65704676
66541.25492139 70992.53769382 74292.46725992 70675.67306462]
Mean: 69500.8798265314
Standard Deviation: 2659.043850886444

That’s right: the Decision Tree model is overfitting so badly that it performs worse than the Linear Regression model.

Using Other Models to Train

Try evaluate using **Random Forest**.
Random Forest -> Training many decision trees on random subsets of the features & average out the predictions.
Building model on top of other model is called Ensembled Learning.

Compare result with other model -> In this case, random forest.

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import cross_val_score
```

```
##############################Random Forest Regressor##############################

forest_reg = RandomForestRegressor()
forest_reg.fit(housing_prepared, housing_labels)

################################Measure Error#####################################

housing_predictions = forest_reg.predict(housing_prepared)
forest_mse = mean_squared_error(housing_labels, housing_predictions)
forest_rmse = np.sqrt(forest_mse)
print('\nMSE:\t', forest_mse)
print('\nRMSE:\t', forest_rmse)

##############################Cross Validation#####################################

scores = cross_val_score(forest_reg, housing_prepared,
                         housing_labels, scoring="neg_mean_squared_error", cv=10)
forest_rmse_scores = np.sqrt(-scores)


def display_scores(scores):
    print('Scores: ', scores)
    print('Mean: ', scores.mean())
    print('Standard Deviation: ', scores.std())


display_scores(forest_rmse_scores)
```

from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import cross_val_score

##############################Random Forest Regressor##############################

forest_reg = RandomForestRegressor()
forest_reg.fit(housing_prepared, housing_labels)

################################Measure Error#####################################

housing_predictions = forest_reg.predict(housing_prepared)
forest_mse = mean_squared_error(housing_labels, housing_predictions)
forest_rmse = np.sqrt(forest_mse)
print('\nMSE:\t', forest_mse)

```
print('\nRMSE:\t', forest_rmse)

################################Cross Validation########################################

scores = cross_val_score(forest_reg, housing_prepared,
                         housing_labels, scoring="neg_mean_squared_error", cv=10)
forest_rmse_scores = np.sqrt(-scores)


def display_scores(scores):
   print('Scores: ', scores)
   print('Mean: ', scores.mean())
   print('Standard Deviation: ', scores.std())


display_scores(forest_rmse_scores)
```

Result:

MSE: 340146682.047111
RMSE: 18443.065961144068

Scores: [47261.57755683 52299.23614666 48710.35142559 50445.00872354 51527.16220838 47253.78261928 46143.81247876 51693.25338437 50155.87512425 49908.86895665]
Mean: 49539.89286243007
Standard Deviation: 2004.333061679457

Wow, this is much better: Random Forests look very promising. However, note that he score on the training set is still much lower than on the validation sets, meaning that the model is still overfitting the training set.

How to solve overfitting problem?

1. Simplify the model
2. Constrain it (Regularize it)
3. or get more training data

Anyway, before going for optimizing the model, try few more models (Eg SVM with different kernel or Neural Network) without changing too much hyperparameters. The goal is to shortlist a few promising models!

Save trained model for References

You should save every model you experiment with, so you can come back easily to any model you want. Make sure you save both the hyperparameters and the trained parameters, as well as the cross-validation scores and perhaps the actual predictions as well. This will allow you to easily compare scores across model types, and compare the types of errors they make. You can easily save Scikit-Learn models by using Python's `pickle` module, or using `sklearn.externals.joblib`, which is more efficient at serializing large NumPy arrays:

```
from sklearn.externals import joblib

joblib.dump(my_model, "my_model.pkl")
# and later...
my_model_loaded = joblib.load("my_model.pkl")
```