100 XP

Exercise - Experiment with more powerful regression models

10 minutes

Sandbox activated! Time remaining: 1 hr 49 min

You have used 2 of 10 sandboxes for today. More sandboxes will be available tomorrow.

learn-notebooks-6e1b5901-cd8d-48dd-988f-115... Py38_default

Regression - Experimenting with additional models

In the previous notebook, we used simple regression models to look at the relationship between features of a bike rentals dataset. In this notebook, we'll experiment with more complex models to improve our regression performance.

Let's start by loading the bicycle sharing data as a **Pandas** DataFrame and viewing the first few rows. We'll also split our data into training and test datasets.

```
# Import modules we'll need for this notebook
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.model_selection import train_test_split
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# load the training dataset
!wget https://raw.githubusercontent.com/MicrosoftDocs/mslearn-introduction-to
bike_data = pd.read_csv('daily-bike-share.csv')
bike_data['day'] = pd.DatetimeIndex(bike_data['dteday']).day
numeric_features = ['temp', 'atemp', 'hum', 'windspeed']
categorical_features = ['season','mnth','holiday','weekday','workingday','wea
bike_data[numeric_features + ['rentals']].describe()
print(bike_data.head())


# Separate features and labels
```

```
# After separating the dataset, we now have numpy arrays named **X** containi
X, y = bike_data[['season','mnth', 'holiday','weekday','workingday','weathers

# Split data 70%-30% into training set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30, ran

print ('Training Set: %d rows\nTest Set: %d rows' % (X_train.shape[0], X_test
```

[1] ✓ 9 sec

```
--2022-03-12 23:41:58--
https://raw.githubusercontent.com/MicrosoftDocs/mslearn-introduction-to-
machine-learning/main/Data/ml-basics/daily-bike-share.csv
Resolving raw.githubusercontent.com... 185.199.110.133, 185.199.109.133,
185.199.108.133, ...
Connecting to raw.githubusercontent.com|185.199.110.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 48800 (48K) [text/plain]
Saving to: ‘daily-bike-share.csv.1’

daily-bike-share.cs 100%[===================>]  47.66K  --.-KB/s    in
0.006s
```

Now we have the following four datasets:

- **X_train**: The feature values we'll use to train the model
- **y_train**: The corresponding labels we'll use to train the model
- **X_test**: The feature values we'll use to validate the model
- **y_test**: The corresponding labels we'll use to validate the model

Now we're ready to train a model by fitting a suitable regression algorithm to the training data.

Experiment with Algorithms

The linear regression algorithm we used last time to train the model has some predictive capability, but there are many kinds of regression algorithm we could try, including:

- **Linear algorithms**: Not just the Linear Regression algorithm we used above (which is technically an *Ordinary Least Squares* algorithm), but other variants such as *Lasso* and *Ridge*.
- **Tree-based algorithms**: Algorithms that build a decision tree to reach a prediction.
- **Ensemble algorithms**: Algorithms that combine the outputs of multiple base algorithms to improve generalizability.

> **Note**: For a full list of Scikit-Learn estimators that encapsulate algorithms for supervised machine learning, see the Scikit-Learn documentation . There are many algorithms to choose from, but for most real-world scenarios, the Scikit-Learn estimator cheat sheet can help you find a suitable starting point.

Try Another Linear Algorithm

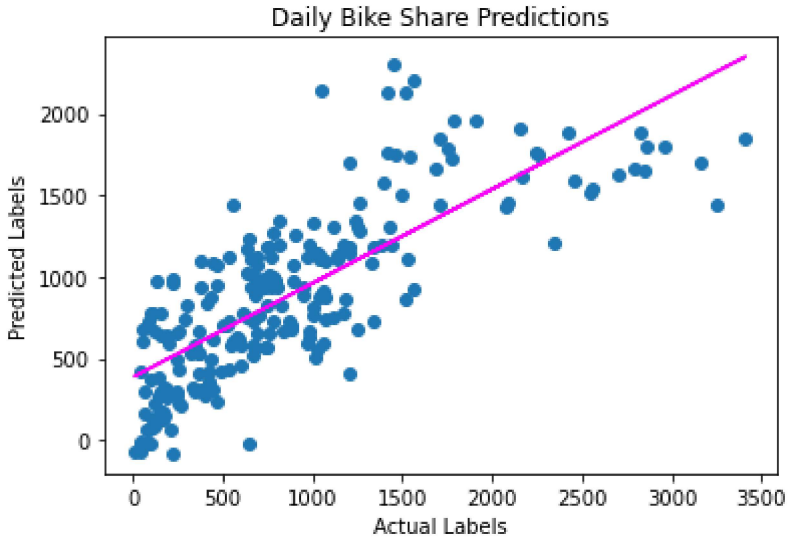
Let's try training our regression model by using a **Lasso** algorithm. We can do this by just changing the estimator in the training code.

[2]

```
from sklearn.linear_model import Lasso

# Fit a lasso model on the training set
model = Lasso().fit(X_train, y_train)
print (model, "\n")

# Evaluate the model using the test data
predictions = model.predict(X_test)
mse = mean_squared_error(y_test, predictions)
print("MSE:", mse)
rmse = np.sqrt(mse)
print("RMSE:", rmse)
r2 = r2_score(y_test, predictions)
print("R2:", r2)

# Plot predicted vs actual
plt.scatter(y_test, predictions)
plt.xlabel('Actual Labels')
plt.ylabel('Predicted Labels')
plt.title('Daily Bike Share Predictions')
# overlay the regression line
z = np.polyfit(y_test, predictions, 1)
p = np.poly1d(z)
plt.plot(y_test,p(y_test), color='magenta')
plt.show()
```

✓ <1 sec

```
Lasso()


MSE: 201155.70593338402

RMSE: 448.5038527519959

R2: 0.605646863782449
```



Try a Decision Tree Algorithm

As an alternative to a linear model, there's a category of algorithms for machine learning that uses a tree-based approach in which the features in the dataset are examined in a series of evaluations, each of which results in a *branch* in a *decision tree* based on the feature value. At the end of each series of branches are leaf-nodes with the predicted label value based on the feature values.

It's easiest to see how this works with an example. Let's train a Decision Tree regression model using the bike rental data. After training the model, the code below will print the model definition and a text representation of the tree it uses to predict label values.

[3]

```
from sklearn.tree import DecisionTreeRegressor
from sklearn.tree import export_text

# Train the model
model = DecisionTreeRegressor().fit(X_train, y_train)
print (model, "\n")

# Visualize the model tree
tree = export_text(model)
print(tree)
```

✓ <1 sec

```
DecisionTreeRegressor()


|--- feature_6 <= 0.45
|   |--- feature_4 <= 0.50
|   |   |--- feature_7 <= 0.32
|   |   |   |--- feature_8 <= 0.41
|   |   |   |   |--- feature_1 <= 2.50
|   |   |   |   |   |--- feature_6 <= 0.29
```

```
|   |   |   |   |   |   |--- feature_1 <= 1.50
|   |   |   |   |   |   |   |--- value: [558.00]
|   |   |   |   |   |   |--- feature_1 >  1.50
|   |   |   |   |   |   |   |--- value: [515.00]
```

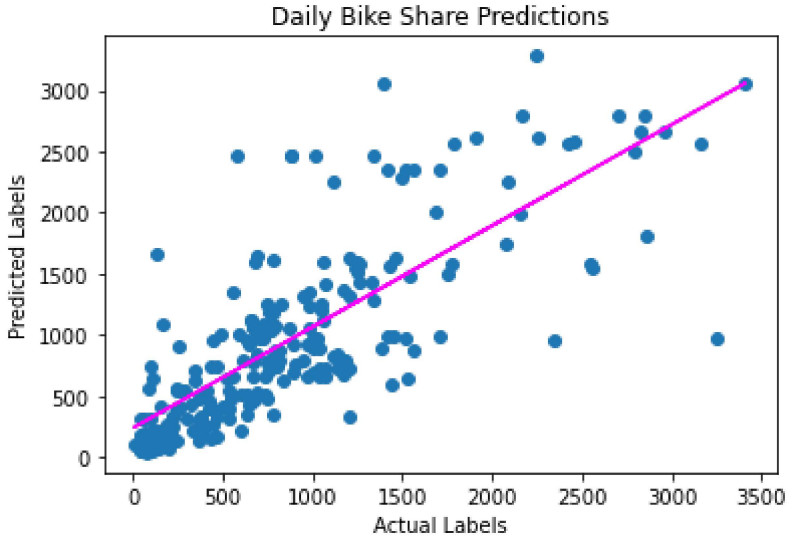
So now we have a tree-based model; but is it any good? Let's evaluate it with the test data.

[4]

```
# Evaluate the model using the test data
predictions = model.predict(X_test)
mse = mean_squared_error(y_test, predictions)
print("MSE:", mse)
rmse = np.sqrt(mse)
print("RMSE:", rmse)
r2 = r2_score(y_test, predictions)
print("R2:", r2)

# Plot predicted vs actual
plt.scatter(y_test, predictions)
plt.xlabel('Actual Labels')
plt.ylabel('Predicted Labels')
plt.title('Daily Bike Share Predictions')
# overlay the regression line
z = np.polyfit(y_test, predictions, 1)
p = np.poly1d(z)
plt.plot(y_test,p(y_test), color='magenta')
plt.show()
```

✓ 1 sec

```
MSE: 245763.30454545454
RMSE: 495.7452012329061
R2: 0.5181964664388643
```



The tree-based model doesn't seem to have improved over the linear model, so what else could we try?

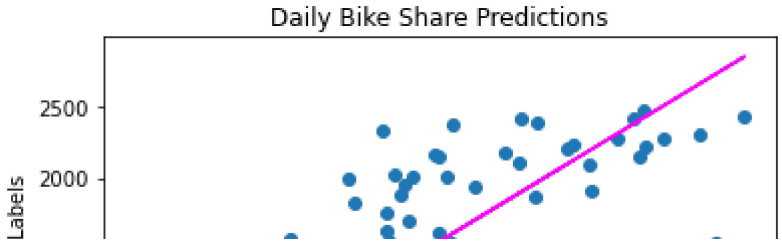
Try an Ensemble Algorithm

Ensemble algorithms work by combining multiple base estimators to produce an optimal model, either by applying an aggregate function to a collection of base models (sometimes referred to a *bagging*) or by building a sequence of models that build on one another to improve predictive performance (referred to as *boosting*).

For example, let's try a Random Forest model, which applies an averaging function to multiple Decision Tree models for a better overall model.

[5]
```
from sklearn.ensemble import RandomForestRegressor

# Train the model
model = RandomForestRegressor().fit(X_train, y_train)
print (model, "\n")

# Evaluate the model using the test data
predictions = model.predict(X_test)
mse = mean_squared_error(y_test, predictions)
print("MSE:", mse)
rmse = np.sqrt(mse)
print("RMSE:", rmse)
r2 = r2_score(y_test, predictions)
print("R2:", r2)

# Plot predicted vs actual
plt.scatter(y_test, predictions)
plt.xlabel('Actual Labels')
plt.ylabel('Predicted Labels')
plt.title('Daily Bike Share Predictions')
# overlay the regression line
z = np.polyfit(y_test, predictions, 1)
p = np.poly1d(z)
plt.plot(y_test,p(y_test), color='magenta')
plt.show()
```
✓ 2 sec

```
RandomForestRegressor()


MSE: 108958.5508340909

RMSE: 330.0887014638503

R2: 0.7863935997253152
```



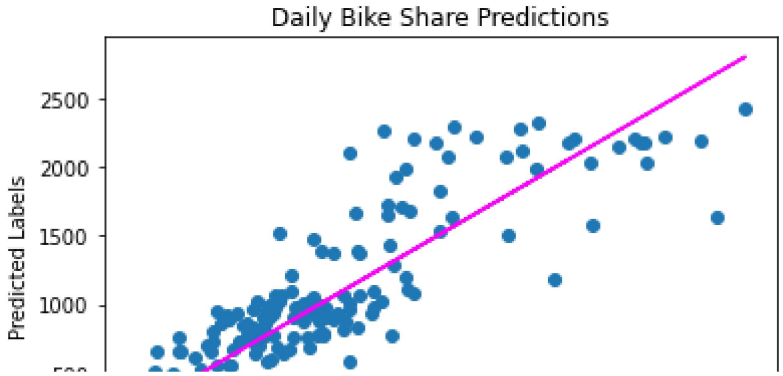
For good measure, let's also try a *boosting* ensemble algorithm. We'll use a Gradient Boosting estimator, which like a Random Forest algorithm builds multiple trees, but instead of building them all independently and taking the average result, each tree is built on the outputs of the previous one in an attempt to incrementally reduce the *loss* (error) in the model.

[6]

```
# Train the model
from sklearn.ensemble import GradientBoostingRegressor

# Fit a lasso model on the training set
model = GradientBoostingRegressor().fit(X_train, y_train)
print (model, "\n")

# Evaluate the model using the test data
predictions = model.predict(X_test)
mse = mean_squared_error(y_test, predictions)
print("MSE:", mse)
rmse = np.sqrt(mse)
print("RMSE:", rmse)
r2 = r2_score(y_test, predictions)
print("R2:", r2)

# Plot predicted vs actual
plt.scatter(y_test, predictions)
plt.xlabel('Actual Labels')
plt.ylabel('Predicted Labels')
plt.title('Daily Bike Share Predictions')
# overlay the regression line
z = np.polyfit(y_test, predictions, 1)
p = np.poly1d(z)
plt.plot(y_test,p(y_test), color='magenta')
plt.show()
```

✓ 2 sec

```
GradientBoostingRegressor()


MSE: 104541.4083974435

RMSE: 323.3286383812042

R2: 0.7950531302364129
```



Summary

Here we've tried a number of new regression algorithms to improve performance. In our notebook we'll look at 'tuning' these algorithms to improve performance.

Further Reading

To learn more about Scikit-Learn, see the Scikit-Learn documentation .

+ Code + Markdown

[ ] Press shift + enter to run

Next unit: Improve models with hyperparameters

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆