

Regression

November 23, 2020

1 Regression

Supervised machine learning techniques involve training a model to operate on a set of *features* and predict a *label* using a dataset that includes some already-known label values. The training process *fits* the features to the known labels to define a general function that can be applied to new features for which the labels are unknown, and predict them. You can think of this function like this, in which y represents the label we want to predict and x represents the features the model uses to predict it.

$$y = f(x)$$

In most cases, x is actually a *vector* that consists of multiple feature values, so to be a little more precise, the function could be expressed like this:

$$y = f([x_1, x_2, x_3, \dots])$$

The goal of training the model is to find a function that performs some kind of calculation to the x values that produces the result y . We do this by applying a machine learning *algorithm* that tries to fit the x values to a calculation that produces y reasonably accurately for all of the cases in the training dataset.

There are lots of machine learning algorithms for supervised learning, and they can be broadly divided into two types:

- **Regression algorithms:** Algorithms that predict a y value that is a numeric value, such as the price of a house or the number of sales transactions.
- **Classification algorithms:** Algorithms that predict to which category, or *class*, an observation belongs. The y value in a classification model is a vector of probability values between 0 and 1, one for each class, indicating the probability of the observation belonging to each class.

In this notebook, we'll focus on *regression*, using an example based on a real study in which data for a bicycle sharing scheme was collected and used to predict the number of rentals based on seasonality and weather conditions. We'll use a simplified version of the dataset from that study.

1.0.1 Explore the Data

The first step in any machine learning project is to explore the data that you will use to train a model. The goal of this exploration is to try to understand the relationships between its attributes;

in particular, any apparent correlation between the *features* and the *label* your model will try to predict. This may require some work to detect and fix issues in the data (such as dealing with missing values, errors, or outlier values), deriving new feature columns by transforming or combining existing features (a process known as *feature engineering*), *normalizing* numeric features (values you can measure or count) so they're on a similar scale, and *encoding* categorical features (values that represent discrete categories) as numeric indicators.

Let's start by loading the bicycle sharing data as a **Pandas** DataFrame and viewing the first few rows.

```
[21]: import pandas as pd

# load the training dataset
bike_data = pd.read_csv('data/daily-bike-share.csv')
bike_data.head()
```

```
[21]:
```

	instant	dteday	season	yr	mnth	holiday	weekday	workingday	\
0	1	1/1/2011	1	0	1	0	6	0	
1	2	1/2/2011	1	0	1	0	0	0	
2	3	1/3/2011	1	0	1	0	1	1	
3	4	1/4/2011	1	0	1	0	2	1	
4	5	1/5/2011	1	0	1	0	3	1	

	weathersit	temp	atemp	hum	windspeed	rentals
0	2	0.344167	0.363625	0.805833	0.160446	331
1	2	0.363478	0.353739	0.696087	0.248539	131
2	1	0.196364	0.189405	0.437273	0.248309	120
3	1	0.200000	0.212122	0.590435	0.160296	108
4	1	0.226957	0.229270	0.436957	0.186900	82

The data consists of the following columns:

- **instant**: A unique row identifier
- **dteday**: The date on which the data was observed - in this case, the data was collected daily; so there's one row per date.
- **season**: A numerically encoded value indicating the season (1:spring, 2:summer, 3:fall, 4:winter)
- **yr**: The year of the study in which the observation was made (the study took place over two years - year 0 represents 2011, and year 1 represents 2012)
- **mnth**: The calendar month in which the observation was made (1:January ... 12:December)
- **holiday**: A binary value indicating whether or not the observation was made on a public holiday)
- **weekday**: The day of the week on which the observation was made (0:Sunday ... 6:Saturday)
- **workingday**: A binary value indicating whether or not the day is a working day (not a weekend or holiday)
- **weathersit**: A categorical value indicating the weather situation (1:clear, 2:mist/cloud, 3:light rain/snow, 4:heavy rain/hail/snow/fog)
- **temp**: The temperature in celsius (normalized)
- **atemp**: The apparent ("feels-like") temperature in celsius (normalized)

- **hum**: The humidity level (normalized)
- **windspeed**: The windspeed (normalized)
- **rentals**: The number of bicycle rentals recorded.

In this dataset, **rentals** represents the label (the y value) our model must be trained to predict. The other columns are potential features (x values).

As mentioned previously, you can perform some *feature engineering* to combine or derive new features. For example, let's add a new column named **day** to the dataframe by extracting the day component from the existing **dteday** column. The new column represents the day of the month from 1 to 31.

```
[22]: bike_data['day'] = pd.DatetimeIndex(bike_data['dteday']).day
      bike_data.head(32)
```

```
[22]:
```

	instant	dteday	season	yr	mnth	holiday	weekday	workingday	\
0	1	1/1/2011	1	0	1	0	6	0	
1	2	1/2/2011	1	0	1	0	0	0	
2	3	1/3/2011	1	0	1	0	1	1	
3	4	1/4/2011	1	0	1	0	2	1	
4	5	1/5/2011	1	0	1	0	3	1	
5	6	1/6/2011	1	0	1	0	4	1	
6	7	1/7/2011	1	0	1	0	5	1	
7	8	1/8/2011	1	0	1	0	6	0	
8	9	1/9/2011	1	0	1	0	0	0	
9	10	1/10/2011	1	0	1	0	1	1	
10	11	1/11/2011	1	0	1	0	2	1	
11	12	1/12/2011	1	0	1	0	3	1	
12	13	1/13/2011	1	0	1	0	4	1	
13	14	1/14/2011	1	0	1	0	5	1	
14	15	1/15/2011	1	0	1	0	6	0	
15	16	1/16/2011	1	0	1	0	0	0	
16	17	1/17/2011	1	0	1	1	1	0	
17	18	1/18/2011	1	0	1	0	2	1	
18	19	1/19/2011	1	0	1	0	3	1	
19	20	1/20/2011	1	0	1	0	4	1	
20	21	1/21/2011	1	0	1	0	5	1	
21	22	1/22/2011	1	0	1	0	6	0	
22	23	1/23/2011	1	0	1	0	0	0	
23	24	1/24/2011	1	0	1	0	1	1	
24	25	1/25/2011	1	0	1	0	2	1	
25	26	1/26/2011	1	0	1	0	3	1	
26	27	1/27/2011	1	0	1	0	4	1	
27	28	1/28/2011	1	0	1	0	5	1	
28	29	1/29/2011	1	0	1	0	6	0	
29	30	1/30/2011	1	0	1	0	0	0	
30	31	1/31/2011	1	0	1	0	1	1	
31	32	2/1/2011	1	0	2	0	2	1	

	weathersit	temp	atemp	hum	windspeed	rentals	day
0	2	0.344167	0.363625	0.805833	0.160446	331	1
1	2	0.363478	0.353739	0.696087	0.248539	131	2
2	1	0.196364	0.189405	0.437273	0.248309	120	3
3	1	0.200000	0.212122	0.590435	0.160296	108	4
4	1	0.226957	0.229270	0.436957	0.186900	82	5
5	1	0.204348	0.233209	0.518261	0.089565	88	6
6	2	0.196522	0.208839	0.498696	0.168726	148	7
7	2	0.165000	0.162254	0.535833	0.266804	68	8
8	1	0.138333	0.116175	0.434167	0.361950	54	9
9	1	0.150833	0.150888	0.482917	0.223267	41	10
10	2	0.169091	0.191464	0.686364	0.122132	43	11
11	1	0.172727	0.160473	0.599545	0.304627	25	12
12	1	0.165000	0.150883	0.470417	0.301000	38	13
13	1	0.160870	0.188413	0.537826	0.126548	54	14
14	2	0.233333	0.248112	0.498750	0.157963	222	15
15	1	0.231667	0.234217	0.483750	0.188433	251	16
16	2	0.175833	0.176771	0.537500	0.194017	117	17
17	2	0.216667	0.232333	0.861667	0.146775	9	18
18	2	0.292174	0.298422	0.741739	0.208317	78	19
19	2	0.261667	0.255050	0.538333	0.195904	83	20
20	1	0.177500	0.157833	0.457083	0.353242	75	21
21	1	0.059130	0.079070	0.400000	0.171970	93	22
22	1	0.096522	0.098839	0.436522	0.246600	150	23
23	1	0.097391	0.117930	0.491739	0.158330	86	24
24	2	0.223478	0.234526	0.616957	0.129796	186	25
25	3	0.217500	0.203600	0.862500	0.293850	34	26
26	1	0.195000	0.219700	0.687500	0.113837	15	27
27	2	0.203478	0.223317	0.793043	0.123300	38	28
28	1	0.196522	0.212126	0.651739	0.145365	123	29
29	1	0.216522	0.250322	0.722174	0.073983	140	30
30	2	0.180833	0.186250	0.603750	0.187192	42	31
31	2	0.192174	0.234530	0.829565	0.053213	47	1

OK, let's start our analysis of the data by examining a few key descriptive statistics. We can use the dataframe's **describe** method to generate these for the numeric features as well as the **rentals** label column.

```
[23]: numeric_features = ['temp', 'atemp', 'hum', 'windspeed']
      bike_data[numeric_features + ['rentals']].describe()
```

```
[23]:
```

	temp	atemp	hum	windspeed	rentals
count	731.000000	731.000000	731.000000	731.000000	731.000000
mean	0.495385	0.474354	0.627894	0.190486	848.176471
std	0.183051	0.162961	0.142429	0.077498	686.622488
min	0.059130	0.079070	0.000000	0.022392	2.000000
25%	0.337083	0.337842	0.520000	0.134950	315.500000

50%	0.498333	0.486733	0.626667	0.180975	713.000000
75%	0.655417	0.608602	0.730209	0.233214	1096.000000
max	0.861667	0.840896	0.972500	0.507463	3410.000000

The statistics reveal some information about the distribution of the data in each of the numeric fields, including the number of observations (there are 731 records), the mean, standard deviation, minimum and maximum values, and the quartile values (the threshold values for 25%, 50% - which is also the median, and 75% of the data). From this, we can see that the mean number of daily rentals is around 848; but there's a comparatively large standard deviation, indicating a lot of variance in the number of rentals per day.

We might get a clearer idea of the distribution of rentals values by visualizing the data. Common plot types for visualizing numeric data distributions are *histograms* and *box plots*, so let's use Python's **matplotlib** library to create one of each of these for the **rentals** column.

```
[24]: import pandas as pd
import matplotlib.pyplot as plt

# This ensures plots are displayed inline in the Jupyter notebook
%matplotlib inline

# Get the label column
label = bike_data['rentals']

# Create a figure for 2 subplots (2 rows, 1 column)
fig, ax = plt.subplots(2, 1, figsize = (9,12))

# Plot the histogram
ax[0].hist(label, bins=100)
ax[0].set_ylabel('Frequency')

# Add lines for the mean, median, and mode
ax[0].axvline(label.mean(), color='magenta', linestyle='dashed', linewidth=2)
ax[0].axvline(label.median(), color='cyan', linestyle='dashed', linewidth=2)

# Plot the boxplot
ax[1].boxplot(label, vert=False)
ax[1].set_xlabel('Rentals')

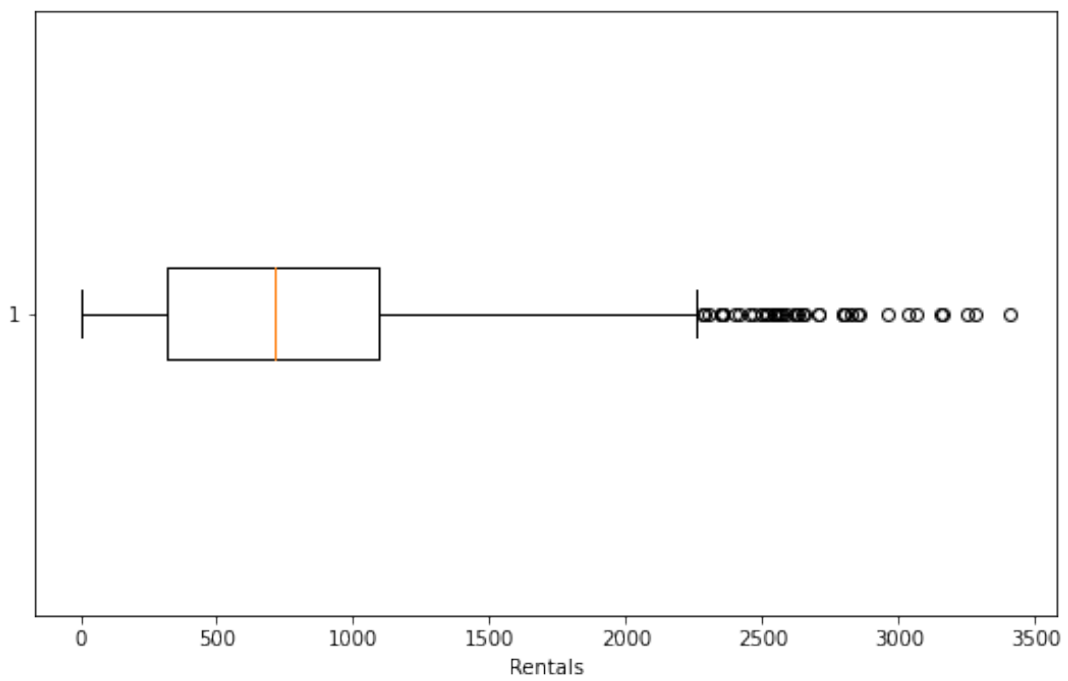
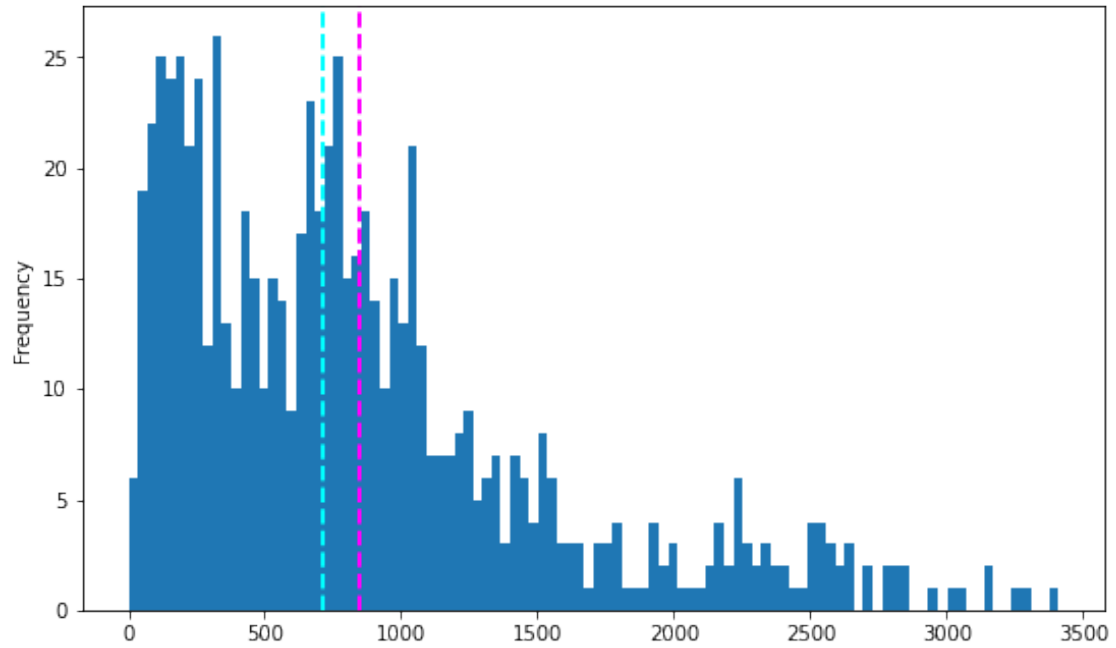
# Add a title to the Figure
fig.suptitle('Rental Distribution')

# Show the figure
fig.show()
```

```
<ipython-input-24-cda7e8772254>:30: UserWarning: Matplotlib is currently using
module://ipykernel.pylab.backend_inline, which is a non-GUI backend, so cannot
```

```
show the figure.  
fig.show()
```

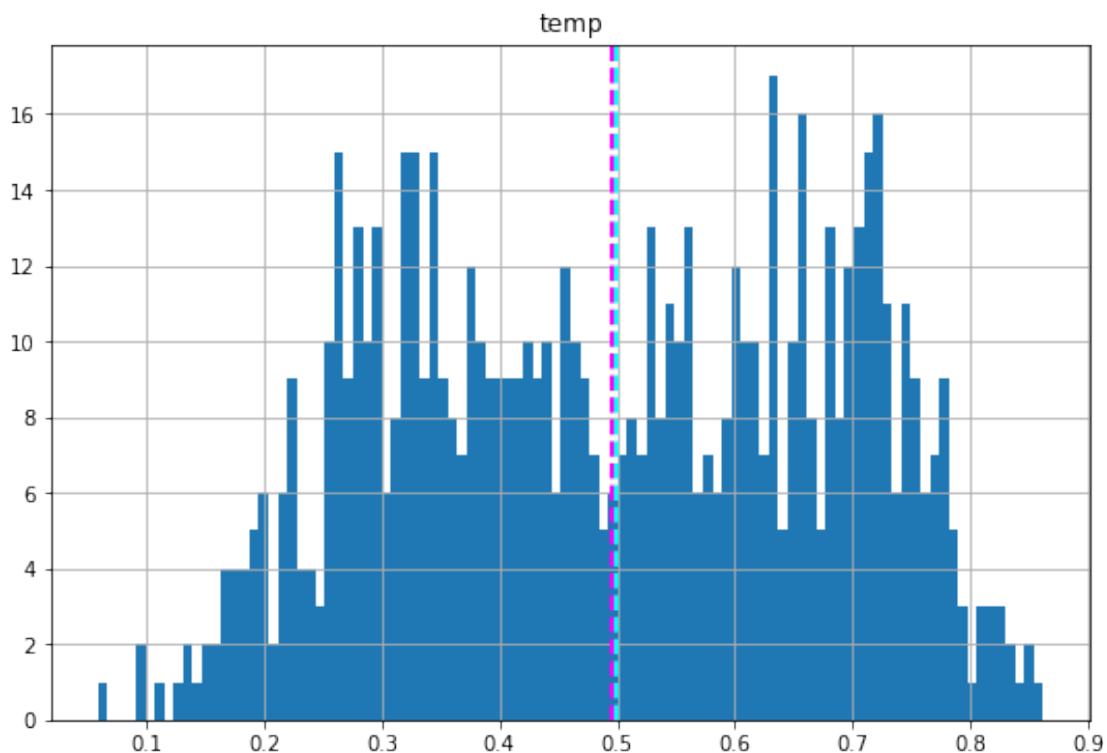
Rental Distribution

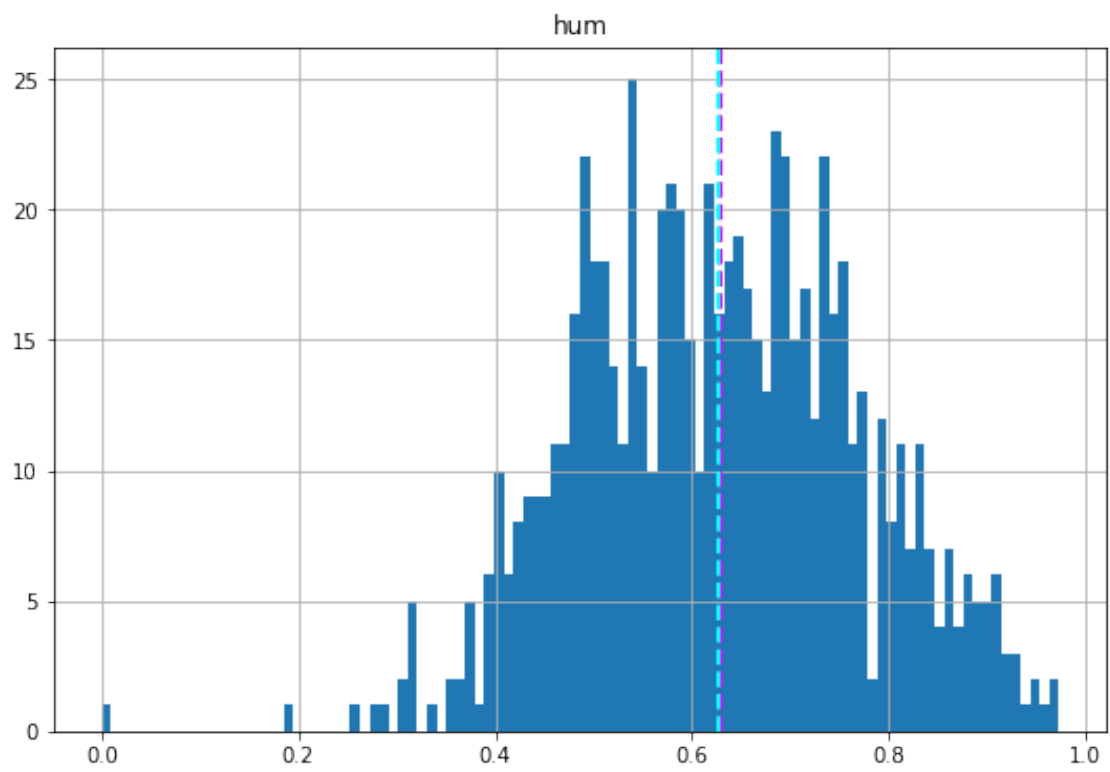
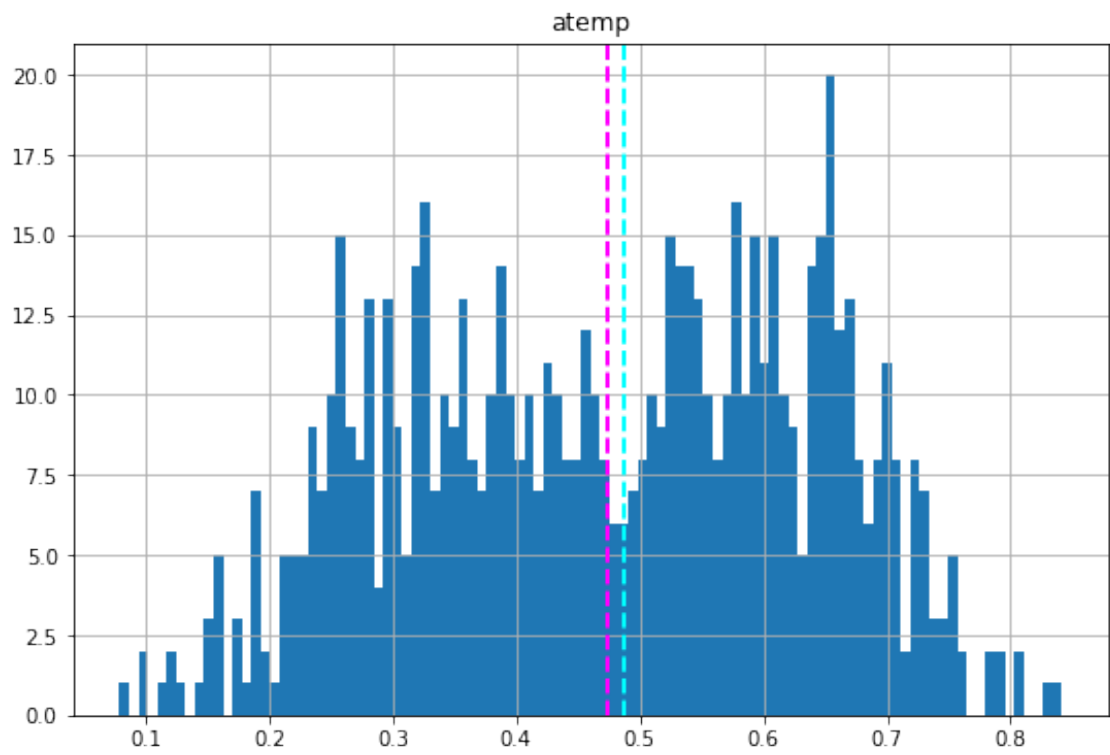


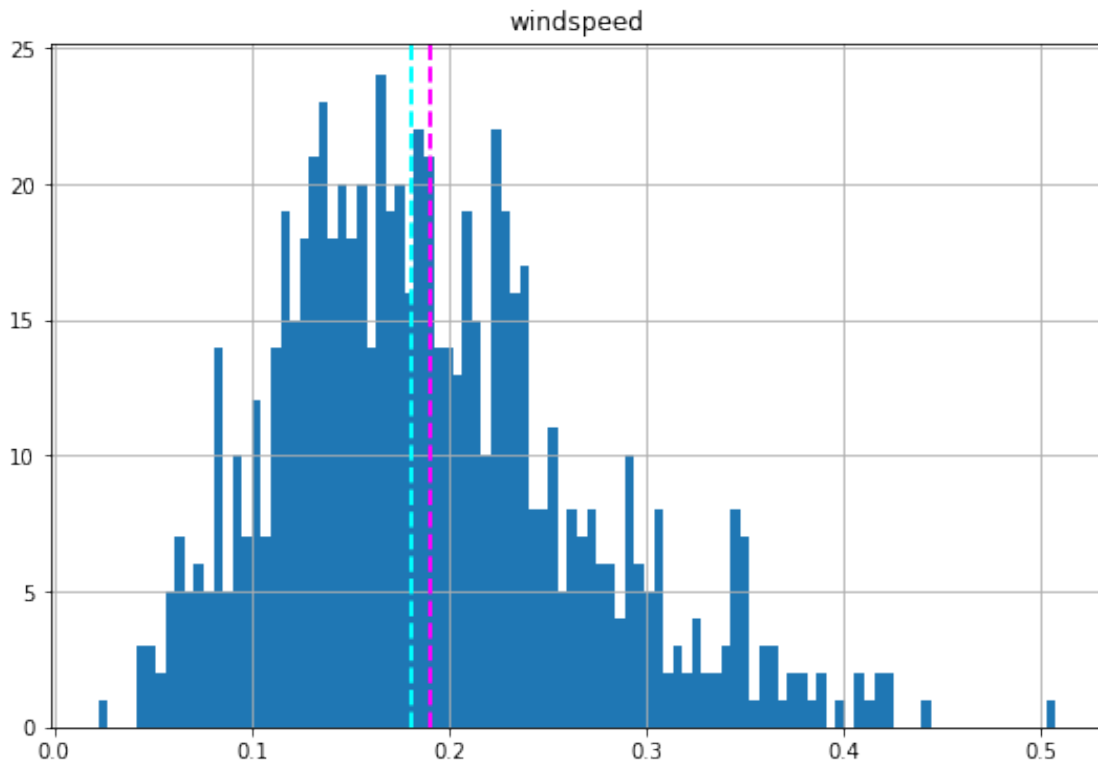
The plots show that the number of daily rentals ranges from 0 to just over 3,400. However, the mean (and median) number of daily rentals is closer to the low end of that range, with most of the data between 0 and around 2,200 rentals. The few values above this are shown in the box plot as small circles, indicating that they are *outliers* - in other words, unusually high or low values beyond the typical range of most of the data.

We can do the same kind of visual exploration of the numeric features. Let's create a histogram for each of these.

```
[25]: # Plot a histogram for each numeric feature
for col in numeric_features:
    fig = plt.figure(figsize=(9, 6))
    ax = fig.gca()
    feature = bike_data[col]
    feature.hist(bins=100, ax = ax)
    ax.axvline(feature.mean(), color='magenta', linestyle='dashed', linewidth=2)
    ax.axvline(feature.median(), color='cyan', linestyle='dashed', linewidth=2)
    ax.set_title(col)
plt.show()
```







The numeric features seem to be more *normally* distributed, with the mean and median nearer the middle of the range of values, coinciding with where the most commonly occurring values are.

Note: The distributions are not truly *normal* in the statistical sense, which would result in a smooth, symmetric “bell-curve” histogram with the mean and mode (the most common value) in the center; but they do generally indicate that most of the observations have a value somewhere near the middle.

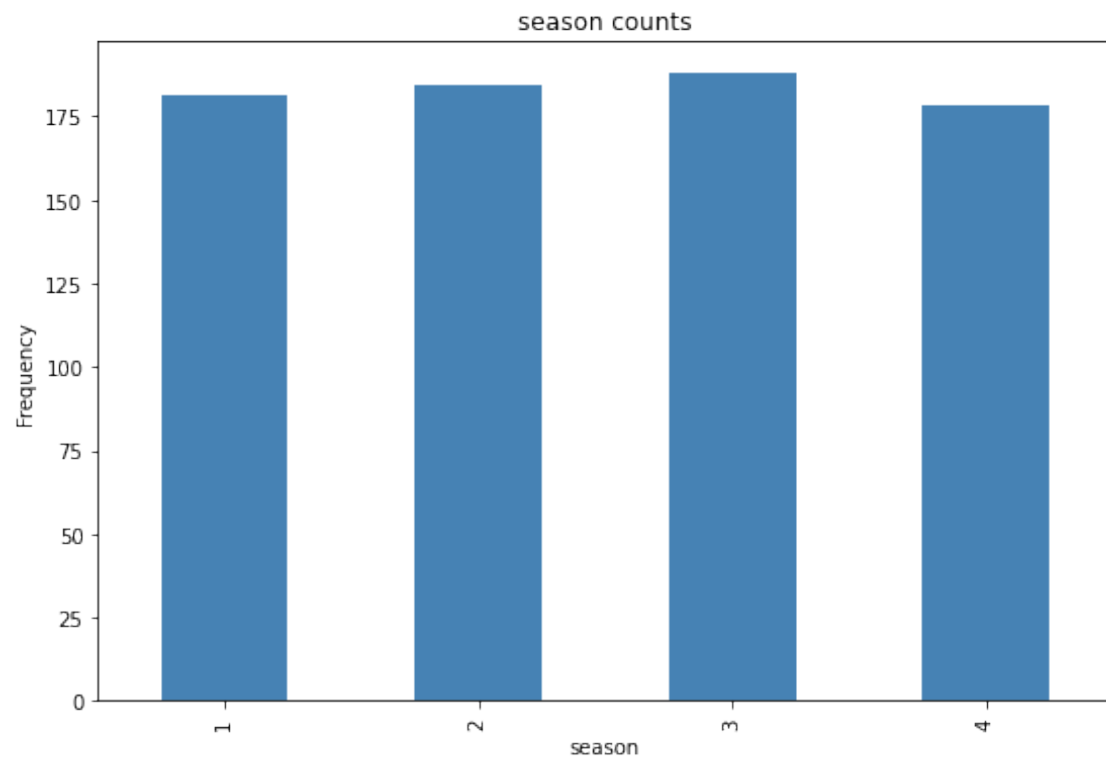
We’ve explored the distribution of the numeric values in the dataset, but what about the categorical features? These aren’t continuous numbers on a scale, so we can’t use histograms; but we can plot a bar chart showing the count of each discrete value for each category.

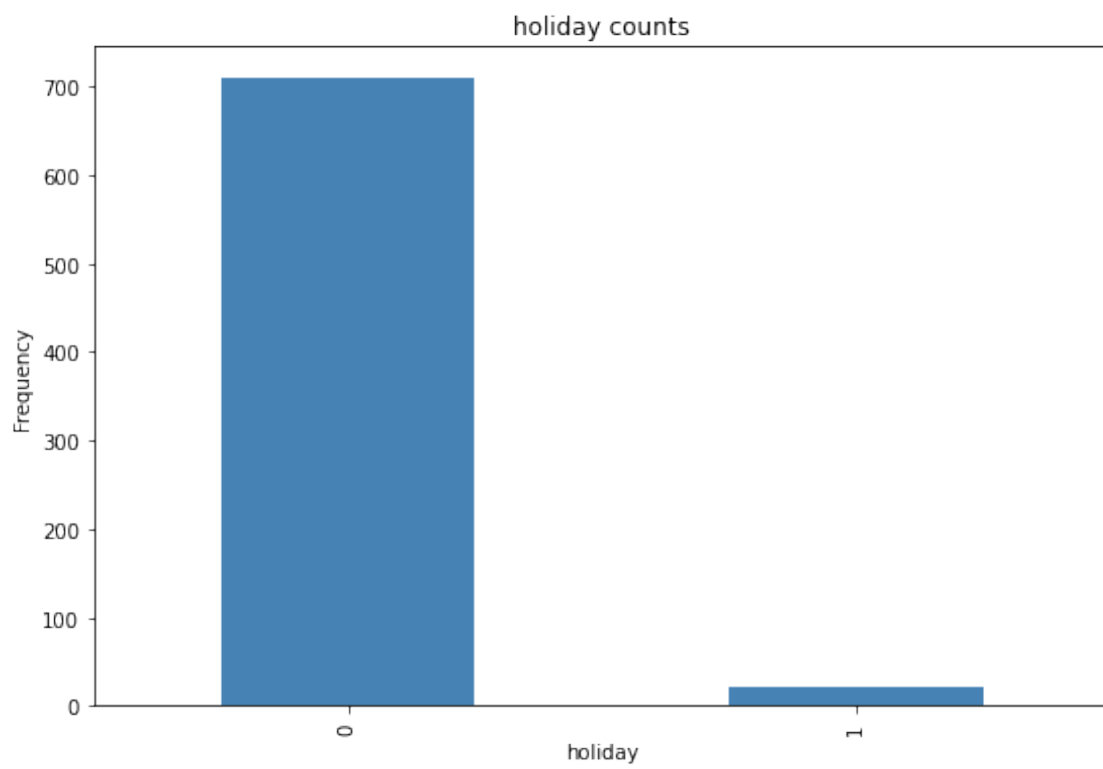
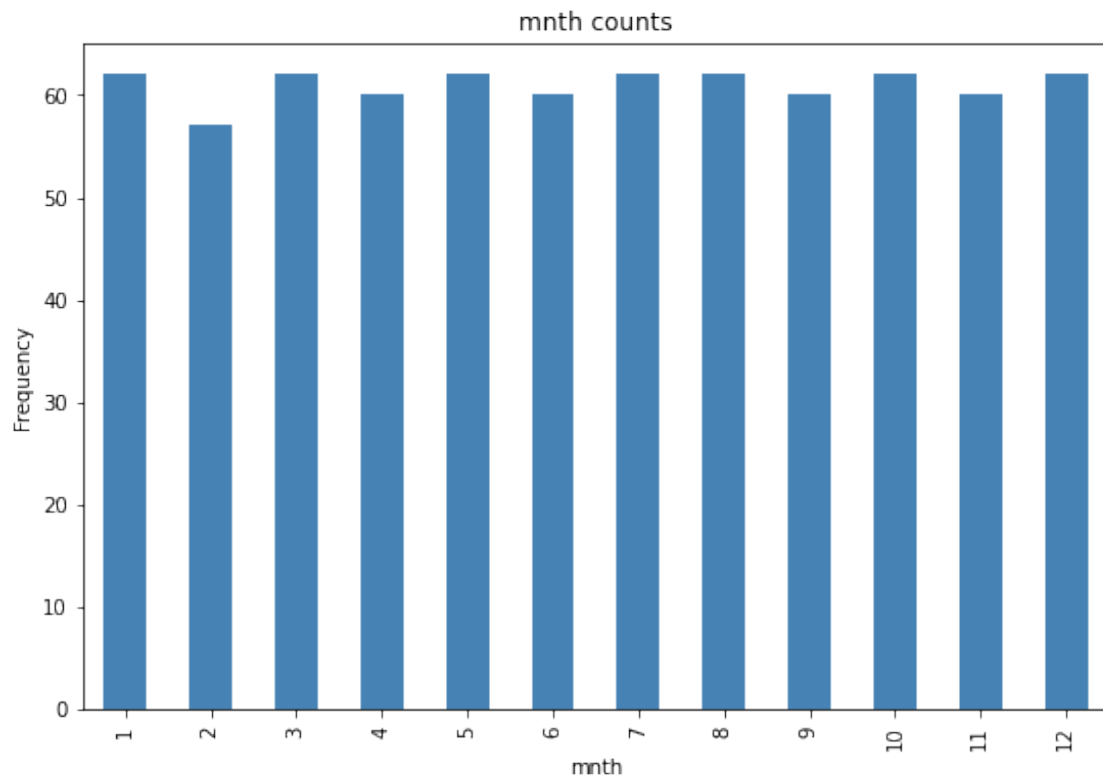
```
[26]: import numpy as np

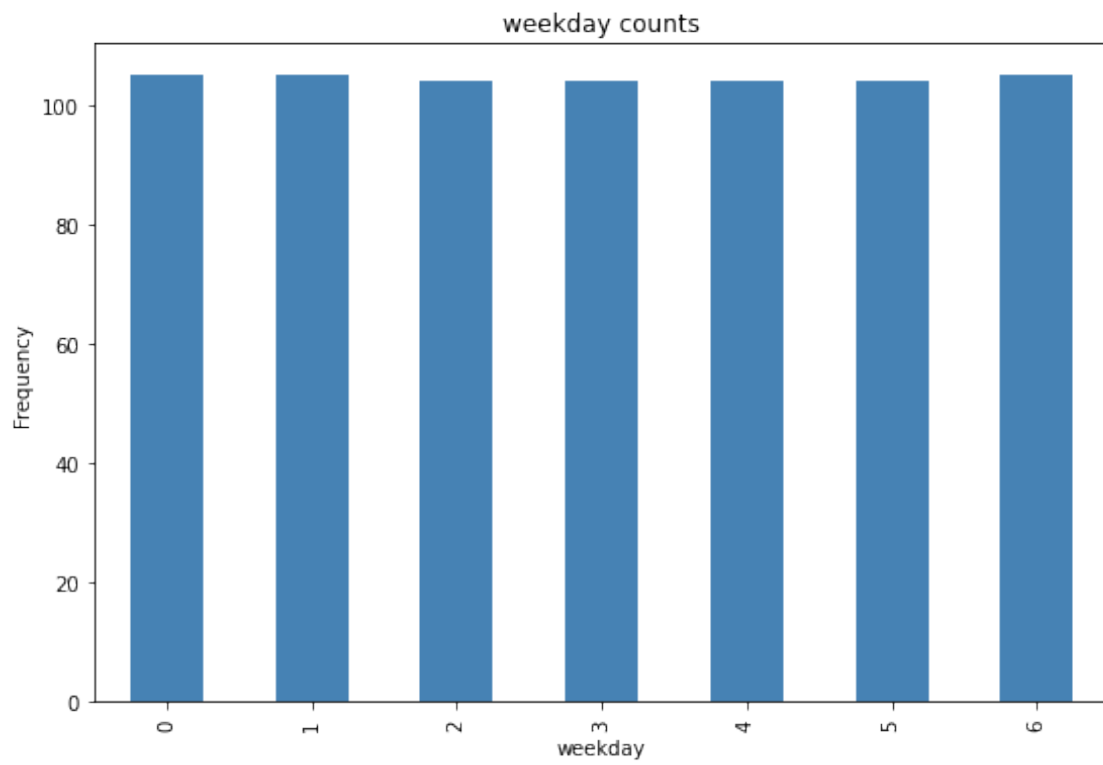
# plot a bar plot for each categorical feature count
categorical_features = [
    'season', 'mnth', 'holiday', 'weekday', 'workingday', 'weathersit', 'day']

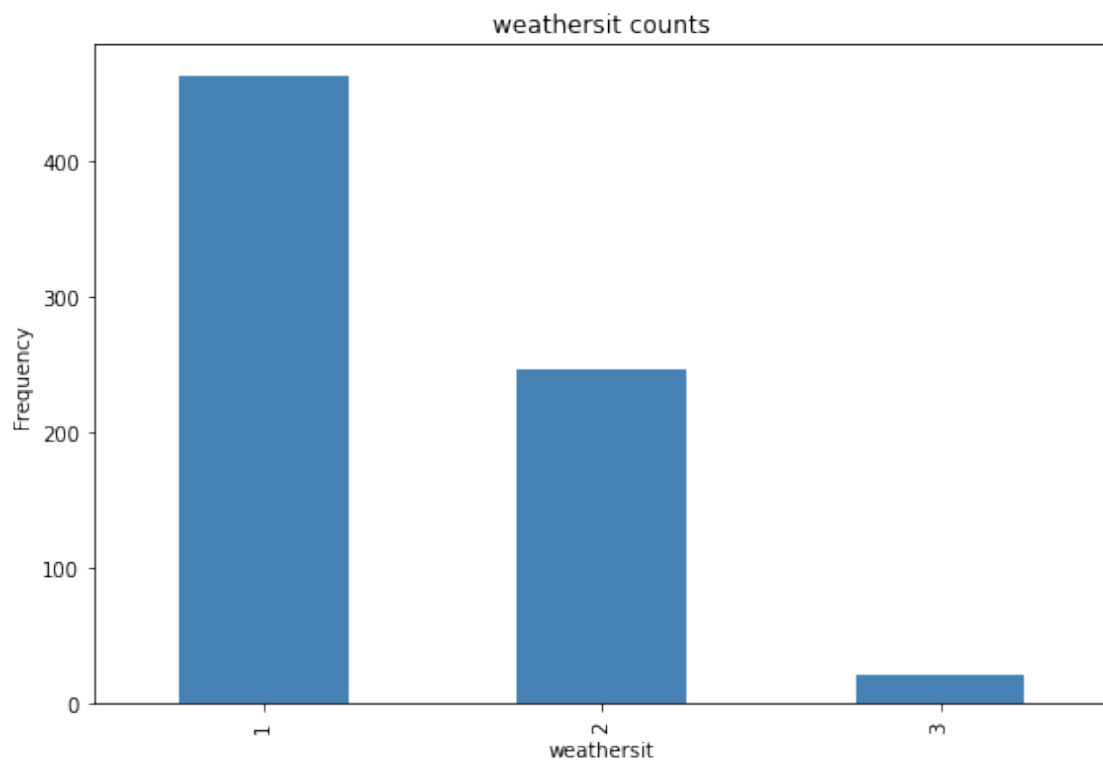
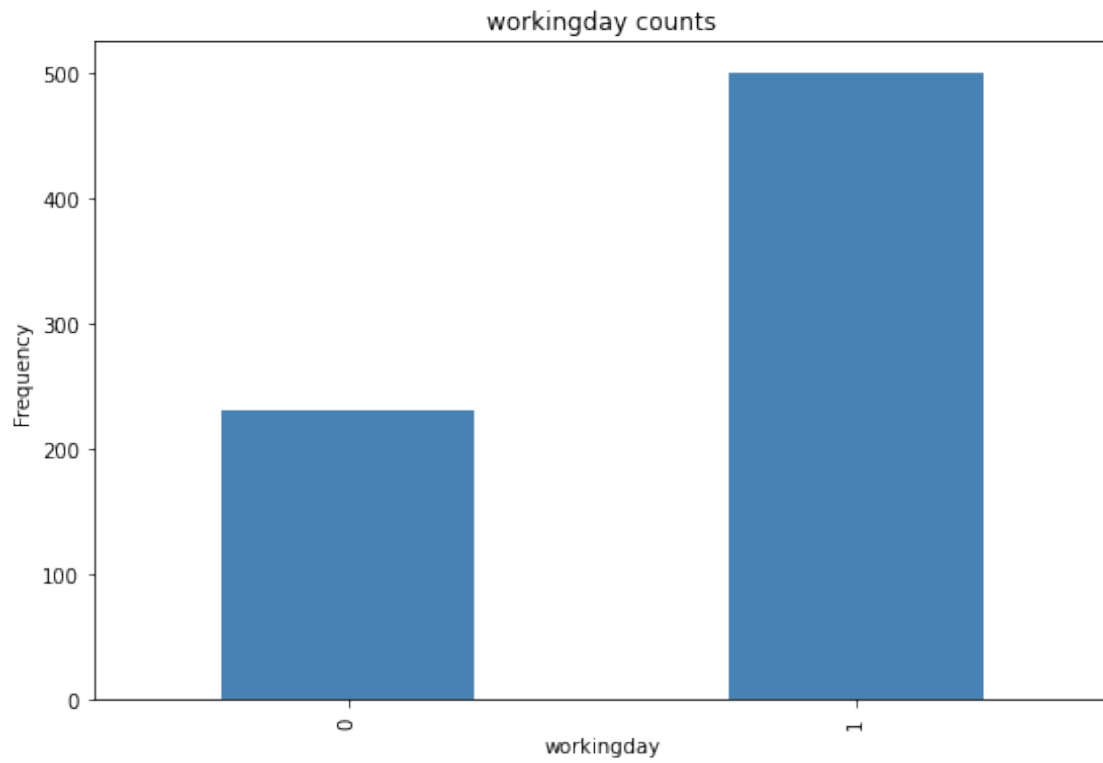
for col in categorical_features:
    counts = bike_data[col].value_counts().sort_index()
    fig = plt.figure(figsize=(9, 6))
    ax = fig.gca()
```

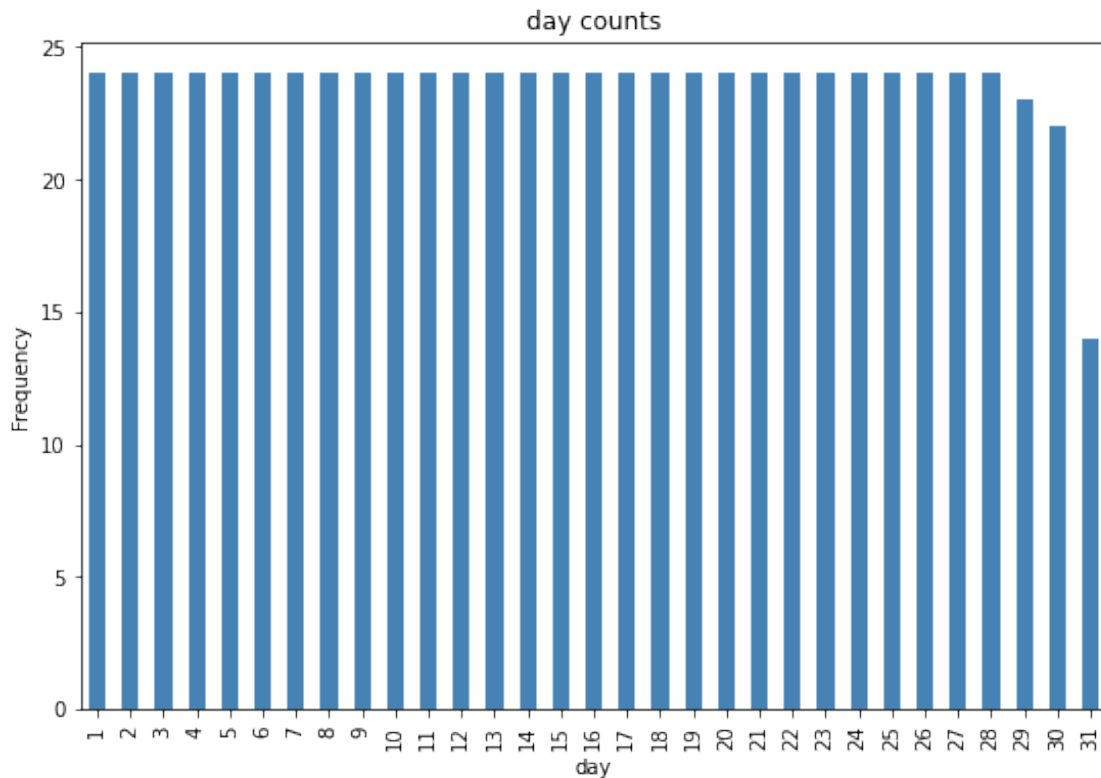
```
counts.plot.bar(ax = ax, color='steelblue')
ax.set_title(col + ' counts')
ax.set_xlabel(col)
ax.set_ylabel("Frequency")
plt.show()
```











Many of the categorical features show a more or less *uniform* distribution (meaning there's roughly the same number of rows for each category). Exceptions to this include:

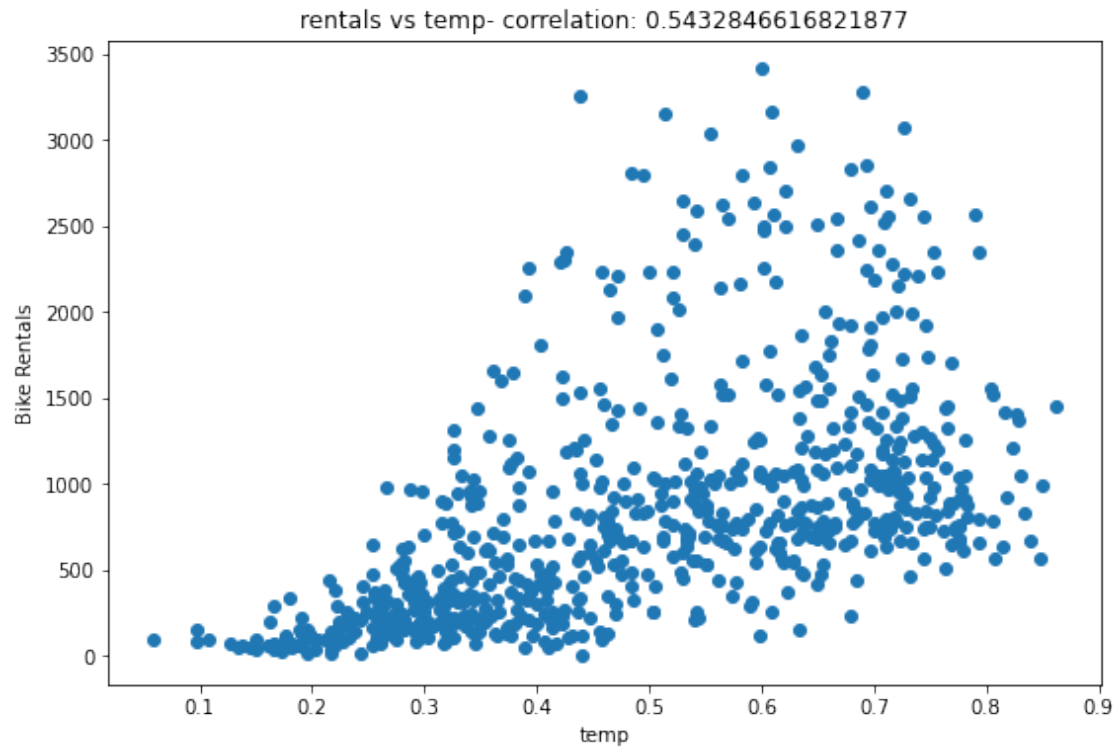
- **holiday**: There are many fewer days that are holidays than days that aren't.
- **workingday**: There are more working days than non-working days.
- **weathersit**: Most days are category 1 (clear), with category 2 (mist and cloud) the next most common. There are comparatively few category 3 (light rain or snow) days, and no category 4 (heavy rain, hail, or fog) days at all.

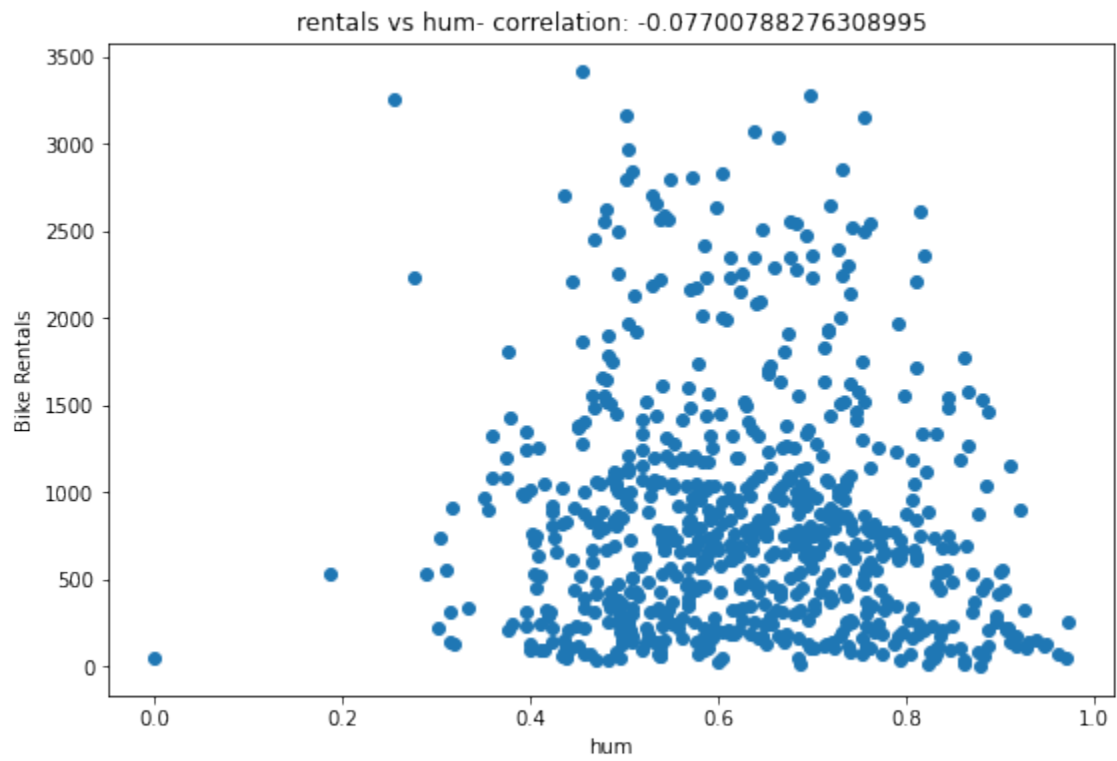
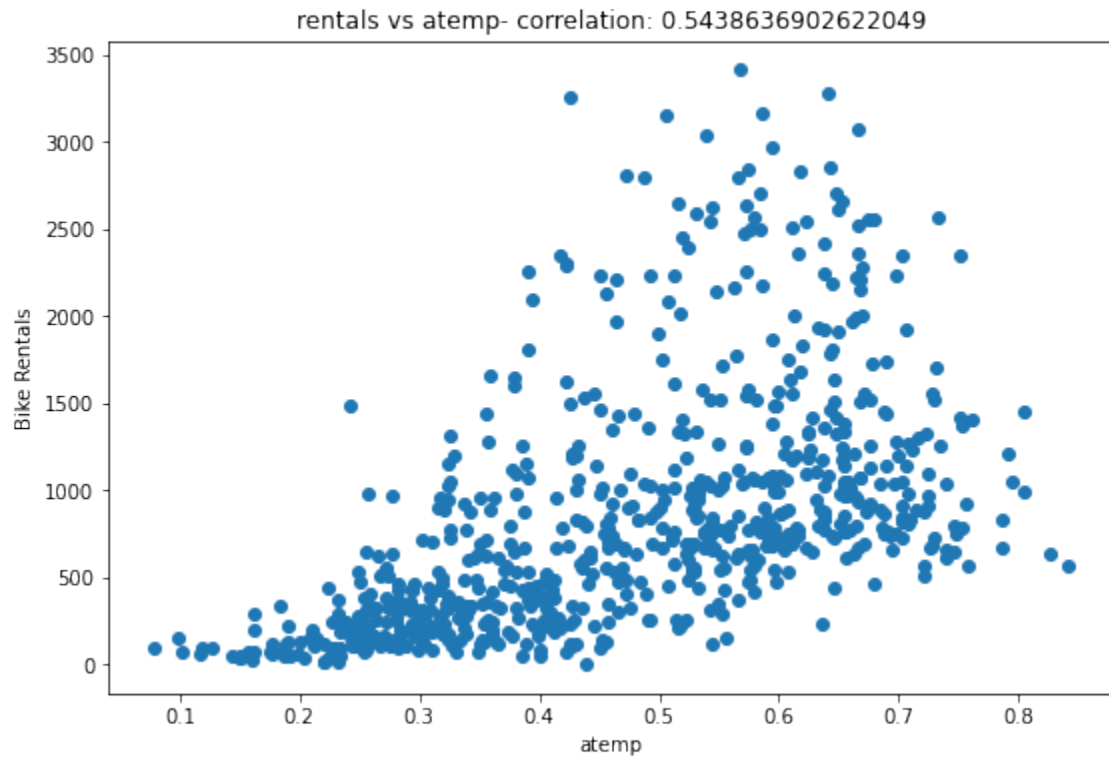
Now that we know something about the distribution of the data in our columns, we can start to look for relationships between the features and the **rentals** label we want to be able to predict.

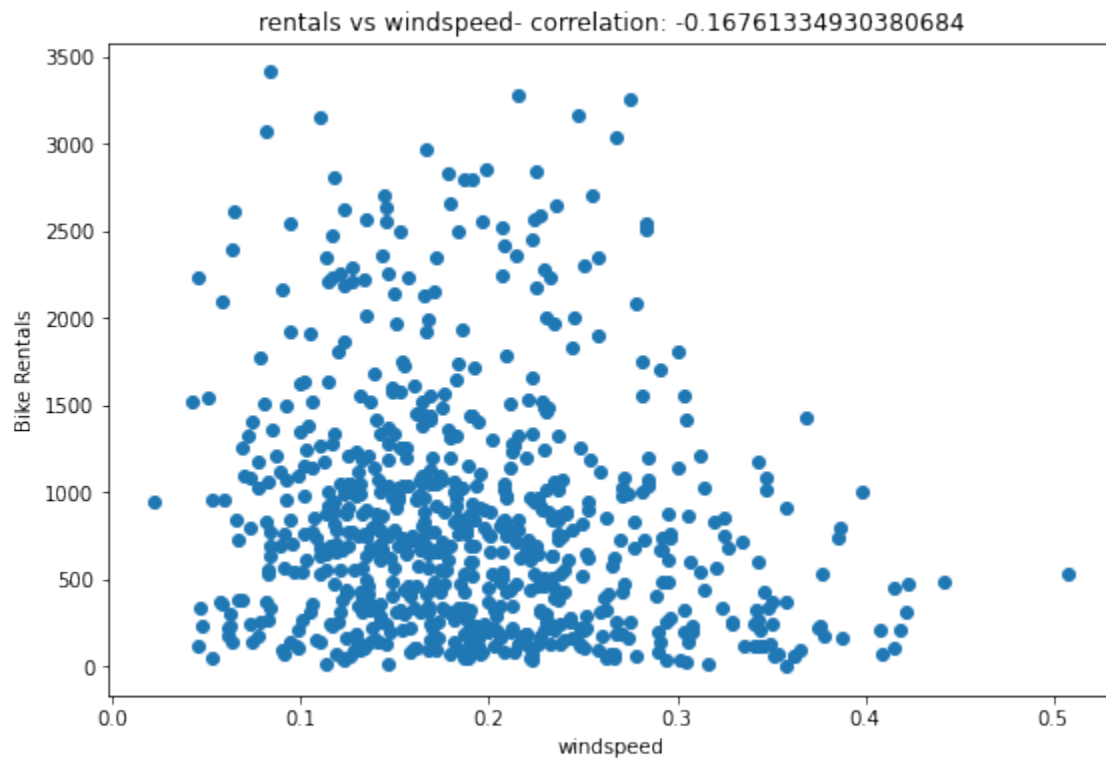
For the numeric features, we can create scatter plots that show the intersection of feature and label values. We can also calculate the *correlation* statistic to quantify the apparent relationship..

```
[27]: for col in numeric_features:
    fig = plt.figure(figsize=(9, 6))
    ax = fig.gca()
    feature = bike_data[col]
    label = bike_data['rentals']
    correlation = feature.corr(label)
    plt.scatter(x=feature, y=label)
```

```
plt.xlabel(col)
plt.ylabel('Bike Rentals')
ax.set_title('rentals vs ' + col + '- correlation: ' + str(correlation))
plt.show()
```



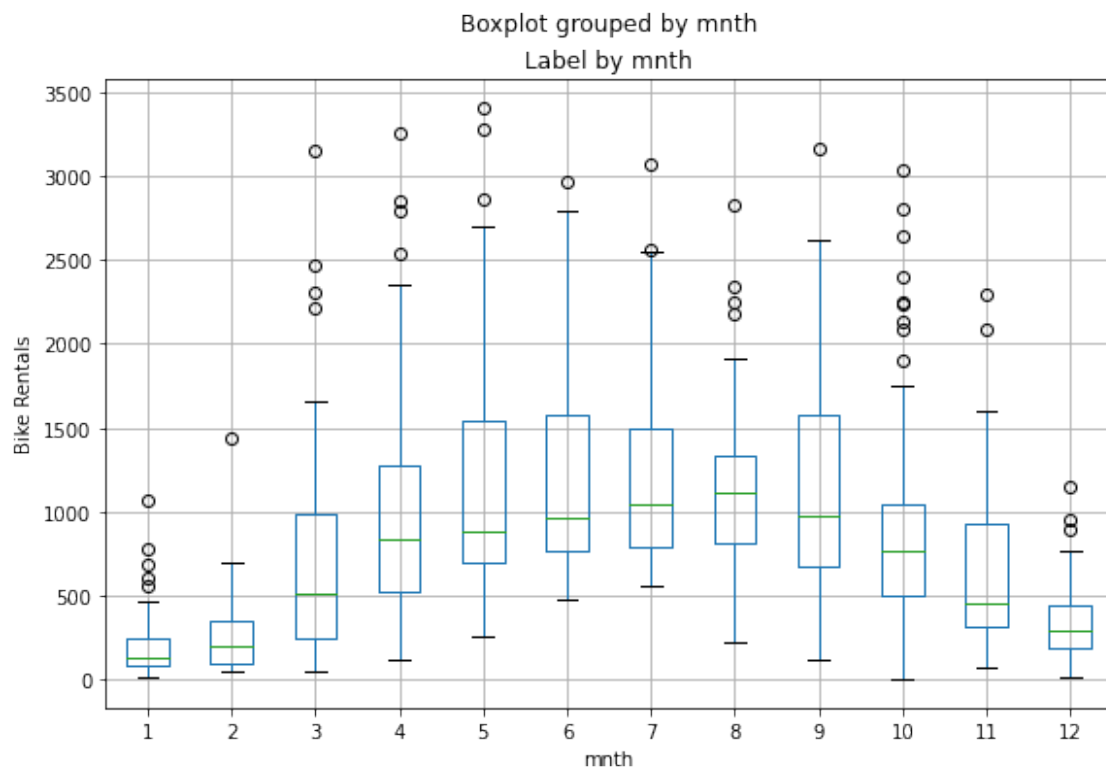
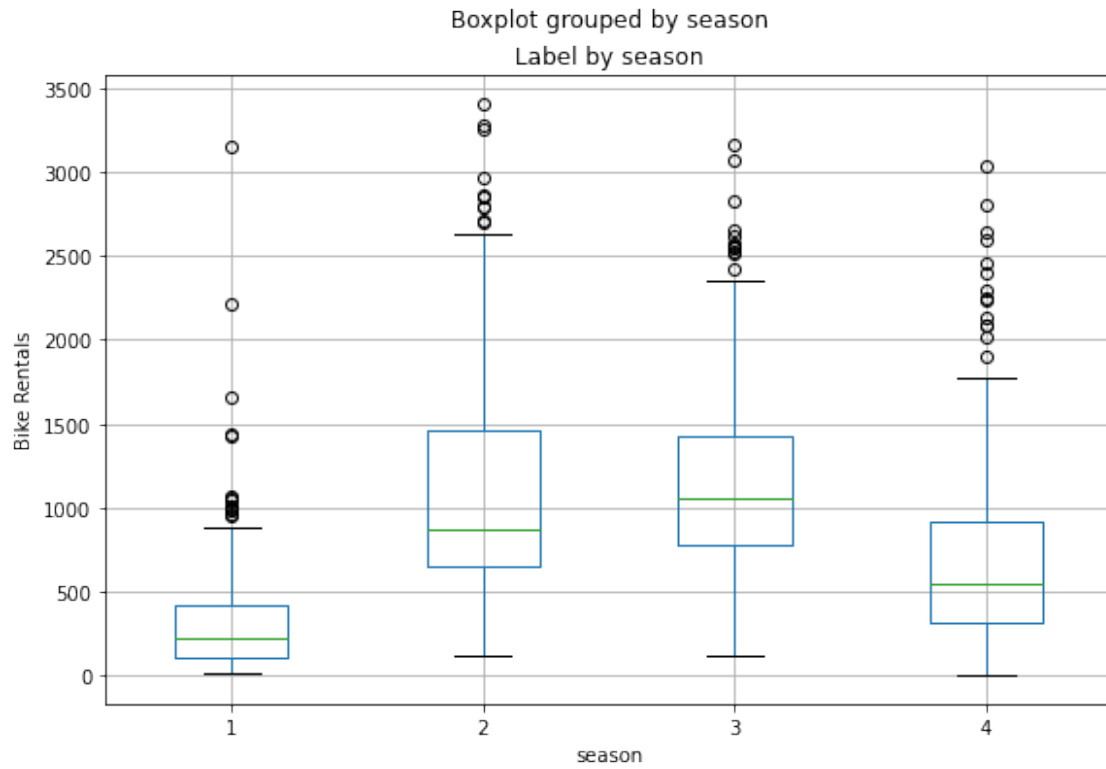


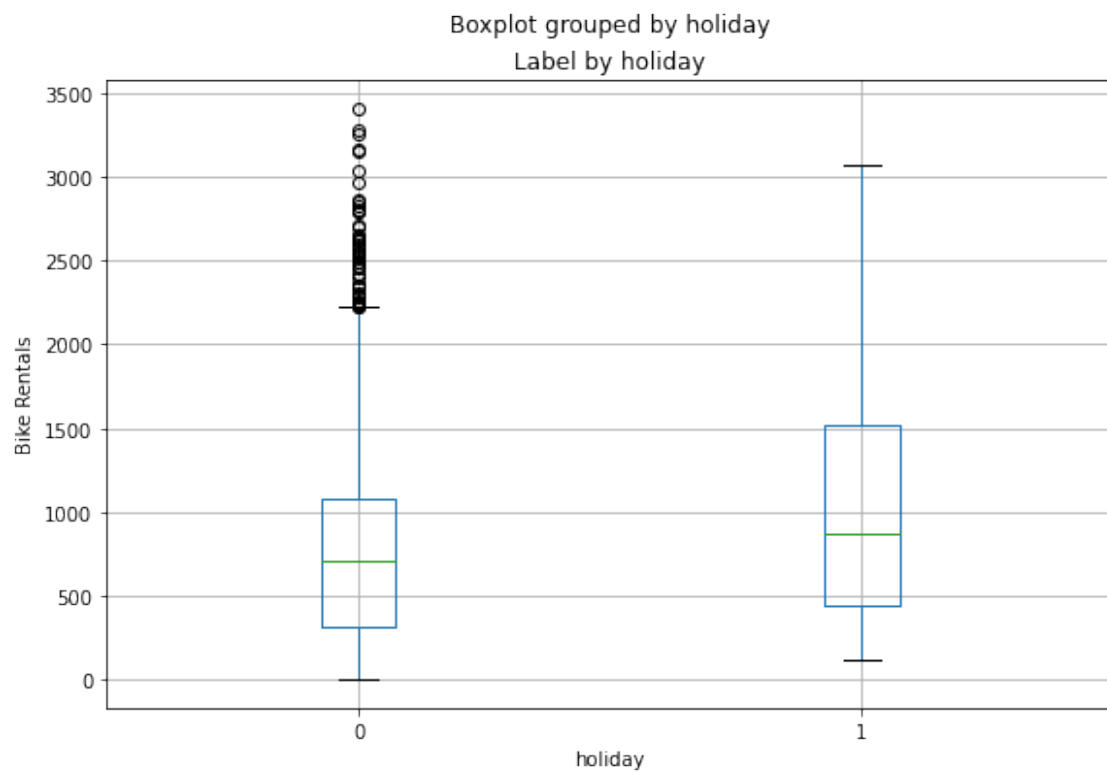


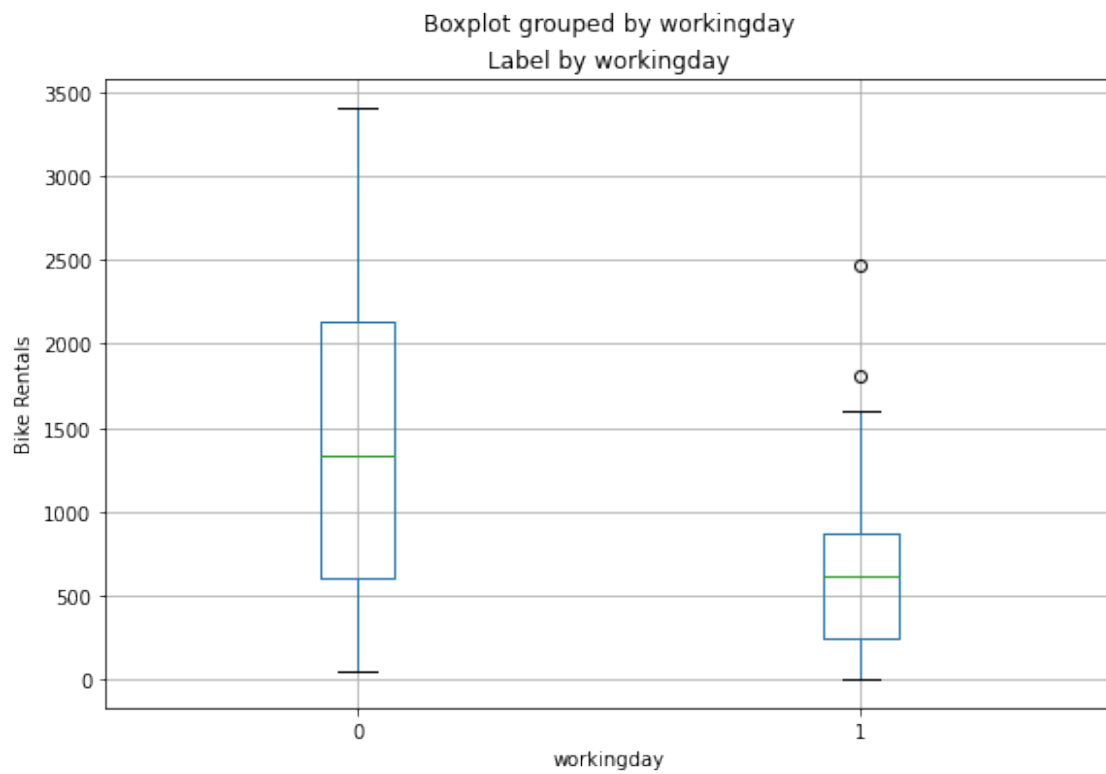
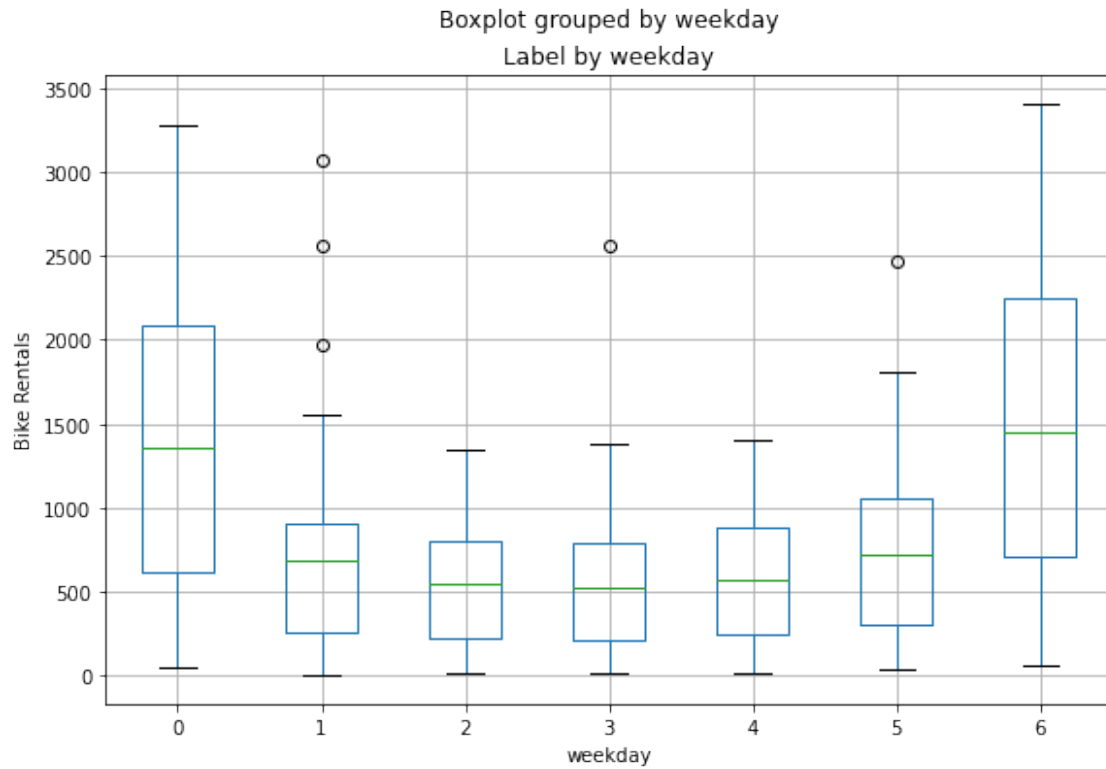
The results aren't conclusive, but if you look closely at the scatter plots for **temp** and **atemp**, you can see a vague diagonal trend showing that higher rental counts tend to coincide with higher temperatures; and a correlation value of just over 0.5 for both of these features supports this observation. Conversely, the plots for **hum** and **windspeed** show a slightly negative correlation, indicating that there are fewer rentals on days with high humidity or windspeed.

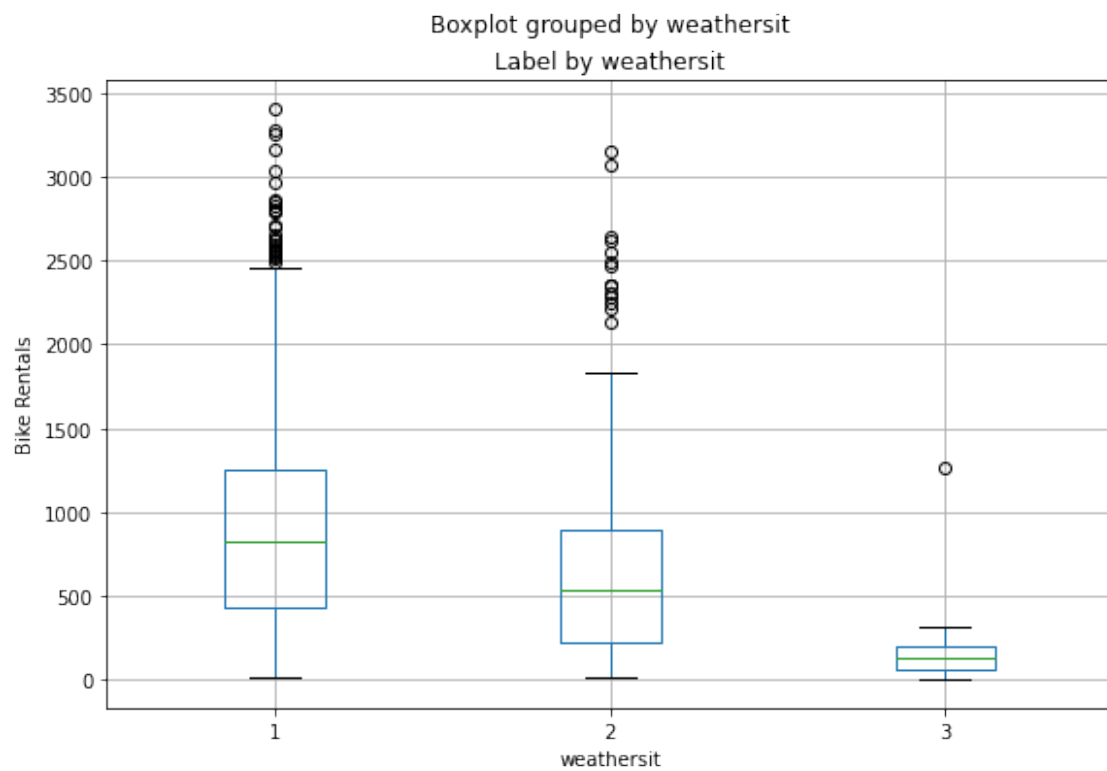
Now let's compare the categorical features to the label. We'll do this by creating box plots that show the distribution of rental counts for each category.

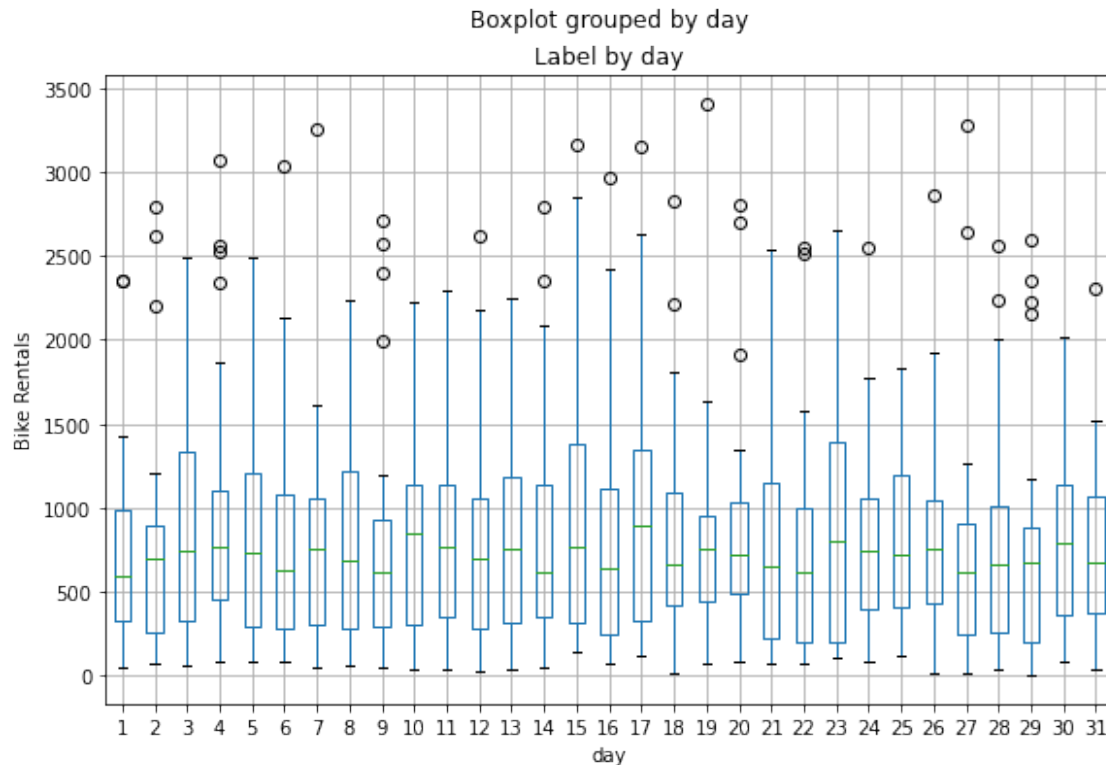
```
[28]: # plot a boxplot for the label by each categorical feature
for col in categorical_features:
    fig = plt.figure(figsize=(9, 6))
    ax = fig.gca()
    bike_data.boxplot(column = 'rentals', by = col, ax = ax)
    ax.set_title('Label by ' + col)
    ax.set_ylabel("Bike Rentals")
plt.show()
```











The plots show some variance in the relationship between some category values and rentals. For example, there's a clear difference in the distribution of rentals on weekends (**weekday** 0 or 6) and those during the working week (**weekday** 1 to 5). Similarly, there are notable differences for **holiday** and **workingday** categories. There's a noticeable trend that shows different rental distributions in summer and fall months compared to spring and winter months. The **weathersit** category also seems to make a difference in rental distribution. The **day** feature we created for the day of the month shows little variation, indicating that it's probably not predictive of the number of rentals.

1.0.2 Train a Regression Model

Now that we've explored the data, it's time to use it to train a regression model that uses the features we've identified as potentially predictive to predict the **rentals** label. The first thing we need to do is to separate the features we want to use to train the model from the label we want it to predict.

```
[29]: # Separate features and labels
X, y = bike_data[['season', 'mnth', '
    ↳ 'holiday', 'weekday', 'workingday', 'weathersit', 'temp', 'atemp', 'hum', '
    ↳ 'windspeed']].values, bike_data['rentals'].values
print('Features:', X[:10], '\nLabels:', y[:10], sep='\n')
```

Features:

```
[[1.      1.      0.      6.      0.      2.      0.344167
  0.363625 0.805833 0.160446 ]
 [1.      1.      0.      0.      0.      2.      0.363478
  0.353739 0.696087 0.248539 ]
 [1.      1.      0.      1.      1.      1.      0.196364
  0.189405 0.437273 0.248309 ]
 [1.      1.      0.      2.      1.      1.      0.2
  0.212122 0.590435 0.160296 ]
 [1.      1.      0.      3.      1.      1.      0.226957
  0.22927  0.436957 0.1869   ]
 [1.      1.      0.      4.      1.      1.      0.204348
  0.233209 0.518261 0.0895652]
 [1.      1.      0.      5.      1.      2.      0.196522
  0.208839 0.498696 0.168726 ]
 [1.      1.      0.      6.      0.      2.      0.165
  0.162254 0.535833 0.266804 ]
 [1.      1.      0.      0.      0.      1.      0.138333
  0.116175 0.434167 0.36195   ]
 [1.      1.      0.      1.      1.      1.      0.150833
  0.150888 0.482917 0.223267 ]]
```

Labels:

```
[331 131 120 108  82  88 148  68  54  41]
```

After separating the dataset, we now have numpy arrays named **X** containing the features, and **y** containing the labels.

We *could* train a model using all of the data; but it's common practice in supervised learning to split the data into two subsets; a (typically larger) set with which to train the model, and a smaller “hold-back” set with which to validate the trained model. This enables us to evaluate how well the model performs when used with the validation dataset by comparing the predicted labels to the known labels.

To randomly split the data, we'll use the **train_test_split** function in the **scikit-learn** library. This library is one of the most widely used machine learning packages for Python.

```
[30]: from sklearn.model_selection import train_test_split

# 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,
    ↪random_state=0)

print ('Training Set: %d, rows\nTest Set: %d rows' % (X_train.size, X_test.
    ↪size))
```

Training Set: 5110, rows

Test Set: 2200 rows

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. We'll use the **scikit-learn** implementation of the *linear regression* algorithm, a common starting point for regression that works by trying to find a linear relationship between the *X* values and the *y* label. The resulting model is a function that conceptually defines a line where every possible *X* and *y* value combination intersect.

```
[31]: # Train the model
from sklearn.linear_model import LinearRegression

# Fit a linear regression model on the training set
model = LinearRegression(normalize=False).fit(X_train, y_train)
print (model)
```

```
LinearRegression()
```

1.0.3 Evaluate the Trained Model

Now that we've trained the model, we can use it to predict rental counts for the features we held back in our validation dataset. Then we can compare these predictions to the actual label values to evaluate how well (or not!) the model is working.

```
[32]: import numpy as np

predictions = model.predict(X_test)
np.set_printoptions(suppress=True)
print('Predicted labels: ', np.round(predictions)[:10])
print('Actual labels    : ', y_test[:10])
```

```
Predicted labels: [1896. 1184. 1007. -28. 314. 385. 475. 590. 1476. -22.]
Actual labels    : [2418  754  222   47  244  145  240  555 3252   38]
```

Comparing each prediction with its corresponding “ground truth” actual value isn't a very efficient way to determine how well the model is predicting. Let's see if we can get a better indication by visualizing a scatter plot that compares the predictions to the actual labels. We'll also overlay a trend line to get a general sense for how well the predicted labels align with the true labels.

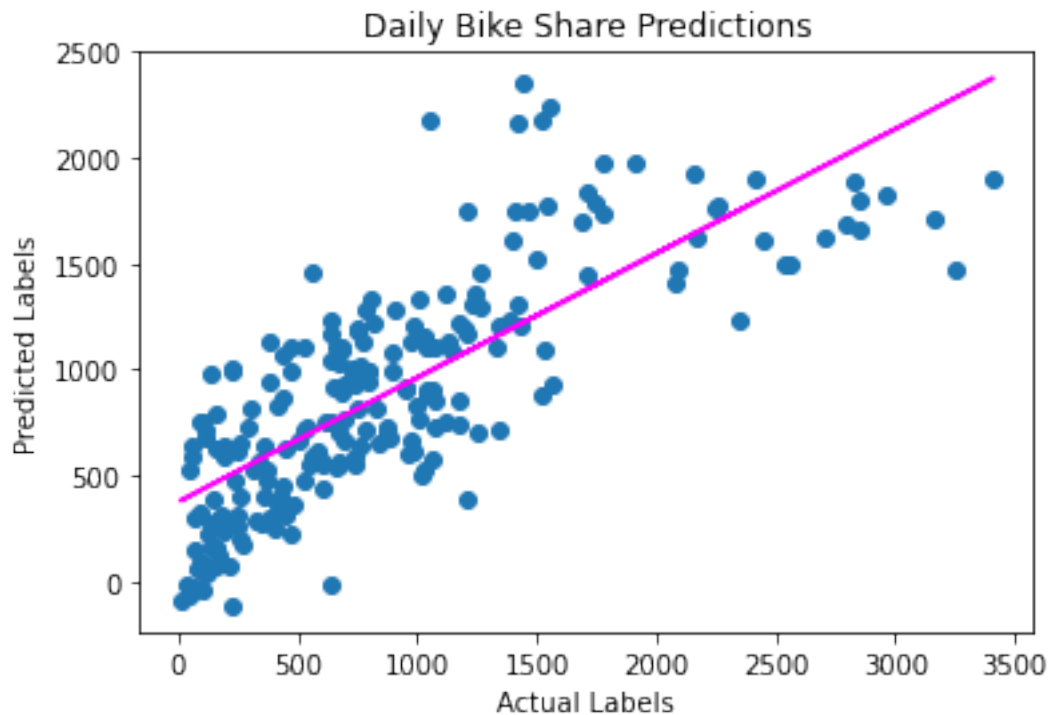
```
[33]: import matplotlib.pyplot as plt

%matplotlib inline

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()
```



There's a definite diagonal trend, and the intersections of the predicted and actual values are generally following the path of the trend line; but there's a fair amount of difference between the ideal function represented by the line and the results. This variance represents the *residuals* of the model - in other words, the difference between the label predicted when the model applies the coefficients it learned during training to the validation data, and the actual value of the validation label. These residuals when evaluated from the validation data indicate the expected level of *error* when the model is used with new data for which the label is unknown.

You can quantify the residuals by calculating a number of commonly used evaluation metrics. We'll focus on the following three:

- **Mean Square Error (MSE):** The mean difference between predicted and actual, squared. This yields a relative metric in which the smaller the value, the better the fit of the model
- **Root Mean Square Error (RMSE):** The square root of the MSE. This yields an absolute metric in the same unit as the label (in this case, numbers of rentals). The smaller the value, the better the model (in a simplistic sense, it represents the average number of rentals by which the predictions are wrong!)
- **Coefficient of Determination (usually known as *R-squared* or **R2**):** A relative metric in which the higher the value, the better the fit of the model. In essence, this metric represents

how much of the variance between predicted and actual label values the model is able to explain.

Note: You can find out more about these and other metrics for evaluating regression models in the [Scikit-Learn documentation](#)

Let's use Scikit-Learn to calculate these metrics for our model, based on the predictions it generated for the validation data.

```
[34]: from sklearn.metrics import mean_squared_error, r2_score

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)
```

```
MSE: 201972.5594703561
RMSE: 449.41357285951665
R2: 0.6040454736919187
```

So now we've quantified the ability of our model to predict the number of rentals. It definitely has *some* predictive power, but we can probably do better!

1.0.4 Preprocess the Data

We trained a model with data that was loaded straight from a source file, with only moderately successful results.

In practice, it's common to perform some preprocessing of the data to make it easier for the algorithm to fit a model to it. There's a huge range of preprocessing transformations you can perform to get your data ready for modeling, but we'll limit ourselves to a few common techniques:

- Scaling numeric features so they're on the same scale. This prevents features with large values from producing coefficients that disproportionately affect the predictions.
- Encoding categorical variables. For example, by using a *one hot encoding* technique you can create individual binary (true/false) features for each possible category value.

To apply these preprocessing transformations, we'll make use of a Scikit-Learn feature named *pipelines*. These enable us to define a set of preprocessing steps that end with an algorithm. You can then fit the entire pipeline to the data, so that the model encapsulates all of the preprocessing steps as well as the regression algorithm. This is useful, because when we want to use the model to predict values from new data, we need to apply the same transformations (based on the same statistical distributions and category encodings used with the training data).

Note: The term *pipeline* is used extensively in machine learning, often to mean very different things! In this context, we're using it to refer to pipeline objects in Scikit-Learn, but you may see it used elsewhere to mean something else.

```
[35]: # Train the model
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LinearRegression
import numpy as np

# Define preprocessing for numeric columns (scale them)
numeric_features = [6,7,8,9]
numeric_transformer = Pipeline(steps=[
    ('scaler', StandardScaler())])

# Define preprocessing for categorical features (encode them)
categorical_features = [0,1,2,3,4,5]
categorical_transformer = Pipeline(steps=[
    ('onehot', OneHotEncoder(handle_unknown='ignore'))])

# Combine preprocessing steps
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)])

# Create preprocessing and training pipeline
pipeline = Pipeline(steps=[('preprocessor', preprocessor),
    ('regressor', LinearRegression(normalize=False))])

# fit the pipeline to train a linear regression model on the training set
model = pipeline.fit(X_train, (y_train))
print (model)
```

```
Pipeline(steps=[('preprocessor',
    ColumnTransformer(transformers=[('num',
    Pipeline(steps=[('scaler',
StandardScaler())]),
    [6, 7, 8, 9]),
    ('cat',
    Pipeline(steps=[('onehot',
OneHotEncoder(handle_unknown='ignore'))]),
    [0, 1, 2, 3, 4, 5]))]),
    ('regressor', LinearRegression())])
```

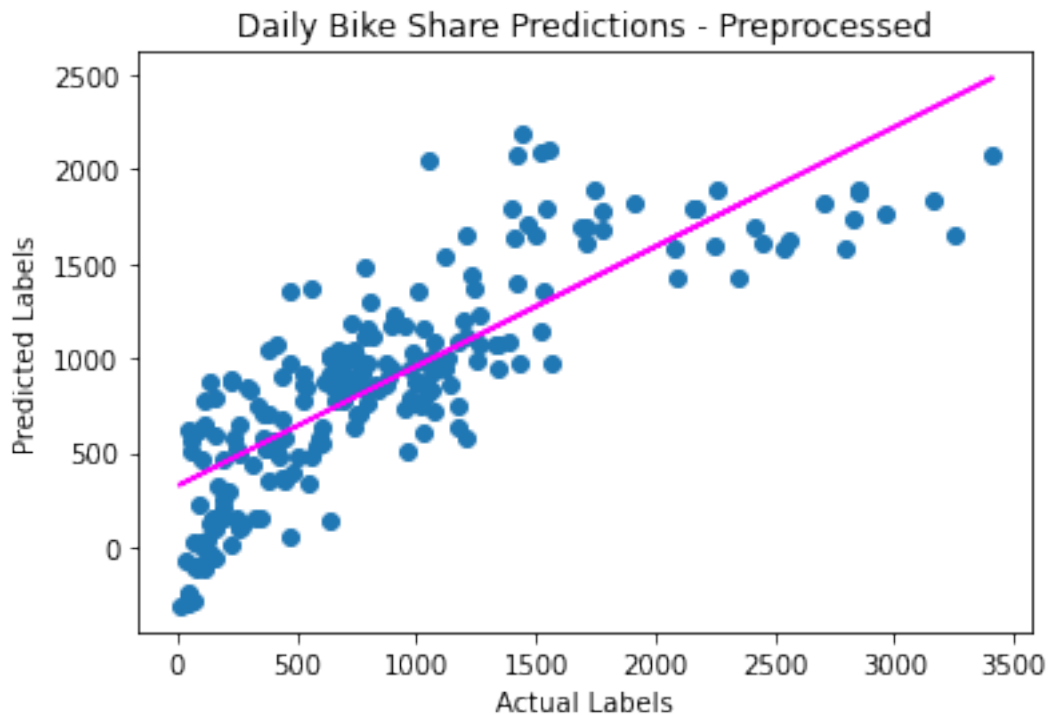
OK, the model is trained, including the preprocessing steps. Let's see how it performs with the validation data.

```
[36]: predictions = model.predict(X_test)
np.set_printoptions(suppress=True)
print('Predicted labels: ', np.round(predictions)[:10])
print('Actual labels: ', y_test[:10])

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)

plt.scatter(y_test, predictions)
plt.xlabel('Actual Labels')
plt.ylabel('Predicted Labels')
plt.title('Daily Bike Share Predictions - Preprocessed')
z = np.polyfit(y_test, predictions, 1)
p = np.poly1d(z)
plt.plot(y_test, p(y_test), color='magenta')
plt.show()
```

Predicted labels: [1694. 985. 894. -303. 150. 150. 581. 474. 1659. -77.]
 Actual labels: [2418 754 222 47 244 145 240 555 3252 38]
 MSE: 174159.78007740033
 RMSE: 417.32455005355285
 R2: 0.658570682060461



It's a little better - geat!

1.0.5 Try a Different Algorithm

The linear regression algorithm we used 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.

Let's try training our regression model by using an ensemble algorithm called **Gradient Boosting**. We can do this by just changing the last step of the pipeline.

```
[37]: from sklearn.ensemble import GradientBoostingRegressor

pipeline = Pipeline(steps=[('preprocessor', preprocessor),
                           ('regressor', GradientBoostingRegressor())])

# train a logistic regression model on the training set
model = pipeline.fit(X_train, (y_train))
print (model)
```

```
Pipeline(steps=[('preprocessor',
                 ColumnTransformer(transformers=[('num',
                                                  Pipeline(steps=[('scaler',
                                                                    StandardScaler())]),
                                                                    [6, 7, 8, 9]),
                                                                    ('cat',
                                                                     Pipeline(steps=[('onehot',
                                                                 OneHotEncoder(handle_unknown='ignore'))]),
                                                                    [0, 1, 2, 3, 4, 5])])),
                 ('regressor', GradientBoostingRegressor())])
```

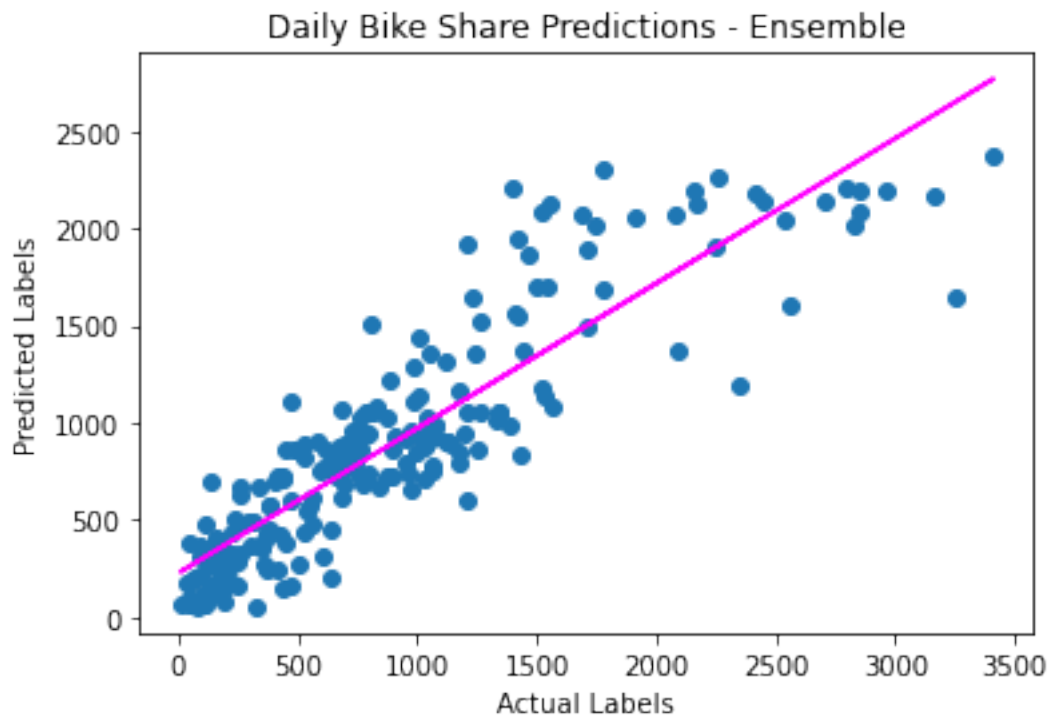
OK, let's see how our ensemble model performs.

```
[38]: 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)
```

```
plt.scatter(y_test, predictions)
plt.xlabel('Actual Labels')
plt.ylabel('Predicted Labels')
plt.title('Daily Bike Share Predictions - Ensemble')
z = np.polyfit(y_test, predictions, 1)
p = np.poly1d(z)
plt.plot(y_test, p(y_test), color='magenta')
plt.show()
```

MSE: 105665.43247669803
 RMSE: 325.06219785865295
 R2: 0.7928495515768801



There's certainly some improvement!

In a real project, you'd likely try a few more algorithms along with more preprocessing; but by now you should have got the general idea. Let's explore how you can use the trained model with new data.

1.0.6 Use the Trained Model

First, let's save the model.

```
[39]: import joblib

# Save the model as a pickle file
filename = './models/bike-share.pkl'
joblib.dump(model, filename)
```

```
[39]: ['./models/bike-share.pkl']
```

Now, we can load it whenever we need it, and use it to predict labels for new data. This is often called *scoring* or *inferencing*.

```
[40]: # Load the model from the file
loaded_model = joblib.load(filename)

# Create a numpy array containing a new observation (for example tomorrow's
↪seasonal and weather forecast information)
X_new = np.array([[1,1,0,3,1,1,0.226957,0.22927,0.436957,0.1869]]).
↪astype('float64')
print('New sample: {}'.format(list(X_new[0])))

# Use the model to predict tomorrow's rentals
result = loaded_model.predict(X_new)
print('Prediction: {:.0f} rentals'.format(np.round(result[0])))
```

```
New sample: [1.0, 1.0, 0.0, 3.0, 1.0, 1.0, 0.226957, 0.22927, 0.436957, 0.1869]
Prediction: 122 rentals
```

1.1 Learn More

To learn more about Scikit-Learn, see the [Scikit-Learn documentation](#).

1.2 Citation

The data used in this exercise is derived from [Capital Bikeshare](#) and is used in accordance with the published [license agreement](#).