2_Train_and_evaluate_a_regression_model

September 5, 2021

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 $\boldsymbol{y}$ represents the label we want to predict and $\boldsymbol{x}$ 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:

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

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.

> **Citation**: The data used in this exercise is derived from Capital Bikeshare and is used in accordance with the published license agreement.

1.1 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.

[1]:
```
import pandas as pd

bike_data = pd.read_csv('daily-bike-share.csv')
bike_data.head()
```

[1]:

	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.

[2]:
```
bike_data['day']= pd.DatetimeIndex(bike_data['dteday']).day
bike_data.head(32)
```

[2]:

	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.

[3]:
```
numeric_features = ['temp', 'atemp', 'hum', 'windspeed']
bike_data[numeric_features + ['rentals']].describe()
```

[3]:

	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
```

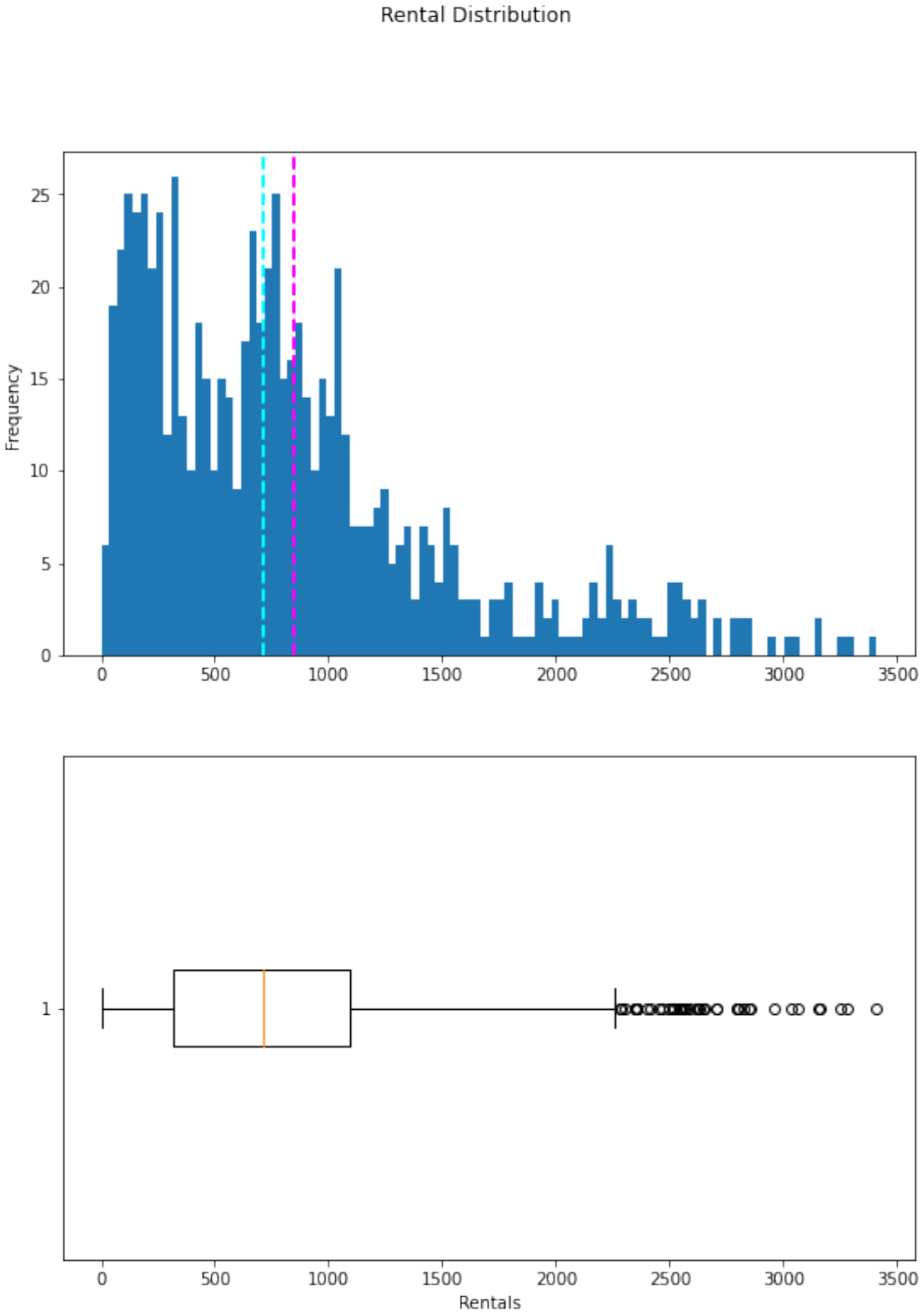
The statistics reveal some information about the distribution of the data in each of the numeric fields, including the number of observations (there are 731 records), the mean, standard deviation, minimum and maximum values, and the quartile values (the threshold values for 25%, 50% - which is also the 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.

[7]:
```
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()
```

```
C:\Users\aduzo\Anaconda3\lib\site-packages\ipykernel_launcher.py:30:
```

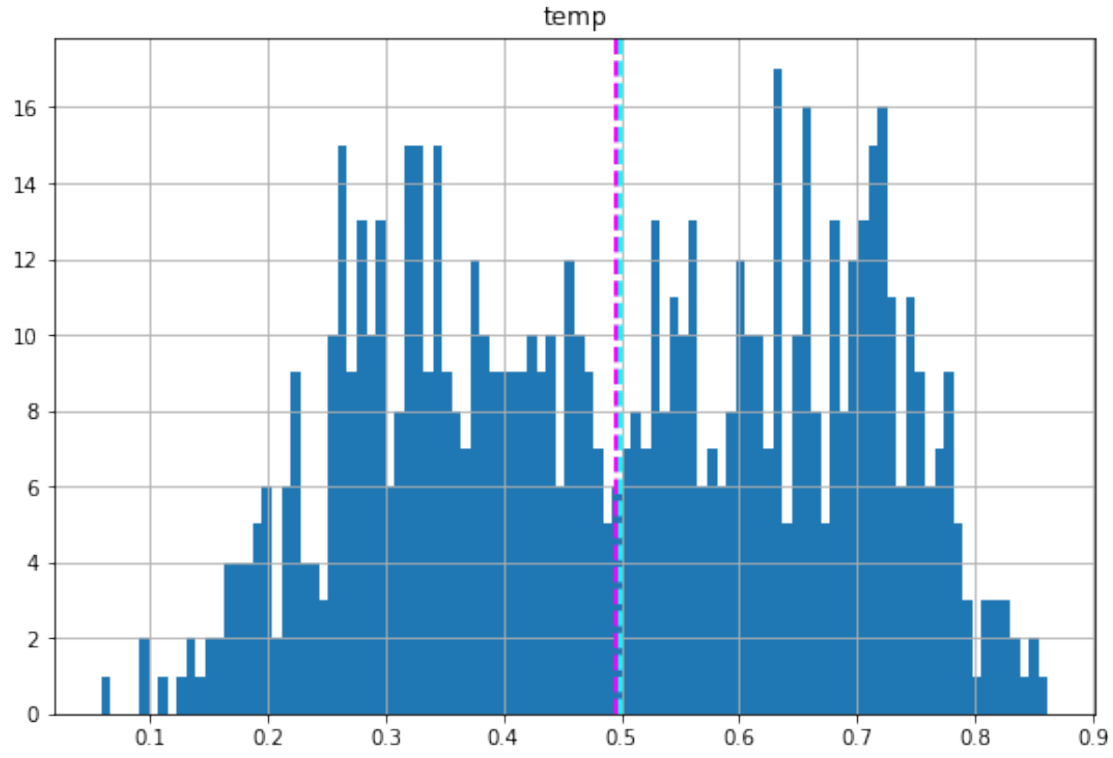
```
UserWarning: Matplotlib is currently using
module://ipykernel.pylab.backend_inline, which is a non-GUI backend, so cannot
show the figure.
```

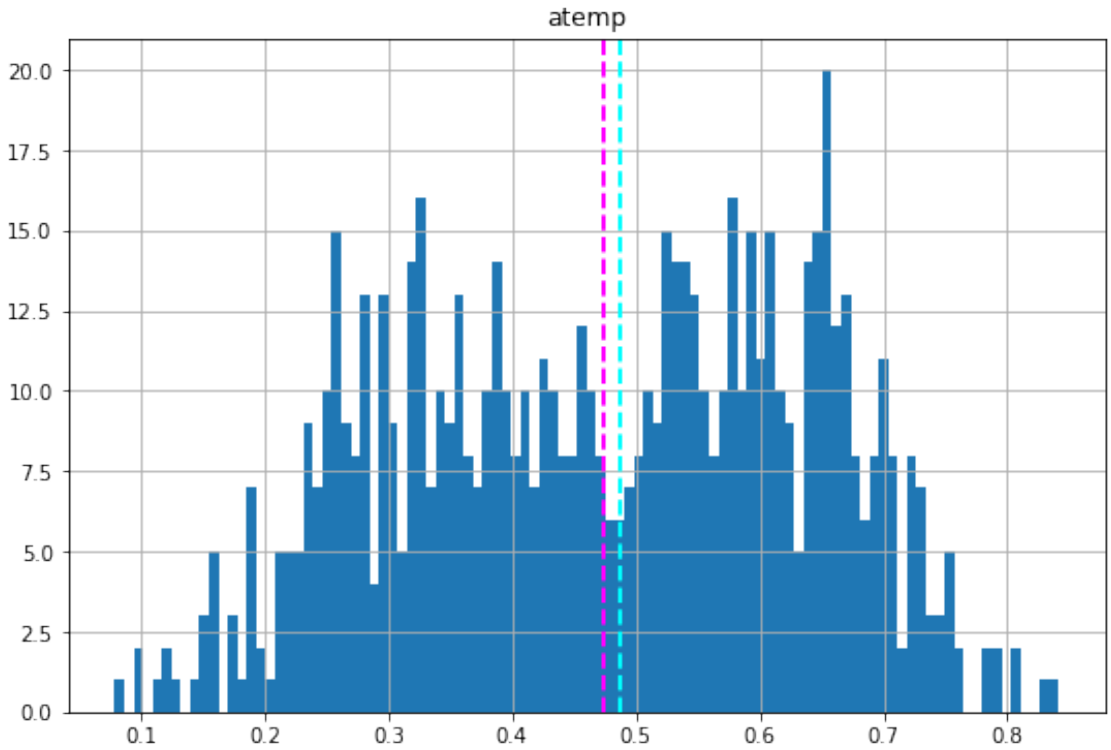


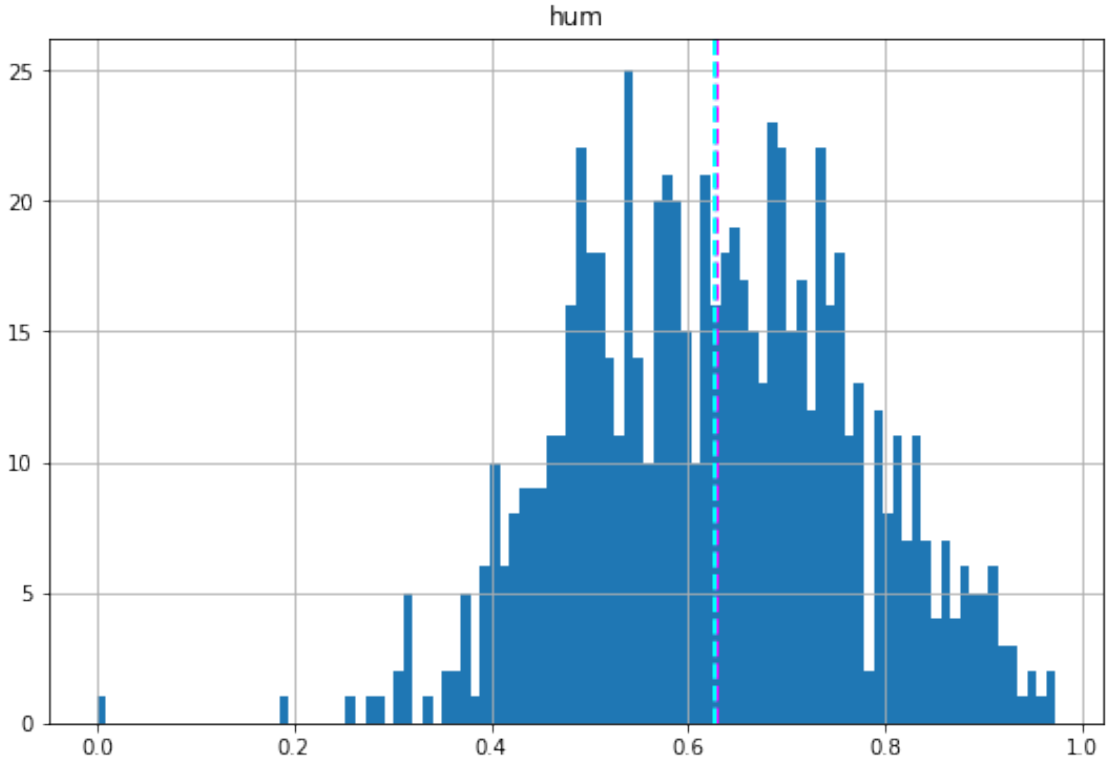
The plots show that the number of daily rentals ranges from 0 to just over 3,400. However, the mean (and median) number of daily rentals is closer to the low end of that range, with most of the data between 0 and around 2,200 rentals. The few values above this are shown in the box plot as small circles,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

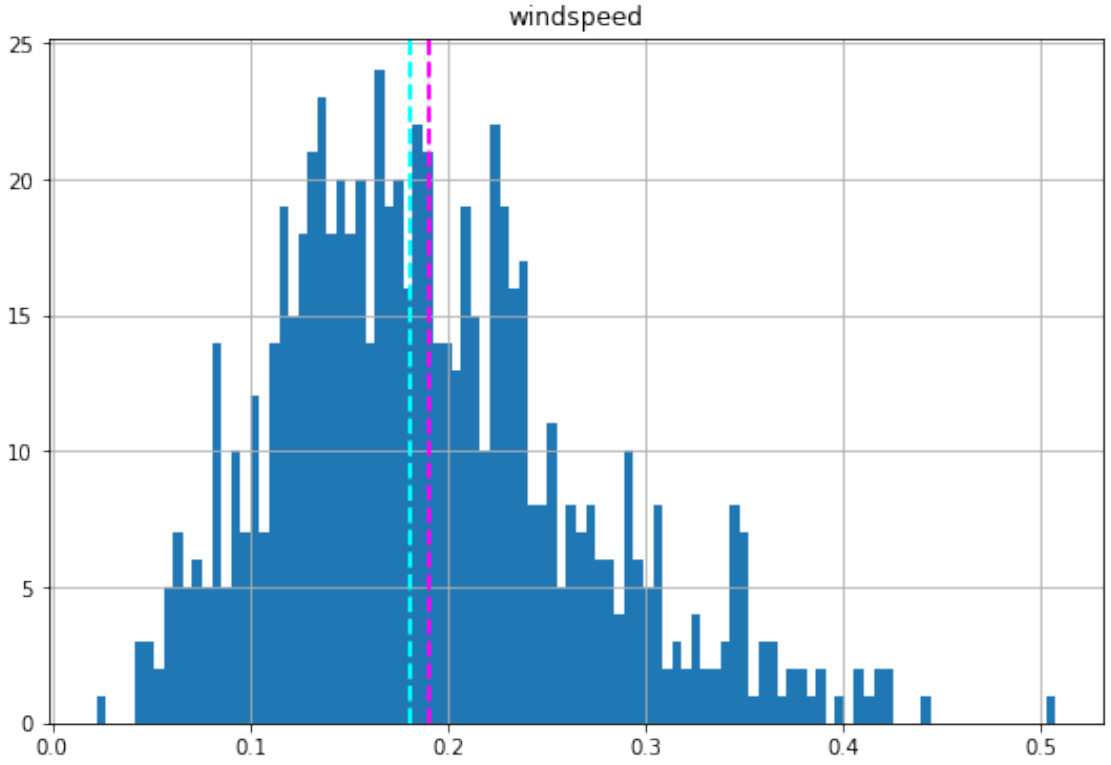
We can do the same kind of visual exploration of the numeric features. Let's create a histogram for each of these.

[8]:
```
# 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()
```




atemp
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8


hum
25
20
15
10
5
0
0.0
0.2
0.4
0.6
0.8
1.0



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.

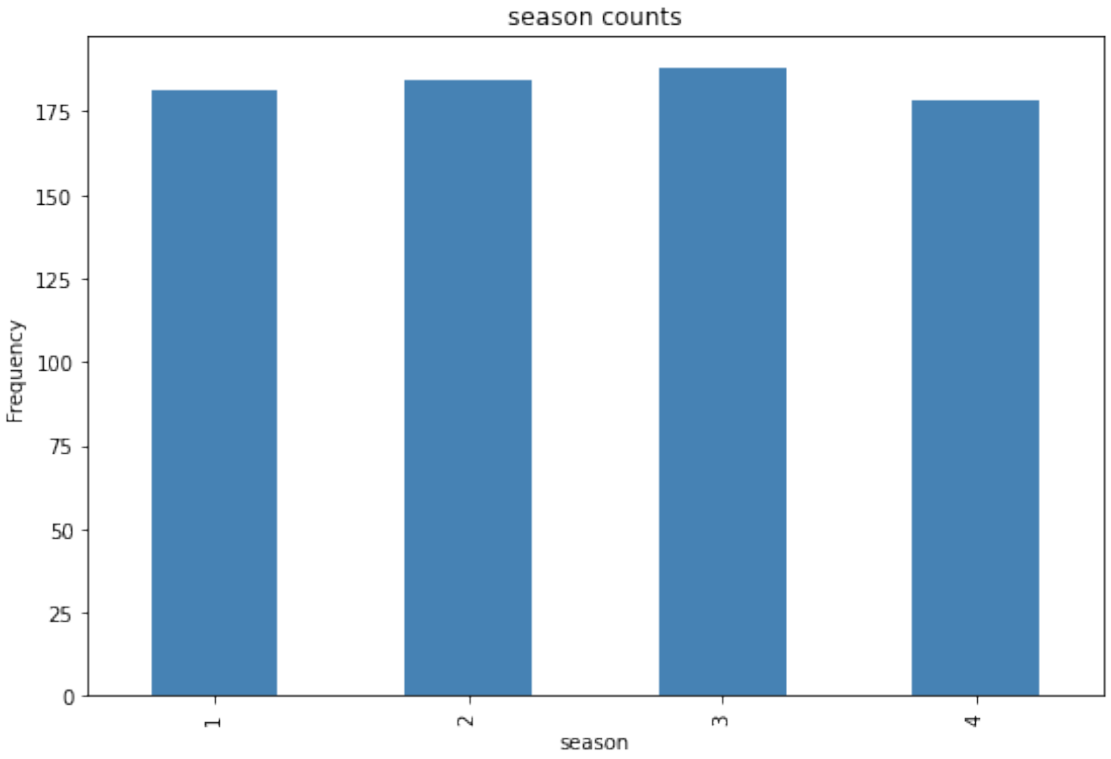
We've explored the distribution of the numeric values in the dataset, but what about the categorical features? These aren't continuous numbers on a scale, so we can't use histograms; but we can plot a bar chart showing the count of each discrete value for each category.

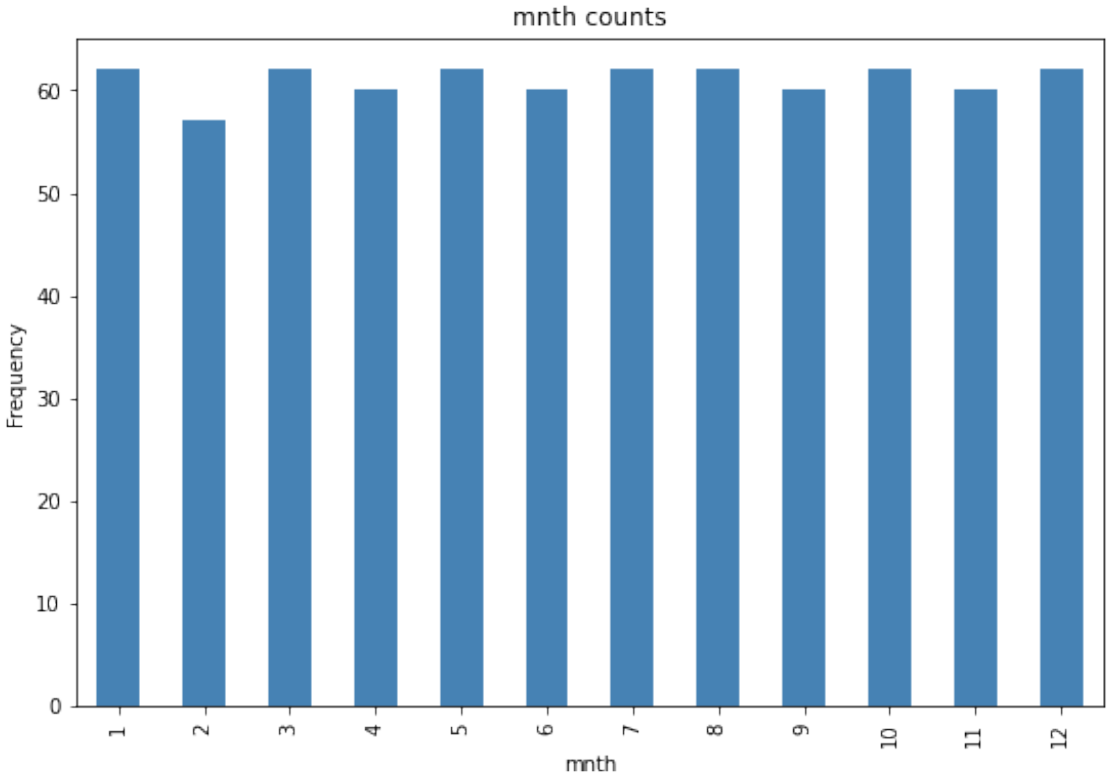
[9]:
```
bike_data['season'].value_counts().sort_index()
```

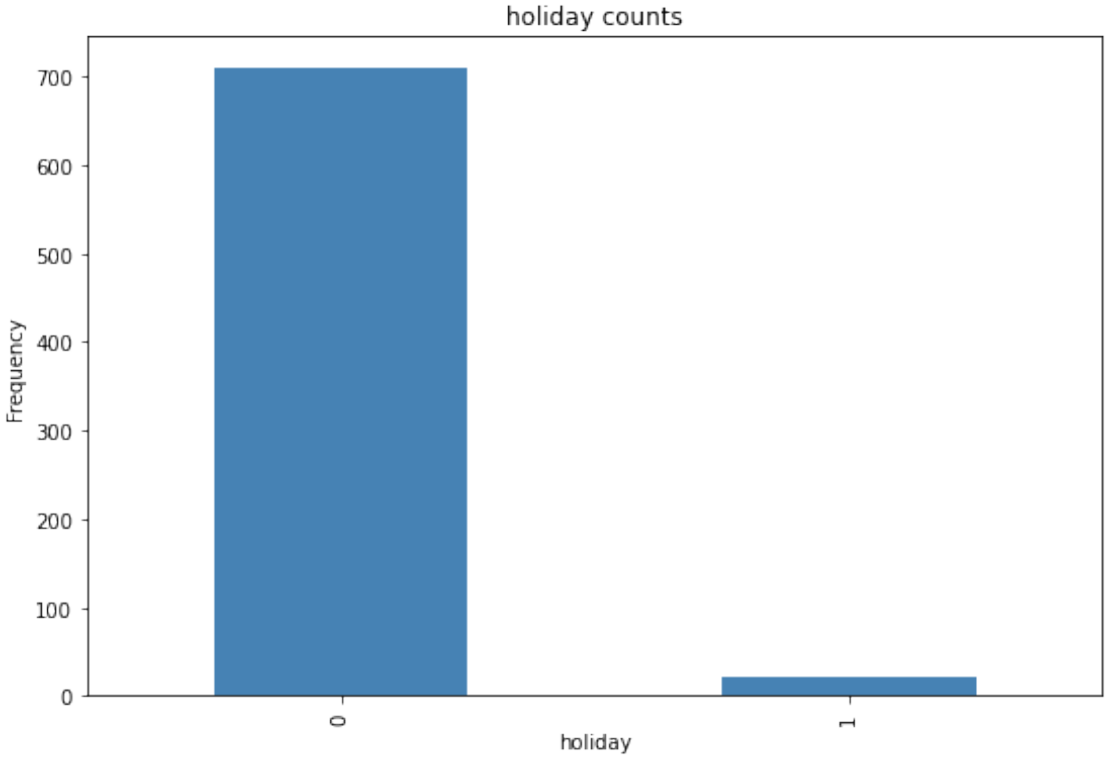
[9]:
```
1    181
2    184
3    188
4    178
Name: season, dtype: int64
```

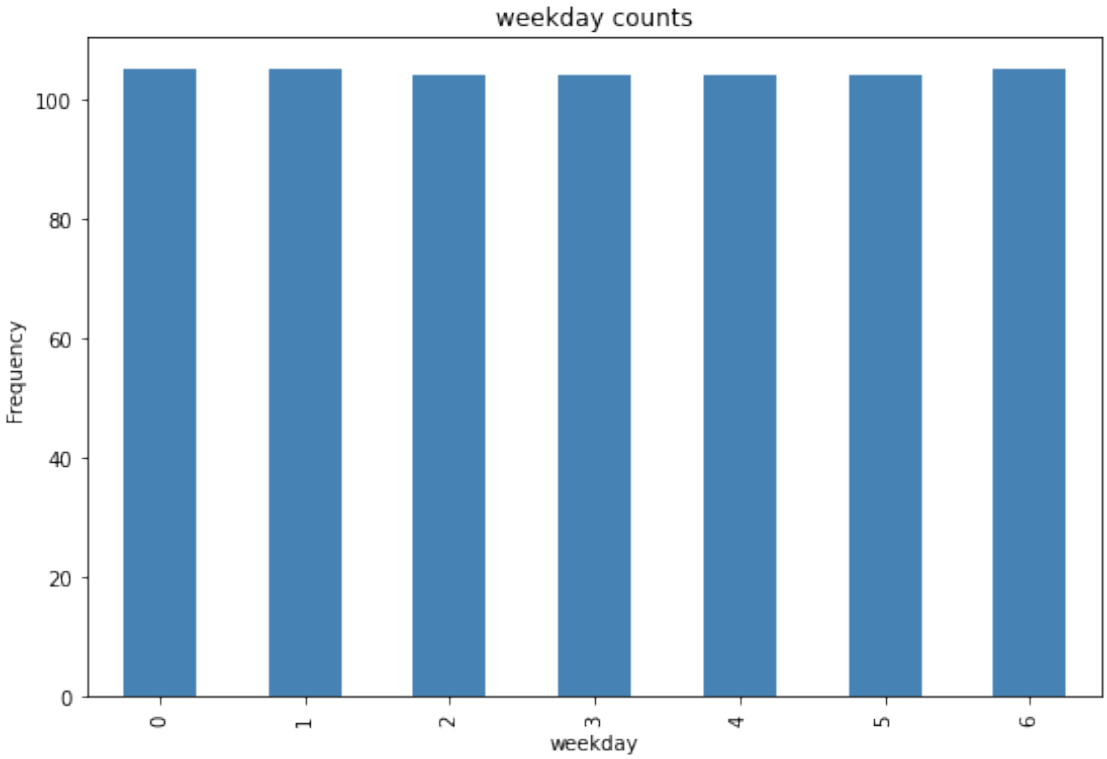
[14]:
```
import numpy as np
# plot a bar plot for each categorical feature count
```

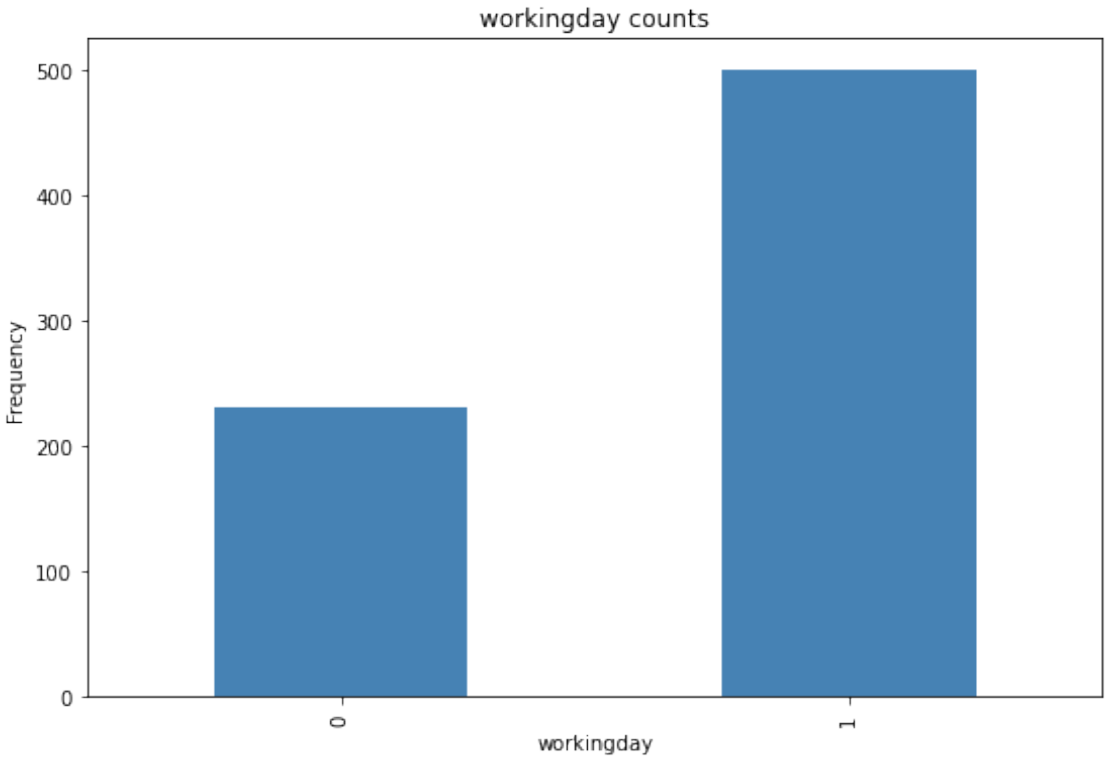
```
categorical_features =␣
 ↪['season','mnth','holiday','weekday','workingday','weathersit', 'day']

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(9)
```

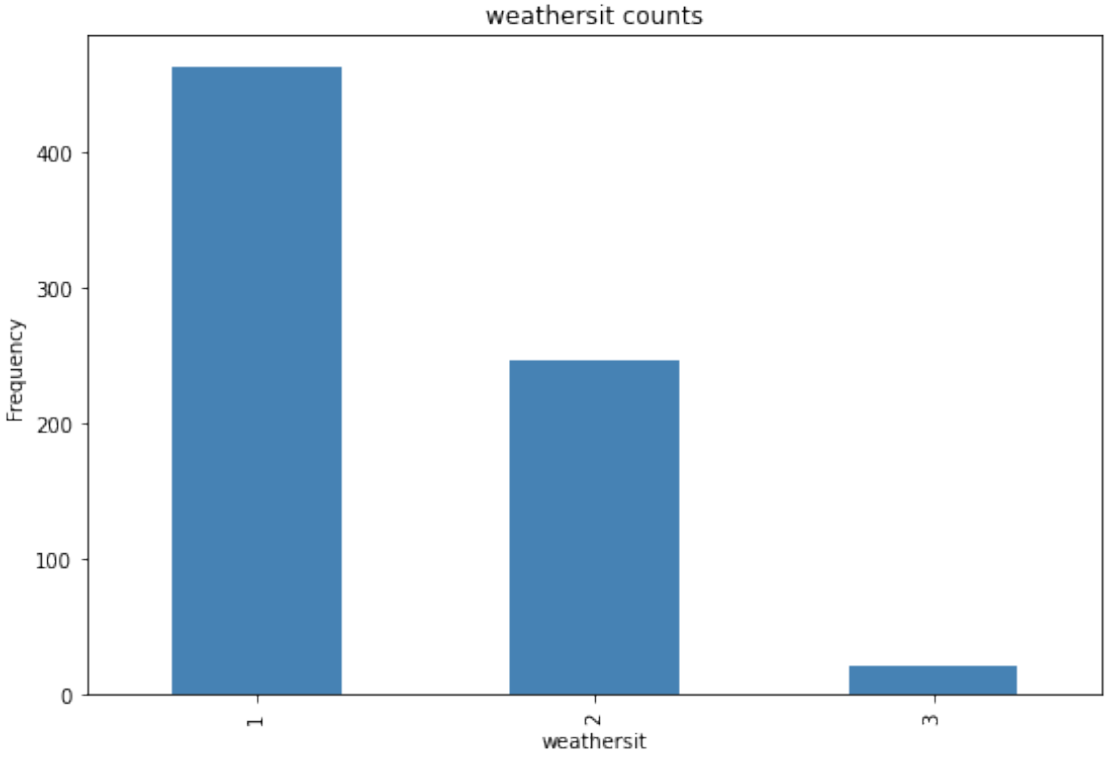


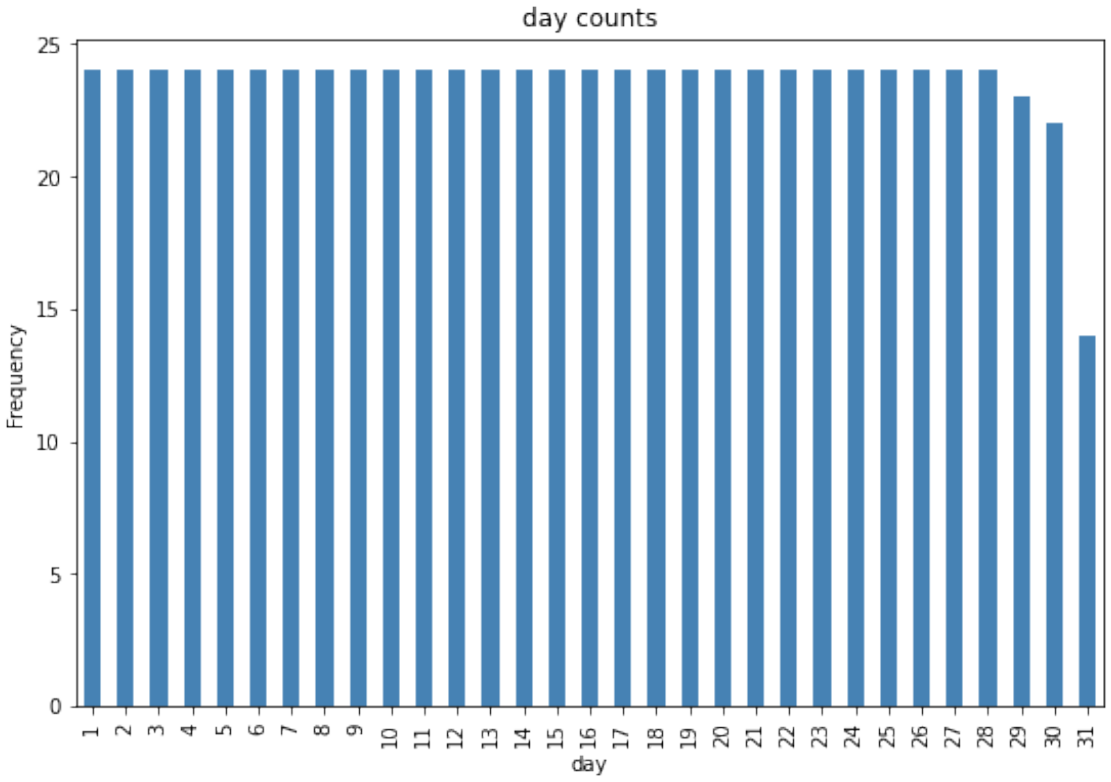

mnth counts
Frequency
0
10
20
30
40
50
60
1
2
3
4
5
6
7
8
9
10
11
12
mnth


holiday counts
Frequency
0
100
200
300
400
500
600
700
0
1
holiday


weekday counts
Frequency
0
20
40
60
80
100
0
1
2
3
4
5
6
weekday


workingday counts
Frequency
500
400
300
200
100
0
0
1
workingday


weathersit counts
Frequency
400
300
200
100
0
1
2
3
weathersit



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:

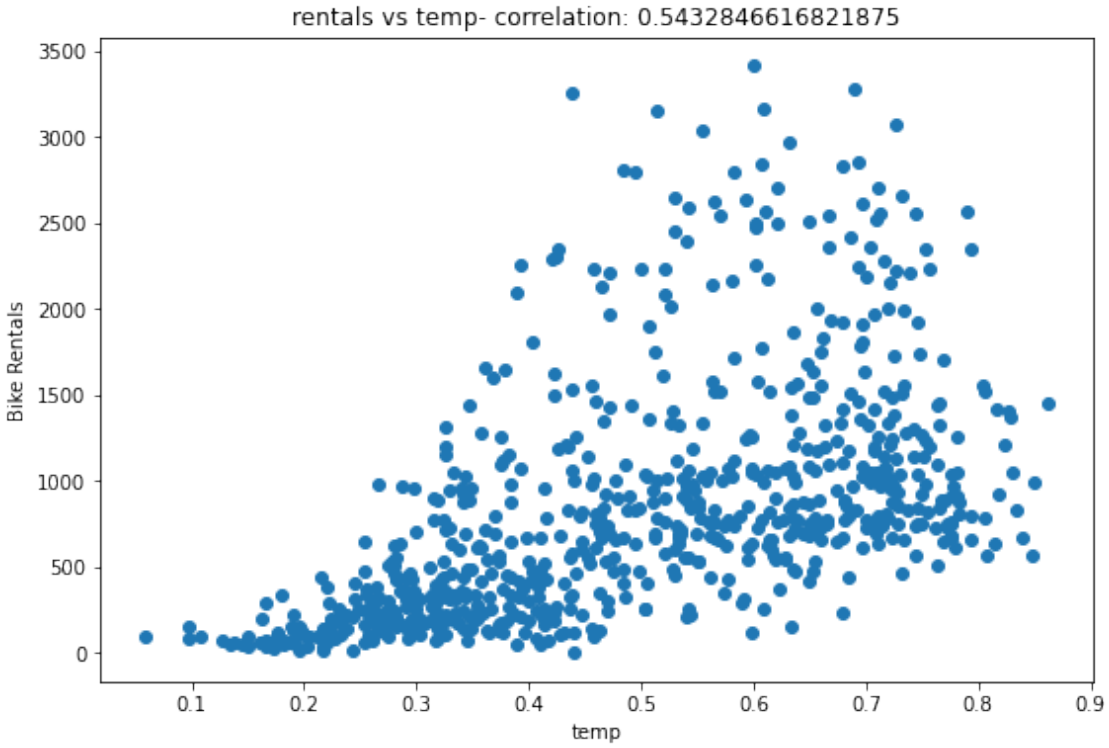
- **holiday**: There are many fewer days that are holidays than days that aren't.
- **workingday**: There are more working days than non-working days.
- **weathersit**: Most days are category *1* (clear), with category *2* (mist and cloud) the next most common. There are comparatively few category *3*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

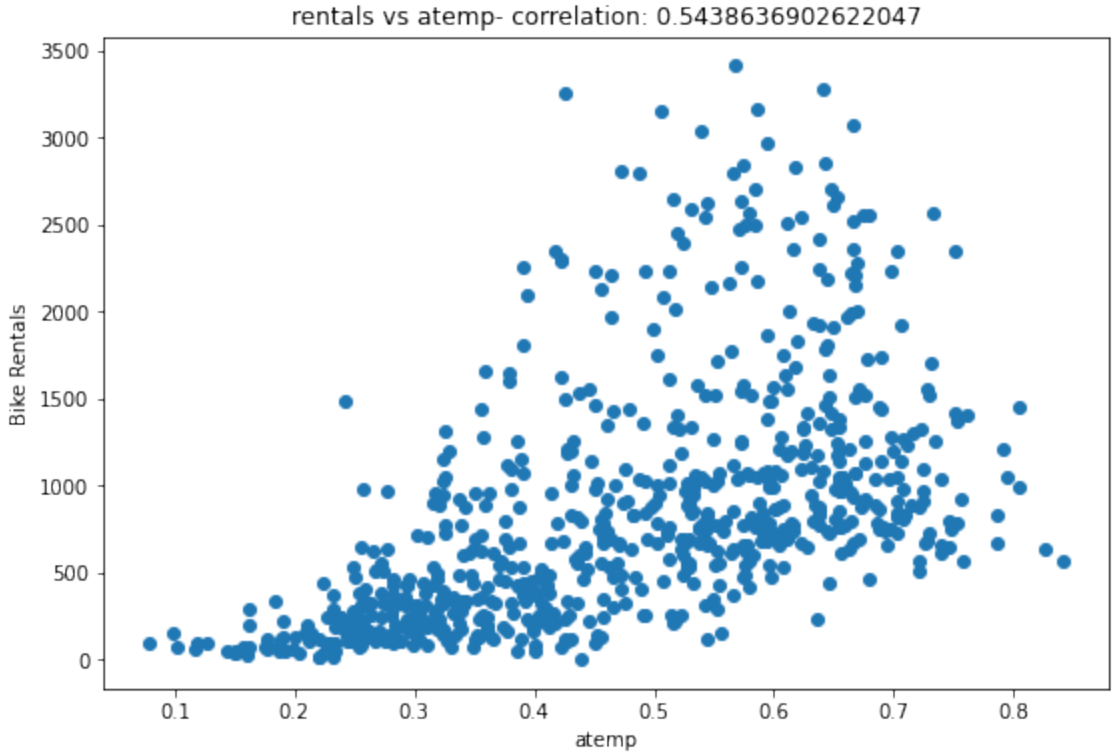
Now that we know something about the distribution of the data in our columns, we can start to look for relationships between the features and the **rentals** label we want to be able to predict.

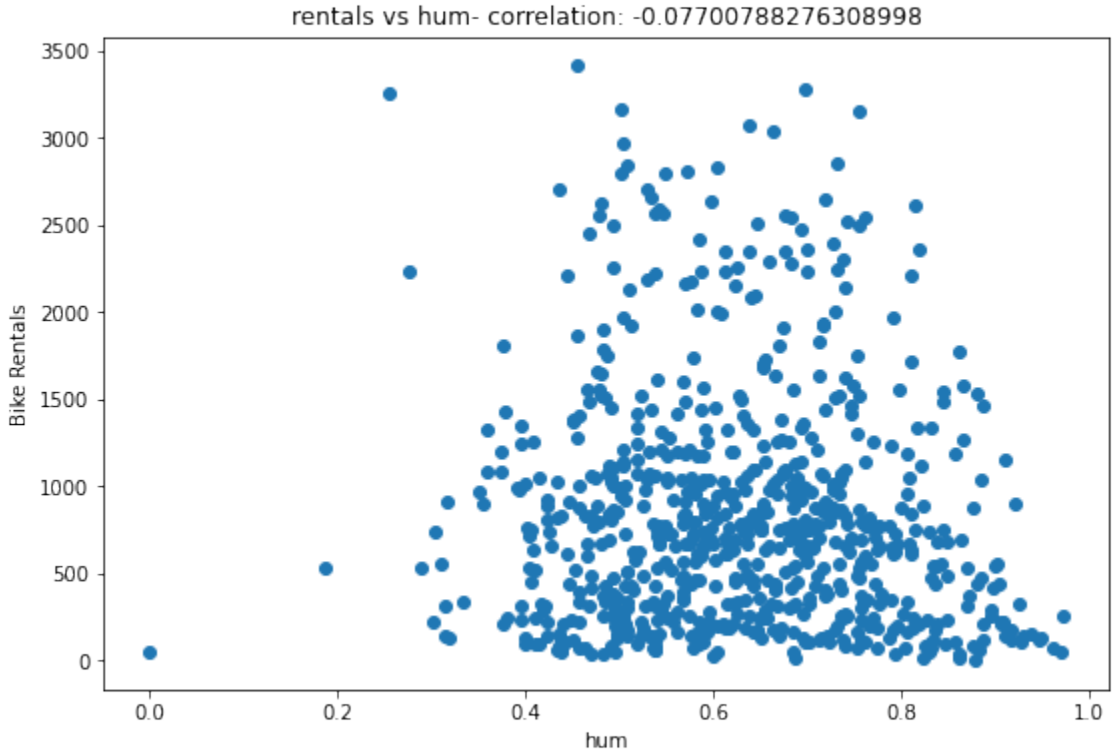
For the numeric features, we can create scatter plots that show the intersection of feature and label values. We can also calculate the *correlation* statistic to quantify the apparent relationship..

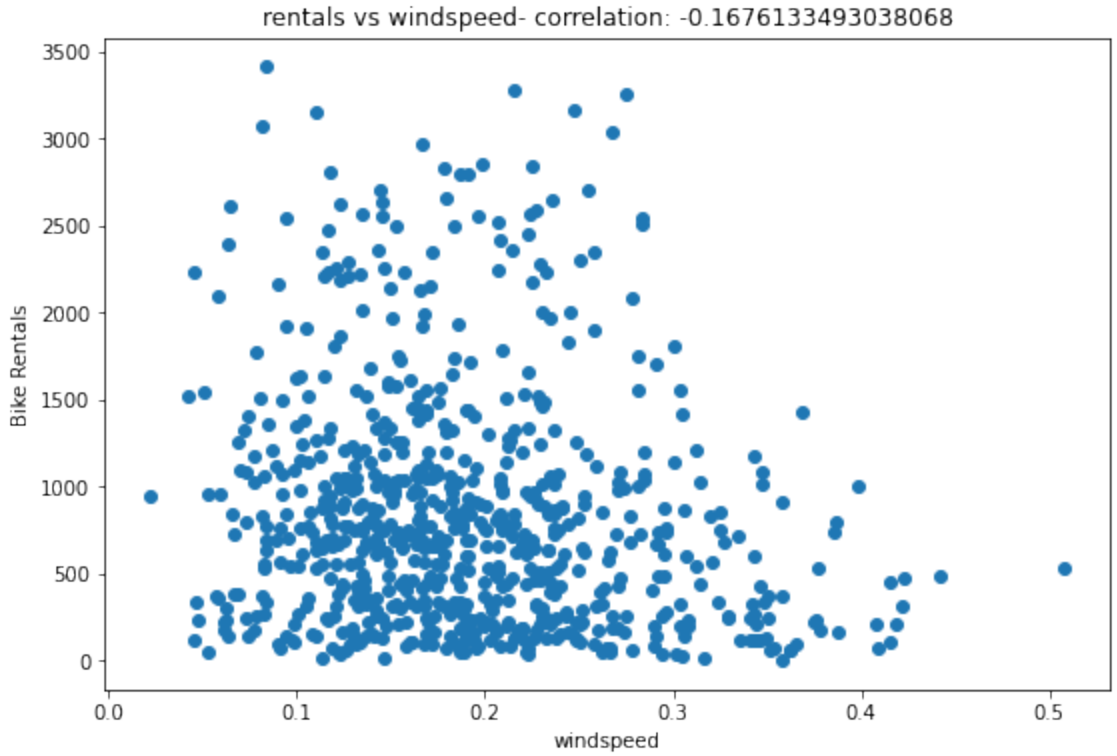
[17]:
```
for col in numeric_features:
    fig = plt.figure(figsize=(9, 6))
    ax = fig.gca()
    feature = bike_data[col]
    label = bike_data['rentals']
    correlation = feature.corr(label)
    plt.scatter(x=feature, y=label)
```

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




rentals vs atemp- correlation: 0.5438636902622047
Bike Rentals
atemp
0
500
1000
1500
2000
2500
3000
3500
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8


rentals vs hum- correlation: -0.07700788276308998
Bike Rentals
hum
0
500
1000
1500
2000
2500
3000
3500
0.0
0.2
0.4
0.6
0.8
1.0



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.

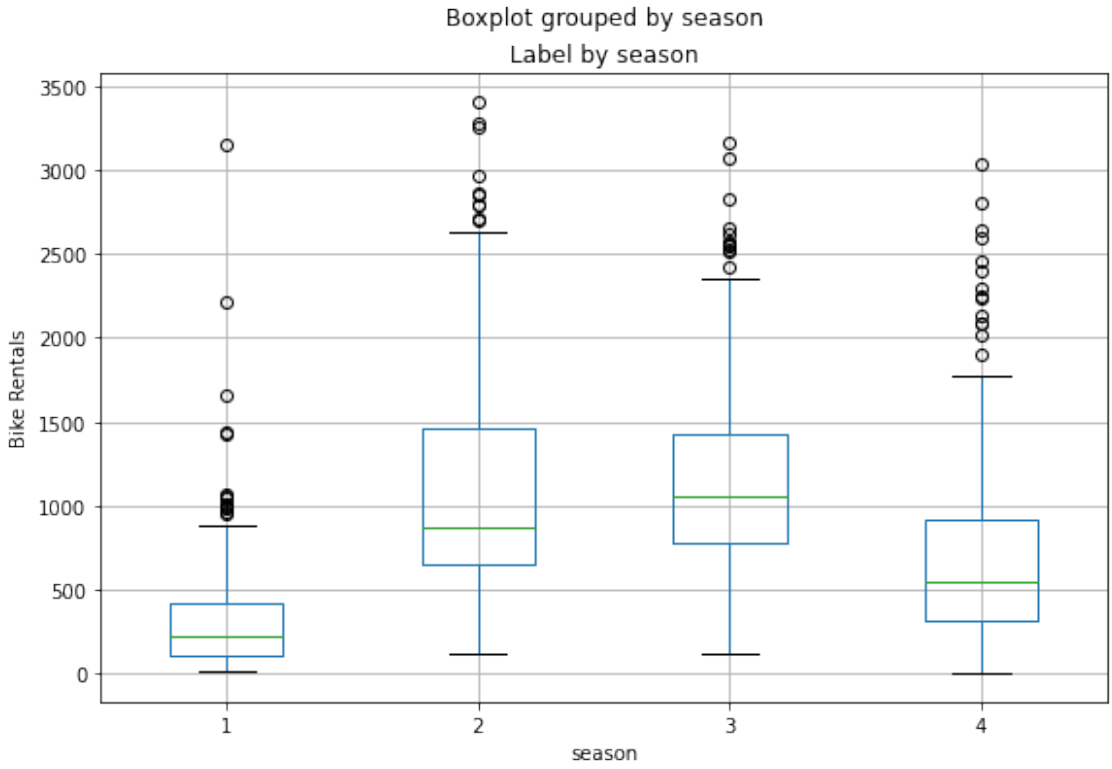
[18]:
```
help(plt.gca)
```

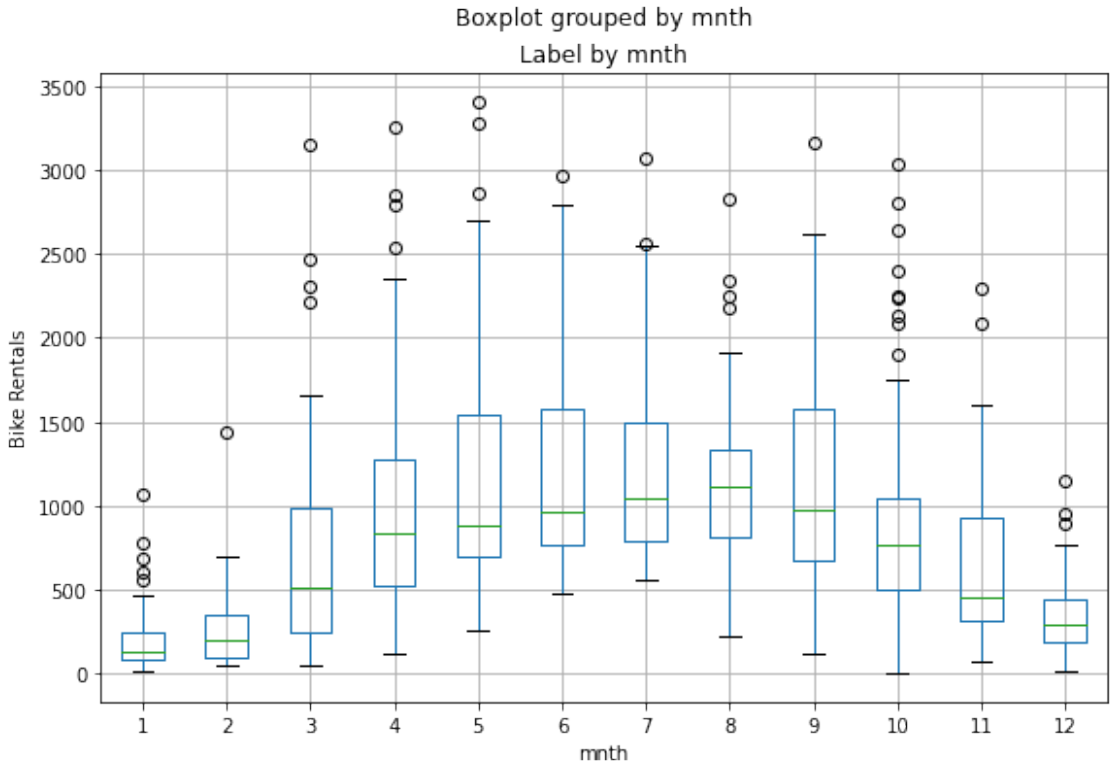
```
Help on function gca in module matplotlib.pyplot:

gca(**kwargs)
    Get the current Axes, creating one if necessary.

    The following kwargs are supported for ensuring the returned Axes
    adheres to the given projection etc., and for Axes creation if
    the active Axes does not exist:

    Properties:
        adjustable: {'box', 'datalim'}
        agg_filter: a filter function, which takes a (m, n, 3) float array and a
```

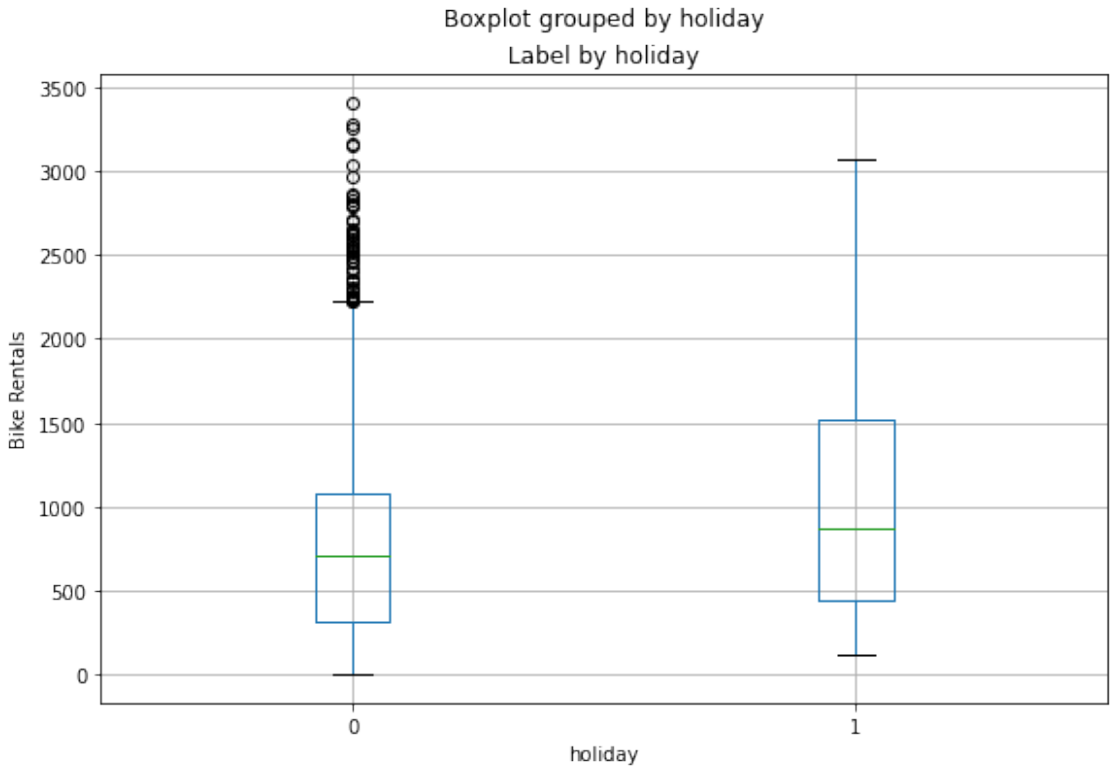
```
dpi value, and returns a (m, n, 3) array
        alpha: scalar or None
        anchor: 2-tuple of floats or {'C', 'SW', 'S', 'SE', …}
        animated: bool
        aspect: {'auto', 'equal'} or float
        autoscale_on: bool
        autoscalex_on: bool
        autoscaley_on: bool
        axes_locator: Callable[[Axes, Renderer], Bbox]
        axisbelow: bool or 'line'
        box_aspect: float or None
        clip_box: `.Bbox`
        clip_on: bool
        clip_path: Patch or (Path, Transform) or None
        contains: unknown
        facecolor or fc: color
        figure: `.Figure`
        frame_on: bool
        gid: str
        in_layout: bool
        label: object
        navigate: bool
        navigate_mode: unknown
        path_effects: `.AbstractPathEffect`
        picker: None or bool or float or callable
        position: [left, bottom, width, height] or `~matplotlib.transforms.Bbox`
        prop_cycle: unknown
        rasterization_zorder: float or None
        rasterized: bool
        sketch_params: (scale: float, length: float, randomness: float)
        snap: bool or None
        title: str
        transform: `.Transform`
        url: str
        visible: bool
        xbound: unknown
        xlabel: str
        xlim: (bottom: float, top: float)
        xmargin: float greater than -0.5
        xscale: {"linear", "log", "symlog", "logit", …} or `.ScaleBase`
        xticklabels: unknown
        xticks: unknown
        ybound: unknown
        ylabel: str
        ylim: (bottom: float, top: float)
        ymargin: float greater than -0.5
        yscale: {"linear", "log", "symlog", "logit", …} or `.ScaleBase`
        yticklabels: unknown
```

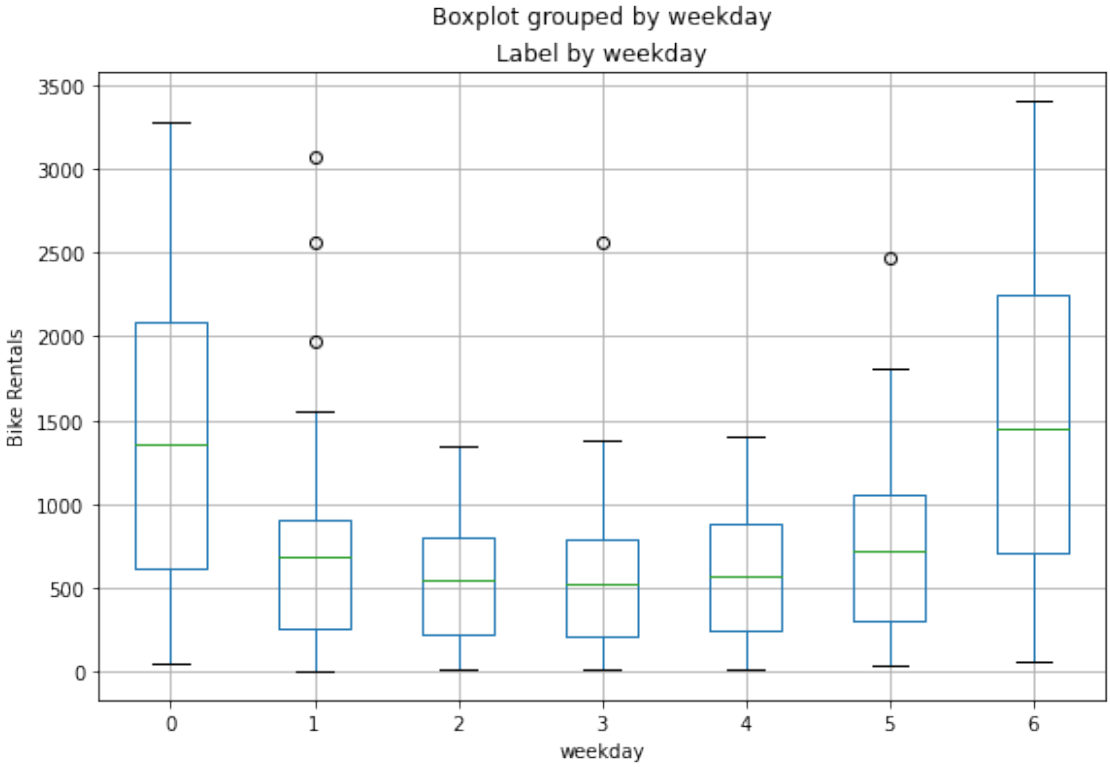
```
        yticks: unknown
        zorder: float
```

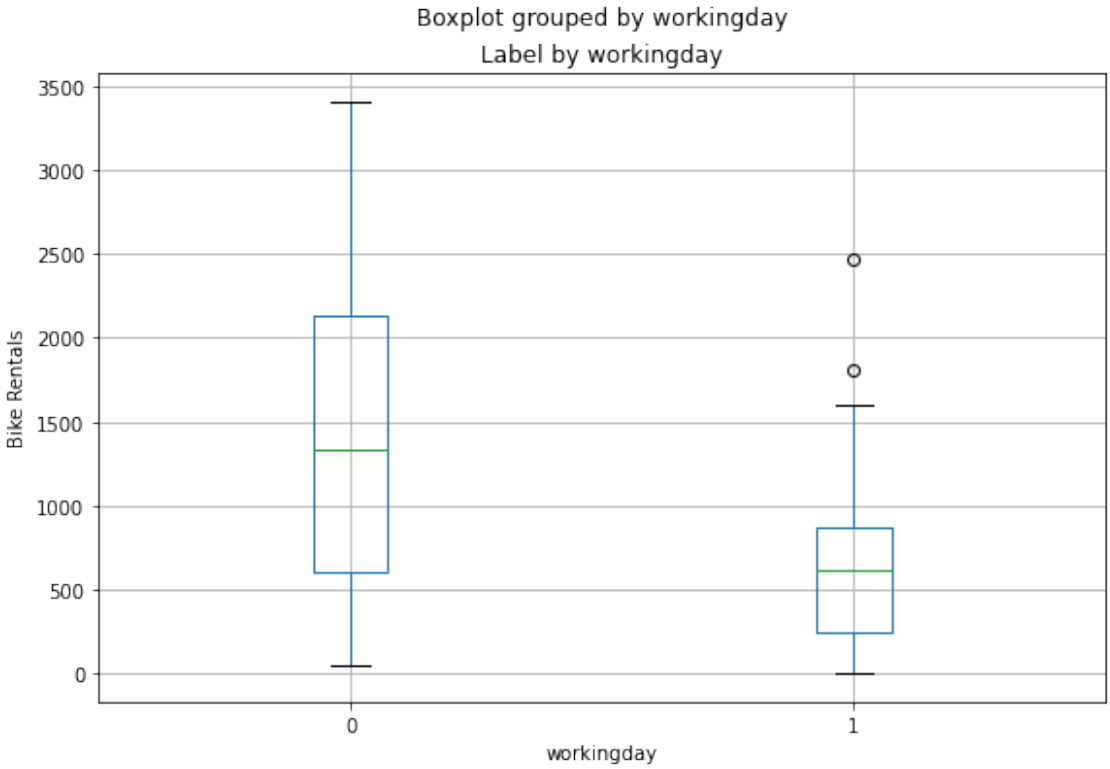
[19]:
```
# plot a boxplot for the label by each categorical feature
for col in categorical_features:
    fig = plt.figure(figsize=(9,6))
    ax = fig.gca()
    bike_data.boxplot(column = 'rentals', by = col, ax =ax)
    ax.set_title('Label by '+ col)
    ax.set_ylabel("Bike Rentals")

plt.show()
```

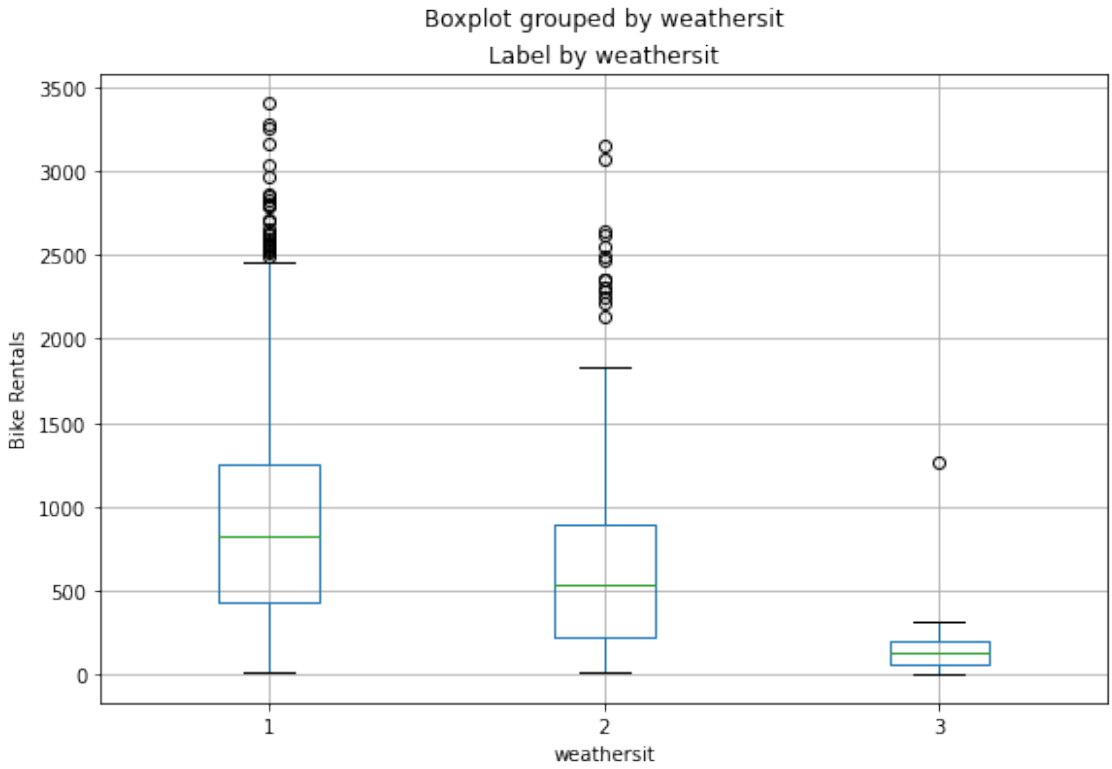


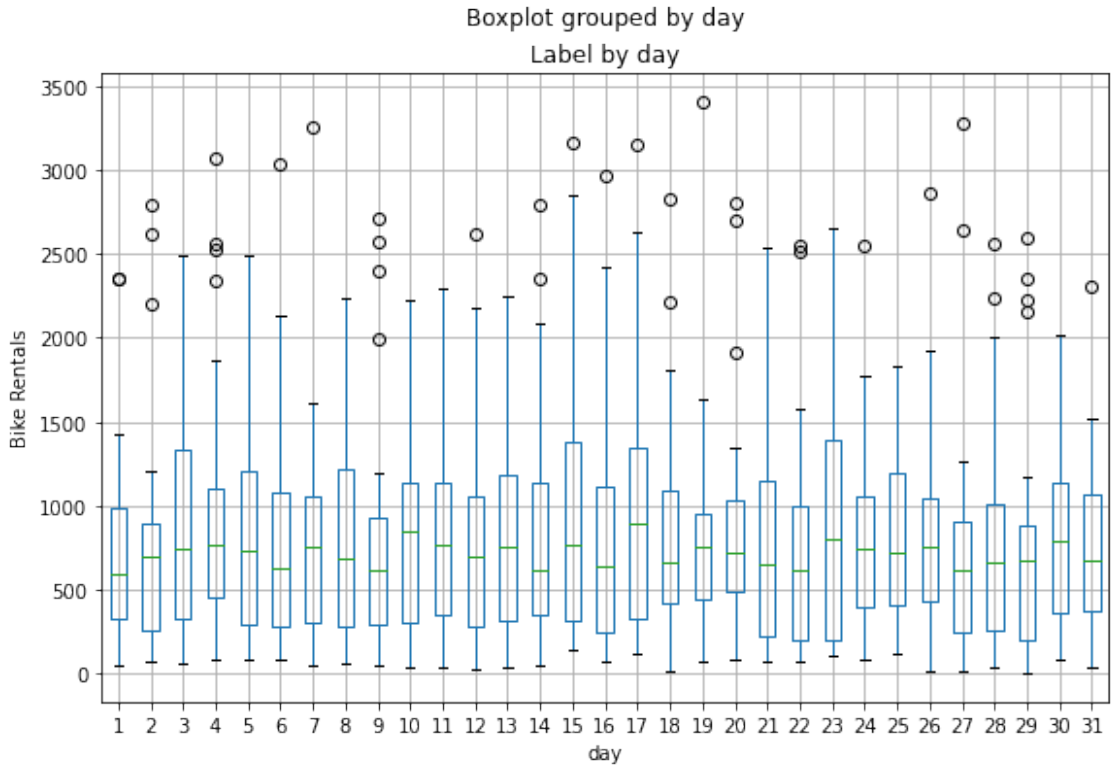

Boxplot grouped by mnth
Label by mnth
Bike Rentals
mnth
0
500
1000
1500
2000
2500
3000
3500
1
2
3
4
5
6
7
8
9
10
11
12


Boxplot grouped by holiday
Label by holiday
Bike Rentals
holiday
0
500
1000
1500
2000
2500
3000
3500
0
1


Boxplot grouped by weekday
Label by weekday
Bike Rentals
3500
3000
2500
2000
1500
1000
500
0
0
1
2
3
4
5
6
weekday


Boxplot grouped by workingday
Label by workingday
Bike Rentals
3500
3000
2500
2000
1500
1000
500
0
0
1
workingday


Boxplot grouped by weathersit
Label by weathersit
Bike Rentals
3500
3000
2500
2000
1500
1000
500
0
1
2
3
weathersit



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.

1.2 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.

[20]:
```
# Seperate 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. It's important to split the data *randomly* (rather than say, taking the first 70% of the data for training and keeping the rest for validation). This helps ensure that the two subsets of data are statistically comparable (so we validate the model with data that has a similar statistical distribution to the data on which it was trained).

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.

[21]:
```
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.shape[0], X_test.
 ↪shape[0]))
```

```
Training Set: 511 rows
Test Set: 220 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 a *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.

In Scikit-Learn, training algorithms are encapsulated in *estimators*, and in this case we'll use the **LinearRegression** estimator to train a linear regression model.

[22]:
```
# train the model
from sklearn.linear_model import LinearRegression

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

```
LinearRegression()
```

1.2.1 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.

[26]:
```
X_test[:3]# remember, only 10 featuree are present, no labels
```

[26]:
```
array([[3.      , 7.      , 0.      , 6.      , 0.      , 1.      ,
        0.686667, 0.638263, 0.585   , 0.208342],
       [3.      , 7.      , 0.      , 4.      , 1.      , 1.      ,
        0.75    , 0.686871, 0.65125 , 0.1592  ],
       [1.      , 1.      , 0.      , 6.      , 0.      , 2.      ,
        0.233333, 0.248112, 0.49875 , 0.157963]])
```

[27]:
```
import numpy as np

# using X_test predict unseen label
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.

[28]:
```
help(np.polyfit)
```

```
Help on function polyfit in module numpy:

polyfit(x, y, deg, rcond=None, full=False, w=None, cov=False)
    Least squares polynomial fit.

    Fit a polynomial ``p(x) = p[0] * x**deg + … + p[deg]`` of degree `deg`
    to points `(x, y)`. Returns a vector of coefficients `p` that minimises
    the squared error in the order `deg`, `deg-1`, … `0`.

    The `Polynomial.fit <numpy.polynomial.polynomial.Polynomial.fit>` class
    method is recommended for new code as it is more stable numerically. See
    the documentation of the method for more information.

    Parameters
    ----------
    x : array_like, shape (M,)
        x-coordinates of the M sample points ``(x[i], y[i])``.
    y : array_like, shape (M,) or (M, K)
        y-coordinates of the sample points. Several data sets of sample
        points sharing the same x-coordinates can be fitted at once by
        passing in a 2D-array that contains one dataset per column.
    deg : int
        Degree of the fitting polynomial
    rcond : float, optional
        Relative condition number of the fit. Singular values smaller than
        this relative to the largest singular value will be ignored. The
        default value is len(x)*eps, where eps is the relative precision of
        the float type, about 2e-16 in most cases.
    full : bool, optional
        Switch determining nature of return value. When it is False (the
        default) just the coefficients are returned, when True diagnostic
        information from the singular value decomposition is also returned.
    w : array_like, shape (M,), optional
        Weights to apply to the y-coordinates of the sample points. For
        gaussian uncertainties, use 1/sigma (not 1/sigma**2).
    cov : bool or str, optional
        If given and not `False`, return not just the estimate but also its
        covariance matrix. By default, the covariance are scaled by
        chi2/sqrt(N-dof), i.e., the weights are presumed to be unreliable
        except in a relative sense and everything is scaled such that the
        reduced chi2 is unity. This scaling is omitted if ``cov='unscaled'``,
```

```
    as is relevant for the case that the weights are 1/sigma**2, with
    sigma known to be a reliable estimate of the uncertainty.

Returns
-------
p : ndarray, shape (deg + 1,) or (deg + 1, K)
    Polynomial coefficients, highest power first.  If `y` was 2-D, the
    coefficients for `k`-th data set are in ``p[:,k]``.

residuals, rank, singular_values, rcond
    Present only if `full` = True.  Residuals is sum of squared residuals
    of the least-squares fit, the effective rank of the scaled Vandermonde
    coefficient matrix, its singular values, and the specified value of
    `rcond`. For more details, see `linalg.lstsq`.

V : ndarray, shape (M,M) or (M,M,K)
    Present only if `full` = False and `cov`=True.  The covariance
    matrix of the polynomial coefficient estimates.  The diagonal of
    this matrix are the variance estimates for each coefficient.  If y
    is a 2-D array, then the covariance matrix for the `k`-th data set
    are in ``V[:,:,k]``


Warns
-----
RankWarning
    The rank of the coefficient matrix in the least-squares fit is
    deficient. The warning is only raised if `full` = False.

    The warnings can be turned off by

    >>> import warnings
    >>> warnings.simplefilter('ignore', np.RankWarning)

See Also
--------
polyval : Compute polynomial values.
linalg.lstsq : Computes a least-squares fit.
scipy.interpolate.UnivariateSpline : Computes spline fits.

Notes
-----
The solution minimizes the squared error

.. math ::
    E = \sum_{j=0}^k |p(x_j) - y_j|^2

in the equations::
```

```
    x[0]**n * p[0] + … + x[0] * p[n-1] + p[n] = y[0]
    x[1]**n * p[0] + … + x[1] * p[n-1] + p[n] = y[1]
    …
    x[k]**n * p[0] + … + x[k] * p[n-1] + p[n] = y[k]

The coefficient matrix of the coefficients `p` is a Vandermonde matrix.

`polyfit` issues a `RankWarning` when the least-squares fit is badly
conditioned. This implies that the best fit is not well-defined due
to numerical error. The results may be improved by lowering the polynomial
degree or by replacing `x` by `x` - `x`.mean(). The `rcond` parameter
can also be set to a value smaller than its default, but the resulting
fit may be spurious: including contributions from the small singular
values can add numerical noise to the result.

Note that fitting polynomial coefficients is inherently badly conditioned
when the degree of the polynomial is large or the interval of sample points
is badly centered. The quality of the fit should always be checked in these
cases. When polynomial fits are not satisfactory, splines may be a good
alternative.

References
----------
.. [1] Wikipedia, "Curve fitting",
       https://en.wikipedia.org/wiki/Curve_fitting
.. [2] Wikipedia, "Polynomial interpolation",
       https://en.wikipedia.org/wiki/Polynomial_interpolation

Examples
--------
>>> import warnings
>>> x = np.array([0.0, 1.0, 2.0, 3.0,  4.0,  5.0])
>>> y = np.array([0.0, 0.8, 0.9, 0.1, -0.8, -1.0])
>>> z = np.polyfit(x, y, 3)
>>> z
array([ 0.08703704, -0.81349206,  1.69312169, -0.03968254]) # may vary

It is convenient to use `poly1d` objects for dealing with polynomials:

>>> p = np.poly1d(z)
>>> p(0.5)
0.6143849206349179 # may vary
>>> p(3.5)
-0.34732142857143039 # may vary
>>> p(10)
22.579365079365115 # may vary
```

```
    High-order polynomials may oscillate wildly:

    >>> with warnings.catch_warnings():
    …     warnings.simplefilter('ignore', np.RankWarning)
    …     p30 = np.poly1d(np.polyfit(x, y, 30))
    …
    >>> p30(4)
    -0.80000000000000204 # may vary
    >>> p30(5)
    -0.99999999999999445 # may vary
    >>> p30(4.5)
    -0.10547061179440398 # may vary

    Illustration:

    >>> import matplotlib.pyplot as plt
    >>> xp = np.linspace(-2, 6, 100)
    >>> _ = plt.plot(x, y, '.', xp, p(xp), '-', xp, p30(xp), '--')
    >>> plt.ylim(-2,2)
    (-2, 2)
    >>> plt.show()
```

[29]:
```
help(np.poly1d)
```

```
Help on class poly1d in module numpy:

class poly1d(builtins.object)
 |  poly1d(c_or_r, r=False, variable=None)
 |
 |  A one-dimensional polynomial class.
 |
 |  A convenience class, used to encapsulate "natural" operations on
 |  polynomials so that said operations may take on their customary
 |  form in code (see Examples).
 |
 |  Parameters
 |  ----------
 |  c_or_r : array_like
 |      The polynomial's coefficients, in decreasing powers, or if
 |      the value of the second parameter is True, the polynomial's
 |      roots (values where the polynomial evaluates to 0).  For example,
 |      ``poly1d([1, 2, 3])`` returns an object that represents
 |      :math:`x^2 + 2x + 3`, whereas ``poly1d([1, 2, 3], True)`` returns
 |      one that represents :math:`(x-1)(x-2)(x-3) = x^3 - 6x^2 + 11x -6`.
 |  r : bool, optional
 |      If True, `c_or_r` specifies the polynomial's roots; the default
 |      is False.
```

```
|  variable : str, optional
|      Changes the variable used when printing `p` from `x` to `variable`
|      (see Examples).
|
|  Examples
|  --------
|  Construct the polynomial :math:`x^2 + 2x + 3`:
|
|  >>> p = np.poly1d([1, 2, 3])
|  >>> print(np.poly1d(p))
|     2
|  1 x + 2 x + 3
|
|  Evaluate the polynomial at :math:`x = 0.5`:
|
|  >>> p(0.5)
|  4.25
|
|  Find the roots:
|
|  >>> p.r
|  array([-1.+1.41421356j, -1.-1.41421356j])
|  >>> p(p.r)
|  array([ -4.44089210e-16+0.j,  -4.44089210e-16+0.j]) # may vary
|
|  These numbers in the previous line represent (0, 0) to machine precision
|
|  Show the coefficients:
|
|  >>> p.c
|  array([1, 2, 3])
|
|  Display the order (the leading zero-coefficients are removed):
|
|  >>> p.order
|  2
|
|  Show the coefficient of the k-th power in the polynomial
|  (which is equivalent to ``p.c[-(i+1)]``):
|
|  >>> p[1]
|  2
|
|  Polynomials can be added, subtracted, multiplied, and divided
|  (returns quotient and remainder):
|
|  >>> p * p
|  poly1d([ 1,  4, 10, 12,  9])
```

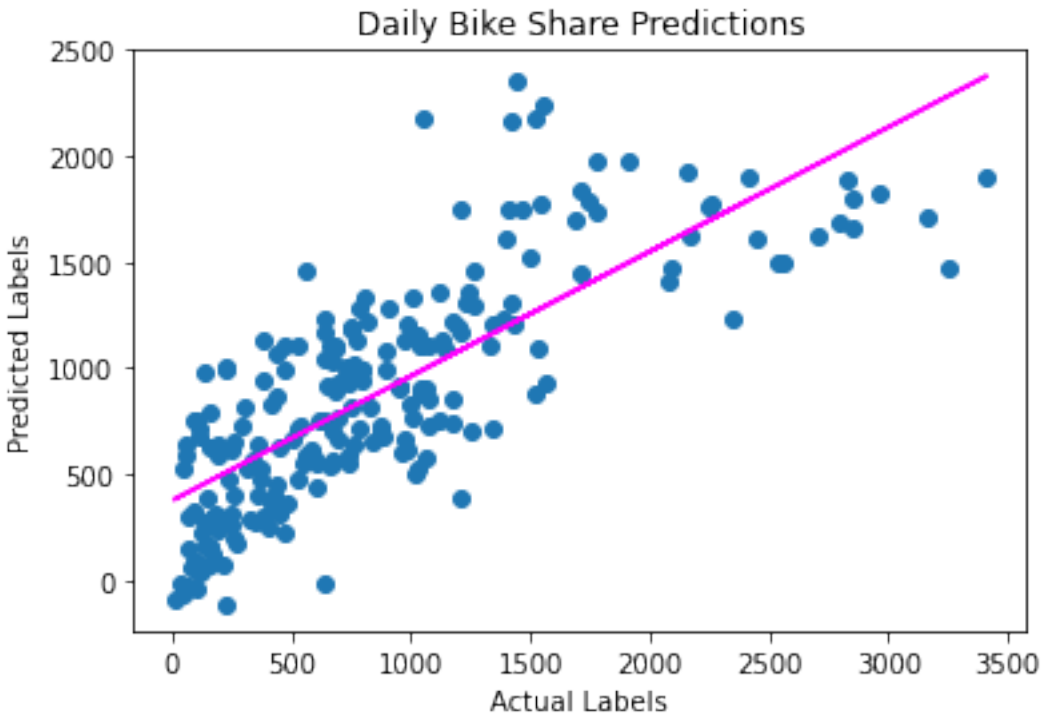
```
|
|  >>> (p**3 + 4) / p
|  (poly1d([ 1.,  4., 10., 12.,  9.]), poly1d([4.]))
|
|  ``asarray(p)`` gives the coefficient array, so polynomials can be
|  used in all functions that accept arrays:
|
|  >>> p**2 # square of polynomial
|  poly1d([ 1,  4, 10, 12,  9])
|
|  >>> np.square(p) # square of individual coefficients
|  array([1, 4, 9])
|
|  The variable used in the string representation of `p` can be modified,
|  using the `variable` parameter:
|
|  >>> p = np.poly1d([1,2,3], variable='z')
|  >>> print(p)
|     2
|  1 z + 2 z + 3
|
|  Construct a polynomial from its roots:
|
|  >>> np.poly1d([1, 2], True)
|  poly1d([ 1., -3.,  2.])
|
|  This is the same polynomial as obtained by:
|
|  >>> np.poly1d([1, -1]) * np.poly1d([1, -2])
|  poly1d([ 1, -3,  2])
|
|  Methods defined here:
|
|  __add__(self, other)
|
|  __array__(self, t=None)
|
|  __call__(self, val)
|      Call self as a function.
|
|  __div__(self, other)
|
|  __eq__(self, other)
|      Return self==value.
|
|  __getitem__(self, val)
|
|  __init__(self, c_or_r, r=False, variable=None)
```

```
|      Initialize self.  See help(type(self)) for accurate signature.
|
|  __iter__(self)
|
|  __len__(self)
|
|  __mul__(self, other)
|
|  __ne__(self, other)
|      Return self!=value.
|
|  __neg__(self)
|
|  __pos__(self)
|
|  __pow__(self, val)
|
|  __radd__(self, other)
|
|  __rdiv__(self, other)
|
|  __repr__(self)
|      Return repr(self).
|
|  __rmul__(self, other)
|
|  __rsub__(self, other)
|
|  __rtruediv__ = __rdiv__(self, other)
|
|  __setitem__(self, key, val)
|
|  __str__(self)
|      Return str(self).
|
|  __sub__(self, other)
|
|  __truediv__ = __div__(self, other)
|
|  deriv(self, m=1)
|      Return a derivative of this polynomial.
|
|      Refer to `polyder` for full documentation.
|
|      See Also
|      --------
|      polyder : equivalent function
|
```

```
|  integ(self, m=1, k=0)
|      Return an antiderivative (indefinite integral) of this polynomial.
|
|      Refer to `polyint` for full documentation.
|
|      See Also
|      --------
|      polyint : equivalent function
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
|
|  c
|      The polynomial coefficients
|
|  coef
|      The polynomial coefficients
|
|  coefficients
|      The polynomial coefficients
|
|  coeffs
|      The polynomial coefficients
|
|  o
|      The order or degree of the polynomial
|
|  order
|      The order or degree of the polynomial
|
|  r
|      The roots of the polynomial, where self(x) == 0
|
|  roots
|      The roots of the polynomial, where self(x) == 0
|
|  variable
|      The name of the polynomial variable
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
```

```
|   __hash__ = None
```

[30]:
```
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)# polynomial  degree of 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 of the squared differences between predicted and actual values. 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.

[31]:
```
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: 201972.55947035592
RMSE: 449.4135728595165
R2: 0.6040454736919191
```

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!

1.3 Summary

Here we've explored our data and fit a basic regression model. In the next notebook, we will try a number of other regression algorithms to improve performance

1.4 Further Reading

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