Practical example

Importing the relevant libraries

In [2]:
```
# For this practical example we will need the following libraries and module
import numpy as np
import pandas as pd
import statsmodels.api as sm
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
import seaborn as sns
sns.set()
```

Loading the raw data

In [3]:
```
# Load the data from a .csv in the same folder
raw_data = pd.read_csv('1.04. Real-life example.csv')

# Let's explore the top 5 rows of the df
raw_data.head()
```

Out[3]:

	Brand	Price	Body	Mileage	EngineV	Engine Type	Registration	Year	Model
0	BMW	4200.0	sedan	277	2.0	Petrol	yes	1991	320
1	Mercedes-Benz	7900.0	van	427	2.9	Diesel	yes	1999	Sprinter 212
2	Mercedes-Benz	13300.0	sedan	358	5.0	Gas	yes	2003	S 500
3	Audi	23000.0	crossover	240	4.2	Petrol	yes	2007	Q7
4	Toyota	18300.0	crossover	120	2.0	Petrol	yes	2011	Rav 4

Preprocessing

Exploring the descriptive statistics of the variables

In [4]:

```python
# Descriptive statistics are very useful for initial exploration of the vari
# By default, only descriptives for the numerical variables are shown
# To include the categorical ones, you should specify this with an argument
raw_data.describe(include='all')

# Note that categorical variables don't have some types of numerical descrip
# and numerical variables don't have some types of categorical descriptives
```

Out[4]:

	Brand	Price	Body	Mileage	EngineV	Engine Type	Registration	
count	4345	4173.000000	4345	4345.000000	4195.000000	4345	4345	4345.0
unique	7	NaN	6	NaN	NaN	4	2	
top	Volkswagen	NaN	sedan	NaN	NaN	Diesel	yes	
freq	936	NaN	1649	NaN	NaN	2019	3947	
mean	NaN	19418.746935	NaN	161.237284	2.790734	NaN	NaN	2006.5
std	NaN	25584.242620	NaN	105.705797	5.066437	NaN	NaN	6.71
min	NaN	600.000000	NaN	0.000000	0.600000	NaN	NaN	1969.0
25%	NaN	6999.000000	NaN	86.000000	1.800000	NaN	NaN	2003.0
50%	NaN	11500.000000	NaN	155.000000	2.200000	NaN	NaN	2008.0
75%	NaN	21700.000000	NaN	230.000000	3.000000	NaN	NaN	2012.0
max	NaN	300000.000000	NaN	980.000000	99.990000	NaN	NaN	2016.0

Determining the variables of interest

In [5]:
```
# For these several lessons, we will create the regression without 'Model'
# Certainly, when you work on the problem on your own, you could create a re
data = raw_data.drop(['Model'],axis=1)

# Let's check the descriptives without 'Model'
data.describe(include='all')
```

Out[5]:

	Brand	Price	Body	Mileage	EngineV	Engine Type	Registration	
count	4345	4173.000000	4345	4345.000000	4195.000000	4345	4345	4345.0(
unique	7	NaN	6	NaN	NaN	4	2	
top	Volkswagen	NaN	sedan	NaN	NaN	Diesel	yes	
freq	936	NaN	1649	NaN	NaN	2019	3947	
mean	NaN	19418.746935	NaN	161.237284	2.790734	NaN	NaN	2006.5ŧ
std	NaN	25584.242620	NaN	105.705797	5.066437	NaN	NaN	6.71
min	NaN	600.000000	NaN	0.000000	0.600000	NaN	NaN	1969.0(
25%	NaN	6999.000000	NaN	86.000000	1.800000	NaN	NaN	2003.0(
50%	NaN	11500.000000	NaN	155.000000	2.200000	NaN	NaN	2008.0(
75%	NaN	21700.000000	NaN	230.000000	3.000000	NaN	NaN	2012.0(
max	NaN	300000.000000	NaN	980.000000	99.990000	NaN	NaN	2016.0(

Dealing with missing values

In [6]:
```
# data.isnull() # shows a df with the information whether a data point is nu
# Since True = the data point is missing, while False = the data point is no
# This will give us the total number of missing values feature-wise
data.isnull().sum()
```

Out[6]:
```
Brand             0
Price           172
Body              0
Mileage           0
EngineV         150
Engine Type       0
Registration      0
Year              0
dtype: int64
```

In [7]:
```
# Let's simply drop all missing values
# This is not always recommended, however, when we remove less than 5% of th
data_no_mv = data.dropna(axis=0)
```

In [8]:
```python
# Let's check the descriptives without the missing values
data_no_mv.describe(include='all')
```

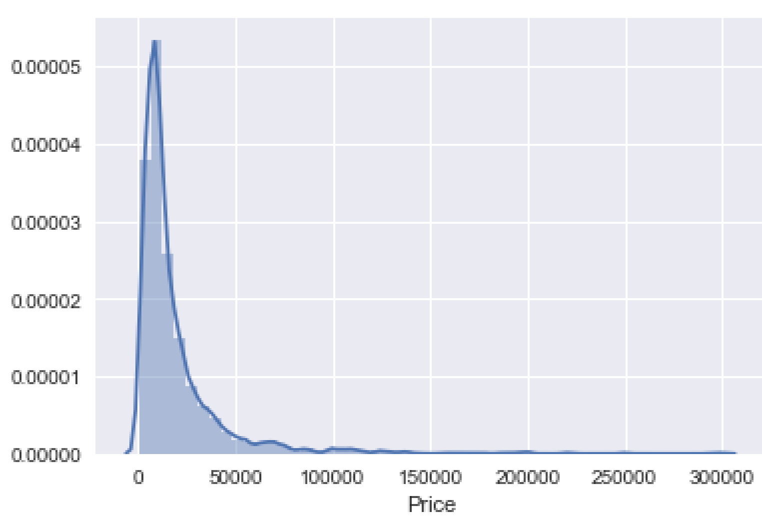
Out[8]:

	Brand	Price	Body	Mileage	EngineV	Engine Type	Registration	
count	4025	4025.000000	4025	4025.000000	4025.000000	4025	4025	4025.00
unique	7	NaN	6	NaN	NaN	4	2	
top	Volkswagen	NaN	sedan	NaN	NaN	Diesel	yes	
freq	880	NaN	1534	NaN	NaN	1861	3654	
mean	NaN	19552.308065	NaN	163.572174	2.764586	NaN	NaN	2006.37
std	NaN	25815.734988	NaN	103.394703	4.935941	NaN	NaN	6.69
min	NaN	600.000000	NaN	0.000000	0.600000	NaN	NaN	1969.00
25%	NaN	6999.000000	NaN	90.000000	1.800000	NaN	NaN	2003.00
50%	NaN	11500.000000	NaN	158.000000	2.200000	NaN	NaN	2007.00
75%	NaN	21900.000000	NaN	230.000000	3.000000	NaN	NaN	2012.00
max	NaN	300000.000000	NaN	980.000000	99.990000	NaN	NaN	2016.00

Exploring the PDFs

In [9]:
```python
# A great step in the data exploration is to display the probability distrib
# The PDF will show us how that variable is distributed
# This makes it very easy to spot anomalies, such as outliers
# The PDF is often the basis on which we decide whether we want to transform
sns.distplot(data_no_mv['Price'])
```

Out[9]: <matplotlib.axes._subplots.AxesSubplot at 0x2b897ab4940>



Dealing with outliers

In [10]:

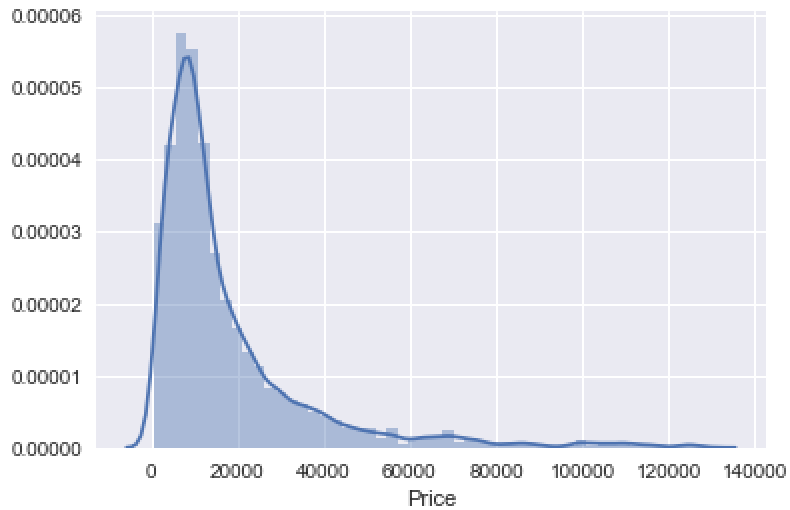
```
# Obviously there are some outliers present

# Without diving too deep into the topic, we can deal with the problem easil
# Here, the outliers are situated around the higher prices (right side of th
# Logic should also be applied
# This is a dataset about used cars, therefore one can imagine how $300,000

# Outliers are a great issue for OLS, thus we must deal with them in some wa
# It may be a useful exercise to try training a model without removing the o

# Let's declare a variable that will be equal to the 99th percentile of the
q = data_no_mv['Price'].quantile(0.99)
# Then we can create a new df, with the condition that all prices must be be
data_1 = data_no_mv[data_no_mv['Price']<q]
# In this way we have essentially removed the top 1% of the data about 'Pric
data_1.describe(include='all')
```

Out[10]:

	Brand	Price	Body	Mileage	EngineV	Engine Type	Registration	
count	3984	3984.000000	3984	3984.000000	3984.000000	3984	3984	3984.0(
unique	7	NaN	6	NaN	NaN	4	2	
top	Volkswagen	NaN	sedan	NaN	NaN	Diesel	yes	
freq	880	NaN	1528	NaN	NaN	1853	3613	
mean	NaN	17837.117460	NaN	165.116466	2.743770	NaN	NaN	2006.2!
std	NaN	18976.268315	NaN	102.766126	4.956057	NaN	NaN	6.67
min	NaN	600.000000	NaN	0.000000	0.600000	NaN	NaN	1969.0(
25%	NaN	6980.000000	NaN	93.000000	1.800000	NaN	NaN	2002.7!
50%	NaN	11400.000000	NaN	160.000000	2.200000	NaN	NaN	2007.0(
75%	NaN	21000.000000	NaN	230.000000	3.000000	NaN	NaN	2011.0(
max	NaN	129222.000000	NaN	980.000000	99.990000	NaN	NaN	2016.0(

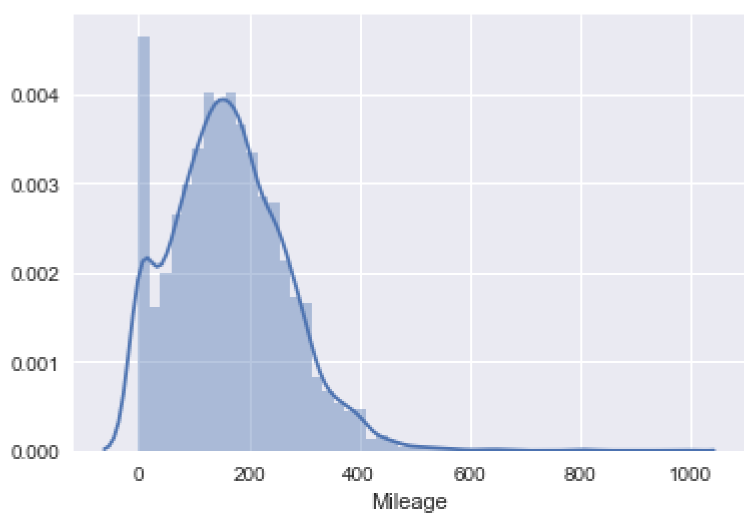
In [11]:
```
# We can check the PDF once again to ensure that the result is still distrib
# however, there are much fewer outliers
sns.distplot(data_1['Price'])
```

Out[11]: <matplotlib.axes._subplots.AxesSubplot at 0x2b897eacfd0>



In [12]:
```
# We can treat the other numerical variables in a similar way
sns.distplot(data_no_mv['Mileage'])
```

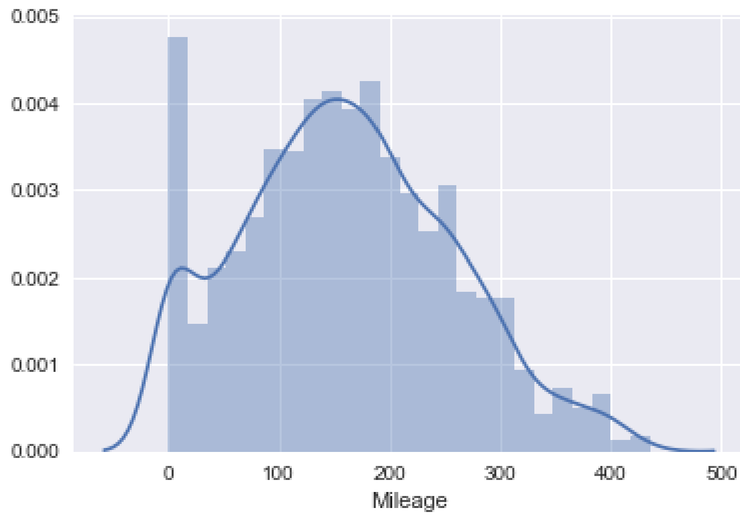
Out[12]: <matplotlib.axes._subplots.AxesSubplot at 0x2b897e36e80>



In [13]:
```
q = data_1['Mileage'].quantile(0.99)
data_2 = data_1[data_1['Mileage']<q]
```

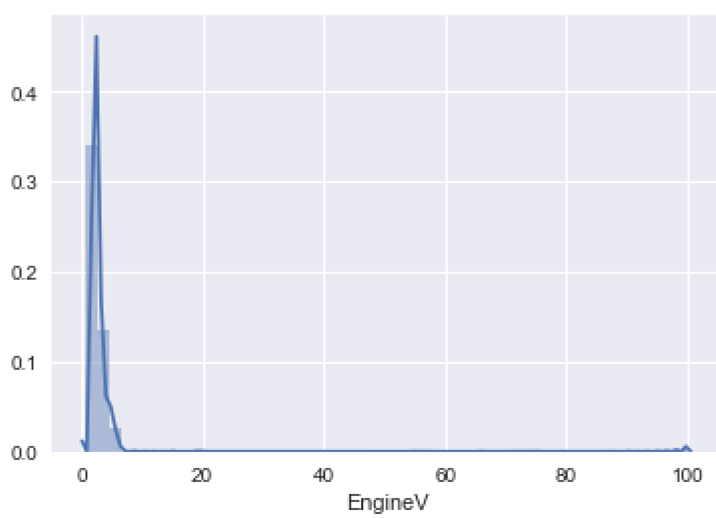
In [14]:
```python
# This plot looks kind of normal, doesn't it?
sns.distplot(data_2['Mileage'])
```

Out[14]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x2b8980359b0>
```



In [15]:
```python
# The situation with engine volume is very strange
# In such cases it makes sense to manually check what may be causing the pro
# In our case the issue comes from the fact that most missing values are ind
# There are also some incorrect entries like 75
sns.distplot(data_no_mv['EngineV'])
```

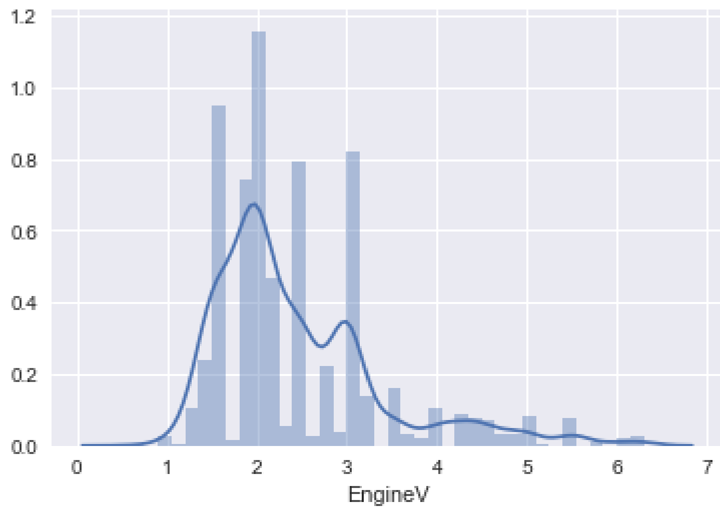
Out[15]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x2b897f681d0>
```



In [16]:
```python
# A simple Google search can indicate the natural domain of this variable
# Car engine volumes are usually (always?) below 6.5l
# This is a prime example of the fact that a domain expert (a person working
# may find it much easier to determine problems with the data than an outsid
data_3 = data_2[data_2['EngineV']<6.5]
```

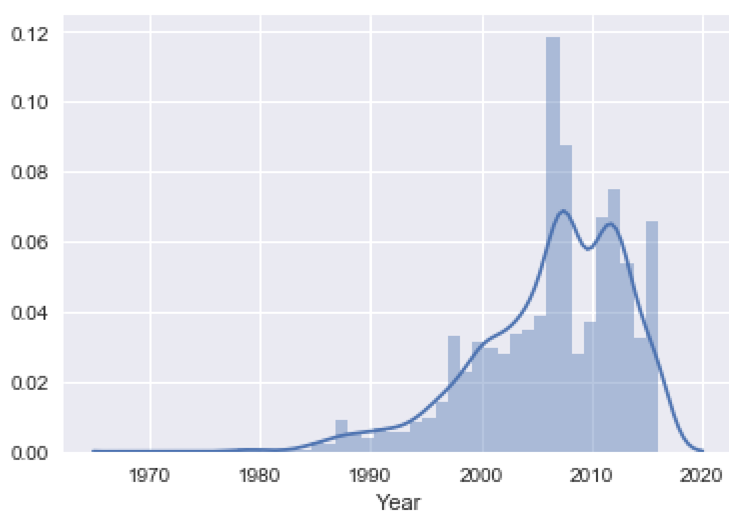
In [17]:
```
# Following this graph, we realize we can actually treat EngineV as a catego
# Even so, in this course we won't, but that's yet something else you may tr
sns.distplot(data_3['EngineV'])
```

Out[17]: <matplotlib.axes._subplots.AxesSubplot at 0x2b8981a0b00>



In [18]:
```
# Finally, the situation with 'Year' is similar to 'Price' and 'Mileage'
# However, the outliers are on the low end
sns.distplot(data_no_mv['Year'])
```

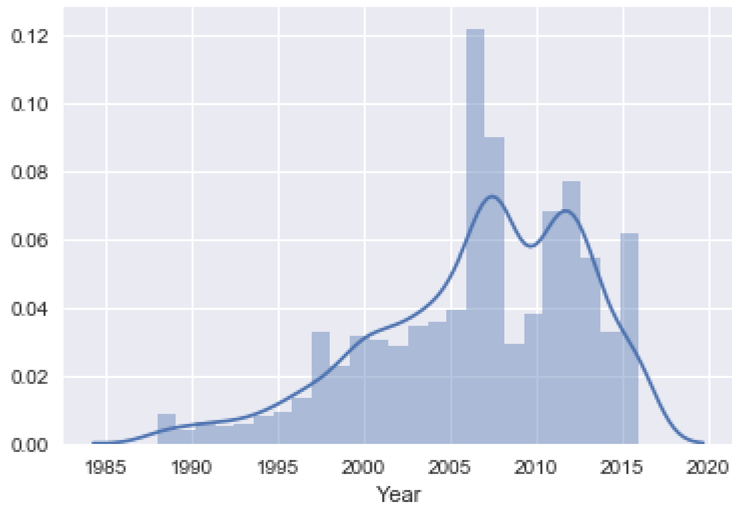
Out[18]: <matplotlib.axes._subplots.AxesSubplot at 0x2b89825e6a0>



In [19]:
```
# I'll simply remove them
q = data_3['Year'].quantile(0.01)
data_4 = data_3[data_3['Year']>q]
```

In [20]:
```
# Here's the new result
sns.distplot(data_4['Year'])
```

Out[20]: <matplotlib.axes._subplots.AxesSubplot at 0x2b898296d30>



In [21]:
```
# When we remove observations, the original indexes are preserved
# If we remove observations with indexes 2 and 3, the indexes will go as: 0,
# That's very problematic as we tend to forget about it (later you will see

# Finally, once we reset the index, a new column will be created containing
# We won't be needing it, thus 'drop=True' to completely forget about it
data_cleaned = data_4.reset_index(drop=True)
```

In [22]:
```
# Let's see what's left
data_cleaned.describe(include='all')
```

Out[22]:

	Brand	Price	Body	Mileage	EngineV	Engine Type	Registration	
count	3867	3867.000000	3867	3867.000000	3867.000000	3867	3867	3867.0(
unique	7	NaN	6	NaN	NaN	4	2	
top	Volkswagen	NaN	sedan	NaN	NaN	Diesel	yes	
freq	848	NaN	1467	NaN	NaN	1807	3505	
mean	NaN	18194.455679	NaN	160.542539	2.450440	NaN	NaN	2006.7(
std	NaN	19085.855165	NaN	95.633291	0.949366	NaN	NaN	6.1(
min	NaN	800.000000	NaN	0.000000	0.600000	NaN	NaN	1988.0(
25%	NaN	7200.000000	NaN	91.000000	1.800000	NaN	NaN	2003.0(
50%	NaN	11700.000000	NaN	157.000000	2.200000	NaN	NaN	2008.0(
75%	NaN	21700.000000	NaN	225.000000	3.000000	NaN	NaN	2012.0(
max	NaN	129222.000000	NaN	435.000000	6.300000	NaN	NaN	2016.0(

Checking the OLS assumptions

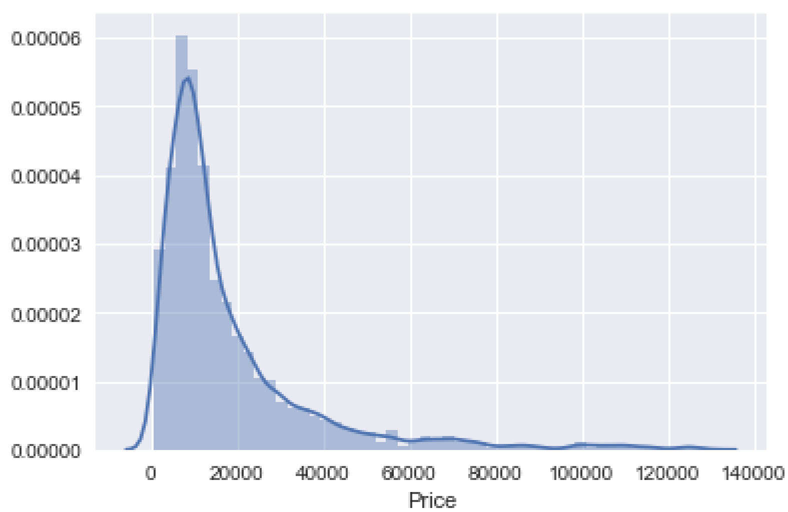
In [23]:

```
# Here we decided to use some matplotlib code, without explaining it
# You can simply use plt.scatter() for each of them (with your current knowl
# But since Price is the 'y' axis of all the plots, it made sense to plot th
f, (ax1, ax2, ax3) = plt.subplots(1, 3, sharey=True, figsize =(15,3)) #share
ax1.scatter(data_cleaned['Year'],data_cleaned['Price'])
ax1.set_title('Price and Year')
ax2.scatter(data_cleaned['EngineV'],data_cleaned['Price'])
ax2.set_title('Price and EngineV')
ax3.scatter(data_cleaned['Mileage'],data_cleaned['Price'])
ax3.set_title('Price and Mileage')


plt.show()
```

In [24]:

```
# From the subplots and the PDF of price, we can easily determine that 'Pric
# A good transformation in that case is a log transformation
sns.distplot(data_cleaned['Price'])
```

Out[24]: <matplotlib.axes._subplots.AxesSubplot at 0x2b8994ccdd8>



Relaxing the assumptions

In [25]:

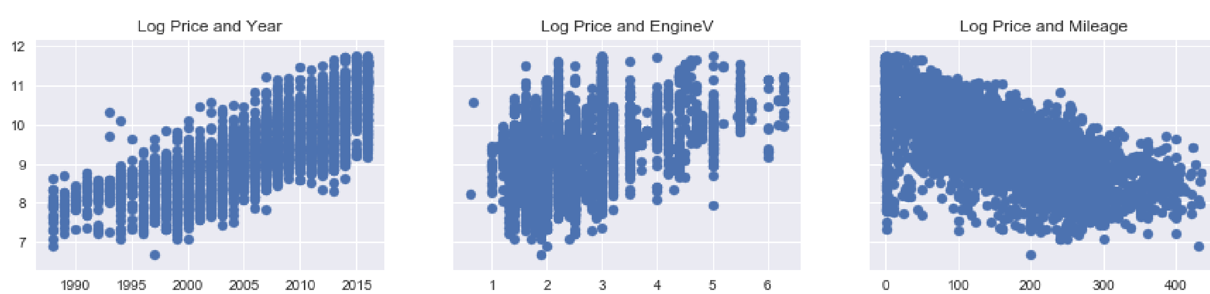
```
# Let's transform 'Price' with a log transformation
log_price = np.log(data_cleaned['Price'])

# Then we add it to our data frame
data_cleaned['log_price'] = log_price
data_cleaned
```

Out[25]:

	Brand	Price	Body	Mileage	EngineV	Engine Type	Registration	Year	log_price
0	BMW	4200.0	sedan	277	2.00	Petrol	yes	1991	8.342840
1	Mercedes-Benz	7900.0	van	427	2.90	Diesel	yes	1999	8.974618
2	Mercedes-Benz	13300.0	sedan	358	5.00	Gas	yes	2003	9.495519
3	Audi	23000.0	crossover	240	4.20	Petrol	yes	2007	10.043249
4	Toyota	18300.0	crossover	120	2.00	Petrol	yes	2011	9.814656
5	Audi	14200.0	vagon	200	2.70	Diesel	yes	2006	9.560997
6	Renault	10799.0	vagon	193	1.50	Diesel	yes	2012	9.287209
7	Volkswagen	1400.0	other	212	1.80	Gas	no	1999	7.244228
8	Renault	11950.0	vagon	177	1.50	Diesel	yes	2011	9.388487

In [26]:

```
# Let's check the three scatters once again
f, (ax1, ax2, ax3) = plt.subplots(1, 3, sharey=True, figsize =(15,3))
ax1.scatter(data_cleaned['Year'],data_cleaned['log_price'])
ax1.set_title('Log Price and Year')
ax2.scatter(data_cleaned['EngineV'],data_cleaned['log_price'])
ax2.set_title('Log Price and EngineV')
ax3.scatter(data_cleaned['Mileage'],data_cleaned['log_price'])
ax3.set_title('Log Price and Mileage')


plt.show()

# The relationships show a clear linear relationship
# This is some good linear regression material

# Alternatively we could have transformed each of the independent variables
```



In [27]:
```
# Since we will be using the log price variable, we can drop the old 'Price'
data_cleaned = data_cleaned.drop(['Price'],axis=1)
```

Multicollinearity

In [28]:
```
# Let's quickly see the columns of our data frame
data_cleaned.columns.values
```

Out[28]:
```
array(['Brand', 'Body', 'Mileage', 'EngineV', 'Engine Type',
       'Registration', 'Year', 'log_price'], dtype=object)
```

In [29]:
```
# sklearn does not have a built-in way to check for multicollinearity
# one of the main reasons is that this is an issue well covered in statistic
# surely it is an issue nonetheless, thus we will try to deal with it

# Here's the relevant module
# full documentation: http://www.statsmodels.org/dev/_modules/statsmodels/st
from statsmodels.stats.outliers_influence import variance_inflation_factor

# To make this as easy as possible to use, we declare a variable where we pu
# all features where we want to check for multicollinearity
# since our categorical data is not yet preprocessed, we will only take the
variables = data_cleaned[['Mileage','Year','EngineV']]

# we create a new data frame which will include all the VIFs
# note that each variable has its own variance inflation factor as this meas
vif = pd.DataFrame()

# here we make use of the variance_inflation_factor, which will basically ou
vif["VIF"] = [variance_inflation_factor(variables.values, i) for i in range(
# Finally, I like to include names so it is easier to explore the result
vif["Features"] = variables.columns
```

In [30]:
```
# Let's explore the result
vif
```

Out[30]:

	VIF	Features
0	3.791584	Mileage
1	10.354854	Year
2	7.662068	EngineV

In [31]:
```
# Since Year has the highest VIF, I will remove it from the model
# This will drive the VIF of other variables down!!!
# So even if EngineV seems with a high VIF, too, once 'Year' is gone that wi
data_no_multicollinearity = data_cleaned.drop(['Year'],axis=1)
```

Create dummy variables

In [32]:
```
# To include the categorical data in the regression, let's create dummies
# There is a very convenient method called: 'get_dummies' which does that se
# It is extremely important that we drop one of the dummies, alternatively w
data_with_dummies = pd.get_dummies(data_no_multicollinearity, drop_first=Tru
```

In [33]:
```
# Here's the result
data_with_dummies.head()
```

Out[33]:

	Mileage	EngineV	log_price	Brand_BMW	Brand_Mercedes-Benz	Brand_Mitsubishi	Brand_Renault
0	277	2.0	8.342840	1	0	0	0
1	427	2.9	8.974618	0	1	0	0
2	358	5.0	9.495519	0	1	0	0
3	240	4.2	10.043249	0	0	0	0
4	120	2.0	9.814656	0	0	0	0

Rearrange a bit

In [34]:
```
# To make our data frame more organized, we prefer to place the dependent va
# Since each problem is different, that must be done manually
# We can display all possible features and then choose the desired order
data_with_dummies.columns.values
```

Out[34]:
```
array(['Mileage', 'EngineV', 'log_price', 'Brand_BMW',
       'Brand_Mercedes-Benz', 'Brand_Mitsubishi', 'Brand_Renault',
       'Brand_Toyota', 'Brand_Volkswagen', 'Body_hatch', 'Body_other',
       'Body_sedan', 'Body_vagon', 'Body_van', 'Engine Type_Gas',
       'Engine Type_Other', 'Engine Type_Petrol', 'Registration_yes'],
      dtype=object)
```

In [35]:
```
# To make the code a bit more parametrized, let's declare a new variable tha
# If you want a different order, just specify it here
# Conventionally, the most intuitive order is: dependent variable, indepeden
cols = ['log_price', 'Mileage', 'EngineV', 'Brand_BMW',
        'Brand_Mercedes-Benz', 'Brand_Mitsubishi', 'Brand_Renault',
        'Brand_Toyota', 'Brand_Volkswagen', 'Body_hatch', 'Body_other',
        'Body_sedan', 'Body_vagon', 'Body_van', 'Engine Type_Gas',
        'Engine Type_Other', 'Engine Type_Petrol', 'Registration_yes']
```

In [36]:
```python
# To implement the reordering, we will create a new df, which is equal to th
data_preprocessed = data_with_dummies[cols]
data_preprocessed.head()
```

Out[36]:

	log_price	Mileage	EngineV	Brand_BMW	Brand_Mercedes-Benz	Brand_Mitsubishi	Brand_Renault
0	8.342840	277	2.0	1	0	0	0
1	8.974618	427	2.9	0	1	0	0
2	9.495519	358	5.0	0	1	0	0
3	10.043249	240	4.2	0	0	0	0
4	9.814656	120	2.0	0	0	0	0

Linear regression model

Declare the inputs and the targets

In [37]:
```python
# The target(s) (dependent variable) is 'log price'
targets = data_preprocessed['log_price']

# The inputs are everything BUT the dependent variable, so we can simply dro
inputs = data_preprocessed.drop(['log_price'],axis=1)
```

Scale the data

In [38]:
```python
# Import the scaling module
from sklearn.preprocessing import StandardScaler

# Create a scaler object
scaler = StandardScaler()
# Fit the inputs (calculate the mean and standard deviation feature-wise)
scaler.fit(inputs)
```

Out[38]:
```
StandardScaler(copy=True, with_mean=True, with_std=True)
```

In [39]:
```python
# Scale the features and store them in a new variable (the actual scaling pr
inputs_scaled = scaler.transform(inputs)
```

Train Test Split

In [40]:
```
# Import the module for the split
from sklearn.model_selection import train_test_split

# Split the variables with an 80-20 split and some random state
# To have the same split as mine, use random_state = 365
x_train, x_test, y_train, y_test = train_test_split(inputs_scaled, targets,
```

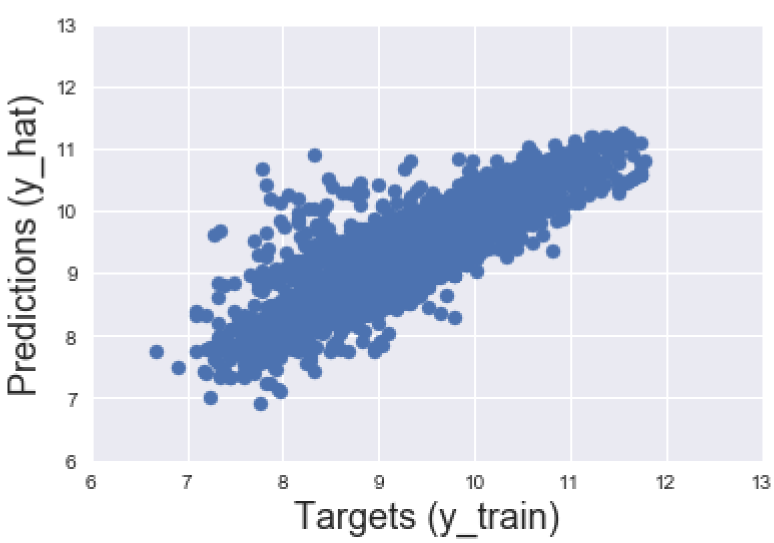
Create the regression

In [41]:
```
# Create a linear regression object
reg = LinearRegression()
# Fit the regression with the scaled TRAIN inputs and targets
reg.fit(x_train,y_train)
```

Out[41]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
```

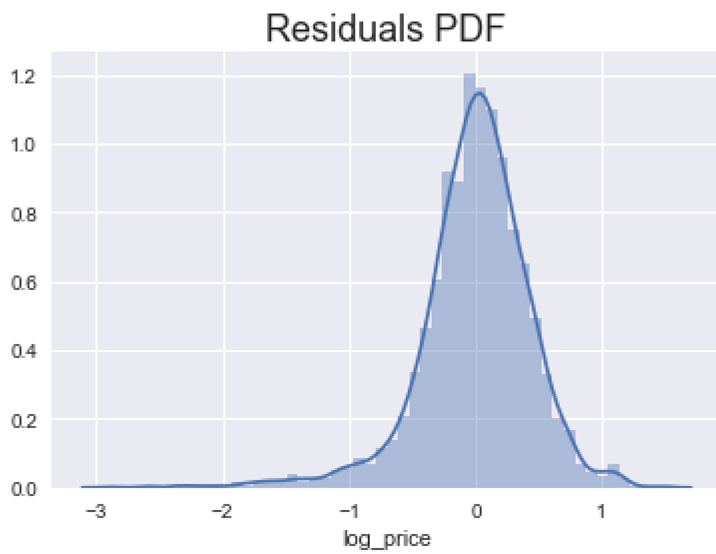
In [42]:
```
# Let's check the outputs of the regression
# I'll store them in y_hat as this is the 'theoretical' name of the predicti
y_hat = reg.predict(x_train)
```

In [43]:
```
# The simplest way to compare the targets (y_train) and the predictions (y_h
# The closer the points to the 45-degree line, the better the prediction
plt.scatter(y_train, y_hat)
# Let's also name the axes
plt.xlabel('Targets (y_train)',size=18)
plt.ylabel('Predictions (y_hat)',size=18)
# Sometimes the plot will have different scales of the x-axis and the y-axis
# This is an issue as we won't be able to interpret the '45-degree line'
# We want the x-axis and the y-axis to be the same
plt.xlim(6,13)
plt.ylim(6,13)
plt.show()
```



In [44]:
```
# Another useful check of our model is a residual plot
# We can plot the PDF of the residuals and check for anomalies
sns.distplot(y_train - y_hat)

# Include a title
plt.title("Residuals PDF", size=18)

# In the best case scenario this plot should be normally distributed
# In our case we notice that there are many negative residuals (far away fro
# Given the definition of the residuals (y_train - y_hat), negative values i
# that y_hat (predictions) are much higher than y_train (the targets)
# This is food for thought to improve our model
```

Out[44]: Text(0.5,1,'Residuals PDF')



In [45]:
```
# Find the R-squared of the model
reg.score(x_train,y_train)

# Note that this is NOT the adjusted R-squared
# in other words... find the Adjusted R-squared to have the appropriate meas
```

Out[45]: 0.744996578792662

Finding the weights and bias

In [46]:
```
# Obtain the bias (intercept) of the regression
reg.intercept_
```

Out[46]: 9.415239458021299

In [47]:
```
# Obtain the weights (coefficients) of the regression
reg.coef_

# Note that they are barely interpretable if at all
```

Out[47]:
```
array([-0.44871341,  0.20903483,  0.0142496 ,  0.01288174, -0.14055166,
       -0.17990912, -0.06054988, -0.08992433, -0.1454692 , -0.10144383,
       -0.20062984, -0.12988747, -0.16859669, -0.12149035, -0.03336798,
       -0.14690868,  0.32047333])
```

In [48]:
```
# Create a regression summary where we can compare them with one-another
reg_summary = pd.DataFrame(inputs.columns.values, columns=['Features'])
reg_summary['Weights'] = reg.coef_
reg_summary
```

Out[48]:

	Features	Weights
0	Mileage	-0.448713
1	EngineV	0.209035
2	Brand_BMW	0.014250
3	Brand_Mercedes-Benz	0.012882
4	Brand_Mitsubishi	-0.140552
5	Brand_Renault	-0.179909
6	Brand_Toyota	-0.060550
7	Brand_Volkswagen	-0.089924
8	Body_hatch	-0.145469
9	Body_other	-0.101444
10	Body_sedan	-0.200630
11	Body_vagon	-0.129887
12	Body_van	-0.168597
13	Engine Type_Gas	-0.121490
14	Engine Type_Other	-0.033368
15	Engine Type_Petrol	-0.146909
16	Registration_yes	0.320473

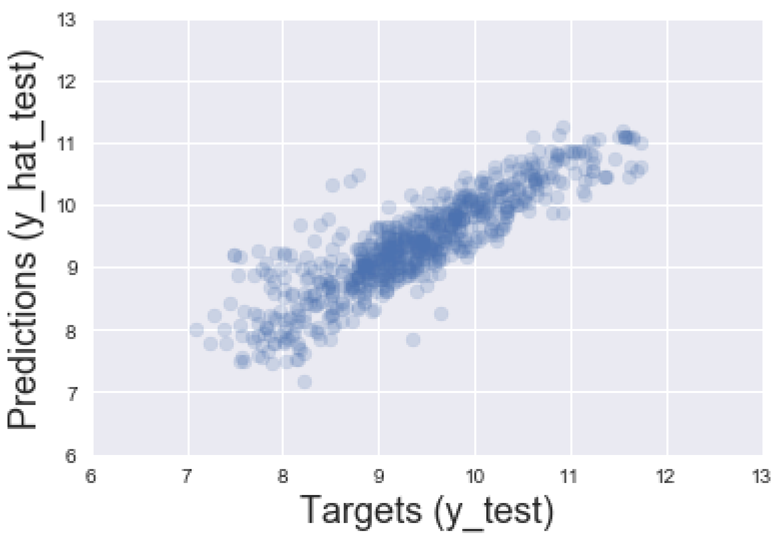
In [49]:
```
# Check the different categories in the 'Brand' variable
data_cleaned['Brand'].unique()

# In this way we can see which 'Brand' is actually the benchmark
```

Out[49]:
```
array(['BMW', 'Mercedes-Benz', 'Audi', 'Toyota', 'Renault', 'Volkswagen',
       'Mitsubishi'], dtype=object)
```

Testing

In [50]:
```
# Once we have trained and fine-tuned our model, we can proceed to testing i
# Testing is done on a dataset that the algorithm has never seen
# Luckily we have prepared such a dataset
# Our test inputs are 'x_test', while the outputs: 'y_test'
# We SHOULD NOT TRAIN THE MODEL ON THEM, we just feed them and find the pred
# If the predictions are far off, we will know that our model overfitted
y_hat_test = reg.predict(x_test)
```

In [51]:
```
# Create a scatter plot with the test targets and the test predictions
# You can include the argument 'alpha' which will introduce opacity to the g
plt.scatter(y_test, y_hat_test, alpha=0.2)
plt.xlabel('Targets (y_test)',size=18)
plt.ylabel('Predictions (y_hat_test)',size=18)
plt.xlim(6,13)
plt.ylim(6,13)
plt.show()
```



In [52]:
```
# Finally, let's manually check these predictions
# To obtain the actual prices, we take the exponential of the log_price
df_pf = pd.DataFrame(np.exp(y_hat_test), columns=['Prediction'])
df_pf.head()
```

Out[52]:

	Prediction
0	10685.501696
1	3499.255242
2	7553.285218
3	7463.963017
4	11353.490075

In [53]:
```
# We can also include the test targets in that data frame (so we can manuall
df_pf['Target'] = np.exp(y_test)
df_pf

# Note that we have a lot of missing values
# There is no reason to have ANY missing values, though
# This suggests that something is wrong with the data frame / indexing
```

Out[53]:

	Prediction	Target
0	10685.501696	NaN
1	3499.255242	7900.0
2	7553.285218	NaN
3	7463.963017	NaN
4	11353.490075	NaN
5	21289.799394	14200.0
6	20159.189144	NaN
7	20349.617702	NaN
8	11581.537864	11950.0
9	33614.617349	NaN
10	7241.068243	NaN
11	5175.769541	10500.0

In [54]:
```
# After displaying y_test, we find what the issue is
# The old indexes are preserved (recall earlier in that code we made a note
# The code was: data_cleaned = data_4.reset_index(drop=True)

# Therefore, to get a proper result, we must reset the index and drop the ol
y_test = y_test.reset_index(drop=True)

# Check the result
y_test.head()
```

Out[54]:
```
0    7.740664
1    7.937375
2    7.824046
3    8.764053
4    9.121509
Name: log_price, dtype: float64
```

In [55]:

```
# Let's overwrite the 'Target' column with the appropriate values
# Again, we need the exponential of the test log price
df_pf['Target'] = np.exp(y_test)
df_pf
```

Out[55]:

	Prediction	Target
0	10685.501696	2300.0
1	3499.255242	2800.0
2	7553.285218	2500.0
3	7463.963017	6400.0
4	11353.490075	9150.0
5	21289.799394	20000.0
6	20159.189144	38888.0
7	20349.617702	16999.0
8	11581.537864	12500.0
9	33614.617349	41000.0
10	7241.068243	12800.0
11	5175.769541	5000.0

In [56]:

```
# Additionally, we can calculate the difference between the targets and the
# Note that this is actually the residual (we already plotted the residuals)
df_pf['Residual'] = df_pf['Target'] - df_pf['Prediction']

# Since OLS is basically an algorithm which minimizes the total sum of squar
# this comparison makes a lot of sense
```

In [57]:
```
# Finally, it makes sense to see how far off we are from the result percenta
# Here, we take the absolute difference in %, so we can easily order the dat
df_pf['Difference%'] = np.absolute(df_pf['Residual']/df_pf['Target']*100)
df_pf
```

Out[57]:

	Prediction	Target	Residual	Difference%
0	10685.501696	2300.0	-8385.501696	364.587030
1	3499.255242	2800.0	-699.255242	24.973402
2	7553.285218	2500.0	-5053.285218	202.131409
3	7463.963017	6400.0	-1063.963017	16.624422
4	11353.490075	9150.0	-2203.490075	24.081859
5	21289.799394	20000.0	-1289.799394	6.448997
6	20159.189144	38888.0	18728.810856	48.160900
7	20349.617702	16999.0	-3350.617702	19.710675
8	11581.537864	12500.0	918.462136	7.347697
9	33614.617349	41000.0	7385.382651	18.013128
10	7241.068243	12800.0	5558.931757	43.429154
11	5175.769541	5000.0	-175.769541	3.515391

In [58]:
```
# Exploring the descriptives here gives us additional insights
df_pf.describe()
```

Out[58]:

	Prediction	Target	Residual	Difference%
count	774.000000	774.000000	774.000000	774.000000
mean	15946.760167	18165.817106	2219.056939	36.256693
std	13133.197604	19967.858908	10871.218143	55.066507
min	1320.562768	1200.000000	-29456.498331	0.062794
25%	7413.644234	6900.000000	-2044.191251	12.108022
50%	11568.168859	11600.000000	142.518577	23.467728
75%	20162.408805	20500.000000	3147.343497	39.563570
max	77403.055224	126000.000000	85106.162329	512.688080

In [59]:

```
# Sometimes it is useful to check these outputs manually
# To see all rows, we use the relevant pandas syntax
pd.options.display.max_rows = 999
# Moreover, to make the dataset clear, we can display the result with only 2
pd.set_option('display.float_format', lambda x: '%.2f' % x)
# Finally, we sort by difference in % and manually check the model
df_pf.sort_values(by=['Difference%'])
```

Out[59]:

	Prediction	Target	Residual	Difference%
698	30480.85	30500.00	19.15	0.06
742	16960.31	16999.00	38.69	0.23
60	12469.21	12500.00	30.79	0.25
110	25614.14	25500.00	-114.14	0.45
367	42703.68	42500.00	-203.68	0.48
369	3084.69	3100.00	15.31	0.49
769	29651.73	29500.00	-151.73	0.51
272	9749.53	9800.00	50.47	0.52
714	23118.07	22999.00	-119.07	0.52
630	8734.58	8800.00	65.42	0.74
380	3473.79	3500.00	26.21	0.75
648	21174.10	21335.00	160.90	0.75