Solution - Secondary Car Market

# Solution: Secondary Car Market

## Import Libraries and Dataset

In [1]:
```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
%matplotlib inline

import seaborn as sns
sns.set_style('darkgrid')
```

In [2]:
```
df = pd.read_csv('data/secondary_car_market.csv', low_memory=False)
```

On import, you may run into the following warning:

> DtypeWarning: Columns (5,6) have mixed types. Specify dtype option on import or set low_memory=False. interactivity=interactivity, compiler=compiler, result=result)

To resolve the warning, we set the parameter `low_memory=False` . According to the Pandas read_csv documentation:

> **low_memory** *: boolean, default True*
>
> Internally process the file in chunks, resulting in lower memory use while parsing, but possibly mixed type inference. To ensure no mixed types either set False, or specify the type with the dtype parameter. Note that the entire file is read into a single DataFrame regardless, use the chunksize or iterator parameter to return the data in chunks. (Only valid with C parser)

In other words, we are reading the dataset in one big chunk, instead of processing it in smaller chunks. If your computer's memory is too low to do this, then read it as normally and just pay extra care during data cleaning in general (you won't run into any issues for this particular challenge).

## Exploratory Analysis and Data Cleaning

We'll start by "getting to know" the dataset. The goal is to understand the dataset at a qualitative level, note anything that should be cleaned up, and spot opportunities for feature engineering.

**Tip:** In our Machine Learning Masterclass, we separate exploratory analysis and data cleaning into separate steps. However, because take-home challenges are meant to be shorter/condensed analyses, feel free to combine these steps. Just remember to keep your code clean and organized.

In [3]:
```
# Example observations
df.head()
```

Out[3]:

| | maker | model | mileage | manufacture_year | transmission | door_count | seat_count | displacement | hor |
|---|---|---|---|---|---|---|---|---|---|
| **0** | audi | a8 | 136702.0 | 2006.0 | auto | 4.0 | 5.0 | 3.94 | 271 |
| **1** | audi | a3 | 102526.0 | 2004.0 | man | 4 | 5 | 1.97 | 138 |
| **2** | audi | a3 | 684.0 | 2016.0 | auto | 5.0 | 5.0 | 1.97 | 148 |
| **3** | audi | s7 | 38525.0 | 2013.0 | auto | 5.0 | 4.0 | 3.99 | 414 |
| **4** | audi | a1 | 45006.0 | 2010.0 | man | 3 | 4 | 1.60 | 103 |

A few useful notes right off the bat:

- We have mostly numeric features, plus a categorical one in `transmission`
- It looks like `door_count` and `seat_count` might be recorded as categorical features, since `'4.0'` and `'4'` appear to be separate classes.
- There's a pretty wide range of mileages, manufacture years, and prices.

Next, we'll check for missing values.

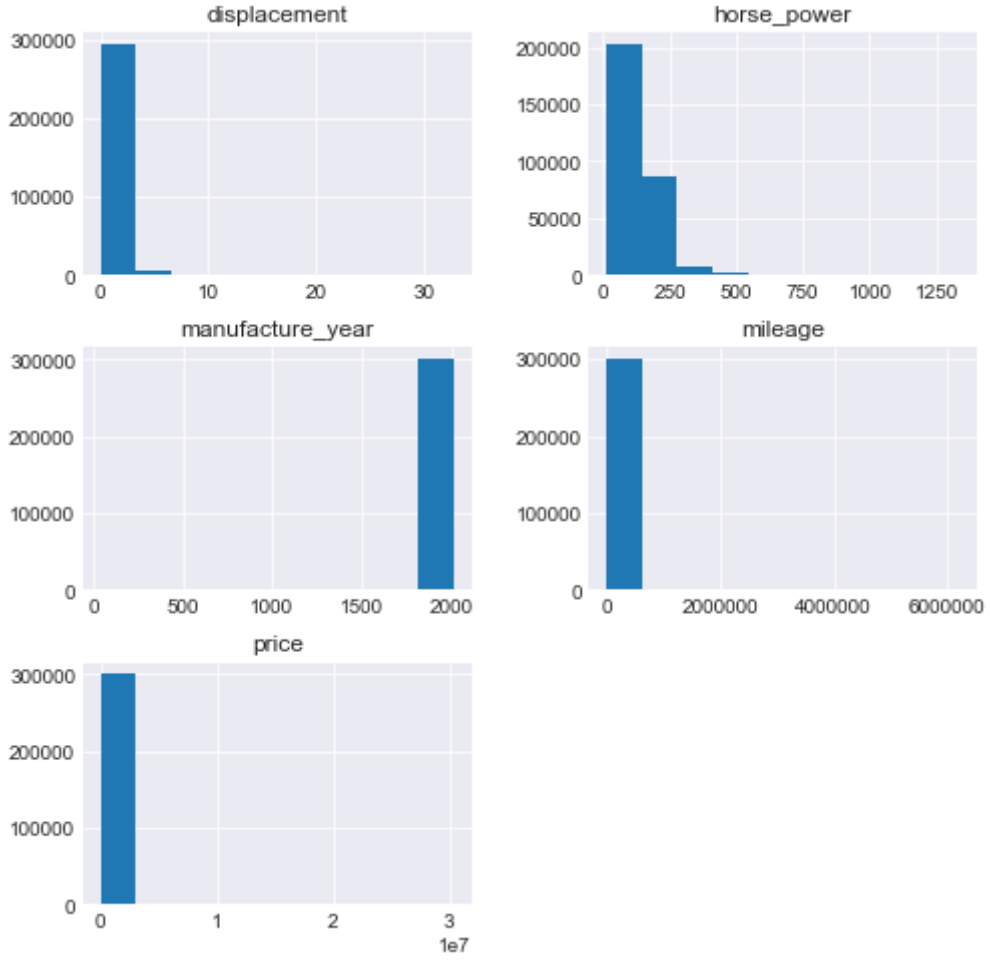
In [4]:
```
# Check for missing values
df.isnull().sum()
```

Out[4]:
```
maker              0
model              0
mileage            0
manufacture_year   0
transmission       0
door_count         0
seat_count         0
displacement       0
horse_power        0
price              0
dtype: int64
```

Next, we'll look at the distributions of numeric features in our dataset.

In [5]:
```
# Numeric feature distributions
df.hist(figsize=(8,8))
```

```
plt.show()
```




In [6]:
```
# Numeric feature summary statistics
df.describe()
```

Out[6]:

| | mileage | manufacture_year | displacement | horse_power | price |
|---|---|---|---|---|---|
| **count** | 3.000000e+05 | 300000.000000 | 300000.000000 | 300000.000000 | 3.000000e+05 |
| **mean** | 6.373923e+04 | 2009.096253 | 1.876543 | 136.943660 | 1.660514e+04 |
| **std** | 8.628841e+04 | 16.946915 | 0.904341 | 63.155779 | 5.958110e+04 |
| **min** | 0.000000e+00 | 39.000000 | 0.010000 | 13.000000 | 0.000000e+00 |
| **25%** | 1.599200e+04 | 2006.000000 | 1.560000 | 101.000000 | 4.797000e+03 |
| **50%** | 5.591400e+04 | 2011.000000 | 1.900000 | 123.000000 | 1.345500e+04 |
| **75%** | 9.693400e+04 | 2014.000000 | 1.980000 | 154.000000 | 2.327100e+04 |
| **max** | 6.213709e+06 | 2017.000000 | 32.770000 | 1337.000000 | 3.042000e+07 |

A few interesting insights:

- It appears that several features have outliers. Let's look into them and see if we have a valid reason to remove them.
- Why aren't `door_count` and `seat_count` considered numeric features?

**First, we'll look for potential outliers.**

In [7]:
```
df.sort_values(by='displacement', ascending=False).head()
```

Out[7]:

| | maker | model | mileage | manufacture_year | transmission | door_count | seat_count | displace |
|---|---|---|---|---|---|---|---|---|
| **216231** | volkswagen | passat | 120937.0 | 2011.0 | man | 4.0 | 5.0 | 32.77 |
| **55310** | audi | a3 | 127381.0 | 2004.0 | auto | None | None | 32.00 |
| **56340** | audi | 200 | 139808.0 | 2008.0 | man | None | None | 32.00 |
| **62867** | audi | a3 | 127381.0 | 2004.0 | auto | None | None | 32.00 |
| **81837** | audi | s3 | 97555.0 | 2007.0 | man | None | None | 32.00 |

Even if you know nothing about cars, a quick Google search for "largest displacement car engine" will show that even an 8-liter engine is considered very large. Another Google search for "audi a3 engine displacement" will show that it comes standard with a 2-liter engine... a far cry from the "32" recorded in the dataset.

Let's confirm this by look at the 99th percentile of displacements in our dataset.

In [8]:
```
df.displacement.quantile(.99)
```

Out[8]:
```
4.16
```

We should also remove any tiny displacements.

In [9]:
```
df.sort_values(by='displacement').head()
```

Out[9]:

| | maker | model | mileage | manufacture_year | transmission | door_count | seat_count | displac |
|---|---|---|---|---|---|---|---|---|
| **10459** | audi | a3 | 18952.0 | 2014.0 | man | 4.0 | 5.0 | 0.01 |
| **103596** | ford | fiesta | 25476.0 | 2008.0 | man | 5.0 | 4.0 | 0.01 |
| **245612** | volkswagen | golf | 14712.0 | 2015.0 | man | 5.0 | 5.0 | 0.02 |
| **2236** | audi | a3 | 9.0 | 2015.0 | man | 4 | 5 | 0.02 |
| **199500** | ford | mondeo | 0.0 | 2015.0 | auto | 5.0 | 5.0 | 0.02 |

In [10]:
```
df.displacement.quantile(.01)
```

Out[10]:
```
1.0
```

In fact, based on our histograms above, it may make sense to perform this outlier filter to each of our numerical variables, even our target variable.

For example, we can see that some cars were listed for absurd prices:

In [11]:
```
df.sort_values(by='price', ascending=False).head()
```

Out[11]:

| | maker | model | mileage | manufacture_year | transmission | door_count | seat_count | displace |
|---|---|---|---|---|---|---|---|---|
| **51433** | audi | a5 | 100662.0 | 2013.0 | auto | 5.0 | 4.0 | 1.97 |
| **50963** | audi | a3 | 111847.0 | 2005.0 | auto | 3.0 | 5.0 | 1.97 |
| **47492** | audi | s5 | 27527.0 | 2009.0 | auto | 2.0 | 4.0 | 3.00 |
| **99059** | audi | a4 | 139808.0 | 2004.0 | man | 5.0 | 5.0 | 1.90 |
| **221647** | volkswagen | golf | 62758.0 | 2007.0 | man | 5.0 | 5.0 | 1.90 |

To keep things simple, let's only keep observations within the 1-percentile to 99-percentile bounds, inclusive. The only exception will be for `manufacture_year`, and we'll only filter for above the 1-percentile bound since the maximum (2017) is reasonable.

This is a conservative way to remove outliers, but it can do wonders for a pricing model.

In [12]:
```
upper_bound = df.quantile(.99)
upper_bound
```

Out[12]:
```
mileage             199021.66
manufacture_year      2016.00
displacement             4.16
horse_power            366.00
price                70189.04
Name: 0.99, dtype: float64
```

In [13]:
```
lower_bound = df.quantile(.01)
lower_bound
```

Out[13]:
```
mileage               1.0
manufacture_year    1994.0
displacement           1.0
horse_power           59.0
price                866.0
Name: 0.01, dtype: float64
```

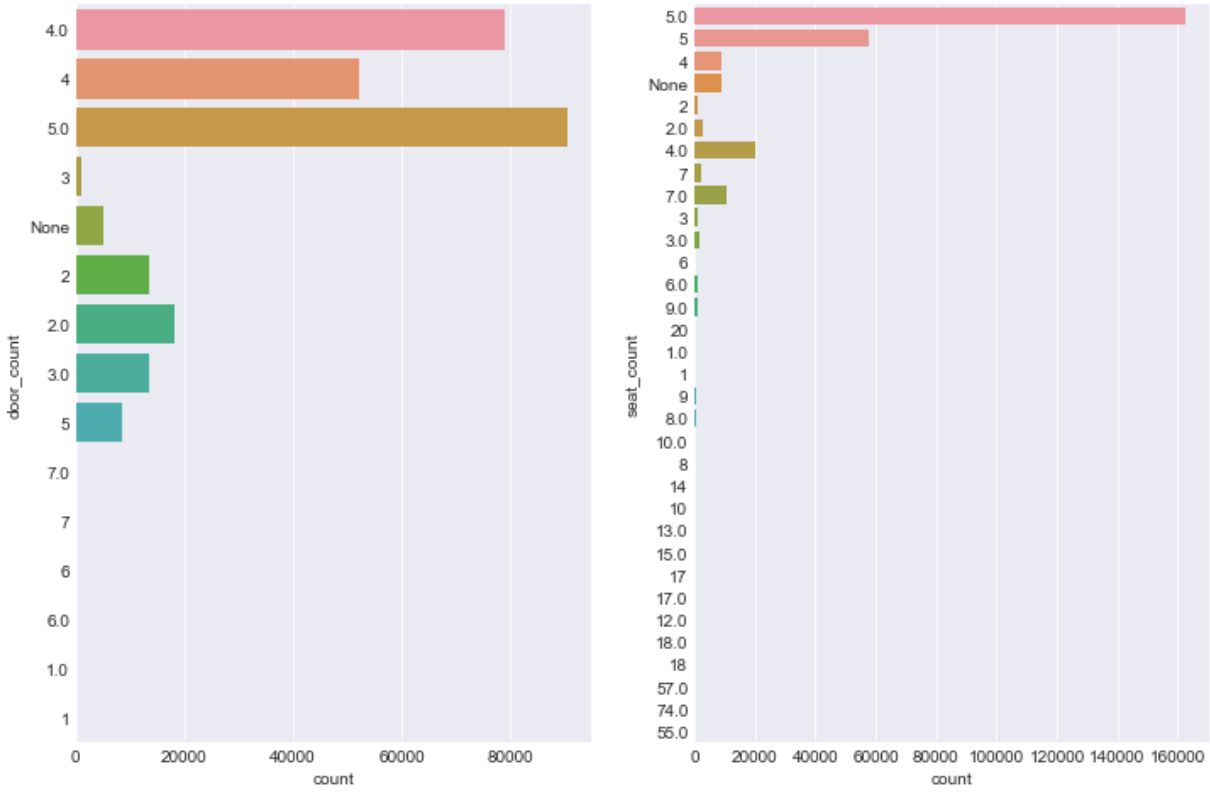
Remember to calculate the bounds before filtering so that they don't change as you incrementally

filter the dataset.

In [14]:
```
df = df[(df.mileage >= lower_bound.mileage) &
        (df.mileage <= upper_bound.mileage)]

df = df[df.manufacture_year >= lower_bound.manufacture_year]

df = df[(df.displacement >= lower_bound.displacement) &
        (df.displacement <= upper_bound.displacement)]

df = df[(df.horse_power >= lower_bound.horse_power) &
        (df.horse_power <= upper_bound.horse_power)]

df = df[(df.price >= lower_bound.price) &
        (df.price <= upper_bound.price)]
```

**Next, we'll investigate why `door_count` and `seat_count` are being treated as categorical features.**

In [15]:
```
# Plot distributions
fig, ax = plt.subplots(1,2, figsize=(12,8))
sns.countplot(y='door_count', data=df, ax=ax[0])
sns.countplot(y='seat_count', data=df, ax=ax[1])
plt.show()
```



Insights about car doors:

- '4' and '4.0' are being recorded as separate classes right now.
- 2-door and 4-door cars should be the most common.
- 3-door and 5-door "hatchback" cars are less common, but plausible.
- There should not be any 1-door cars or 58-door cars. These are clearly data-collection errors. We should remove these.
- The `None` values should probably be treated as missing values. Since the count of these values is relatively tiny, we're safe to just remove them.

Insights about seat counts:

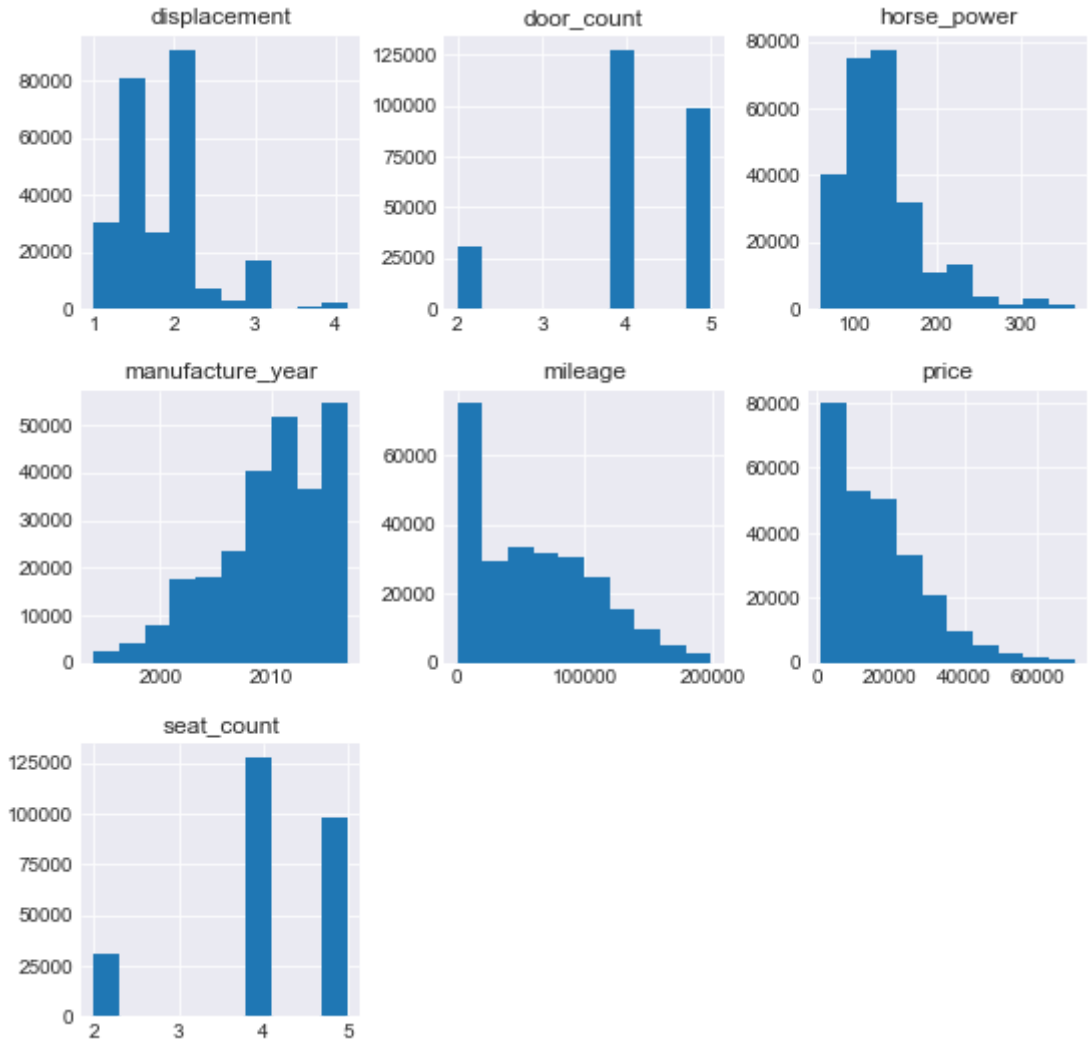
- '4' and '4.0' are being recorded as separate classes right now.
- We will keep cars with seat counts of 2,4,5, and 7.
- We will treat `None` values as missing values. Since the count of these values is relatively tiny, we're safe to just remove them.

We'll clean these features like so:

In [16]:
```
# Drop 'None'
df = df[(df.door_count != 'None') &
        (df.seat_count != 'None')]

# Convert values to numeric
df['door_count'] = df.door_count.astype(float)
df['seat_count'] = df.door_count.astype(float)

# Keep reasonable values
df = df[df.door_count.isin([2,3,4,5])]
df = df[df.seat_count.isin([2,4,5,7])]
```

After extensive data cleaning, our distributions should look more reasonable.

In [17]:
```
# Numeric feature distributions
df.hist(figsize=(9,9))
plt.show()
```




## Feature Engineering

For feature engineering, we'll create an "Age of Car" feature, a "New Car" indicator variable for cars with 0 mileage, and dummy variables.

We'll also drop the `manufacture_year` feature before saving our analytical base table, since that feature won't be helpful when predicting car prices in the *future*.

In [18]:
```
# Age of car
df['car_age'] = 2018 - df.manufacture_year

# Indicator feature for brand new cars
```

```
df['new_car'] = (df.mileage == 0) * 1

# Creat dummy variables and ABT
abt = pd.get_dummies(df.drop('manufacture_year', axis=1))
```

In general, on take-home challenges, it won't be useful to spend too much time on feature engineering (unless you are expected to already understand the domain at a deep level).

Instead, employers will be looking for you to acknowledge this step and create a few key, "low-hanging fruit" features.

In [19]:
```
abt.shape
```

Out[19]:
```
(257025, 108)
```

## Machine Learning

Since we've done the hard part of data cleaning and feature engineering first, training a model will be relatively straightforward. (If you're curious, you can try training this model *without* data cleaning first... you'll see that the performance sinks dramatically.)

**Note:** In take-home challenges, you typically do not need spend a large amount of time optimizing your model. Just pick a reasonable model (we prefer random forests), explain why (because they work well out-of-the-box, can model non-linear relationships, and are quite robust to outliers), and make a note of what you would try if given more time (tune hyperparameters, try other algorithms, etc).

In [20]:
```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import r2_score, mean_absolute_error
```

First, we'll split our dataset into a test set and a training set.

In [21]:
```
# Train / Test split
y = abt.price
X = abt.drop('price', axis=1)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=12
```

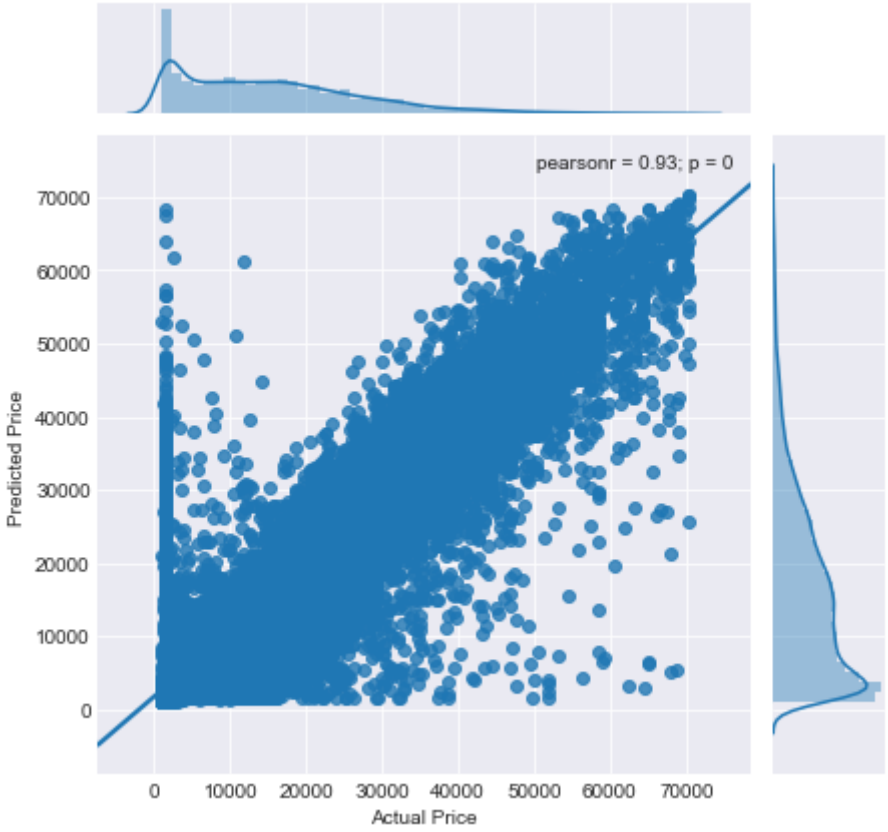
We'll fit and train a basic random forest.

In [22]:
```
# Train a basic random forest model
rf = RandomForestRegressor(random_state=1234)
```

```
rf.fit(X_train, y_train)

# Make prediction on test set
pred = rf.predict(X_test)
```

Next, we'll plot the model performance.

In [23]:
```
sns.jointplot(y_test, pred, kind='reg')
plt.xlabel('Actual Price')
plt.ylabel('Predicted Price')
plt.show()
```




In [24]:
```
# R^2
r2_score(y_test, pred)
```

Out[24]:
```
0.8650955940379994
```

In [25]:
```
mean_absolute_error(y_test, pred)
```

Out[25]:
```
2552.122907377165
```

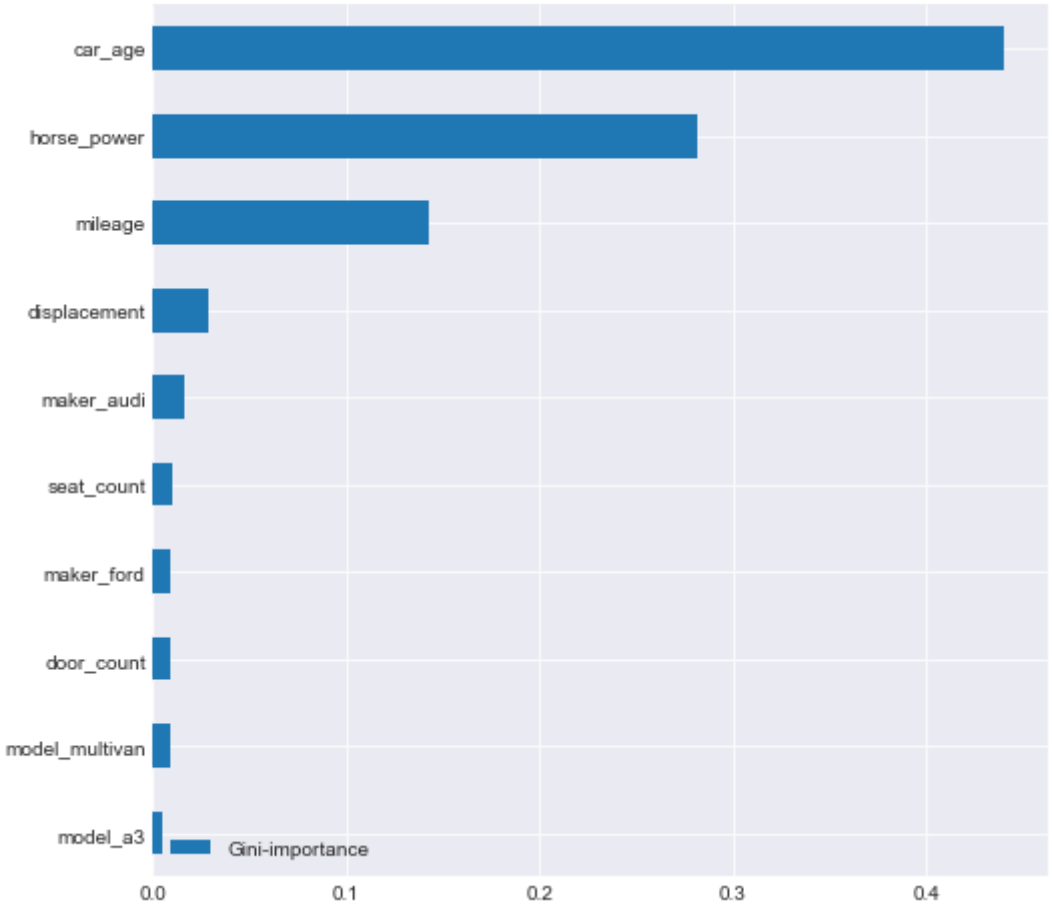
Our model performs quite well, especially given the limited time to train and optimize it.

Finally, we'll plot the feature importances.

In [26]:
```
# Helper function for plotting feature importances
```

```
def plot_feature_importances(columns, feature_importances, show_top_n=10)
    feats = dict( zip(columns, feature_importances) )
    imp = pd.DataFrame.from_dict(feats, orient='index').rename(columns={(
    imp.sort_values(by='Gini-importance').tail(show_top_n).plot(kind='bar
    plt.show()
```

In [27]:
```
plot_feature_importances(X_train.columns, rf.feature_importances_)
```




The age of the car, its horsepower, and its total mileage were the most important drivers of its price.