Regularization Techniques

Estimated time needed: **60** minutes

The purpose of Regularization techniques is to reduce the degree of overfitting that can occur in Regression models. Overfitting leads to poor ability of the model to make predictions on the new, unseen data. As we saw in the previous Regression Lessons, with a creation of extra features, such as through polynomial regression, a model can become easily overfit. To reduce the overfitting, we can regularize the model, or in other words, we can decrease its degrees of freedom. A simple way to regularize polynomial model is to reduce the number of polynomial degrees. For a linear regression model, regularization is typically achieved by constraining the weights of the model. Regularizer imposes a penalty on the size of the coefficients of the model.

In this lab, we will cover three types of regularizers:

- Ridge regression
- Lasso regression
- Elastic Net

Each one has its own advantages and disadvantages. Lasso will eliminate many features and reduce overfitting in your linear model. Ridge will reduce the impact of the features that are not important in predicting your target. Elastic Net combines feature elimination from Lasso and feature coefficient reduction from the Ridge model to improve your model's predictions.

The common features of all these regularizers include using cross-validation to select hyperparameters and applying data normalization to improve the performance.

Objectives

After completing this lab you will be able to:

- Understand the advantages and disadvantages of Ridge, Lasso and Elastic Net Regressions
- Apply Ridge, Lasso and Elastic Net Regressions
- Perform hyperparameters Grid Search on a model using validation data

---

Setup

For this lab, we will be using the following libraries:

- `pandas` for managing the data.
- `numpy` for mathematical operations.

- `seaborn` for visualizing the data.
- `matplotlib` for visualizing the data.
- `sklearn` for machine learning and machine-learning-pipeline related functions.
- `scipy` for statistical computations.

Import the required libraries

The following required modules are pre-installed in the Skills Network Labs environment. However if you run this notebook commands in a different Jupyter environment (e.g. Watson Studio or Ananconda) you will need to install these libraries by removing the `#` sign before `!mamba` in the code cell below.

In [1]:
```
# All Libraries required for this lab are listed below. The libraries
# !mamba install -qy pandas==1.3.4 numpy==1.21.4 seaborn==0.9.0 matpl
# Note: If your environment doesn't support "!mamba install", use "!p
```

In [2]:
```
!pip install -U scikit-learn
```

```
Requirement already satisfied: scikit-learn in /home/jupyterlab/cond
a/envs/python/lib/python3.7/site-packages (0.20.1)
Collecting scikit-learn
  Downloading scikit_learn-1.0.2-cp37-cp37m-manylinux_2_17_x86_64.man
ylinux2014_x86_64.whl (24.8 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 24.8/24.8 MB 43.9 MB/s
eta 0:00:0000:0100:01
Collecting joblib>=0.11
  Downloading joblib-1.2.0-py3-none-any.whl (297 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 298.0/298.0 kB 34.0 MB/s
eta 0:00:00
Requirement already satisfied: scipy>=1.1.0 in /home/jupyterlab/cond
a/envs/python/lib/python3.7/site-packages (from scikit-learn) (1.7.3)
Requirement already satisfied: numpy>=1.14.6 in /home/jupyterlab/cond
a/envs/python/lib/python3.7/site-packages (from scikit-learn) (1.21.
6)
Collecting threadpoolctl>=2.0.0
  Downloading threadpoolctl-3.1.0-py3-none-any.whl (14 kB)
Installing collected packages: threadpoolctl, joblib, scikit-learn
  Attempting uninstall: scikit-learn
    Found existing installation: scikit-learn 0.20.1
    Uninstalling scikit-learn-0.20.1:
      Successfully uninstalled scikit-learn-0.20.1
Successfully installed joblib-1.2.0 scikit-learn-1.0.2 threadpoolctl-
3.1.0
```

In [3]:
```
#import sklearn; print("Scikit-Learn", sklearn.__version__)
```

In [4]:
```
# Surpress warnings:
def warn(*args, **kwargs):
    pass
import warnings
warnings.warn = warn
```

In [5]:
```
import pandas as pd
import numpy as np

import seaborn as sns
import matplotlib.pylab as plt
%matplotlib inline

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression,Ridge,Lasso,Elastic
from sklearn.metrics import r2_score
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import scale
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
from sklearn.feature_selection import SelectKBest, f_regression
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV
from sklearn.decomposition import PCA
```

First, let's define some functions that will help us in the future analysis.

Below function will calculate the R^2 on each feature given the input of the model.

In [6]:
```
def get_R2_features(model,test=True):
    #X: global
    features=list(X)
    features.remove("three")

    R_2_train=[]
    R_2_test=[]

    for feature in features:
        model.fit(X_train[[feature]],y_train)

        R_2_test.append(model.score(X_test[[feature]],y_test))
        R_2_train.append(model.score(X_train[[feature]],y_train))

    plt.bar(features,R_2_train,label="Train")
    plt.bar(features,R_2_test,label="Test")
    plt.xticks(rotation=90)
    plt.ylabel("$R^2$")
    plt.legend()
    plt.show()
    print("Training R^2 mean value {} Testing R^2 mean value {} ".for
    print("Training R^2 max value {} Testing R^2 max value {} ".forma
```

Below function will plot the estimated coefficients for each feature and find R^2 on training and testing sets.

In [7]:
```
def plot_coef(X,model,name=None):


    plt.bar(X.columns[2:],abs(model.coef_[2:]))
    plt.xticks(rotation=90)
    plt.ylabel("$coefficients$")
    plt.title(name)
    plt.show()
    print("R^2 on training  data ",model.score(X_train, y_train))
    print("R^2 on testing data ",model.score(X_test,y_test))
```

Below function plots the distribution of two inputs.

In [8]:
```
def  plot_dis(y,yhat):

    plt.figure()
    ax1 = sns.distplot(y, hist=False, color="r", label="Actual Value'
    sns.distplot(yhat, hist=False, color="b", label="Fitted Values" ,
    plt.legend()

    plt.title('Actual vs Fitted Values')
    plt.xlabel('Price (in dollars)')
    plt.ylabel('Proportion of Cars')
    plt.show()
    plt.close()
```

Reading and understanding our data

For this lab, we will be using the car sales dataset, hosted on IBM Cloud object storage. The dataset contains all the information about cars, the name of the manufacturer, the year it was launched, all car technical parameters, and the sale price. This dataset has already been pre-cleaned and encoded (using one-hot and label encoders) in the Linear Regression Notebook.

Let's read the data into *pandas* data frame and look at the first 5 rows using the `head()` method.

In [9]:
```
data = pd.read_csv('https://cf-courses-data.s3.us.cloud-object-storag
data.head()
```

Out[9]:

	diesel	gas	std	turbo	convertible	hardtop	hatchback	sedan	wagon	4wd
0	0.0	1.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
1	0.0	1.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
2	0.0	1.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
3	0.0	1.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0
4	0.0	1.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0

5 rows × 36 columns

We can find more information about the features and types using the `info()` method.

In [10]:
```
data.info()
```

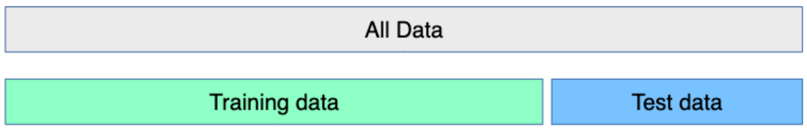
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 205 entries, 0 to 204
Data columns (total 36 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   diesel       205 non-null    float64
 1   gas          205 non-null    float64
 2   std          205 non-null    float64
 3   turbo        205 non-null    float64
 4   convertible  205 non-null    float64
 5   hardtop      205 non-null    float64
 6   hatchback    205 non-null    float64
 7   sedan        205 non-null    float64
 8   wagon        205 non-null    float64
 9   4wd          205 non-null    float64
 10  fwd          205 non-null    float64
 11  rwd          205 non-null    float64
 12  dohc         205 non-null    float64
 13  dohcv        205 non-null    float64
 14  l            205 non-null    float64
 15  ohc          205 non-null    float64
 16  ohcf         205 non-null    float64
 17  ohcv         205 non-null    float64
 18  rotor        205 non-null    float64
 19  eight        205 non-null    float64
 20  five         205 non-null    float64
 21  four         205 non-null    float64
 22  six          205 non-null    float64
 23  three        205 non-null    float64
 24   twelve       205 non-null    float64
 25   two          205 non-null    float64
 26   wheelbase    205 non-null    float64
 27   curbweight   205 non-null    float64
 28   enginesize   205 non-null    float64
 29   boreratio    205 non-null    float64
 30   horsepower   205 non-null    float64
 31   carlength    205 non-null    float64
 32   carwidth     205 non-null    float64
 33   citympg      205 non-null    float64
 34   highwaympg   205 non-null    float64
 35   price        205 non-null    float64
dtypes: float64(36)
memory usage: 57.8 KB
```

Data Preparation

Let's first split our data into `X` features and `y` target.

In [11]:
```
X = data.drop('price', axis=1)
y = data.price
```

Now that we have split our data into training and testing sets, the training data is used for your model to recognize patterns using some criteria,the test data set it used to evaluate your model, as shown in the following image:



source scikit-learn.org

Now, we split our data, using `train_test_split` function, into the training and testing sets, allocating 30% of the data for testing.

In [12]:
```
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.
print("number of test samples :", X_test.shape[0])
print("number of training samples:",X_train.shape[0])
```

```
number of test samples : 21
number of training samples: 184
```

Linear Regression

In linear regression we are trying to find the value of $\mathbf{w}$ that minimizes the Mean Squared Error (MSE), we can represent this using the following expression:

$$\hat{\mathbf{w}} = \underset{\mathbf{w}}{\operatorname{argmin}} ||\mathbf{y} - \mathbf{X}\mathbf{w}||^2\_2$$

Where $\mathbf{y}$ is the target, $\mathbf{X}$ is the training set and $\mathbf{w}$ is the parameter weights. The resulting $\hat{\mathbf{w}}$ is the best value to minimize the MSE, i.e., the distance between the target $\mathbf{y}$ and the estimate $\mathbf{X}\mathbf{w}$. We do this by fitting the model.

Let's create a `LinearRegression` object, called `lm`.

In [13]:
```
lm = LinearRegression()
```

Now, let's fit the model with multiple features on our X_train and y_train data.

In [14]:
```
lm.fit(X_train, y_train)
```

Out[14]:
```
LinearRegression()
```

We apply `predict( )` function on the testing data set.

In [15]:
```
predicted = lm.predict(X_test)
```

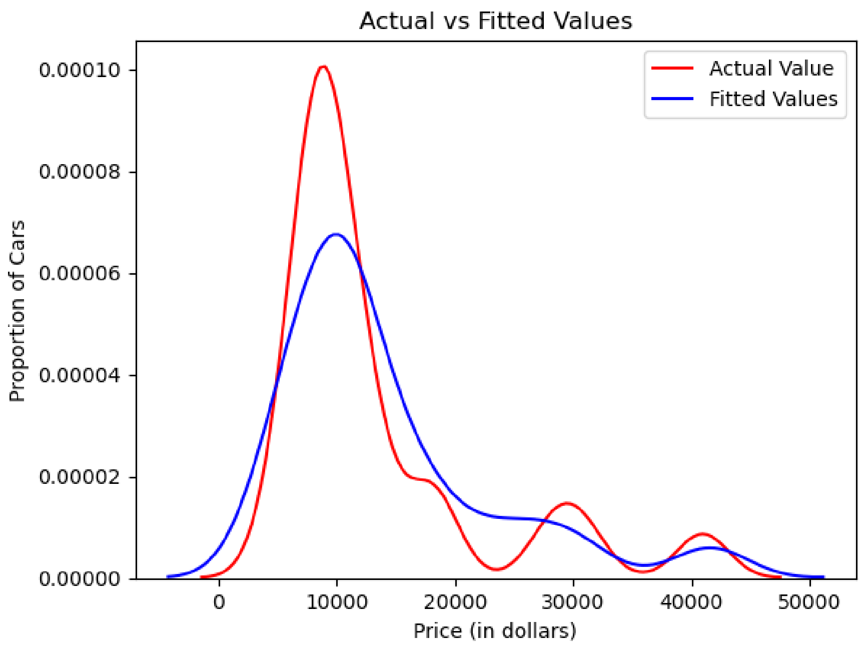
Let's calculate the `$R^2$` on both, training and testing data sets.

In [16]:
```
print("R^2 on training  data ",lm.score(X_train, y_train))
print("R^2 on testing data ",lm.score(X_test,y_test))
```

```
R^2 on training  data  0.9092101381197338
R^2 on testing data  0.9472499250320212
```

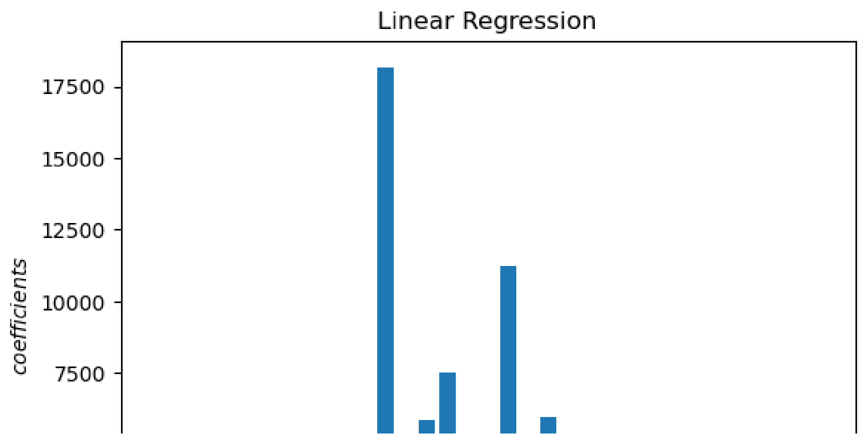
We can plot a distribution of the predicted values vs the actual values.

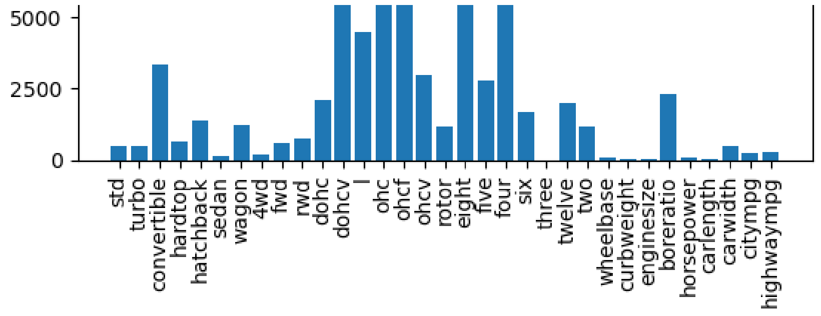
In [17]:
```
plot_dis(y_test,predicted)
```



We can view the estimated coefficients for the linear regression problem and drop the top two coefficients, as they are two large.

In [18]:
```
plot_coef(X,lm,name="Linear Regression")
```





```
R^2 on training  data  0.9092101381197338
R^2 on testing data  0.9472499250320212
```

Ridge Regression

Let's review the Ridge Regression. Ridge Regression makes the prior assumption that our coefficients are normally distributed around zero. A regularization term, alpha, is added to the cost function. This forces the learning algorithm to not only fit the data but also keep the model weights as small as possible. The variance of the distribution is inversely proportional to the parameter alpha. This is also called the L2 regularizer , as it adds a L2 penalty to the minimization term, as shown here:

$$\hat{\mathbf{w}} = \underset{\mathbf{w}}{\mathrm{argmin}} ||\mathbf{y} - \mathbf{X}\mathbf{w}||^2\_2 + \alpha ||\mathbf{w}||\_2$$

We minimize the MSE, but we also penalize large weights by including their magnitude $||\mathbf{w}||\_2$ in the minimization term. This additional minimization term makes the model less susceptible to noise and makes the weights smaller. Alpha controls the takeoff between MSE and penalization or regularization term and is chosen via cross-validation.

Let's see how the parameter alpha changes the model. Note, here our test data will be used as validation data. Also, the regularization term should only be added to the cost function during the training.

Let's create a Ridge Regression object, setting the regularization parameter (alpha) to 0.01.

In [19]:
```
rr = Ridge(alpha=0.01)
rr
```

Out[19]:
```
Ridge(alpha=0.01)
```

Like regular regression, you can fit the model using the `fit()` method.

In [20]:
```
rr.fit(X_train, y_train)
```

Out[20]:
```
Ridge(alpha=0.01)
```

Similarly, you can obtain a prediction:

In [21]:
```
rr.predict(X_test)
```

Out[21]:
```
array([30178.77172992, 22179.93145434, 11229.58960483, 11790.4033714
9,
       26348.13785546,  5439.13547145,  9054.02541015,  7265.3655856
3,
       10591.48456189, 10390.82134687, 17471.12024994,  7010.4743022
8,
       16547.06078383, 10468.27937016, 41540.38102791,  5390.3465915
2,
        5109.76377302, 15373.1930261 , 10703.56615831, 11448.3324270
2,
       10565.49269055])
```

We can calculate the R^2 on the training and testing data.

In [22]:

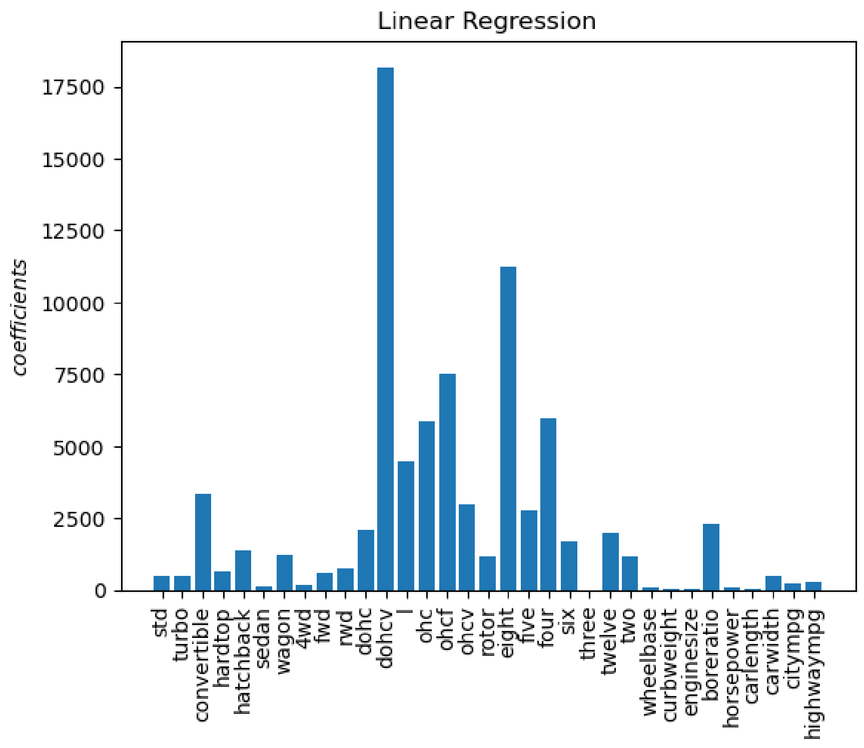
```
print("R^2 on training  data ",rr.score(X_train, y_train))
print("R^2 on testing data ",rr.score(X_test,y_test))
```

```
R^2 on training  data  0.9091956531801182
R^2 on testing data  0.9478784615596495
```

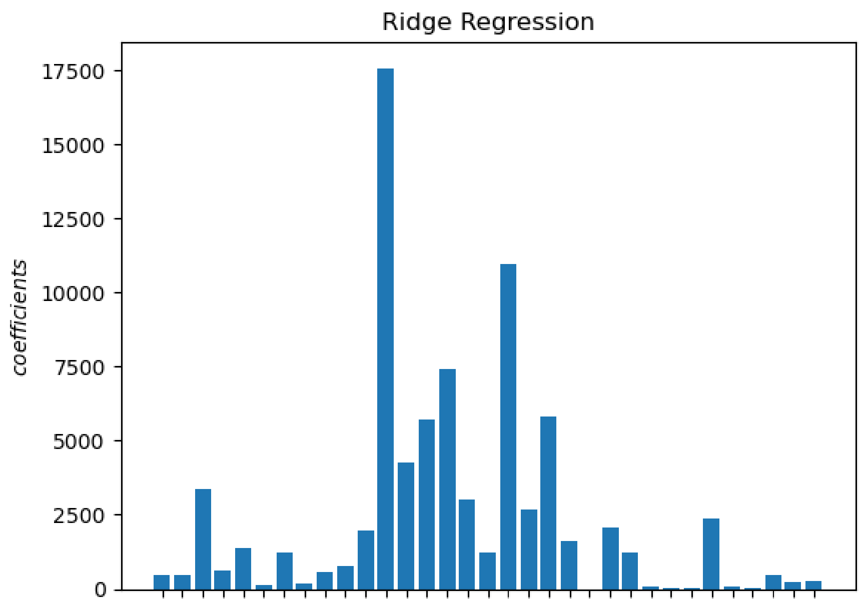
Now let's compare the Ridge Regression and the Linear Regression models. The results on the R^2 are about the same, and the coefficients seem to be smaller.

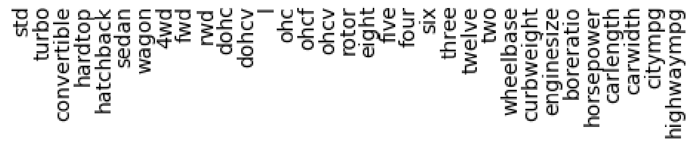
In [23]:

```
plot_coef(X,lm,name="Linear Regression")
plot_coef(X,rr,name="Ridge Regression")
```



```
R^2 on training  data  0.9092101381197338
R^2 on testing data  0.9472499250320212
```



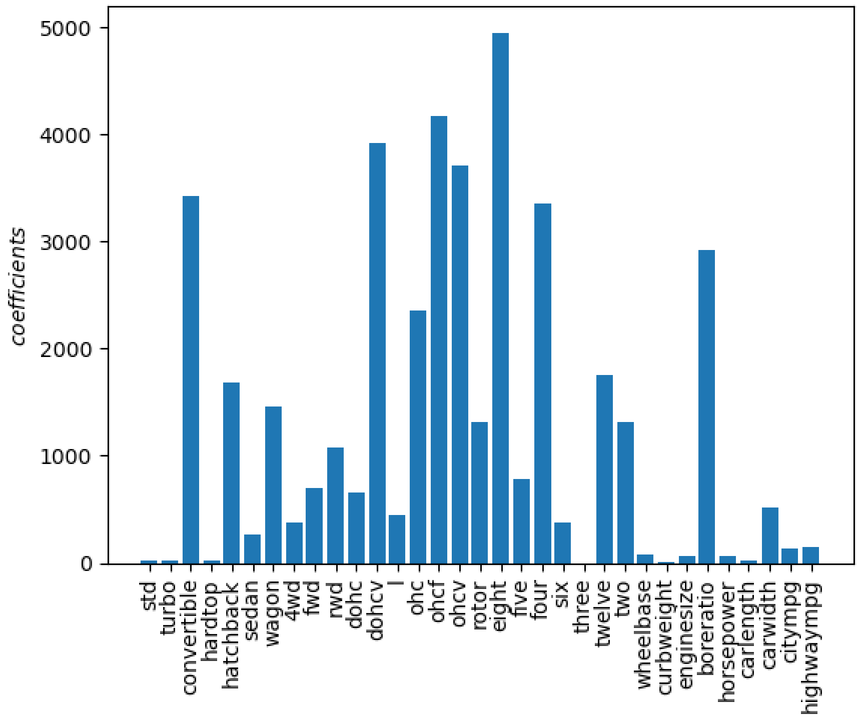


```
R^2 on training  data  0.9091956531801182
R^2 on testing data  0.9478784615596495
```

If we increase alpha, the coefficients get smaller, but the results are not as good as our previous value of alpha.

In [24]:

```python
rr = Ridge(alpha=1)
rr.fit(X_train, y_train)
plot_coef(X,rr)
```

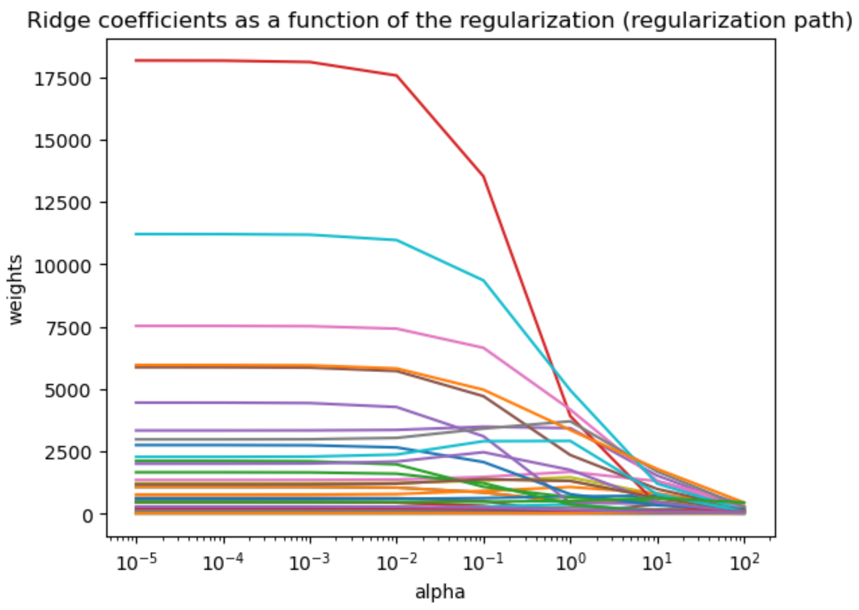


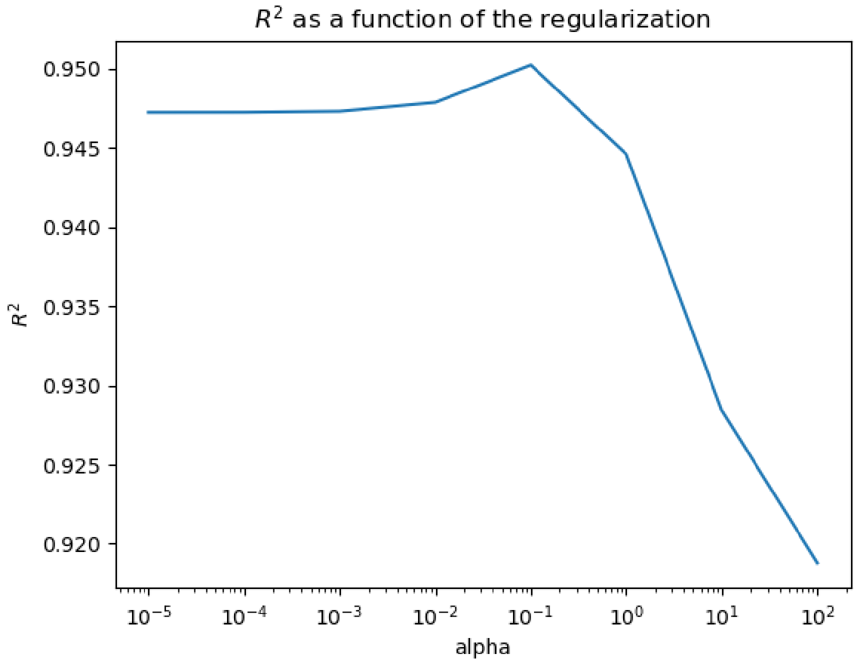
```
R^2 on training  data  0.8991374778636105
R^2 on testing data  0.9446031107273959
```

In general, we see that if we increase alpha, the coefficients get smaller, but the model performance relationship gets more complex. As a result, we use the validation data to select a value for alpha. Here, we plot the coefficients and R^2 of the test data on the vertical axes and alpha on the horizontal axis, as well the R^2 using the test data.

In [25]:

```python
alphas = [0.00001,0.0001,0.001,0.01,0.1,1,10,100]
R_2=[]
coefs = []
for alpha in alphas:
    ridge = Ridge(alpha=alpha)
    ridge.fit(X_train, y_train)
    coefs.append(abs(ridge.coef_))
    R_2.append(ridge.score(X_test,y_test))


ax = plt.gca()
ax.plot(alphas, coefs)
ax.set_xscale("log")
plt.xlabel("alpha")
plt.ylabel("weights")
plt.title("Ridge coefficients as a function of the regularization (re
plt.show()


ax = plt.gca()
ax = plt.gca()
ax.plot(alphas, R_2)
ax.set_xscale("log")
plt.xlabel("alpha")
plt.ylabel("$R^2$")
plt.title("$R^2$ as a function of the regularization")
plt.show()
```





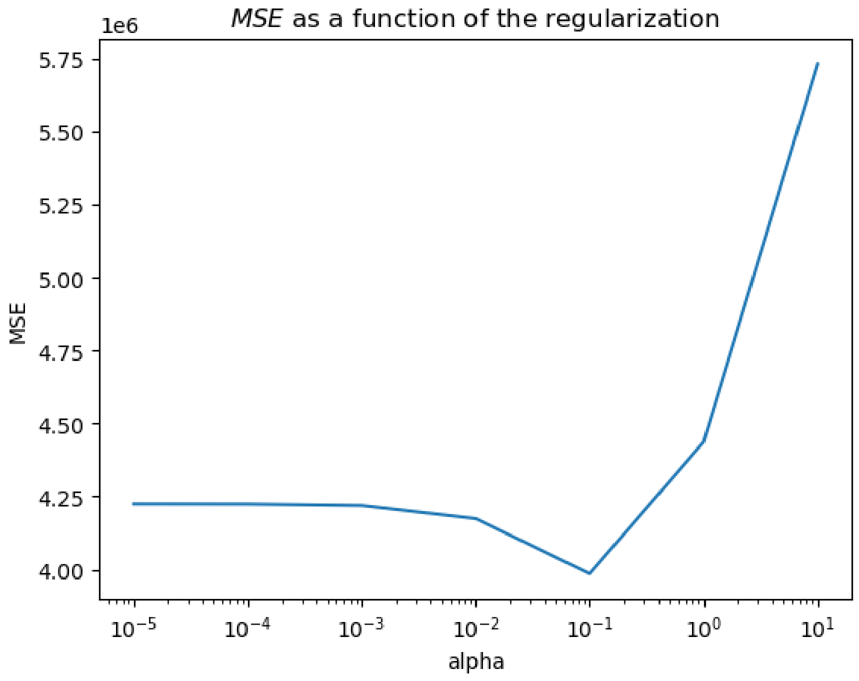
As we increase alpha, the coefficients get smaller but the $R^2$ peaks when alpha is 1.

Exercise 1

In this Exercise, plot the MSE as a function of alpha. What pattern do you notice?

In [27]:

```
# Enter your code and run the cell
alphas = [0.00001,0.0001,0.001,0.01,0.1,1,10]
MEAN_SQE=[]

for alpha in alphas:
    ridge = Ridge(alpha=alpha)
    ridge.fit(X_train, y_train)
    MEAN_SQE.append(mean_squared_error(ridge.predict(X_test),y_test))

ax = plt.gca()
ax.plot(alphas, MEAN_SQE)
ax.set_xscale("log")
plt.xlabel("alpha")
plt.ylabel("MSE")
plt.title("$MSE$ as a function of the regularization")
plt.show()
```



Answers (Click Here) A small alpha leads to over-fitting but as alpha gets larger the MSE decreases. When alpha gets too large the MSE increases leading to underfitting. The optimal point seems to be in the middle .

Pipeline

We can also create a Pipeline object and apply a set of transforms sequentially. Then, we can apply Polynomial Features, perform data standardization then apply Ridge regression. Data Pipelines simplify the steps of processing the data. We use the module `Pipeline` to create a pipeline. We also use `StandardScaler` step in our pipeline. Scaling our data is necessary step in Ridge regression as it will penalize features with a large magnitude.

Now, we create a pipeline object.

In [28]:
```
Input=[ ('polynomial', PolynomialFeatures(include_bias=False,degree=2
pipe = Pipeline(Input)
```

We fit the object.

In [29]:
```
pipe.fit(X_train, y_train)
```

Out[29]:
```
Pipeline(steps=[('polynomial', PolynomialFeatures(include_bias=Fals
e)),
                ('ss', StandardScaler()), ('model', Ridge(alpha=1))])
```

We can calculate the score on the test data.

In [30]:

```
predicted=pipe.predict(X_test)
pipe.score(X_test, y_test)
```

Out[30]:
```
0.9075262214621758
```

Looking for hyperparameters can get difficult with loops. The problem will get worse as we add more transforms such as polynomial transform. Therefore, we can use `GridSearchCV` to make things simpler.

GridSearchCV

To search for the best combination of hyperparameters we can create a `GridSearchCV()` function as a dictionary of parameter values. The parameters of pipelines can be set by using the name of the key, separated by "__", then the parameter. Here, we look for different polynomial degrees and different values of alpha.

In [31]:
```
param_grid = {
    "polynomial__degree": [1,2,3,4],
    "model__alpha":[0.0001,0.001,0.01,0.1,1,10]
}
```

Keys of the dictionary are the model "key name __" followed by the parameter as an attribute.

polynomial__degree: is the degree of the polynomial; in this case 1, 2, 3, 4 and 5.

model__alpha : Regularization strength; must be a positive float.

We create a `GridSearchCV` object and fit it. The method trains the model and the hyperparameters are selected via exhaustive search over the specified values.

In [32]:
```
search = GridSearchCV(pipe, param_grid, n_jobs=2)

search.fit(X_train, y_train)
search
```

Out[32]:
```
GridSearchCV(estimator=Pipeline(steps=[('polynomial',
                                         PolynomialFeatures(include_bi
as=False)),
                                        ('ss', StandardScaler()),
                                        ('model', Ridge(alpha=1))]),
             n_jobs=2,
             param_grid={'model__alpha': [0.0001, 0.001, 0.01, 0.1,
1, 10],
                         'polynomial__degree': [1, 2, 3, 4]})
```

We can input the results into *pandas* `DataFrame()` as a dictionary with keys as column headers and values as columns and display the results.

In [33]:
```
pd.DataFrame(search.cv_results_).head()
```

Out[33]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_model_alpl
0	0.022561	0.018511	0.004366	0.003803	0.00

1	0.025547	0.009405	0.004660	0.003634	0.00
2	0.126840	0.034384	0.011123	0.004579	0.00
3	1.315031	0.328870	0.095507	0.013646	0.00
4	0.008750	0.005370	0.007882	0.007624	0.0

There are some other useful attributes:

`best_score_` : mean cross-validated score of the `best_estimator` .

`best_params_dict` : parameter setting that gives the best results on the hold-out data.

In [34]:
```python
print("best_score_: ",search.best_score_)
print("best_params_: ",search.best_params_)
```

```
best_score_:  0.8681282282391243
best_params_:  {'model__alpha': 10, 'polynomial__degree': 2}
```

We can call `predict()` on the estimator with the best found parameters.

In [35]:
```python
predict = search.predict(X_test)

predict
```

Out[35]:
```
array([34464.68303015, 23419.34979609,  9723.38208924, 13433.4255162
5,
       25239.67969507,  6532.54733115,  7468.55986929,  7687.5089222
6,
        9114.59988701,  9541.72323357, 15453.55023026,  6815.6403158
3,
       16958.71711106, 10290.1730189 , 42072.30176794,  6356.5335292
3,
        1536.17210277, 13705.75183086,  9758.24171738,  9210.4005416
7,
        9999.70716056])
```

We can find the best model.

In [36]:
```python
best=search.best_estimator_
best
```

Out[36]:
```
Pipeline(steps=[('polynomial', PolynomialFeatures(include_bias=Fals
e)),
                ('ss', StandardScaler()), ('model', Ridge(alpha=1
0))])
```

As we can see from the above output, it is five degree polynomial with alpha value of 0.0001. Now, let's make a prediction.

In [37]:
```python
predict = best.predict(X_test)
predict
```

Out[37]:
```
array([34464.68303015, 23419.34979609,  9723.38208924, 13433.4255162
5,
        25239.67969507,  6532.54733115,  7468.55986929,  7687.5089222
6,
         9114.59988701,  9541.72323357, 15453.55023026,  6815.6403158
3,
        16958.71711106, 10290.1730189 , 42072.30176794,  6356.5335292
3,
         1536.17210277, 13705.75183086,  9758.24171738,  9210.4005416
7,
         9999.70716056])
```

We can calculate the R^2 on the test data.

In [38]:
```
best.score(X_test, y_test)
```

Out[38]:
```
0.9441203657848715
```

As we see, using Ridge Regression polynomial function works better than all other models. Finely, we can train our model on the entire data set!

In [39]:
```
best.fit(X,y)
```

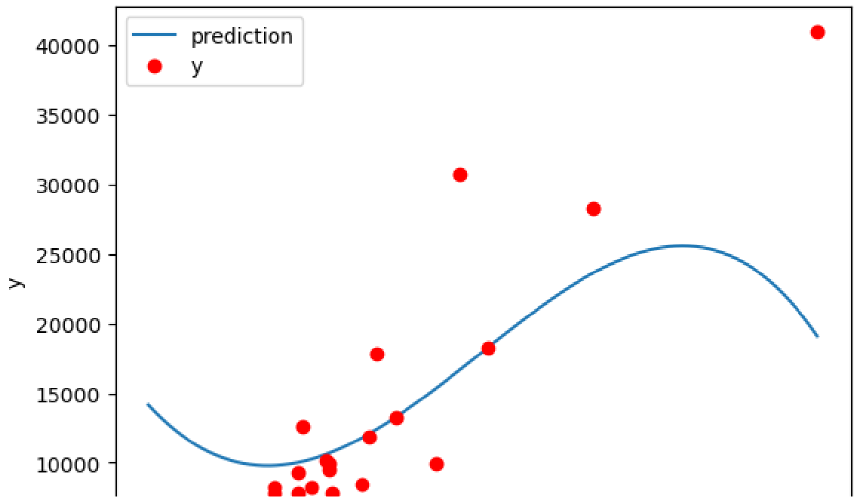
Out[39]:
```
Pipeline(steps=[('polynomial', PolynomialFeatures(include_bias=Fals
e)),
                ('ss', StandardScaler()), ('model', Ridge(alpha=1
0))])
```

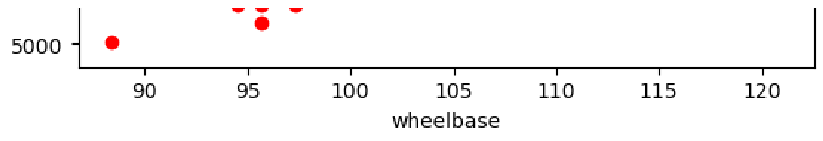
Exercise 2

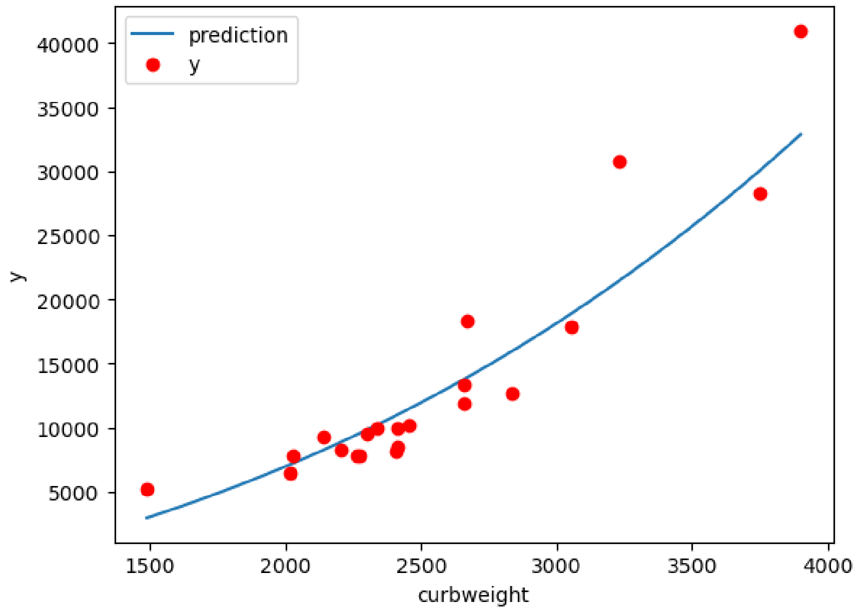
Perform grid search on the following features and plot the results by completing the following lines of code:

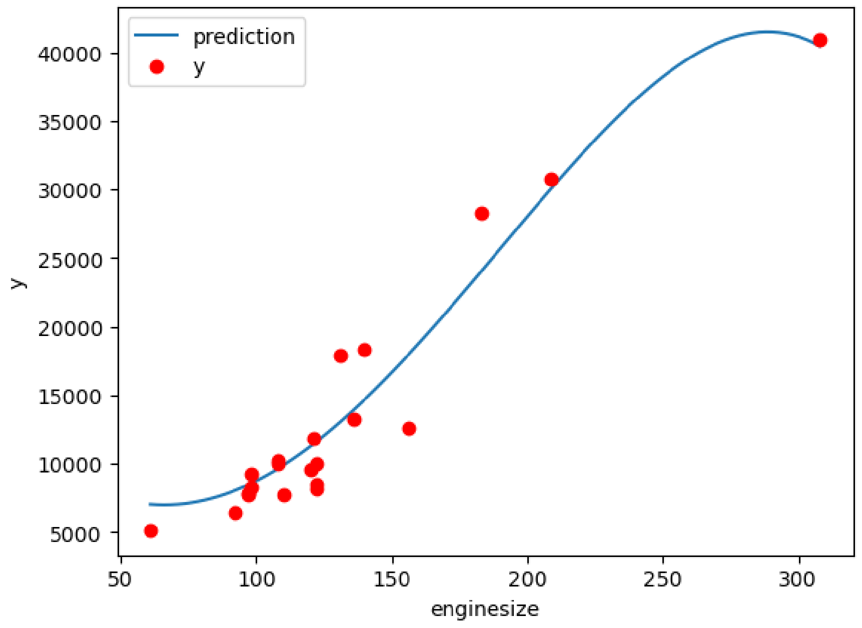
In [40]:
```
# Run the cell
columns=['wheelbase', 'curbweight', 'enginesize', 'boreratio', 'horse
       'carlength', 'carwidth', 'citympg']
```

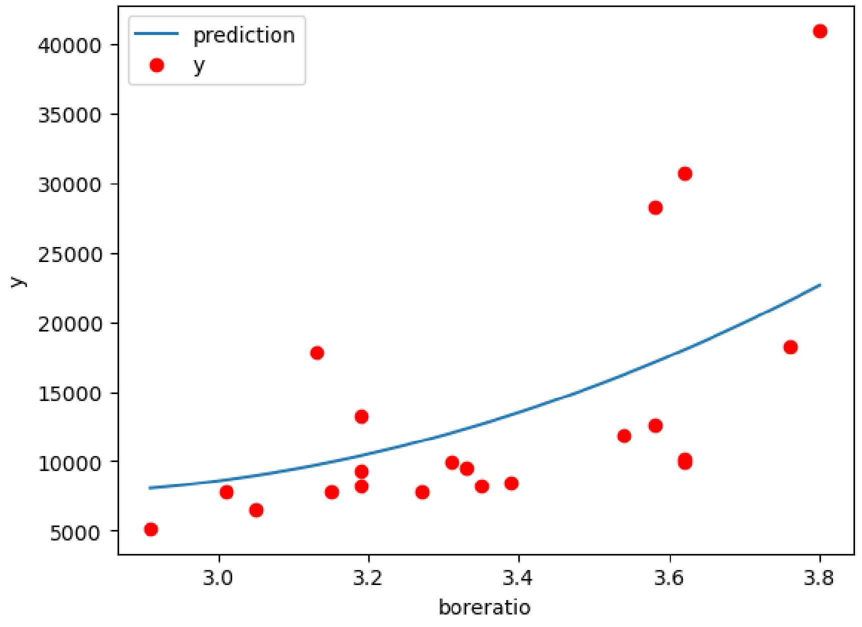
In [42]:
```
# Enter your code and run the cell
for column in columns:
    search.fit(X_train[[column]], y_train)
    x=np.linspace(X_test[[column]].min(), X_test[[column]].max(),num=
    plt.plot(x,search.predict(x.reshape(-1,1)),label="prediction")
    plt.plot(X_test[column],y_test,'ro',label="y")
    plt.xlabel(column)
    plt.ylabel("y")
    plt.legend()
    plt.show()
```

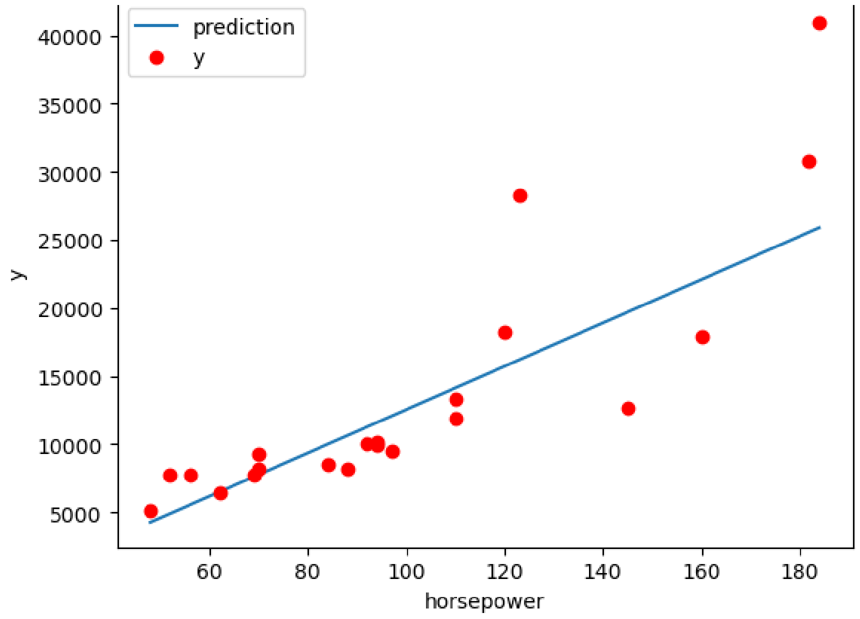


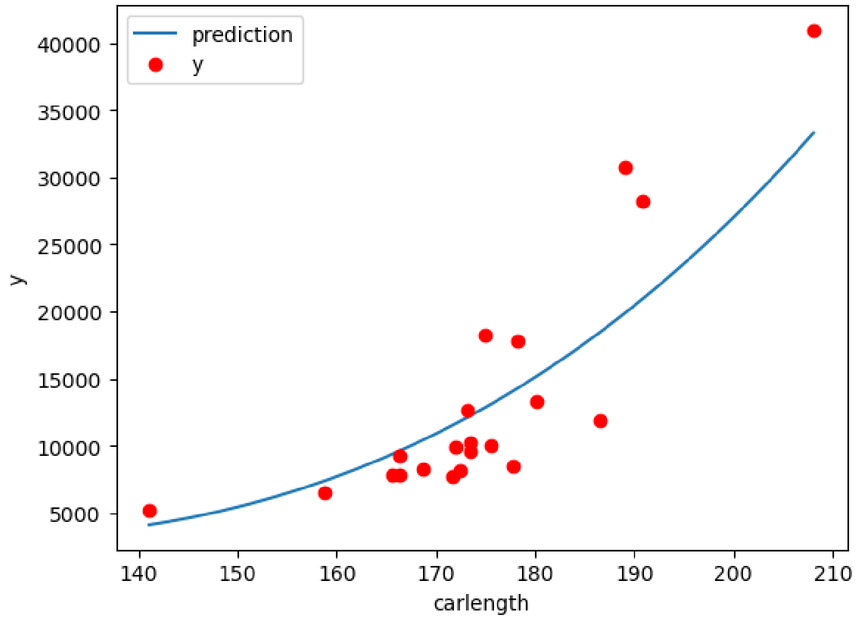

5000
90
95
100
105
110
115
120
wheelbase

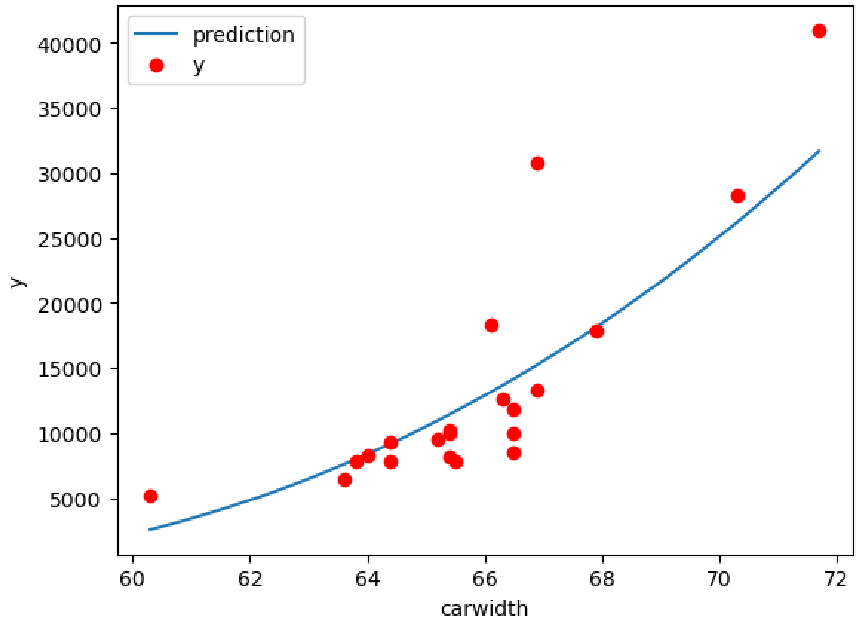

prediction
y
40000
35000
30000
25000
20000
15000
10000
5000
y
1500
2000
2500
3000
3500
4000
curbweight


prediction
y
40000
35000
30000
25000
20000
15000
10000
5000
y
50
100
150
200
250
300
enginesize

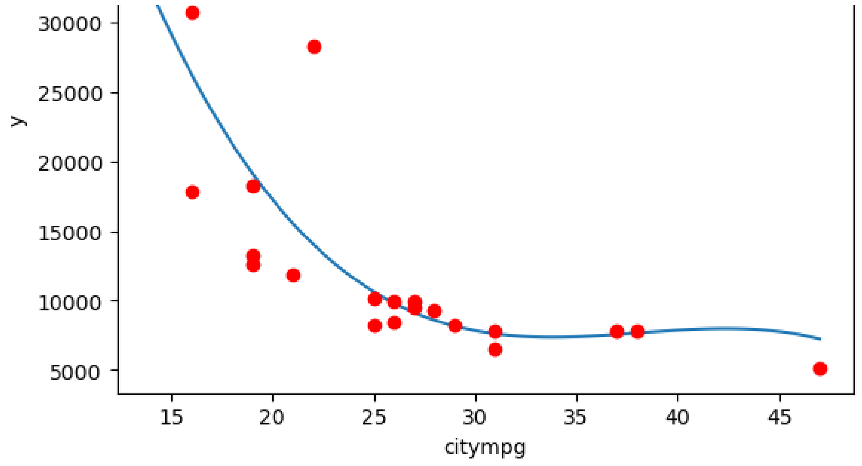

prediction
y
40000
35000
30000
25000
20000
15000
10000
5000
y
3.0
3.2
3.4
3.6
3.8
boreratio


prediction
y
40000
35000
30000
25000
20000
15000
10000
5000
y
60
80
100
120
140
160
180
horsepower


prediction
y
40000
35000
30000
25000
20000
15000
10000
5000
y
140
150
160
170
180
190
200
210
carlength


prediction
y
40000
35000
30000
25000
20000
15000
10000
5000
y
60
62
64
66
68
70
72
carwidth

prediction
y
40000
35000



Lasso Regression

In this section, let's review the Lasso (Least Absolute Shrinkage and Selection Operator) Regression. Lasso Regression makes the prior assumption that our coefficients have Laplace (double-exponential) distribution around zero. The scale parameter of the distribution is inversely proportional to the parameter alpha. The main advantage of LASSO Regression is that many coefficients are set to zero, therefore they are not required. This has many advantages, one of them is that you may not need to collect and/or store all of the features. This may save resources. For example, if the feature was some medical test, you would no longer need to perform that test. Let's see how the parameter alpha changes the model. We minimize the MSE, but we also penalize large weights by including their sum of absolute values $||\mathbf{w}||\_1$, symbolically:

$$\hat{\mathbf{w}} = \underset{\mathbf{w}}{\text{argmin}} ||\mathbf{y} - \mathbf{X}\mathbf{w}||^2\_2 + \alpha||\mathbf{w}||\_1$$

This regularization or penalty term makes many coefficients zero, making the model easy to understand and can also be used for feature selection. There are some drawbacks to this technique. It takes longer time to train and the solution may not be unique. Alpha controls the trade-off between MSE and penalization or regularization term and is chosen via cross-validation. Let's see how the parameter alpha changes the model. Note, as before, our test data will be used as validation data. Let's create a Ridge Regression object, setting the regularization parameter (alpha) to 0.01.

In [43]:
```
la = Lasso(alpha=0.1)
la.fit(X_train,y_train)
la
```

Out[43]:
```
Lasso(alpha=0.1)
```

Let's make a prediction.

In [44]:
```
predicted = la.predict(X_test)
predicted
```

Out[44]:
```
array([30207.01664593, 22253.28978157, 11234.4293658 , 11825.5141138
1,
       26349.51670682,  5442.67340587,  9115.38970742,  7262.3783497
1,
       10602.22642602, 10426.6507622 , 17480.62216135,  6988.6068362
2,
       16570.42746876, 10462.5969077 , 41586.07832506,  5399.2518604
3,
        3317.54502062, 15367.96191917, 10698.17086097, 11451.7264468
4,
       10548.16668782])
```

Let's calculate the R^2 on the training and testing data and see how it performs compared to the other methods.

In [45]:

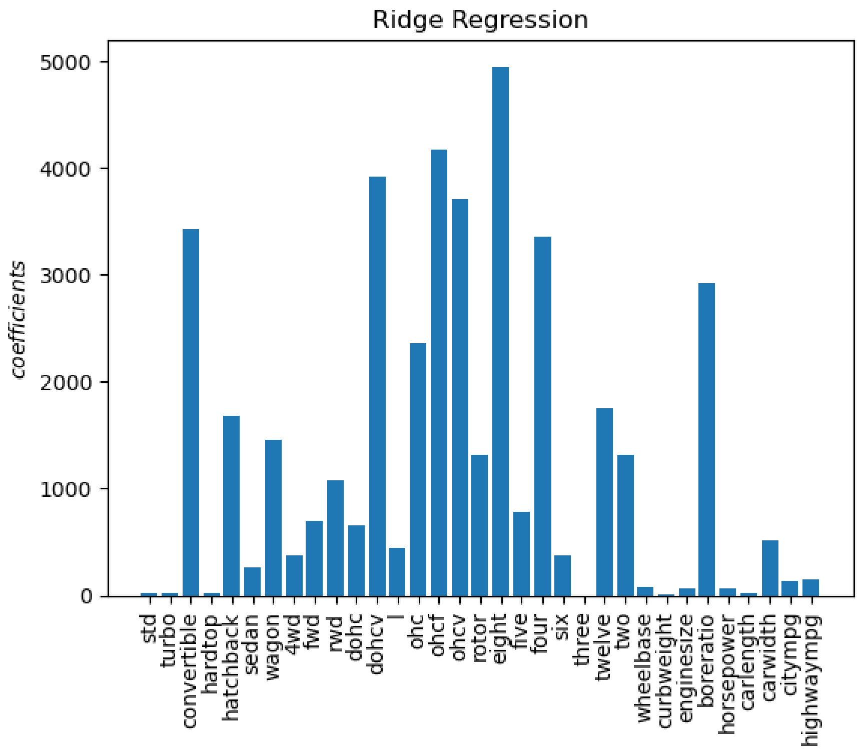
```
print("R^2 on training  data ",lm.score(X_train, y_train))
print("R^2 on testing data ",lm.score(X_test,y_test))
```

```
R^2 on training  data  0.9092101381197338
R^2 on testing data  0.9472499250320212
```

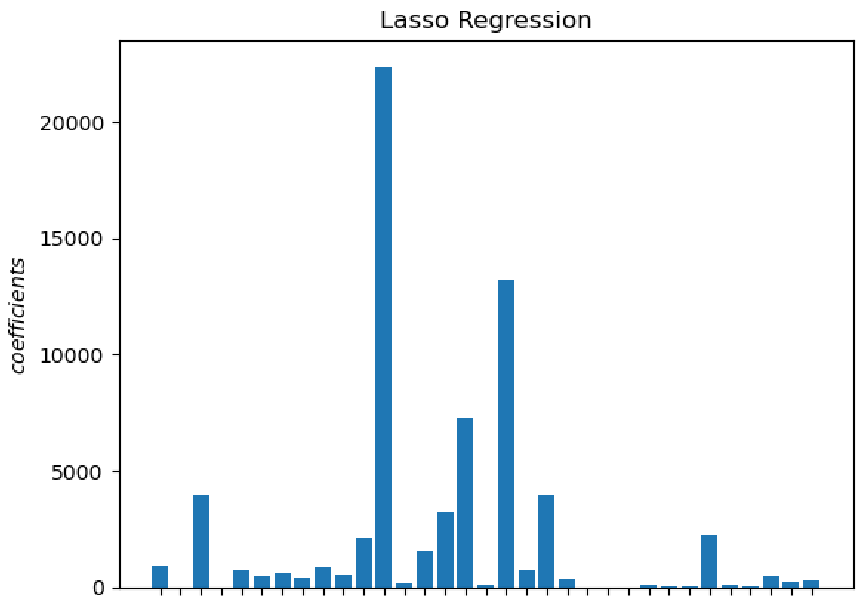
If we compare the Lasso Regression to the Ridge Regression model we see that the results on the R^2 are slightly worse, but most of the coefficients are zero.

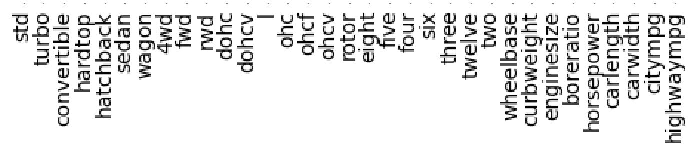
In [46]:

```
plot_coef(X,rr,name="Ridge Regression")
plot_coef(X,la,name="Lasso Regression")
```



```
R^2 on training  data  0.8991374778636105
R^2 on testing data  0.9446031107273959
```



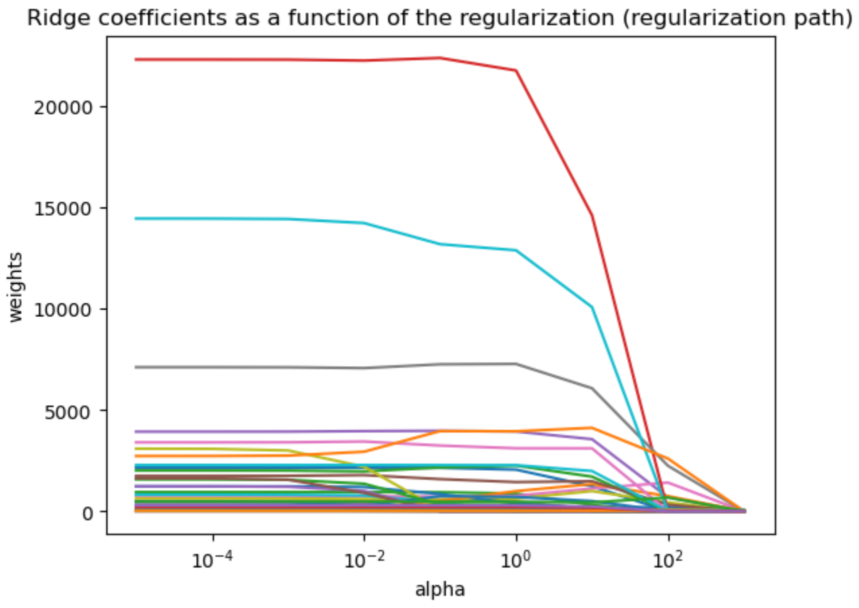


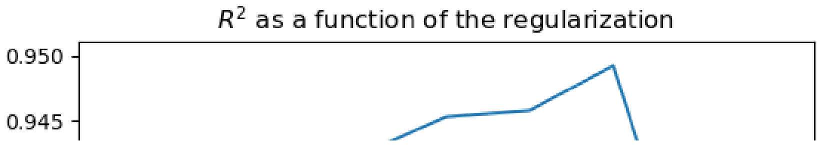
```
R^2 on training  data  0.9092098726971483
R^2 on testing data  0.9453258869077994
```

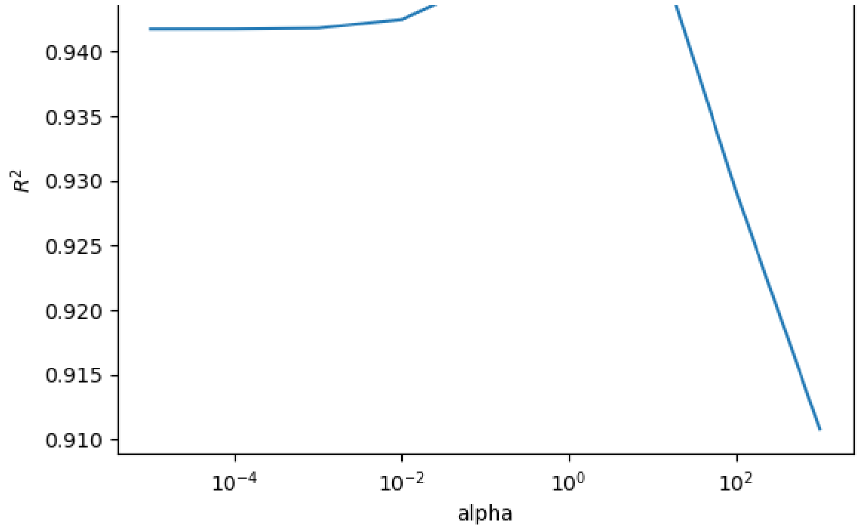
Similar to the Ridge Regression, if we increase the value of alpha, the coefficients will get smaller. Additionally, many coefficients become zero. Moreover, the model performance relationship becomes more complex. As a result, we use the validation data to select a value for alpha. Here, we plot the coefficients and R^2 of the test data on the vertical axes and alpha values on the horizontal axis.

In [47]:

```python
alphas = [0.00001,0.0001,0.001,0.01,0.1,1,10,100,1000]
R_2=[]
coefs = []
for alpha in alphas:
    la=Lasso(alpha=alpha)

    la.fit(X_train, y_train)
    coefs.append(abs(la.coef_))
    R_2.append(la.score(X_test,y_test))


ax = plt.gca()
ax.plot(alphas, coefs)
ax.set_xscale("log")
plt.xlabel("alpha")
plt.ylabel("weights")
plt.title("Ridge coefficients as a function of the regularization (re
plt.show()


ax = plt.gca()
ax.plot(alphas, R_2)
ax.set_xscale("log")
plt.xlabel("alpha")
plt.ylabel("$R^2$")
plt.title("$R^2$ as a function of the regularization")
plt.show()
```







Pipeline

We can also create a Pipeline object and apply a set of transforms sequentially. Then, we can apply polynomial features, perform data standardization, then apply Lasso Regression. We also use `StandardScaler` as a step in our pipeline. Scaling your data is necessary step in LASSO Regression, as it will penalize features with a large magnitudes.

We start by creating a pipeline object.

In [48]:
```
Input=[ ('polynomial', PolynomialFeatures(include_bias=False,degree=2
pipe = Pipeline(Input)
```

Then we fit the object, and make our predictions.

In [49]:
```
pipe.fit(X_train, y_train)
pipe.predict(X_test)
```

Out[49]:
```
array([31090.35502413, 23135.62504126,  9457.4650858 , 11222.6446508
9,
       28503.13820242,  6821.35081552,  7716.68381925,  7436.3373659
3,
        7805.79226801, 10151.89082851, 16262.8074276 ,  8278.7525784
9,
       16684.95563795, 10658.46233607, 38313.26144272,  7039.4378737
,
        8259.35558463, 12989.37400724,  9251.96079447,  8253.9293395
9,
        9674.77573004])
```

We can calculate the R^2 on the training and testing data sets.

In [50]:
```
print("R^2 on training  data ",pipe.score(X_train, y_train))
print("R^2 on testing data ",pipe.score(X_test,y_test))
```

```
R^2 on training  data  0.941181185417151
R^2 on testing data  0.953382982655632
```

As we see, some individual features perform similarly to using all the features (we removed the feature `three`). Additionally, we see the smaller coefficients seem to correspond to a larger R^2, therefore larger coefficients correspond to overfiting.

GridSearchCV

To search for the best combination of hyperparameters, we can create a `GridSearchCV()` function as a dictionary of parameter values. The parameters of pipelines can be set by using the name of the key, separated by "__", then the parameter. Here, we look for different polynomial degrees and different values of alpha.

In [51]:
```
param_grid = {
    "polynomial__degree": [ 1, 2,3,4,5],
    "model__alpha":[0.0001,0.001,0.01,0.1,1,10]
}
```

To search for the best combination of hyperparameters, we create a `GridSearchCV` object with a dictionary of parameter values.

In [52]:
```
search = GridSearchCV(pipe, param_grid, n_jobs=2)
search.fit(X_train, y_train)
```

Out[52]:
```
GridSearchCV(estimator=Pipeline(steps=[('polynomial',
                                        PolynomialFeatures(include_bi
as=False)),
                                       ('ss', StandardScaler()),
                                       ('model', Lasso(alpha=1, tol=
0.2))]),
             n_jobs=2,
             param_grid={'model__alpha': [0.0001, 0.001, 0.01, 0.1,
1, 10],
                         'polynomial__degree': [1, 2, 3, 4, 5]})
```

Now, we can find the best model.

In [53]:
```
best=search.best_estimator_
best
```

Out[53]:
```
Pipeline(steps=[('polynomial',
                 PolynomialFeatures(degree=3, include_bias=False)),
                ('ss', StandardScaler()), ('model', Lasso(alpha=10, t
ol=0.2))])
```

We can calculate the R^2 on the test data.

In [54]:
```
best.score(X_test,y_test)
```

Out[54]:
```
0.9390483610147886
```

Elastic Net

In this section, let's review the Elastic Net Regression. It combines L1 and L2 priors as regularizes or penalties. So, we can combine the two as follows:

$$\hat{\mathbf{w}} = \underset{\mathbf{w}}{\operatorname{argmin}} ||\mathbf{y} - \mathbf{X}\mathbf{w}||^2\_2 + \alpha\rho||\mathbf{w}||\_1 0.5\alpha(1-\rho)||\mathbf{w}||^2\_2$$

Additionally to the alpha term (α), we have a mixing parameter, ρ, such that $0 \leq \rho \leq 1$. For ρ=0, the penalty is an L2 regularization . For $\rho = 0$, it is L1 regularization; otherwise, it is a combination of L1 and L2. In *scikit-learn* the parameter is called `l1_ratio` . Unlike the Ridge Regression, Elastic Net finds zero coefficients. In many cases Elastic Net performs better than Lasso, as it includes features that are correlated with one another. One drawback of the

Elastic Net is you have two hyperparameters. Lets create a model where `alpha=0.1` and `l1_ratio=0.5` and fit the data with this model.

In [55]:
```python
enet = ElasticNet(alpha=0.1, l1_ratio=0.5)
enet.fit(X_train,y_train)
```

Out[55]:
```
ElasticNet(alpha=0.1)
```

Let's make a prediction.

In [56]:
```python
predicted=enet.predict(X_test)
predicted
```

Out[56]:
```
array([27876.23069332, 19692.47948228, 11278.55904242, 11378.1176281
7,
       25990.32173225,  5511.46549415,  7591.3310787 ,  7796.5095087
4,
        9762.62529929,  8646.72495203, 17018.24269379,  7412.9503681
2,
       17031.98094445, 10732.9267982 , 37491.12185882,  5254.6679723
5,
        -286.43084016, 16075.95423556, 11330.87482645, 10579.4638010
1,
       11539.66267619])
```

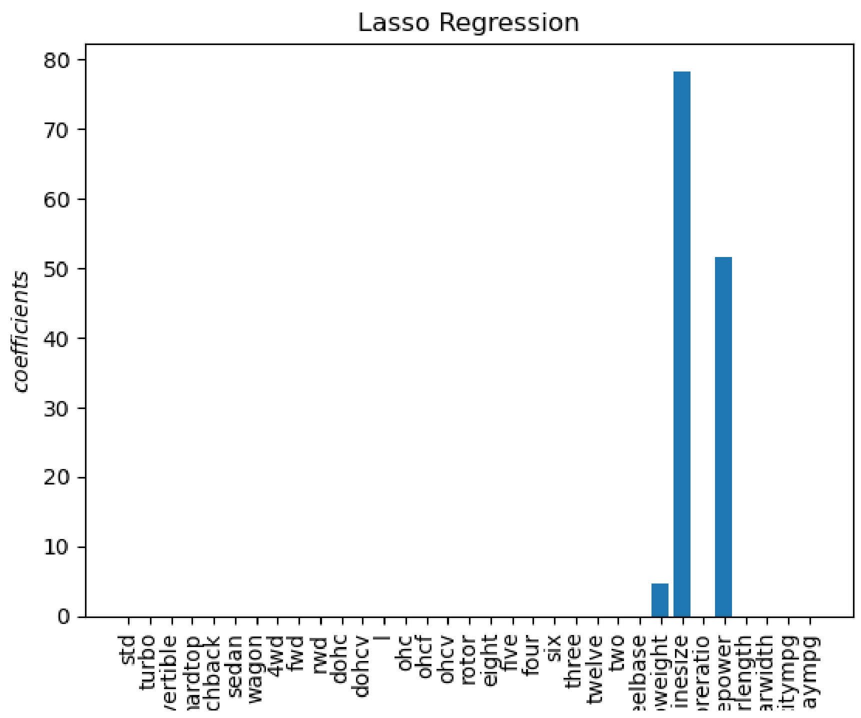
Let's calculate the R^2 on the test data.

In [57]:
```python
print("R^2 on training  data ", enet.score(X_train, y_train))
print("R^2 on testing data ", enet.score(X_test,y_test))
```

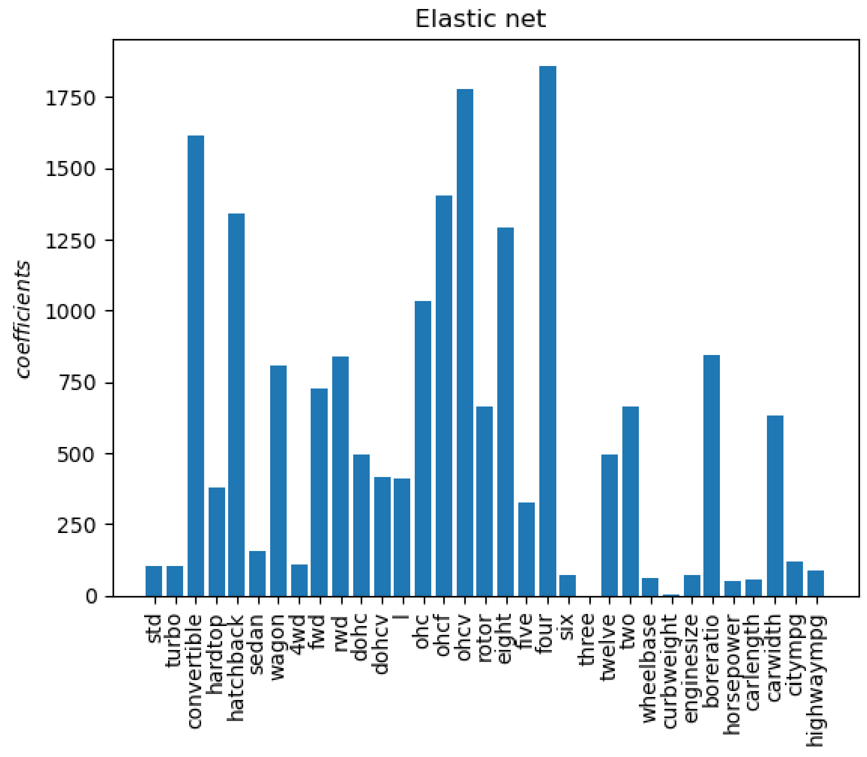
```
R^2 on training  data  0.8726983414262406
R^2 on testing data  0.9289334382162672
```

If we compare the Elastic Net to Lasso Regression and Ridge Regression, we see the results on the R^2 are better than the Elastic Net and many of the coefficients are zero.

In [58]:
```python
plot_coef(X,la,name="Lasso Regression")
plot_coef(X,enet,name="Elastic net ")

## graph that leads to error
```



```
R^2 on training  data  0.799083710857619
R^2 on testing data  0.910818845927178
```

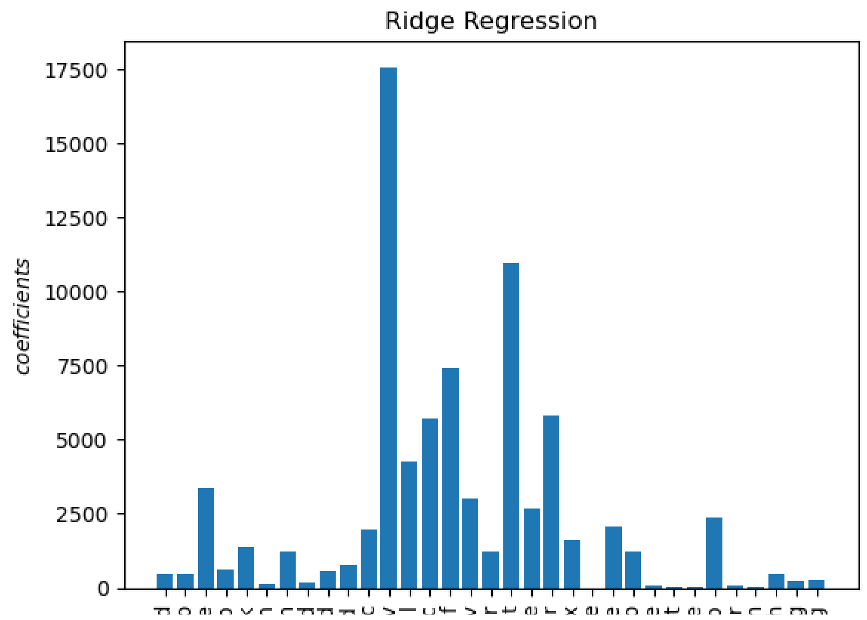


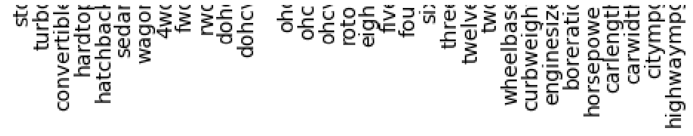
```
R^2 on training  data  0.8726983414262406
R^2 on testing data  0.9289334382162672
```

Exercise 3

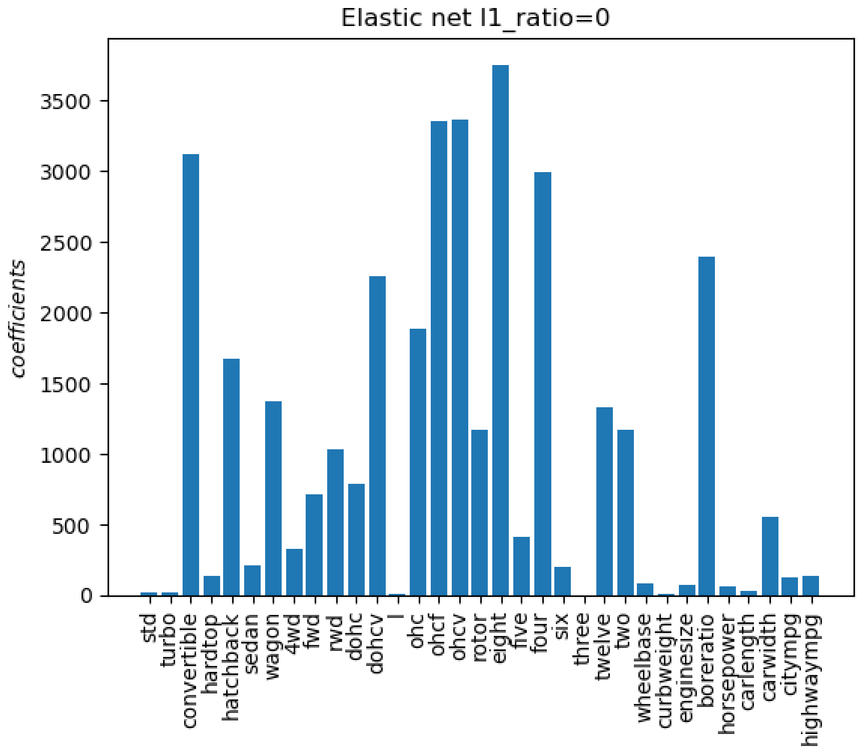
Create and fit the Elastic Net model and the Ridge Regression models and plot the coefficients for both models using the `plot_coef()` function.

In [60]:
```python
# Enter your code and run the cell
enet = ElasticNet(alpha=0.01, l1_ratio=0)
enet.fit(X_train,y_train)
rr = Ridge(alpha=0.01)
rr.fit(X_train,y_train)
plot_coef(X,rr,name="Ridge Regression")

plot_coef(X,enet,name="Elastic net l1_ratio=0 ")
```





```
R^2 on training  data  0.9091956531801182
R^2 on testing data  0.9478784615596495
```



```
R^2 on training  data  0.8941741868792898
R^2 on testing data  0.9407240278163358
```

Exercise 4

Create a Pipeline object, apply polynomial features (degree = 2), perform data standardization, then apply Elastic Net with `alpha=0.1` and `l1_ratio=0.1` parameters. Fit the model using the training data, then calculate the R^2 on the training and testing data.

In [61]:

```
# Enter your code and run the cell
Input=[ ('polynomial', PolynomialFeatures(include_bias=False,degree=2
pipe = Pipeline(Input)
pipe.fit(X_train, y_train)
print("R^2 on training  data ",pipe.score(X_train, y_train))
print("R^2 on testing data ",pipe.score(X_test,y_test))
```

```
R^2 on training  data  0.9703466188354504
R^2 on testing data  0.947428988234126
```

Exercise 5

Search for the best combination of hyperparameters by creating a `GridSearchCV` object for Elastic Net Regression. Find the best parameter values using the pipeline object, as used in the above examples.
Use `param_grid` , then find thee R^2 on the test data using the best estimator.

In [62]:

```
param_grid = {
    "polynomial__degree": [ 1, 2,3,4,5],
    "model__alpha":[0.0001,0.001,0.01,0.1,1,10],
    "model__l1_ratio":[0.1,0.25,0.5,0.75,0.9]
}
```

In [63]:

```
# Enter your code and run the cell
Input=[ ('polynomial', PolynomialFeatures(include_bias=False,degree=
pipe = Pipeline(Input)
search = GridSearchCV(pipe, param_grid, n_jobs=2)
search.fit(X_test, y_test)
best=search.best_estimator_
best.score(X_test,y_test)
```

Out[63]:

```
0.9800717238596846
```

Principal Component Analysis (Optional)

In this example, we will explore Principal Component Analysis to reduce the dimensionality of our data. We will do so by creating a Pipeline object first, then applying standard scaling and performing PCA, and then applying Elastic Net Regularization with the following parametrs: `tol=0.2` , `alpha=0.1` and `l1_ratio=0.1` . Finally, we will fit the model using the training data, then calculate the R^2 on the training and testing data sets.

Before adding PCA as a prep-processing step, we have to standardize our data. Scaling the features makes them have the same standard deviation.

In [64]:

```
scaler = StandardScaler()
X_train[:] = scaler.fit_transform(X_train)
X_train.columns = [f'{c} (scaled)' for c in X_train.columns]
```

Now, let's perform PCA.

In [65]:

```
pca = PCA()
pca.fit(X_train)
```

Out[65]:

```
PCA()
```

We can find the projection of the dataset onto the principal components, let's call it X_train_hat , this is our "new" dataset, it is the same shape as the original dataset.

In [66]:

```
X_train_hat = pca.transform(X_train)
print(X_train_hat.shape)
```

```
(184, 35)
```

Let's look at the new dataset as a dataframe.

In [67]:

```
X_train_hat_PCA = pd.DataFrame(columns=[f'Projection  on Component {i
X_train_hat_PCA.head()
```

Out[67]:

	Projection on Component 1	Projection on Component 2	Projection on Component 3	Projection on Component 4	Projection on Component 5	Projection on Component 6
0	3.294148	-1.989679	-2.182446	-1.156530	-0.715546	-0.605020
1	-2.801332	-0.496152	-0.654240	0.175846	0.267657	0.914558
2	5.993408	0.535847	0.270387	0.833262	0.898968	1.739295

2	-3.995408	-0.555847	-0.270587	-0.855262	0.898968	1.759295
3	-3.462434	0.060153	-0.841084	-0.505797	-0.771184	0.138637
4	4.664168	-0.948145	-1.440491	-0.739345	-1.235190	-0.889695

5 rows × 35 columns

Now, let's see how much variance can be explained using these principal components (PCs).

In [68]:
```
plt.plot(pca.explained_variance_ratio_)
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.title("Component-wise variance and cumulative explained variance"
```

Out[68]:
```
Text(0.5, 1.0, 'Component-wise variance and cumulative explained vari
ance')
```