NonLinear_regression_house_pricing

December 21, 2023

Before you turn this problem in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel→Restart) and then **run all cells** (in the menubar, select Cell→Run All).

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE" and remove every line containing the expression: "raise …" (if you leave such a line your code will not run).

Do not remove any cell from the notebook you downloaded. You can add any number of cells (and remove them if not more necessary).

0.1 IMPORTANT: make sure to rerun all the code from the beginning to obtain the results for the final version of your notebook, since this is the way we will do it before evaluating your notebook!!!

Fill in your name and id number (numero matricola) below:

[1]:
```
NAME = "Tommaso Bergamasco"
ID_number = int("2052409")

import IPython
assert IPython.version_info[0] >= 3, "Your version of IPython is too old,␣
 ↪please update it."
```

---

1 Regression on House Pricing Dataset: Variable Selection & Regularization

We will consider a reduced version of a dataset containing house sale prices for King County, which includes Seattle. It includes homes sold between May 2014 and May 2015.

Dataset used: https://www.kaggle.com/harlfoxem/housesalesprediction

{ Kaggle competition on house prices: https://www.kaggle.com/c/house-prices-advanced-regression-techniques }

For each house we know 18 house features (e.g., number of bedrooms, number of bathrooms, etc.) plus its price, that is what we would like to predict.

[2]:
```
#import all packages needed
from __future__ import division
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from scipy import stats

import warnings
warnings.filterwarnings('ignore')

np.random.seed(ID_number)
```

[3]:
```
url = "https://raw.githubusercontent.com/LucaZancato/ML2020-2021/main/
 ↪kc_house_data.csv"

data = pd.read_csv(url, sep=',')

# Remove the data samples with missing values (NaN)
data = data.dropna()
# Remove the columns we are not going to use
data = data.drop(columns=['id', 'date'])
# Have a brief description of the dataset
data.describe()
```

[3]:

	price	bedrooms	bathrooms	sqft_living	sqft_lot
count	3.164000e+03	3164.000000	3164.000000	3164.000000	3.164000e+03
mean	5.354358e+05	3.381163	2.071903	2070.027813	1.525054e+04
std	3.809004e+05	0.895472	0.768212	920.251879	4.254457e+04
min	7.500000e+04	0.000000	0.000000	380.000000	6.490000e+02
25%	3.150000e+05	3.000000	1.500000	1430.000000	5.453750e+03
50%	4.450000e+05	3.000000	2.000000	1910.000000	8.000000e+03
75%	6.402500e+05	4.000000	2.500000	2500.000000	1.122250e+04
max	5.350000e+06	8.000000	6.000000	8010.000000	1.651359e+06

	floors	waterfront	view	condition	grade
count	3164.000000	3164.000000	3164.000000	3164.000000	3164.000000
mean	1.434893	0.009798	0.244311	3.459229	7.615676
std	0.507792	0.098513	0.776298	0.682592	1.166324
min	1.000000	0.000000	0.000000	1.000000	3.000000
25%	1.000000	0.000000	0.000000	3.000000	7.000000
50%	1.000000	0.000000	0.000000	3.000000	7.000000
75%	2.000000	0.000000	0.000000	4.000000	8.000000
max	3.500000	1.000000	4.000000	5.000000	12.000000

	sqft_above	sqft_basement	yr_built	yr_renovated	zipcode
count	3164.000000	3164.000000	3164.000000	3164.000000	3164.000000

```
mean   1761.252212     308.775601  1967.489254     94.668774  98077.125158
std     815.934864     458.977904    28.095275    424.439427     54.172937
min     380.000000       0.000000  1900.000000      0.000000  98001.000000
25%    1190.000000       0.000000  1950.000000      0.000000  98032.000000
50%    1545.000000       0.000000  1969.000000      0.000000  98059.000000
75%    2150.000000     600.000000  1990.000000      0.000000  98117.000000
max    6720.000000    2620.000000  2015.000000   2015.000000  98199.000000

                lat         long  sqft_living15     sqft_lot15
count  3164.000000  3164.000000    3164.000000    3164.000000
mean     47.557868  -122.212337    1982.544564   13176.302465
std       0.140789     0.139577     686.256670   25413.180755
min      47.177500  -122.514000     620.000000     660.000000
25%      47.459575  -122.324250    1480.000000    5429.500000
50%      47.572500  -122.226000    1830.000000    7873.000000
75%      47.680250  -122.124000    2360.000000   10408.250000
max      47.777600  -121.315000    5790.000000  425581.000000
```

[4]:
```
# Print first 5 datapoints of the dataset
data.head()
```

[4]:
```
      price  bedrooms  bathrooms  sqft_living  sqft_lot  floors  waterfront  \
0  221900.0         3       1.00         1180      5650     1.0           0
1  538000.0         3       2.25         2570      7242     2.0           0
2  180000.0         2       1.00          770     10000     1.0           0
3  604000.0         4       3.00         1960      5000     1.0           0
4  510000.0         3       2.00         1680      8080     1.0           0

   view  condition  grade  sqft_above  sqft_basement  yr_built  yr_renovated  \
0     0          3      7        1180              0      1955             0
1     0          3      7        2170            400      1951          1991
2     0          3      6         770              0      1933             0
3     0          5      7        1050            910      1965             0
4     0          3      8        1680              0      1987             0

   zipcode      lat     long  sqft_living15  sqft_lot15
0    98178  47.5112 -122.257         1340.0      5650.0
1    98125  47.7210 -122.319         1690.0      7639.0
2    98028  47.7379 -122.233         2720.0      8062.0
3    98136  47.5208 -122.393         1360.0      5000.0
4    98074  47.6168 -122.045         1800.0      7503.0
```

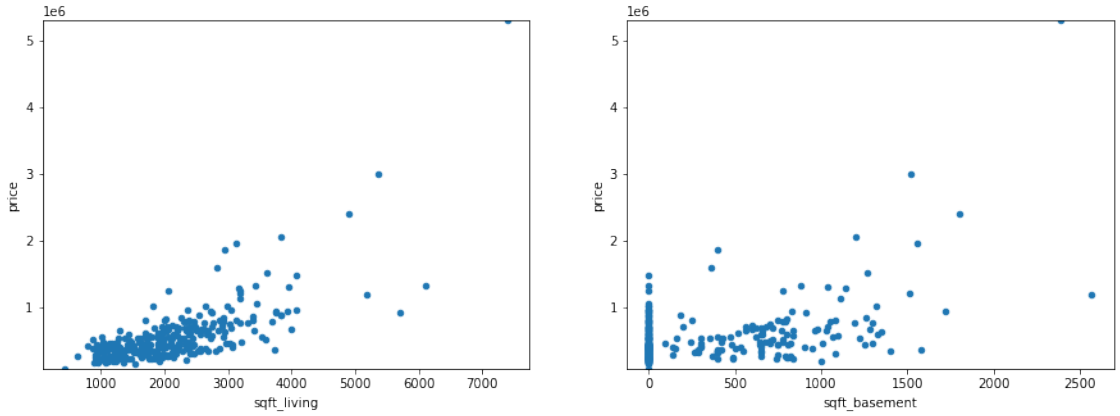
[5]:
```
# Let's look at all the possible independent variables and get an idea of our␣
 ↪data. Do not forget we are going
# to predict the variable 'price' using all the other features
data.columns
```

[5]:
```
Index(['price', 'bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot', 'floors',
       'waterfront', 'view', 'condition', 'grade', 'sqft_above',
       'sqft_basement', 'yr_built', 'yr_renovated', 'zipcode', 'lat', 'long',
       'sqft_living15', 'sqft_lot15'],
      dtype='object')
```

[6]:
```
# Let's split data into train and test using sklearn built-in function:␣
 ↪train_test_split (have a look at the
# documentation)
m_t, m = 300, len(data)
m_test = m - m_t

from sklearn.model_selection import train_test_split
train_data, test_data = train_test_split(data, test_size=m_test/m,␣
 ↪random_state=ID_number)
```

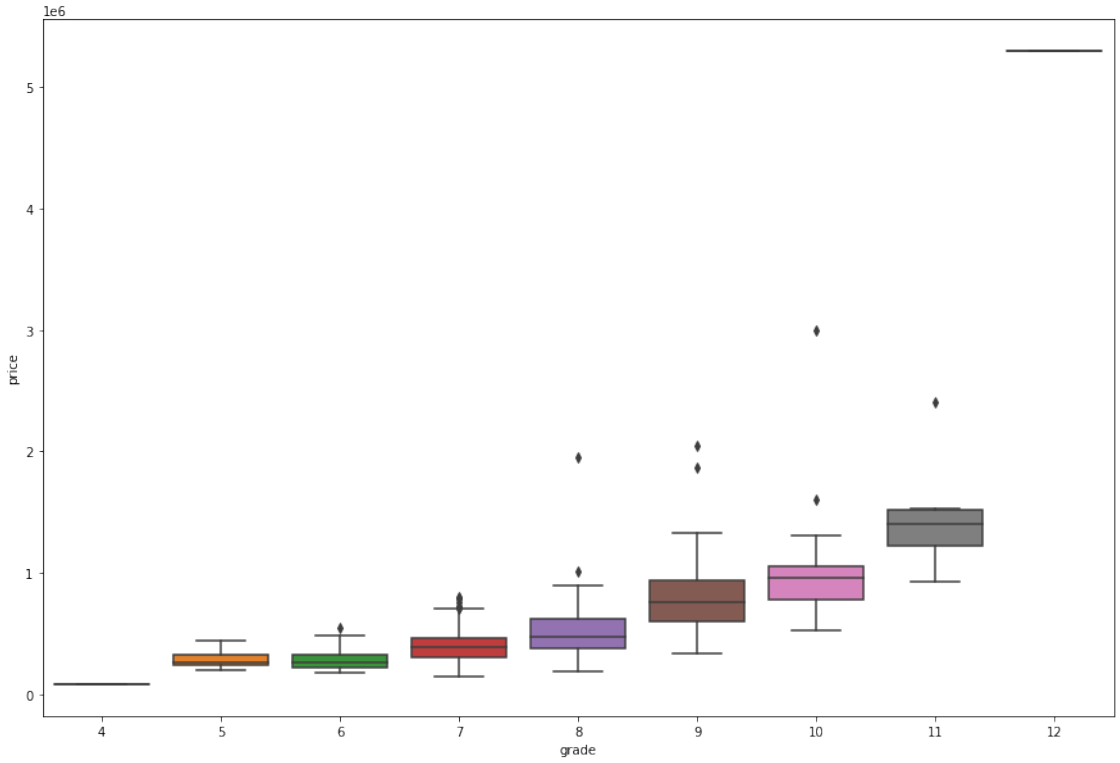
[7]:
```
################## Experimental Cell to use log transformation on reg variable␣
 ↪################
################## Remove if necessary␣
 ↪#########################################################
# train_data['price'] = np.log(train_data['price'])
# test_data['price'] = np.log(test_data['price'])
```

[8]:
```
# Let's check the price trend as a function of the sqrt_living and␣
 ↪sqrt_basement (separately)

def plot_single_feature_vs_y(feature, train_data, ax=None, y='price'):
    reduced_data = pd.concat([train_data[y], train_data[feature]], axis=1)
    reduced_data.plot.scatter(x=feature, y=y, ylim=(train_data[y].min(),␣
 ↪train_data[y].max()), ax=ax)

fig, axes = plt.subplots(1,2, figsize=(15,5))
plot_single_feature_vs_y('sqft_living', train_data, ax=axes[0])
plot_single_feature_vs_y('sqft_basement', train_data, ax=axes[1])

# Note 'sqft_basement' might not be easily used to predict Y (many values are␣
 ↪zero while 'price' has different values)
```



[9]:
```
# Let's explore data and features distributions using box plots
def box_plot_single_feature_vs_y(feature, train_data):
    plt.figure(figsize=(15,10))
    reduced_data = pd.concat([train_data['price'], train_data[feature]], axis=1)
    sns.boxplot(x=feature, y="price", data=reduced_data)
```

[10]:
```
# Prices over 'grade'

box_plot_single_feature_vs_y('grade', train_data)
```

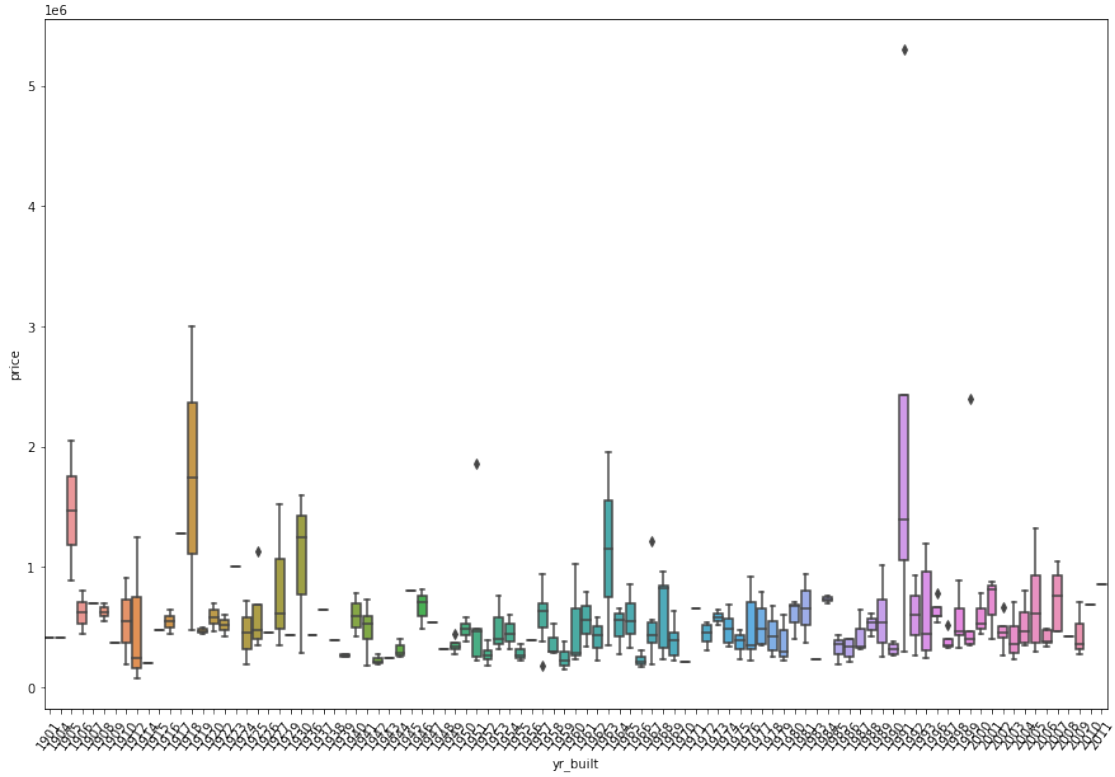


[11]:
```
# Prices over 'yr_built'

box_plot_single_feature_vs_y('yr_built', train_data)
plt.xticks(rotation=55)
```

[11]:
```
(array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
        17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
        34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
        51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
        68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
        85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96]),
 [Text(0, 0, '1901'),
  Text(1, 0, '1904'),
  Text(2, 0, '1905'),
  Text(3, 0, '1906'),
  Text(4, 0, '1907'),
  Text(5, 0, '1908'),
  Text(6, 0, '1909'),
  Text(7, 0, '1910'),
  Text(8, 0, '1912'),
  Text(9, 0, '1914'),
  Text(10, 0, '1915'),
  Text(11, 0, '1916'),
  Text(12, 0, '1917'),
  Text(13, 0, '1918'),
  Text(14, 0, '1919'),
  Text(15, 0, '1920'),
  Text(16, 0, '1922'),
  Text(17, 0, '1923'),
  Text(18, 0, '1924'),
  Text(19, 0, '1925'),
  Text(20, 0, '1926'),
  Text(21, 0, '1927'),
  Text(22, 0, '1929'),
  Text(23, 0, '1930'),
  Text(24, 0, '1936'),
  Text(25, 0, '1937'),
  Text(26, 0, '1938'),
  Text(27, 0, '1939'),
  Text(28, 0, '1940'),
  Text(29, 0, '1941'),
  Text(30, 0, '1942'),
  Text(31, 0, '1943'),
  Text(32, 0, '1944'),
  Text(33, 0, '1945'),
  Text(34, 0, '1946'),
```

```
Text(35, 0, '1947'),
Text(36, 0, '1948'),
Text(37, 0, '1949'),
Text(38, 0, '1950'),
Text(39, 0, '1951'),
Text(40, 0, '1952'),
Text(41, 0, '1953'),
Text(42, 0, '1954'),
Text(43, 0, '1955'),
Text(44, 0, '1956'),
Text(45, 0, '1957'),
Text(46, 0, '1958'),
Text(47, 0, '1959'),
Text(48, 0, '1960'),
Text(49, 0, '1961'),
Text(50, 0, '1962'),
Text(51, 0, '1963'),
Text(52, 0, '1964'),
Text(53, 0, '1965'),
Text(54, 0, '1966'),
Text(55, 0, '1967'),
Text(56, 0, '1968'),
Text(57, 0, '1969'),
Text(58, 0, '1970'),
Text(59, 0, '1971'),
Text(60, 0, '1972'),
Text(61, 0, '1973'),
Text(62, 0, '1974'),
Text(63, 0, '1975'),
Text(64, 0, '1976'),
Text(65, 0, '1977'),
Text(66, 0, '1978'),
Text(67, 0, '1979'),
Text(68, 0, '1980'),
Text(69, 0, '1981'),
Text(70, 0, '1983'),
Text(71, 0, '1984'),
Text(72, 0, '1985'),
Text(73, 0, '1986'),
Text(74, 0, '1987'),
Text(75, 0, '1988'),
Text(76, 0, '1989'),
Text(77, 0, '1990'),
Text(78, 0, '1991'),
Text(79, 0, '1992'),
Text(80, 0, '1993'),
Text(81, 0, '1996'),
```

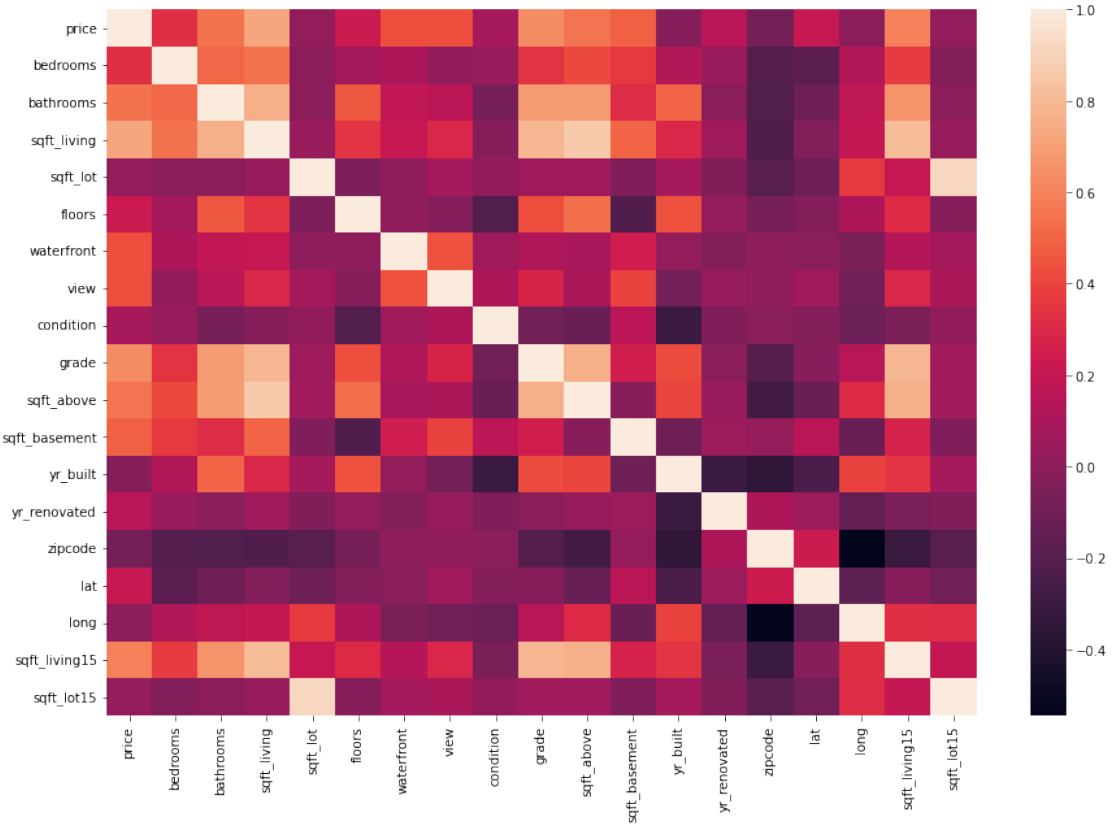
```
Text(82, 0, '1997'),
Text(83, 0, '1998'),
Text(84, 0, '1999'),
Text(85, 0, '2000'),
Text(86, 0, '2001'),
Text(87, 0, '2002'),
Text(88, 0, '2003'),
Text(89, 0, '2004'),
Text(90, 0, '2005'),
Text(91, 0, '2006'),
Text(92, 0, '2007'),
Text(93, 0, '2008'),
Text(94, 0, '2009'),
Text(95, 0, '2010'),
Text(96, 0, '2011')])
```



Note: 'grade' seems to correlate well with the regression variable 'price' (the higher the 'grade' the higher the 'price'). On the other hand it is not clear whether there is correlation between 'yr_built' and 'price'. You can try to inspect other features and how they correlate with 'price'.

[12]:
```
# Let's' try to make the process we followed up to now a little bit more␣
↪systematic: we will use a pandas
```

```
# built-in function to plot the correlation matrix between all the features and␣
 ↪the regression variables.

def plot_correlation_matrix(df_train):
    plt.subplots(figsize=(15, 10))
    corr_matrix = df_train.corr()
    ax = sns.heatmap(corr_matrix, vmax=1, square=False)
    ax.set_ylim(19, 0)
    return corr_matrix

corr_matrix = plot_correlation_matrix(train_data)
```



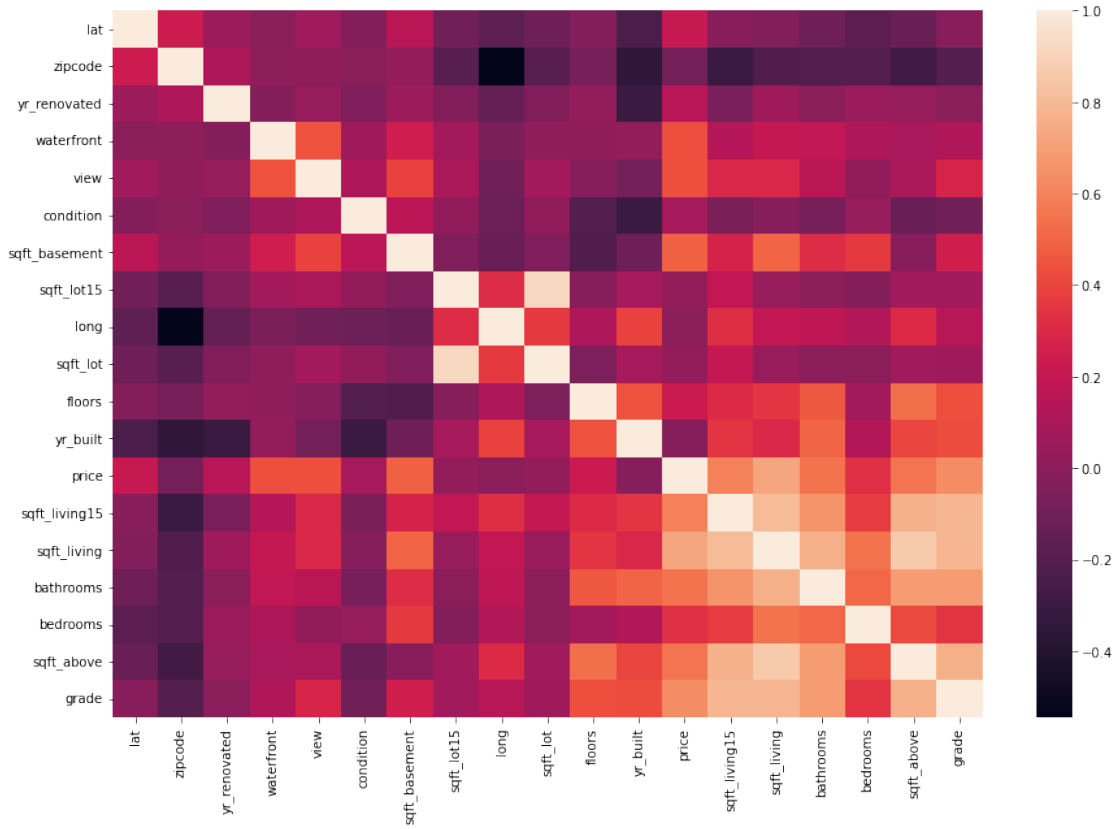
Warning: if you see white lines in the plot above, try re-reunning the notebook with a different ID_number (adding 1 to your own should fix the issue). Otherwise an error will be generated in the next cell.

This is for data-visualization only. The rest of the code should run without any problem with any ID_number.

[13]:
```
# Previous correlation matrix is not ordered, we need to sort its entries such␣
 ↪that we can cluster the most
```

```
# correlated variables. In this way it will be easier to read the correlation␣
 ↪matrix.

import scipy
import scipy.cluster.hierarchy as sch

def cluster_corr(corr_array, inplace=False) -> pd.DataFrame:
    """
    Rearranges the correlation matrix, corr_array, so that groups of highly
    correlated variables are next to eachother

    :param corr_array: pandas.DataFrame or numpy.ndarray a NxN correlation␣
 ↪matrix

    :returns: A NxN correlation matrix with the columns and rows rearranged
    """
    pairwise_distances = sch.distance.pdist(corr_array)
    linkage = sch.linkage(pairwise_distances, method='complete')
    cluster_distance_threshold = pairwise_distances.max()/2
    idx_to_cluster_array = sch.fcluster(linkage, cluster_distance_threshold,
                                        criterion='distance')
    idx = np.argsort(idx_to_cluster_array)

    if not inplace:
        corr_array = corr_array.copy()

    if isinstance(corr_array, pd.DataFrame):
        return corr_array.iloc[idx, :].T.iloc[idx, :]
    return corr_array[idx, :][:, idx]

plt.subplots(figsize=(15, 10))
corr_matrix = cluster_corr(corr_matrix, inplace=False)
ax = sns.heatmap(corr_matrix, vmax=1, square=False)
ax.set_ylim(19, 0)
```

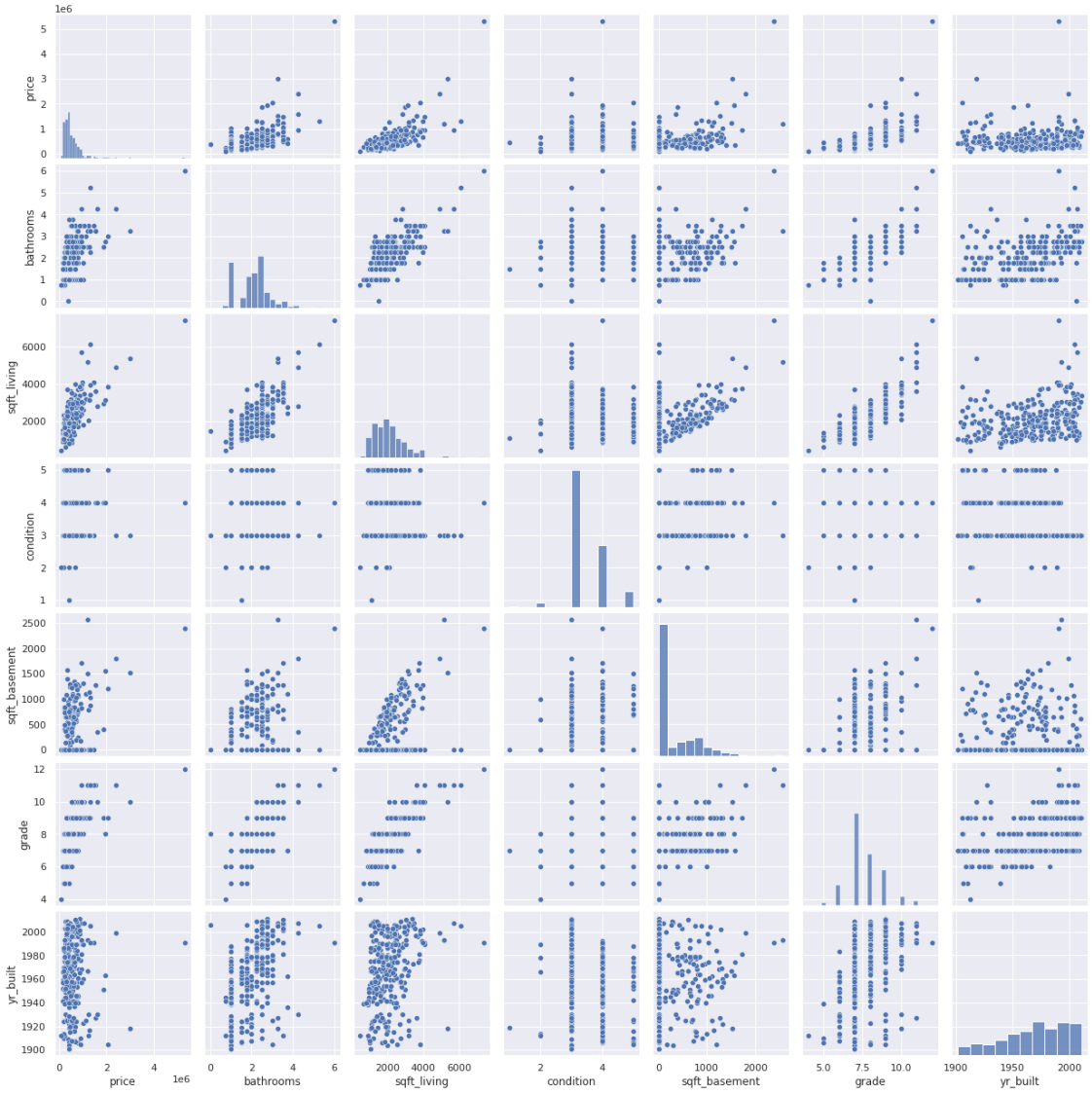
[13]: (19.0, 0.0)



Note the 'clusters' along the diagonal (there is high correlation among the following variables): - 'sqft_above', 'sqft_living15', 'grade', 'bathrooms', 'bedrooms', 'sqft_living' - 'view' and 'price' - 'sqft_lot' and 'sqft_lot15'

Actually, these correlations are so strong that it might indicate a situation of multicollinearity. This means these varaibles are in some sense redundant (they give almost the same information) and might not be useful to build our final linear model.

[14]:
```
# Let's have a look at some scatter plots (in the main diagonal there is a␣
 ↪histogram with the actual data)
sns.set()
cols = ['price', 'bathrooms', 'sqft_living', 'condition', 'sqft_basement',␣
 ↪'grade', 'yr_built']
sns.pairplot(train_data[cols], size = 2.5)
plt.show()
# As we see in the correlation matrix 'condition' and 'yr_build' are very␣
 ↪poorly correlated
```



Note: We can remove features that we believe are much correlated with others (as we described earlier, they can be thought as redundant). For example we can remove some variables (but not all of them!) in the following list ['sqft_above', 'sqft_living15', 'grade', 'bathrooms', 'bedrooms', 'sqft_living'], and keep only some of them. As a furhter example the very same logic applies to ['sqft_lot', 'sqft_lot15'] features (we can get rid of one of the two). Don't forget we should keep into account how much each feature is correlated with the regression variable too.

[15]:
```
# Let's now standardize the data as we did in the past homeworks
features_names = train_data.columns[1:]
x_train, y_train = train_data[features_names].values.astype('float'),␣
 ↪train_data['price'].values.astype('float')
x_test, y_test = test_data[features_names].values.astype('float'),␣
 ↪test_data['price'].values.astype('float')
```

```
from sklearn import preprocessing
scaler_x = preprocessing.StandardScaler().fit(x_train)
x_train = scaler_x.transform(x_train)

scaler_y = preprocessing.StandardScaler().fit(y_train.reshape(-1,1))
y_train = scaler_y.transform(y_train.reshape(-1,1)).reshape(-1,)

x_test = scaler_x.transform(x_test)
y_test = scaler_y.transform(y_test.reshape(-1,1)).reshape(-1,)
```

[16]:
```
# TODO 1
# Write a function to compute the Least-Squares estimate using␣
 ↪LinearRegression() from Scikit-learn given x_train and
# y_train. The function must return the COD both for training and test dataset␣
 ↪AND must return a vector containing
# all the model parameters (both bias b and coefficients w)
from sklearn import linear_model
def solve_LS_problem(x_train : np.ndarray, y_train : np.array, x_test : np.
 ↪ndarray, y_test : np.array) -> tuple:
    '''
    Funtion used to compute the LS estimate given train data. This function␣
 ↪uses Scikit-learn to get both the LS
    soluion and other required quantities.
    :param x_train: input data used to get the linear model predictions
    :param y_train: output data to be predicted
    :param x_test: test features used to assess model performance
    :param y_test: test output to be predicted to assess model performance

    :returns: (COD_train, COD_test, w)
        WHERE
        COD_train : Coefficient of determination for the training dataset␣
 ↪(float)
        COD_test : Coefficient of determination for the test dataset (float)
        w : parameters of the linear model (the bias is contained, return it as␣
 ↪the first element of w)
            of shape (#parameters + 1,)
    '''

    # YOUR CODE HERE

    # Add the bias in front of input sets. (This changes also the dimension of␣
 ↪the coef_ w)
    x_train = np.hstack((np.ones((x_train.shape[0],1)),x_train))
    x_test = np.hstack((np.ones((x_test.shape[0],1)),x_test))
```

```
    # Define the Linear Regression Model
    lin_reg_model = linear_model.LinearRegression().fit(x_train, y_train)

    # Extract the coef_ attribute of the Linear Regression Model
    w = lin_reg_model.coef_ # dimension (19,1)

    # Compute the CODs for both train and test
    COD_train = lin_reg_model.score(x_train, y_train)
    COD_test = lin_reg_model.score(x_test, y_test)

    return (COD_train, COD_test, w)
```

[17]:
```
COD_train_LS_full, COD_test_LS_full, w_LS_full = solve_LS_problem(x_train,␣
 ↪y_train, x_test, y_test)
print(f"Coefficient of determination on training data: {COD_train_LS_full:.4f}")
print(f"Coefficient of determination on test data:     {COD_test_LS_full:.4f}")

assert w_LS_full.shape == (19,)
assert type(COD_train_LS_full) == np.float64 and COD_train_LS_full <= 1.0
assert type(COD_test_LS_full) == np.float64 and COD_test_LS_full <= 1.0
```

```
Coefficient of determination on training data: 0.7573
Coefficient of determination on test data:     0.5731
```

[18]:
```
# TODO 2
# Based on the observations we made earlier looking at the dataset (correlation␣
 ↪and scatter plots) which variables
# would you choose to predict the price? Choose the 4 most important features␣
 ↪based on your intuition.
# Here we plot features and their indeces for your ease of use
print({index: feature for index, feature in enumerate(features_names)})
```

```
{0: 'bedrooms', 1: 'bathrooms', 2: 'sqft_living', 3: 'sqft_lot', 4: 'floors', 5:
'waterfront', 6: 'view', 7: 'condition', 8: 'grade', 9: 'sqft_above', 10:
'sqft_basement', 11: 'yr_built', 12: 'yr_renovated', 13: 'zipcode', 14: 'lat',
15: 'long', 16: 'sqft_living15', 17: 'sqft_lot15'}
```

[19]:
```
hand_selected_features_indeces = None # Replace with a list of 4 indeces and␣
 ↪then solve the reduced (in the number of
                                      # features) LS problem using the function␣
 ↪we built before
COD_train_LS_reduced, COD_test_LS_reduced, w_LS_reduced = None, None, None  #␣
 ↪Replace with proper values

# YOUR CODE HERE

# Good choices for the variables respect 3 characteristics:
```

```
    # 1. Strong Correlation w.r.t. price
    # 2. Weak Correlations between them
    # 3. Not a lot of 0

# Look at the corr. matrix: sqft_living and grade are strongly correlated to␣
 ↪the price and
# intuitively they do not correlate much to each other.
# Then I choose the average sqft_living of the nearest 15 houses (since it can␣
 ↪be an indicator
# of wealthy neighborhoods) and the view which is intuitively not very
# corraleted to the other paramenters.
hand_selected_features_indeces = [2,6,8,16]

# Now we extract only the "right" features in the input datasets
x_train_reduced = x_train[:,hand_selected_features_indeces]
x_test_reduced = x_test[:,hand_selected_features_indeces]


# And now we can call the function we wrote above
COD_train_LS_reduced, COD_test_LS_reduced, w_LS_reduced =␣
 ↪solve_LS_problem(x_train_reduced,y_train,x_test_reduced,y_test)

print(f"Coefficient of determination on training data: {COD_train_LS_reduced:.
 ↪4f}")
print(f"Coefficient of determination on test data:     {COD_test_LS_reduced:.
 ↪4f}")
```

```
Coefficient of determination on training data: 0.5854
Coefficient of determination on test data:     0.5540
```

[20]:
```
assert w_LS_reduced.shape == (5,)
```

1.1 Best-Subset Selection

What if we try a brute force approach? Are the features you selected the same as the ones you would get by looking at all the possible combinations?

In the next cell we are going to split x_train into a ("new") training dataset (x_train_BSS) and validation dataset (x_val_BSS) to perform best-subset selection (remember the validation dataset is used to find the best generalizing model among the ones, with different number of features, you trained using x_train_BSS).

We are going to choose subsets of features going from 1 to $n_{sub} = 4$. In theory we should try all the possible combinations of size $1, 2, \dots, 18$ but the number of models to train and validate would be huge! For the sake of simplicity we will choose all the possible subsets of dimension 1 to 4.

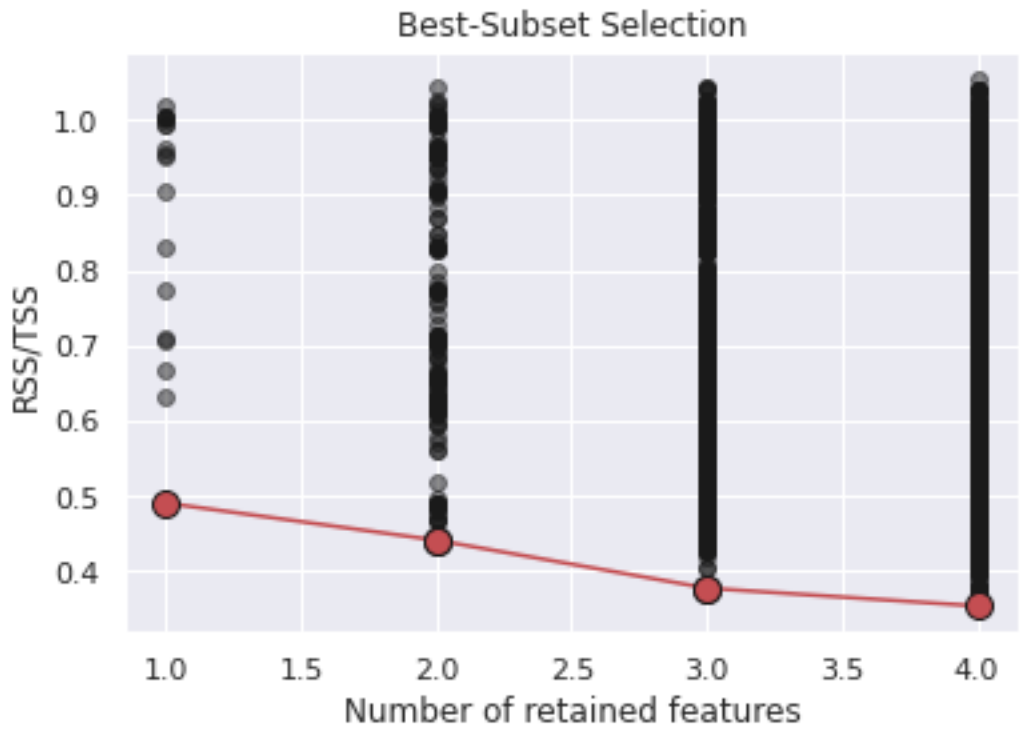
Steps: 1. Compute the LS estimate using all the possible subsets of k features 2. Compute the prediction error on the validation dataset 3. Choose the subset of k^* features giving the lowest validation error

[21]:
```
# TODO 3
# Let's get the validation dataset from the training dataset (don't forget they␣
 ↪must be disjoint, of course we could
# have splitted them before, during the pre-processing step)
import itertools

m_train_BSS, m_val_BSS = m_t // 2, m_t - m_t // 2

x_train_BSS, y_train_BSS = x_train[:m_train_BSS], y_train[:m_train_BSS]
x_val_BSS, y_val_BSS = x_train[m_train_BSS:], y_train[m_train_BSS:]


nsub = 4
features_idx_dict, validation_err_dict = {}, {}
for k in range(1,nsub + 1):
    features_idx = list(itertools.combinations(range(x_train_BSS.shape[1]), k))
    validation_error = np.zeros(len(features_idx),)
    for j in range(len(features_idx)):
        # You should use the function you built in previous TODO
        # YOUR CODE HERE

        # Once again we reduce our input sets in order to include only␣
 ↪features_idx
        x_train_BSS_reduced = x_train_BSS[:,features_idx[j]]
        x_val_BSS_reduced = x_val_BSS[:,features_idx[j]]

        # Now we call the function on such reduced data set
        COD_train_BSS, COD_val_BSS, w_BSS =␣
 ↪solve_LS_problem(x_train_BSS_reduced,y_train_BSS,x_val_BSS_reduced,y_val_BSS)

        validation_error[j] = 1 - COD_val_BSS
    print(f'Number of models trained for {k} chosen features:␣
 ↪{len(features_idx)}')

    features_idx_dict.update({k: features_idx})
    validation_err_dict.update({k: validation_error})

validation_err_min_per_size = {k+1: np.min(val_errs) for k, val_errs in␣
 ↪enumerate(validation_err_dict.values())}
validation_err_argmin_per_size = {k+1: np.argmin(val_errs) for k, val_errs in␣
 ↪enumerate(validation_err_dict.values())}
```

```
Number of models trained for 1 chosen features: 18
Number of models trained for 2 chosen features: 153
Number of models trained for 3 chosen features: 816
Number of models trained for 4 chosen features: 3060
```

[22]:
```
number_subsets_per_group = [int(scipy.special.comb(18, k)) for k in range(1,␣
 ↪nsub + 1)]
for k, n_SubSet in zip(range(1, nsub + 1), number_subsets_per_group):
    assert len(validation_err_dict[k]) == n_SubSet
```

[23]:
```
# Plot the validation score for each model
plt.figure(2)
for k in range(1,nsub+1):
    plt.scatter(k*np.ones(validation_err_dict[k].shape),␣
 ↪validation_err_dict[k], color='k', alpha=0.5)
    if k > 1:
        plt.plot([k-1, k], [validation_err_min_per_size[k-1],␣
 ↪validation_err_min_per_size[k]], color='r',marker='o',
           markeredgecolor='k', markerfacecolor = 'r', markersize = 10)
plt.xlabel('Number of retained features')
plt.ylabel('RSS/TSS')
plt.title('Best-Subset Selection')
plt.show()
```



[24]:
```
# TODO 4
# Pick the number of features for the best subset according to figure above,␣
 ↪select the best subset using the results
```

```
# above and learn the model on the entire training data (x_train); eventually␣
 ↪compute COD on training (x_train) and
# on test data (x_test).

# Now pick the number of features according to best subset
opt_num_features = 4
opt_features_idx =␣
 ↪features_idx_dict[opt_num_features][validation_err_argmin_per_size[opt_num_features]]

# You should use the function you built in previous TODOs
COD_train_BSS, COD_test_BSS, w_hat_BSS = None, None, None # Replace with the␣
 ↪proper quantities
# YOUR CODE HERE

# We have to train the algorithm upon all the columns of x_train, but we have␣
 ↪to consider
# of course only the columns of interest indexed by opt_features_idx
x_train_4_features = x_train[:,opt_features_idx]
x_test_4_features = x_test[:,opt_features_idx]

# Now we can call the LS solver on such data sets
COD_train_BSS, COD_test_BSS, w_hat_BSS =␣
 ↪solve_LS_problem(x_train_4_features,y_train,x_test_4_features,y_test)

# Let's print the indices of the features from best subset
print(f'Best features indexes: {opt_features_idx}')
print(f'Best features names: {str({features_names[i] for i in␣
 ↪opt_features_idx})}')

# Let's print model performance on train and test datasets (measured using COD)
print(f"Coefficient of determination on training data: {COD_train_BSS:.4f}")
print(f"Coefficient of determination on test data:     {COD_test_BSS:.4f}")
```

```
Best features indexes: (2, 5, 11, 14)
Best features names: {'waterfront', 'lat', 'sqft_living', 'yr_built'}
Coefficient of determination on training data: 0.6985
Coefficient of determination on test data:     0.6161
```

[25]:
```
assert len(opt_features_idx) == opt_num_features
```

1.1.1 Ridge regression

Recall that for linear models with scalar output we have $h(x) =< w, x >$ and that the Empirical error $L_S(h)$ can be written in terms of the vector of parameters w, in the form:

$$L_S(w) = \frac{1}{m_t}\|Y - Xw\|^2$$

where Y and X are the matrices whose $i-$th rows are, respectively, the output data y_i and the input vectors $x_i^\top$.

In the case of Ridge regression we add a regularization term to the RSS term so that our Empirical error becomes:

$$L_S(w) = \frac{1}{m_t}\|Y - Xw\|^2 + \lambda\|w\|^2 \propto \|Y - Xw\|^2 + \underbrace{\lambda * m_t}_{:=\alpha}\|w\|^2$$

The Ridge Least Squares solution is given by the expression:

$$\hat{w}_{Ridge} = \arg\min_w L_S(w) = (X^\top X + \alpha I)^{-1} X^\top Y$$

Note: what has changed w.r.t. the LS solution? Do we need to worry about invertibility of the matrix we need to invert? - Prove that adding a positive multiple of identity to a semi definite positive matrix you get a positive definite matrix. - Prove that a positive definite matrix is *always* invertible.

1.1.2 Note that:

w.r.t. the LS solution ($\hat{w}_{LS} = (X^T X)^\dagger X^T Y$) we now have a "perturbation" of the matrix $X^T X$. However we don't need to worry about the invertibility since $\alpha > 0$ for costruction which lead us to the case of a Semi Positive matrix summed to a positive multiple of an Identity matrix which is always invertible (see next 2 proofs).

1.1.3 Proof 1

Prove that if $P \geq 0 \Rightarrow P' = P + \lambda\mathbb{1} > 0, \quad \forall\lambda > 0.$

P is associated to a Jordan form like:

$$J_P = \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_j \end{bmatrix} \text{ where } \lambda_1, \dots, \lambda_j \geq 0$$

and hence it's trivial to see that P' will be associated to:

$$J_{P'} = \begin{bmatrix} \lambda'_1 & & \\ & \ddots & \\ & & \lambda'_j \end{bmatrix} \text{ where } \lambda'_1, \dots, \lambda'_j > 0$$

Since the new eigenvalues are obtained as sum of a positive number with a number ≥ 0. ■

1.1.4 Proof 2

A matrix P is invertible iff $rankP > 0$, but as we have already seen the matrix P can be associated to:

$$J_P = \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_j \end{bmatrix} \text{ where } \lambda_1, ..., \lambda_j > 0$$

And hence it holds $rankP = rankJ_P = \lambda_1 \cdot ... \cdot \lambda_j > 0$. ■

---

[26]:
```
# TODO 5
# Write a function which computes the optimal parameters w_hat, solution to the␣
 ↪LS regularized problem described
# earlier.
# We assume w_hat contains the bias term b (as described in class), so you will␣
 ↪need to create the proper input data
# adding a fictitious feature containing only ones (it is assumed you add the␣
 ↪ones vector before all your features
# so that b_hat is in the first position of w_hat).
# Then write a function "solve_ridge_LS_problem", similar in spirit to␣
 ↪"solve_LS_problem" (see function description
# for details on the input and return values).
def compute_LS_optimal_ridge_ERM_coefficients(x_train : np.ndarray, y_train :␣
 ↪np.ndarray, alpha : float) -> np.ndarray:
    '''
    This function estimates the optimal Ridge LS coefficients given the input␣
 ↪and output training data x, y.
    This function assumes the bias term b is condensed with the other␣
 ↪coefficients, therefore a column of ones is
    stacked (place it in front of the feature vector) to the input features and␣
 ↪the size of the returned optimal
    coefficient is: number of features + 1

    :param x_train: input features
    :param y_train: output to be predicted
    :param alpha: regularization parameter

    :returns: a column vector containing w_hat, solution to the ridge ERM LS␣
 ↪problem
    '''
    # YOUR CODE HERE

    # Add the bias to input training data set
    x_train = np.hstack((np.ones((x_train.shape[0],1)),x_train))

    # Define important variables:
```

```
    # Matrix X^TX (as in LS problem)
    A_matrix = x_train.T @ x_train

    # Matrix alphaI
    alpha_I = alpha * np.eye(A_matrix.shape[0])

    # vector b (as in LS problem)
    b = x_train.T @ y_train

    # Compute w_hat using the quantities defined above
    w_hat = np.linalg.inv(A_matrix + alpha_I) @ b

    # Reshape in a column vector?
    # Doing this ruins the consistency with the dimensions of the sklearn␣
 ↪implementations
    # w_hat = w_hat.reshape((w_hat.shape[0],1))

    return w_hat


def solve_ridge_LS_problem(x_train : np.ndarray, y_train : np.array, x_test :␣
 ↪np.ndarray, y_test : np.array,
                           alpha : float) -> tuple:
    '''
    Funtion used to compute the ridge LS estimate given train data and alpha␣
 ↪hyper-parameter.
    This function uses Scikit-learn to get both the ridge LS soluion and other␣
 ↪required quantities.
    Note you could have implemented this function using your own␣
 ↪"compute_LS_optimal_ridge_ERM_coefficients"
    but it is faster and easier for you to compute training loss and test one␣
 ↪using Scikit-learn implementation.
    :param x_train: input data used to get the linear model predictions
    :param y_train: output data to be predicted
    :param x_test: test features used to assess model performance
    :param y_test: test output to be predicted to assess model performance
    :param alpha: regularization hyper-parameter

    :returns: (COD_train, COD_test, w)
        WHERE
        COD_train : Coefficient of determination for the training dataset
        COD_test : Coefficient of determination for the test dataset
        w : parameters of the linear model (the bias is contained, return it as␣
 ↪the first element of w)
    '''
```

```
    # YOUR CODE HERE

    # Add the bias
    x_train = np.hstack((np.ones((x_train.shape[0],1)),x_train))
    x_test = np.hstack((np.ones((x_test.shape[0],1)),x_test))

    # Define the Ridge model
    ridge_model = linear_model.Ridge(alpha=alpha).fit(x_train, y_train)

    # Compute the CODs
    COD_train = ridge_model.score(x_train,y_train)
    COD_test = ridge_model.score(x_test,y_test)

    # Extract the coefficients w
    w = ridge_model.coef_ # dimension (19,)

    return (COD_train, COD_test, w)


alpha = 0.1
w_hat_ridge_hand = compute_LS_optimal_ridge_ERM_coefficients(x_train, y_train,␣
 ↪alpha = alpha)
print(f"w_hat \n {w_hat_ridge_hand}")
print(w_hat_ridge_hand.shape)
```

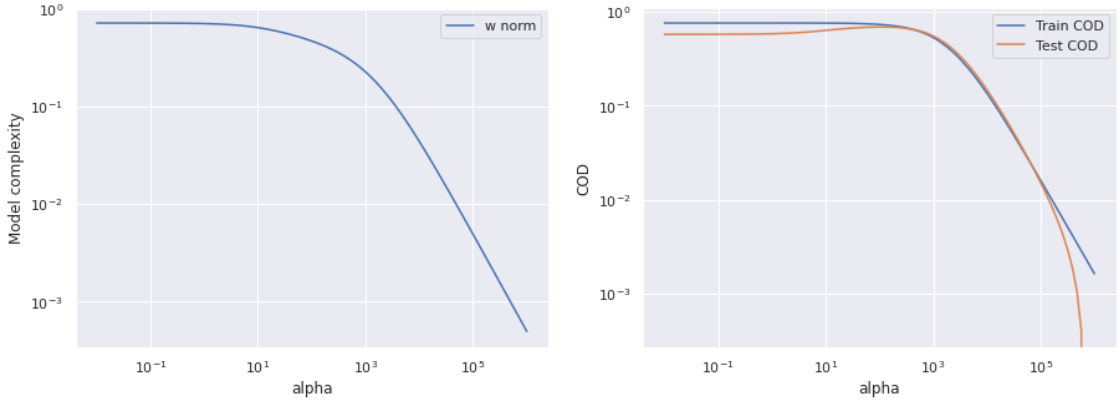
```
w_hat
 [ 6.09965562e-15 -7.02207903e-02  1.46454401e-01  2.15795633e-01
  2.03108256e-01  6.52360572e-02  2.89491731e-01  1.70505616e-02
  3.81403707e-02  3.03853575e-01  1.54713094e-01  1.60410278e-01
 -2.96094157e-01  5.20195787e-02 -7.64241464e-02  1.78171709e-01
 -1.25172898e-02 -3.57201034e-02 -1.79654969e-01]
(19,)
```

[27]:
```
# Compare your ridge regression solution with sklearn one
ridge = linear_model.Ridge(alpha=alpha)
ridge.fit(x_train, y_train)
ridge.coef_.shape, ridge.intercept_.shape
w_hat_ridge_sklearn = np.concatenate((ridge.intercept_.reshape(-1,), ridge.
 ↪coef_))

print(f"w_hat_sklearn \n {w_hat_ridge_sklearn}")
COD_train_ridge, COD_test_ridge, w_hat_ridge = solve_ridge_LS_problem(x_train,␣
 ↪y_train, x_test, y_test, alpha)

# Let's print model performance on train and test datasets (measured using COD)
print(f"Coefficient of determination on training data: {COD_train_ridge:.4f}")
```

```
print(f"Coefficient of determination on test data:     {COD_test_ridge:.4f}")

assert np.isclose(w_hat_ridge_hand, w_hat_ridge_sklearn, atol=1e-8).all()
assert np.isclose(w_hat_ridge_hand, w_hat_ridge, atol=1e-8).all()
```

```
w_hat_sklearn
 [ 6.01333618e-15 -7.02207903e-02  1.46454401e-01  2.15795633e-01
  2.03108256e-01  6.52360572e-02  2.89491731e-01  1.70505616e-02
  3.81403707e-02  3.03853575e-01  1.54713094e-01  1.60410278e-01
 -2.96094157e-01  5.20195787e-02 -7.64241464e-02  1.78171709e-01
 -1.25172898e-02 -3.57201034e-02 -1.79654969e-01]
Coefficient of determination on training data: 0.7573
Coefficient of determination on test data:     0.5741
```

[28]:
```
# Let's print model train and test metric as a function of the regularization␣
 ↪parameter alpha (which is constraining
# model complexity, the higher the norm of w the higher model complexity)

alphas = np.logspace(-2, 6, 100)
# alphas = np.linspace(1e-2, 1e6, 100)
ridge_results = [solve_ridge_LS_problem(x_train, y_train, x_test, y_test, a)␣
 ↪for a in alphas]

train_CODs = list(zip(*ridge_results))[0]
test_CODs = list(zip(*ridge_results))[1]
all_w_hat = list(zip(*ridge_results))[2]

fig, axes = plt.subplots(1,2, figsize=(15, 5))
axes[0].plot(alphas, list(map(np.linalg.norm, all_w_hat)), label='w norm')
axes[0].legend()
axes[0].set_xlabel('alpha')
axes[0].set_ylabel('Model complexity')

axes[1].plot(alphas, train_CODs, label='Train COD')
axes[1].plot(alphas, test_CODs, label='Test COD')
axes[1].legend()
axes[1].set_xlabel('alpha')
axes[1].set_ylabel('COD')

# If you would like to see these plots in log scale in x or y or both just␣
 ↪uncomment the following lines
axes[0].set_xscale('log')
axes[0].set_yscale('log')
axes[1].set_xscale('log')
axes[1].set_yscale('log')
```

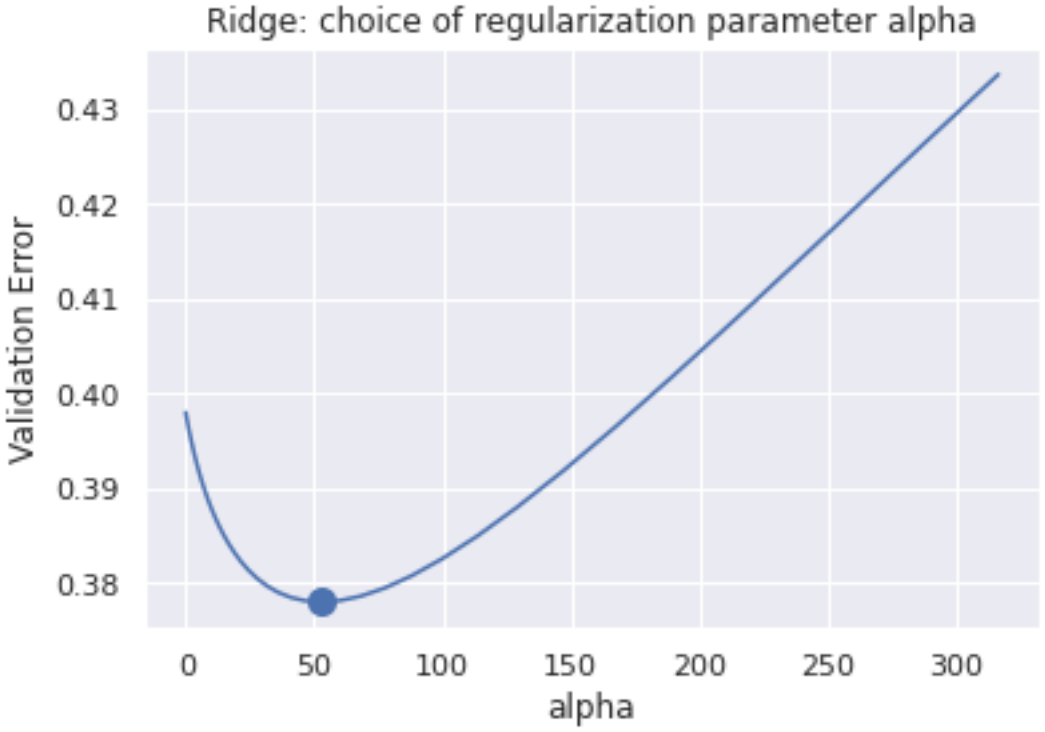


[29]:
```
# TODO 6
# How to choose the optimal alpha? This is an hyper-parameter, usually it is estimated dthrough Cross-validation.
# In this TODO we will implement by hand the Cross Validation procedure to estimate hyper-parameter alpha.
# The function we are going to implement is pretty general and can be applied both to Ridge and Lasso (see next)!

from sklearn.model_selection import KFold

def CV_by_hand(num_folds, model_class, other_model_hyper_parameters, loss, hyper_param_range, x_train, y_train):
    kf = KFold(n_splits = num_folds)
    loss_kfold = np.zeros(len(hyper_param_range),) # on 0 for every candidate lambda
    for i in range(len(hyper_param_range)):
        for train_index, validation_index in kf.split(x_train):
            # In order to complete the following 2 lines have a look at the documentation of
            # sklearn.model_selection.KFold (no need to insert new lines, just replace "None" with the correct
            # quantity)
            x_train_kfold, x_val_kfold = x_train[train_index], x_train[validation_index] # Assign the correct values
            y_train_kfold, y_val_kfold = y_train[train_index], y_train[validation_index] # Assign the correct values
            # YOUR CODE HERE
            # See assignments above

            # Initialize the model with the hyper-parameters you are willing to test (in this case we are interested
```

```
            # on alpha alone, but we might need to change the number of␣
 ↪iterations (or other hyper-parameters,
            # depending on the model we are using: Ridge, Lasso, etc.) to solve␣
 ↪the ERM problem. Therefore we need
            # to pass such an information using the dictionary␣
 ↪"other_model_hyper_parameters")
            model_kfold = model_class(alpha=hyper_param_range[i],␣
 ↪**other_model_hyper_parameters)
            # think of model_class as e.g. linear_model.Ridge

            # Fit the model using training data from the k-fold
            # YOUR CODE HERE
            model_kfold.fit(x_train_kfold,y_train_kfold)

            # Compute the loss using the validation data from the k-fold
            loss_kfold[i] += loss(y_val_kfold, x_val_kfold, model_kfold)
            # think of loss as e.g. the squared norm loss

    loss_kfold /= m_t # m_t = size of the union of the k S_i of size m_t/k

    return loss_kfold


n_alphas, num_folds = 100, 5
alphas = np.logspace(-3,2.5,num = n_alphas)
model_class = linear_model.Ridge
other_model_hyper_parameters = {}
loss = lambda y_val, x_val, model: np.linalg.norm(y_val - model.
 ↪predict(x_val))**2

# Perform CV with your implemented function
loss_ridge_kfold = CV_by_hand(num_folds, model_class,␣
 ↪other_model_hyper_parameters, loss, alphas, x_train, y_train)

assert loss_ridge_kfold.shape == (n_alphas,)

# Choose the regularization parameter that minimizes the validation loss
best_index, ridge_alpha_opt = None, None # Replace with the correct quatities
# YOUR CODE HERE
best_index = np.argmin(loss_ridge_kfold)
ridge_alpha_opt = alphas[best_index]
```

[30]:
```
plt.figure()
plt.plot(alphas, loss_ridge_kfold, color='b')
plt.scatter(ridge_alpha_opt, loss_ridge_kfold[best_index], color='b',␣
 ↪marker='o', linewidths=5)
```

```
plt.xlabel('alpha')
plt.ylabel('Validation Error')
plt.title('Ridge: choice of regularization parameter alpha')
plt.show()

COD_train_ridge_opt, COD_test_ridge_opt, w_hat_ridge_opt =␣
 ↪solve_ridge_LS_problem(x_train, y_train, x_test, y_test,

 ↪  ridge_alpha_opt)


print(f"Best value of the regularization parameter: {ridge_alpha_opt:.4f}")
# Let's print model performance on train and test datasets (measured using COD)
print(f"Coefficient of determination on training data: {COD_train_ridge_opt:.
 ↪4f}")
print(f"Coefficient of determination on test data:     {COD_test_ridge_opt:.
 ↪4f}")

np.random.seed(ID_number) # Do not consider this line, it is only used for␣
 ↪evaluation purposes
```




```
Best value of the regularization parameter: 52.7500
Coefficient of determination on training data: 0.7438
```

```
Coefficient of determination on test data:     0.6802
```

1.2 LASSO

In the following we will apply a different regularization to our linear model: LASSO - Least Absolute Shrinkage and Selection Operator (l1 regularization)

You will code the same function as above, using functions from the Scikit-learn module, to solve the lasso LS problem for a fixed value of the regularization hyper-parameter.

After that, use the routine `lasso_path` from `sklearn.linear_regression` to compute the "lasso path" for different values of the regularization parameter λ. You should first fix a grid of possible values of lambda (the variable `lasso_lams`). For each entry of the vector `lasso_lams` you should compute the corresponding model (The $i-$th column of the vector `lasso_coefs` should contain the coefficients of the linear model computed using `lasso_lams[i]` as regularization parameter).

Be careful that the grid should be chosen appropriately.

Note: the parameter λ is called α in the Lasso model from sklearn.

[31]:
```
# TODO 7
# As we did for ridge regression and LS, write a function to solve the Lasso LS␣
 ↪Problem exploiting sklearn
def solve_lasso_LS_problem(x_train : np.ndarray, y_train : np.array, x_test :␣
 ↪np.ndarray, y_test : np.array,
                           lam : float) -> tuple:
    '''
    Funtion used to compute the LASSO LS estimate given train data and lambda␣
 ↪hyper-parameter.
    This function uses Scikit-learn to get both the LASSO LS solution and other␣
 ↪required quantities.
    :param x_train: input data used to get the linear model predictions
    :param y_train: output data to be predicted
    :param x_test: test features used to assess model performance
    :param y_test: test output to be predicted to assess model performance
    :param lam: regularization hyper-parameter (what is called alphas in␣
 ↪sklearn)

    :returns: (COD_train, COD_test, w)
        WHERE
        COD_train : Coefficient of determination for the training dataset
        COD_test : Coefficient of determination for the test dataset
        w : parameters of the linear model (the bias is contained, return it as␣
 ↪the first element of w)
    '''

    # YOUR CODE HERE

    # Add the bias
```

```
    x_train = np.hstack((np.ones((x_train.shape[0],1)),x_train))
    x_test = np.hstack((np.ones((x_test.shape[0],1)),x_test))

    # Define the Ridge model
    lasso_model = linear_model.Lasso(alpha=lam).fit(x_train, y_train)

    # Compute the CODs
    COD_train = lasso_model.score(x_train,y_train)
    COD_test = lasso_model.score(x_test,y_test)

    # Extract the coefficients w
    w = lasso_model.coef_

    return (COD_train, COD_test, w)


lam = 0.1 # regularization hyper-parameter
COD_train_lasso, COD_test_lasso, w_hat_lasso = solve_lasso_LS_problem(x_train,␣
 ↪y_train, x_test, y_test, lam)

# Let's print model performance on train and test datasets (measured using COD)
print(f"Coefficient of determination on training data: {COD_train_lasso:.4f}")
print(f"Coefficient of determination on test data:     {COD_test_lasso:.4f}")
```

```
Coefficient of determination on training data: 0.6876
Coefficient of determination on test data:     0.6730
```

[32]:
```
assert w_hat_lasso.shape == (19,)
```

[33]:
```
# TODO 8
from sklearn.linear_model import lasso_path
# Select a grid of possible regularization parameters (be careful how this is␣
 ↪chosen, you may have to refine
# the choice after seeing the results)

num_lambdas = 100
# Below the logspace(-10,0) will be considered and so do I.
# Note that the array must be fed to the lasso_path method "with coordinate␣
 ↪descent"
lasso_lams = np.logspace(-10,0,num=num_lambdas)[::-1] # Replace with a proper␣
 ↪interval (no need to add any line)
# YOUR CODE HERE

# Use the function lasso_path (see documentation) to compute the "lasso path",␣
 ↪passing as input the lambda values
# you have specified in lasso_lams
# YOUR CODE HERE
```

```
_, lasso_coefs, _ = lasso_path(x_train, y_train, alphas=lasso_lams)

# Recall that lasso_coefs is a matrix (18 x num_lambdas) where coefficients on␣
 ↪the i-th column are
# computed using the i-th lambda.
# the bigger the lambda the sparser the column
```

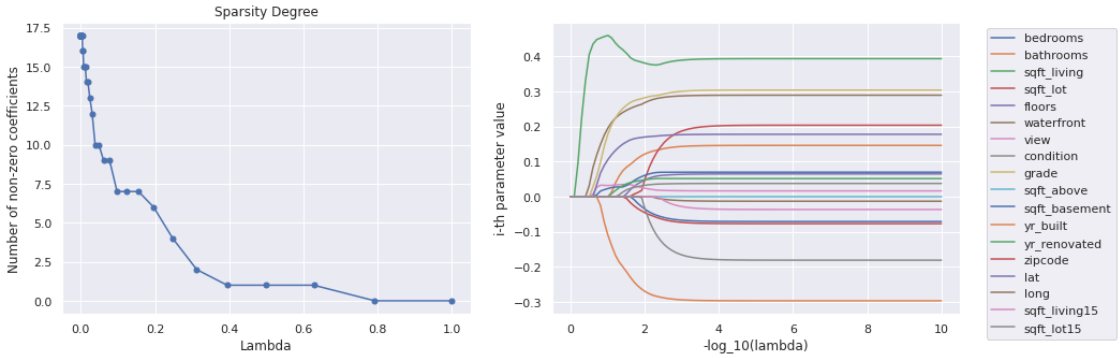
[34]:
```
assert lasso_coefs.shape == (18, num_lambdas)
```

Evaluate the sparsity in the estimated coefficients as a function of the regularization parameter λ: to this purpose, compute the number of non-zero entries in the estimated coefficient vector.

[35]:
```
number_non_zero_coeffs = np.zeros(len(lasso_lams),)
# The number of non zero coeffs must be evaluated for each lambda
# YOUR CODE HERE

# axis=0 select the non-zero elements for each column of lasso_coefs
number_non_zero_coeffs = np.array(np.count_nonzero(lasso_coefs, axis=0))
print(number_non_zero_coeffs)

fig, axes = plt.subplots(1,2, figsize=(15, 5))
axes[0].plot(lasso_lams, number_non_zero_coeffs, marker='o', markersize=5)
axes[0].set_xlabel('Lambda')
axes[0].set_ylabel('Number of non-zero coefficients')
axes[0].set_title('Sparsity Degree')

neg_log_alphas_lasso = -np.log10(lasso_lams) # This is used only to make a nice␣
 ↪plot (you can directly use: lasso_lams as x value)
for i, coef_l in enumerate(lasso_coefs):
    l1 = axes[1].plot(neg_log_alphas_lasso, coef_l, label=features_names[i])
axes[1].legend(bbox_to_anchor=(1.05, 1))
axes[1].set_xlabel('-log_10(lambda)')
axes[1].set_ylabel('i-th parameter value')
```

```
[ 0  0  1  1  1  2  4  6  7  7  7  9  9 10 10 12 13 14 14 15 15 15 16 16
 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17
 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17
 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17
 17 17 17 17]
```

[35]:
```
Text(0, 0.5, 'i-th parameter value')
```



[36]:
```
assert len(number_non_zero_coeffs) == num_lambdas
```

1.2.1 TODO 9: explain the results in the figures above (max 5 lines)

What does each plot mean? Did you observe what you would have expected from the theory? Type your answer in the next cell (no code needed)

1.2.2 YOUR CODE HERE

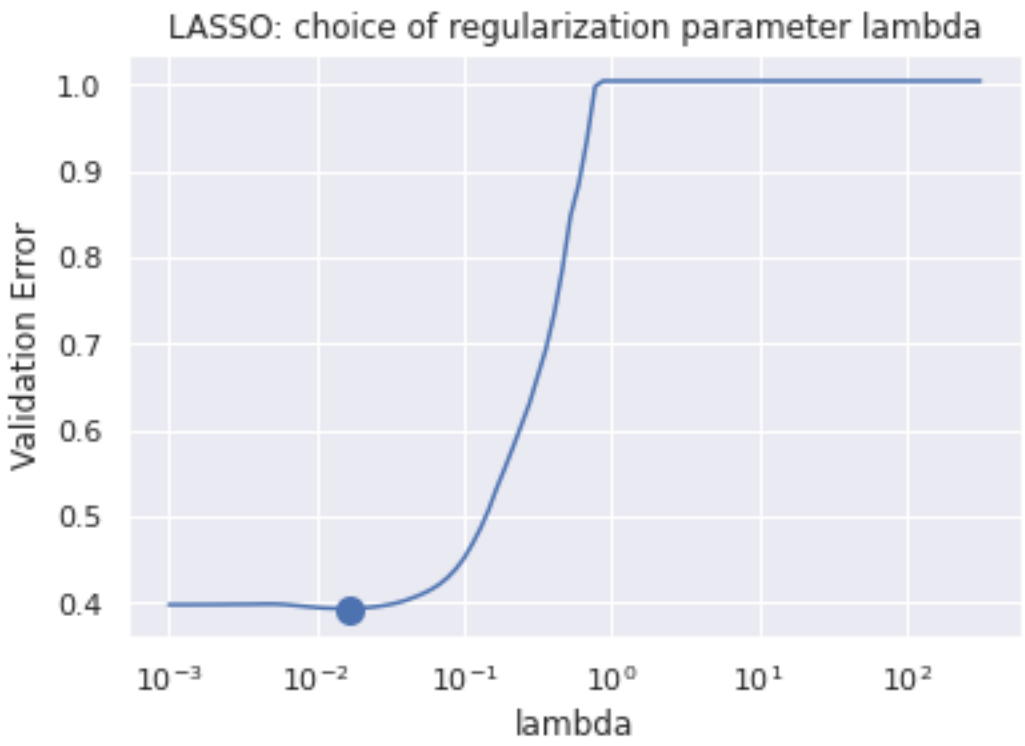
- The first plot shows how the number of non-zero coeffs decrease when λ increases (more Regularization). Note also as such plot must be decreasing.
- The second plot shows how every feature's value goes to 0 when increasing λ (or decreasing $1/\lambda$).

This is consistent with the theory: increasing λ leads to a larger penalization of solutions (vectors w) with larger components w_i. Therefore for $\lambda \to \infty$ the l_1-norm $||w||_1 \to 0$.

[37]:
```
# TODO 10
# Use Cross-Validation to find the optimal lam. You should use the CV function you implemented earlier.
# Once the omptimal lambda has been found, the following cell will automatically print its training and test error
num_folds, num_lambdas = 5, 100
lambdas = np.logspace(-3,2.5,num=num_lambdas) # Replace with a proper interval (no need to add any line)
# YOUR CODE HERE

model_class = linear_model.Lasso
other_model_hyper_parameters = {} # {'max_iter':10000} use this if you want to increase the number of iteration
                                  # of the optimization not required
loss = lambda y_val, x_val, model: np.linalg.norm(y_val - model.predict(x_val))**2

# Perform CV with your implemented function
```

```
loss_lasso_kfold = CV_by_hand(num_folds, model_class,␣
 ↪other_model_hyper_parameters, loss, lambdas, x_train, y_train)

# Do not worry if you get the warning ("Objective did not converge"). You may␣
 ↪try to increase the number of iterations,
# but the execution time will increase.
```

[38]:
```
assert loss_lasso_kfold.shape == (num_lambdas,)

# Choose the regularization parameter that minimizes the validation loss
best_index = np.argmin(loss_lasso_kfold)
lasso_lambda_opt = lambdas[best_index]


plt.figure()
plt.plot(lambdas, loss_lasso_kfold, color='b')
plt.scatter(lasso_lambda_opt, loss_lasso_kfold[best_index], color='b',␣
 ↪marker='o', linewidths=5)
plt.xlabel('lambda')
plt.ylabel('Validation Error')
plt.title('LASSO: choice of regularization parameter lambda')
plt.xscale('log')
plt.show()

COD_train_lasso_opt, COD_test_lasso_opt, w_hat_lasso_opt =␣
 ↪solve_lasso_LS_problem(x_train, y_train, x_test, y_test,

 ↪  lasso_lambda_opt)

print(f"Best value of the regularization parameter: {lasso_lambda_opt:.4f}")
# Let's print model performance on train and test datasets (measured using COD)
print(f"Coefficient of determination on training data: {COD_train_lasso_opt:.
 ↪4f}")
print(f"Coefficient of determination on test data:     {COD_test_lasso_opt:.
 ↪4f}")
print("Total number of coefficients:", len(w_hat_lasso_opt))
print("Number of non-zero coefficients:", sum(w_hat_lasso_opt != 0))
```




```
Best value of the regularization parameter: 0.0167
Coefficient of determination on training data: 0.7477
Coefficient of determination on test data:     0.6746
Total number of coefficients: 19
Number of non-zero coefficients: 14
```

[39]:
```
# Let's print some performance metrics of the models we defined throughout the␣
 ↪notebook
columns = ['COD_Train', 'COD_Test']
dict_results = {'LS_full': [COD_train_LS_full, COD_test_LS_full],
                'LS_reduced_hand': [COD_train_LS_reduced, COD_test_LS_reduced],
                'LS_reduced_BSS': [COD_train_BSS, COD_test_BSS],
                'ridge_opt': [COD_train_ridge_opt, COD_test_ridge_opt],
                'lasso_opt': [COD_train_lasso_opt, COD_test_lasso_opt]
               }
results = pd.DataFrame.from_dict(dict_results, orient='index', columns=columns)
results['Gen_gap'] = np.abs(results['COD_Train'] - results['COD_Test'])
results
```

[39]:

	COD_Train	COD_Test	Gen_gap
LS_full	0.757283	0.573130	0.184153
LS_reduced_hand	0.585423	0.553983	0.031440
LS_reduced_BSS	0.698454	0.616088	0.082365

```
ridge_opt          0.743763   0.680153   0.063610
lasso_opt          0.747747   0.674568   0.073179
```

[40]:
```
# Let's compare the final coefficients
ind = np.arange(1, len(w_hat_lasso_opt) + 1)  # the x locations for the groups
width = 0.25       # the width of the bars
fig, ax = plt.subplots(figsize=(15,7))
rects1 = ax.bar(ind,             w_LS_full,        width, color='r',
                label=f'LS norm_coeff {np.linalg.norm(w_LS_full):.4}, #␣
 ↪non-zero coeffs {sum(w_LS_full != 0)}')
rects2 = ax.bar(ind + width,     w_hat_ridge_opt , width, color='y',
                label=f'Ridge norm_coeff {np.linalg.norm(w_hat_ridge_opt):.4} #␣
 ↪non-zero coeffs {sum(w_hat_ridge_opt != 0)}')
rects3 = ax.bar(ind + 2 * width, w_hat_lasso_opt , width, color='g',
                label=f'Lasso norm_coeff {np.linalg.norm(w_hat_lasso_opt):.4} #␣
 ↪non-zero coeffs {sum(w_hat_lasso_opt != 0)}')
plt.xlabel('Coefficient Idx')
plt.ylabel('Coefficient Value')
plt.title('LS, Ridge and Lasso Coefficient')
plt.legend()
```

[40]:
```
<matplotlib.legend.Legend at 0x7fcca5e54a90>
```

1.2.3 TODO 11

Compare and comment the results obtained so far: did you get what you would have expected from the theory? Type your answer in the next cell (no code needed)

1.2.4 YOUR CODE HERE

Consider the above table which compares the CODs and and the Gaps.

- We can note that `LS_FULL`, which makes use of all the information (possibly redundant), achieves the best `COD_TRAIN`. On the other hand `LS_FULL` has also the highest `GEN_GAP` which is a sign that the model (at its full complexity) is overfitting the Training Data.
- In order to avoid overfitting we have to use more data (not this case) or alternatively to look for "simpler" models (decrease $|\mathcal{H}|$). To do this we must increase our inductive bias which means namely to add constraints on the possible hypothesis h or equivalently on the coefficient's vectors w.
- Let's ignore `LS_REDUCED_HAND` which is simply a naive implementation of the more powerful `LS_REDUCED_BSS`. We can note that even selecting only the 4 most significant features leads to increased `COD_TEST` (w.r.t. `LS_FULL`) and to a significant decreasing of the `GEN_GAP`. `LS_REDUCED_BSS` is therefore the best model so far.
- Note that `LSS_REDUCED_BSS` is basically a "brute force variable selection", therefore we expect the `LASSO_OPT` to do a better job than just cut away 14/18 features (as the `LSS_REDUCED_BSS` does). This is indeed what happens since the LASSO has higher CODs and lower gap.
- Note also that `RIDGE_OPT` is probably the best model of all since it is slightly better than the `LASSO_OPT` in terms of CODs and gap, but we "pay" this precision by using all of the 18 features instead of the 14 features of the LASSO.
- Regarding the bar plot, note how both the Ridge and the LASSO coeff. achieves a smaller norm and/or the use of a smaller number of features than the LS. (they are "simpler" models than LS).
- Finally note that the trivial constraints $L_S^{LASSO}, L_S^{Ridge} \geq L_S^{LS-FULL}$ hold.

1.2.5 How do LS, Ridge and LASSO reject redundant/useless features?

In the next TODOs we are going to create two new hand-crafted datasets: - first we simply replicate the same features a certain number of times (so that we will have redundant features) - second we simply add random features to the true 18 features we have

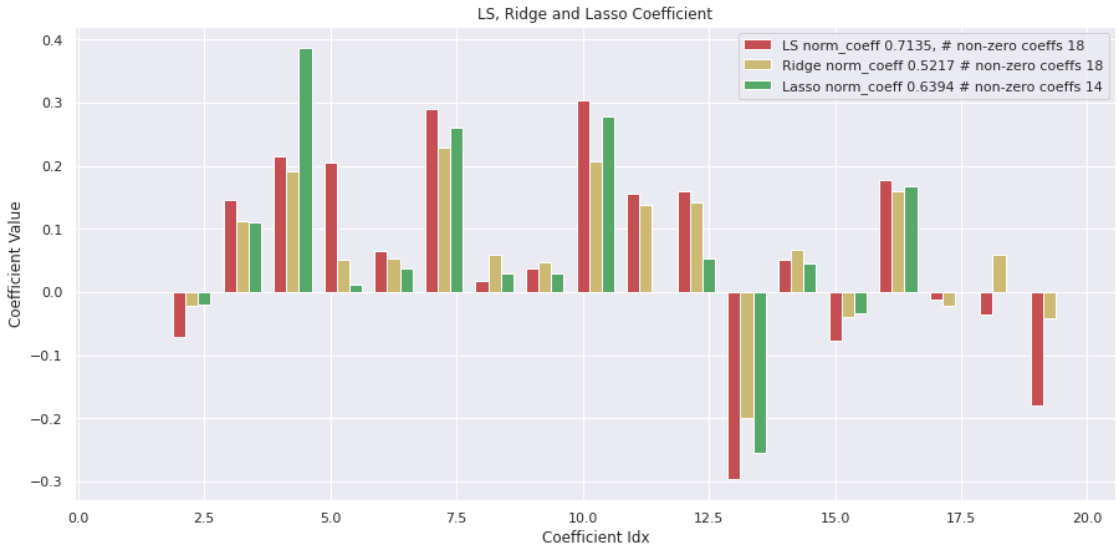
Let's see how the models we studied so far behaves in presence of this kind of nuisances.

[41]:
```
# Let's first create a function to train and print in one shot all the␣
 ↪quantities we are interested on: Training COD,
# Test COD, model parameters.

def solve_Ls_Ridge_Lasso(x_train : np.ndarray, y_train : np.array, x_test : np.
 ↪ndarray, y_test : np.array):
    # Solve Ordinary LS
    COD_train_LS_full, COD_test_LS_full, w_LS_full = solve_LS_problem(x_train,␣
 ↪y_train, x_test, y_test)
    num_folds, n_alphas, num_lambdas = 5, 100, 100
    loss = lambda y_val, x_val, model: np.linalg.norm(y_val - model.
 ↪predict(x_val))**2
```

```
   # Solve Ridge
   alphas = np.logspace(-3,2.5, num = n_alphas)
   loss_ridge_kfold = CV_by_hand(num_folds, linear_model.Ridge, {}, loss,␣
↪alphas, x_train, y_train)
   best_index_ridge = np.argmin(loss_ridge_kfold)
   ridge_alpha_opt = alphas[best_index_ridge]
   COD_train_ridge_opt, COD_test_ridge_opt, w_hat_ridge_opt =␣
↪solve_ridge_LS_problem(x_train, y_train, x_test, y_test,
                                                                        ␣
↪      ridge_alpha_opt)

   # Solve LASSO
   lambdas = np.logspace(-10,0, num = num_lambdas)
   loss_lasso_kfold = CV_by_hand(num_folds, linear_model.Lasso, {}, loss,␣
↪lambdas, x_train, y_train)
   best_index = np.argmin(loss_lasso_kfold)
   lasso_lambda_opt = lambdas[best_index]
   COD_train_lasso_opt, COD_test_lasso_opt, w_hat_lasso_opt =␣
↪solve_lasso_LS_problem(x_train, y_train, x_test, y_test,
                                                                        ␣
↪      lasso_lambda_opt)

   # The following is simply a copy and paste of what we have done earlier
   columns = ['COD_Train', 'COD_Test']
   dict_results = {
                   'LS_full': [COD_train_LS_full, COD_test_LS_full],
                   'ridge_opt': [COD_train_ridge_opt, COD_test_ridge_opt],
                   'lasso_opt': [COD_train_lasso_opt, COD_test_lasso_opt]
                  }
   results = pd.DataFrame.from_dict(dict_results, orient='index',␣
↪columns=columns)
   results['Gen_gap'] = np.abs(results['COD_Train'] - results['COD_Test'])

   # Let's compare the final coefficients
   ind = np.arange(1, len(w_hat_lasso_opt) + 1)  # the x locations for the␣
↪groups
   width = 0.25       # the width of the bars
   fig, ax = plt.subplots(figsize=(15,7))
   rects1 = ax.bar(ind,             w_LS_full,        width, color='r',
                   label=f'LS norm_coeff {np.linalg.norm(w_LS_full):.4}, #␣
↪non-zero coeffs {sum(w_LS_full != 0)}')
   rects2 = ax.bar(ind + width,     w_hat_ridge_opt , width, color='y',
                  label=f'Ridge norm_coeff {np.linalg.norm(w_hat_ridge_opt):.
↪4} # non-zero coeffs {sum(w_hat_ridge_opt != 0)}')
   rects3 = ax.bar(ind + 2 * width, w_hat_lasso_opt , width, color='g',
```

```
                    label=f'Lasso norm_coeff {np.linalg.norm(w_hat_lasso_opt):.
↪4} # non-zero coeffs {sum(w_hat_lasso_opt != 0)}')
    plt.xlabel('Coefficient Idx')
    plt.ylabel('Coefficient Value')
    plt.title('LS, Ridge and Lasso Coefficient')
    plt.legend()

    return results, fig
```

[42]:
```
old_results, old_fig = solve_Ls_Ridge_Lasso(x_train, y_train, x_test, y_test)
```



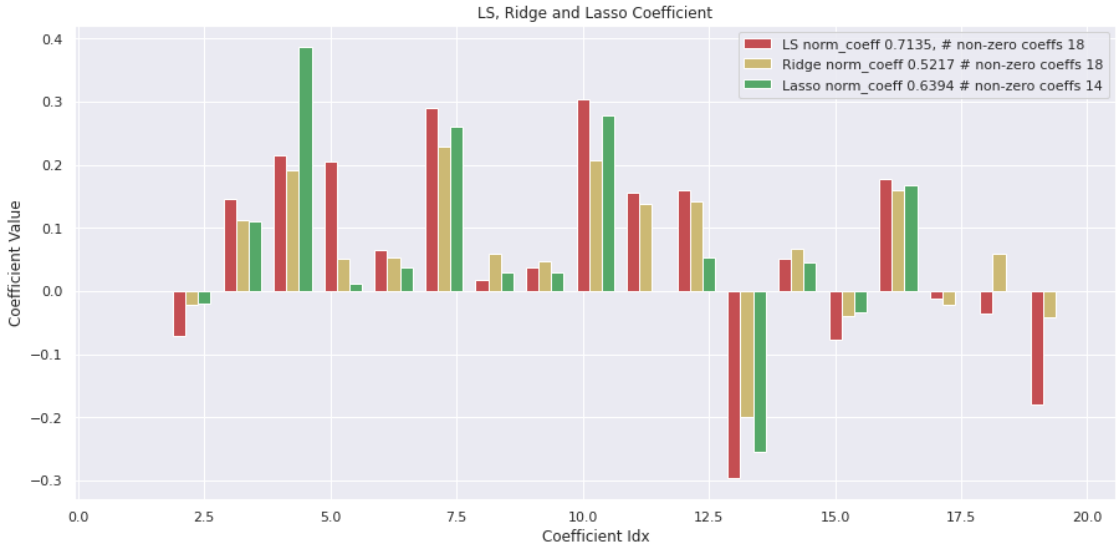
[43]:
```
# We should now see the very same results we obtained earlier
old_results
```

[43]:

	COD_Train	COD_Test	Gen_gap
LS_full	0.757283	0.573130	0.184153
ridge_opt	0.743763	0.680153	0.063610
lasso_opt	0.748509	0.673403	0.075106

[44]:
```
# We should now see the very same results we obtained earlier
old_fig
```

[44]:

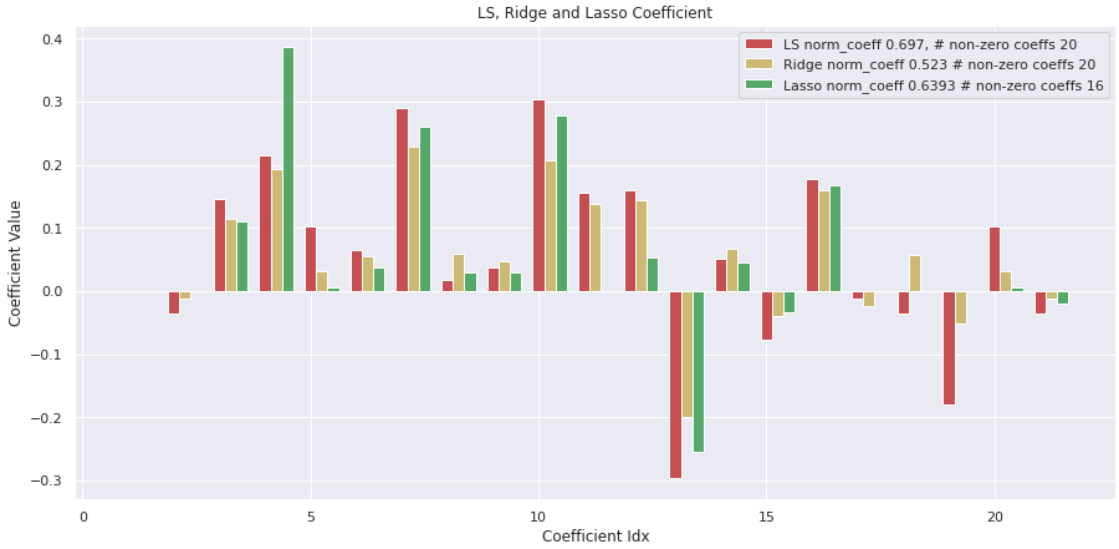


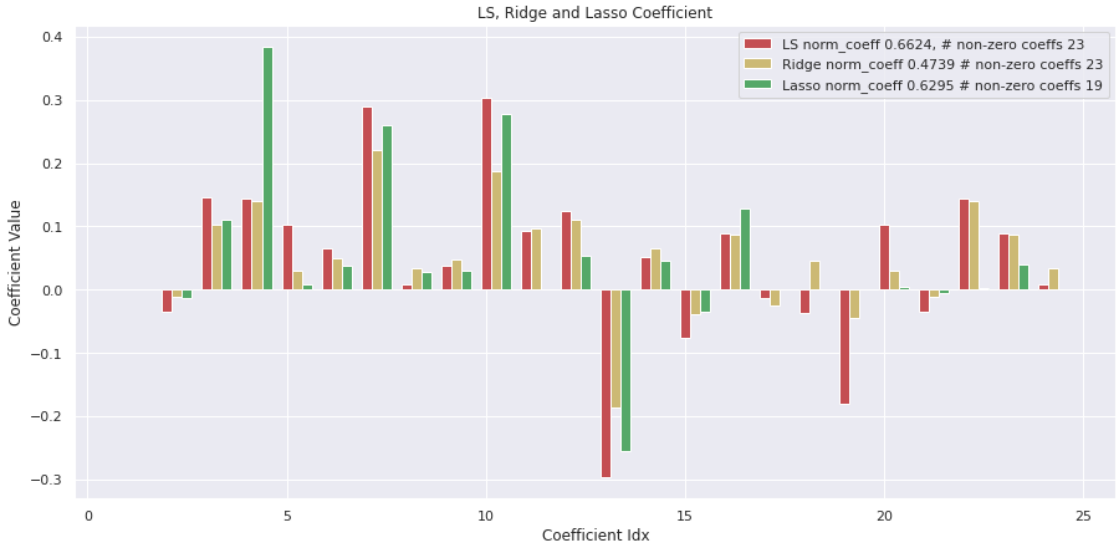
[45]:
```python
# TODO 12
# Write a function that replicates a single feature (chosen at random) from a dataset x_train
def replicate_a_random_feature(x_train : np.array, x_test : np.array) -> np.array:
    '''
    This function replicates a randomly chosen feature from the ones of a given dataset and return a dataset
    containing all the old features + the copied one (better placed in the last position!) - this operation must be
    done for the test dataset too.
    :param x_train: Features we are willing to replicate of shape (m_t, n_feats)
    :param x_test: Features we are willing to replicate of shape (m_test, n_feats)

    :returns: (new_x_train, new_x_test)
        WHERE
            new_x_train: New set of train features with a replicated feature randomly chosen
                         (its shape is (m_t, n_feats + 1))
            new_x_test: New set of test features with a replicated feature randomly chosen
                         (its shape is (m_test, n_feats + 1))
    '''
    # YOUR CODE HERE

    # Define a random index to select 1 of the 18 features
    rand_ind = np.random.choice(18)
```

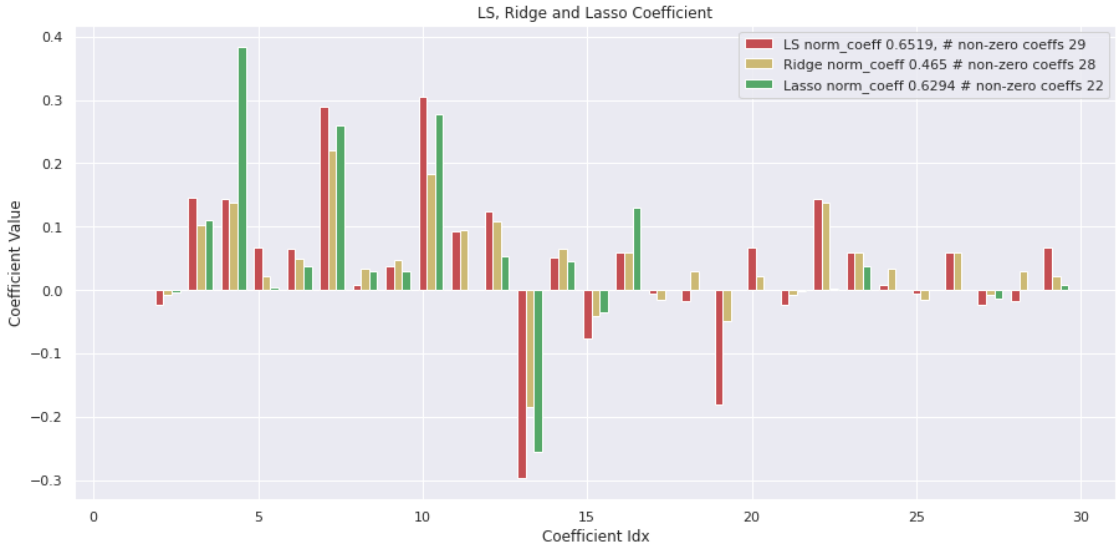
```
    # create new train dataset
    new_x_train = np.hstack((x_train,x_train[:,rand_ind].reshape((-1,1))))

    # create new test dataset.
    new_x_test = np.hstack((x_test, x_test[:,rand_ind].reshape((-1,1))))

    return (new_x_train, new_x_test)

# Write a function that adds a single random feature (normalized: zero mean and␣
 ↪unit variance) to the dataset
def add_random_feature(x_train : np.array, x_test : np.array) -> np.array:
    '''
    This function adds (better in the last position!) a random feature␣
 ↪(normalized: zero mean and unit variance) to a
    given dataset (this operation must be done for the test dataset too).␣
 ↪Random train and test features are
    extracted from the same normalized gaussian distribution but they are not␣
 ↪the same realization.
    :param x_train: Features from the train dataset (m_t, n_feats)
    :param x_test: Features from the test dataset (m_test, n_feats)

    :returns: (new_x_train, new_x_test)
        WHERE
            new_x_train: New set of train features with a single normalized␣
 ↪feature added
                         (its shape is (m_t, n_feats + 1))
            new_x_test: New set of test features with a single normalized␣
 ↪feature added
                         (its shape is (m_test, n_feats + 1))
    '''
    # YOUR CODE HERE

    # Create random column vector for train dataset
    random_feature_column_train = np.random.normal(0,1,(x_train.shape[0],1))
    # Create modified train dataset
    new_x_train = np.hstack((x_train, random_feature_column_train))

    # Create random column vector for test dataset
    random_feature_column_test = np.random.normal(0,1,(x_test.shape[0],1))
    # Create modified test dataset
    new_x_test = np.hstack((x_test, random_feature_column_test))

    return (new_x_train, new_x_test)
```

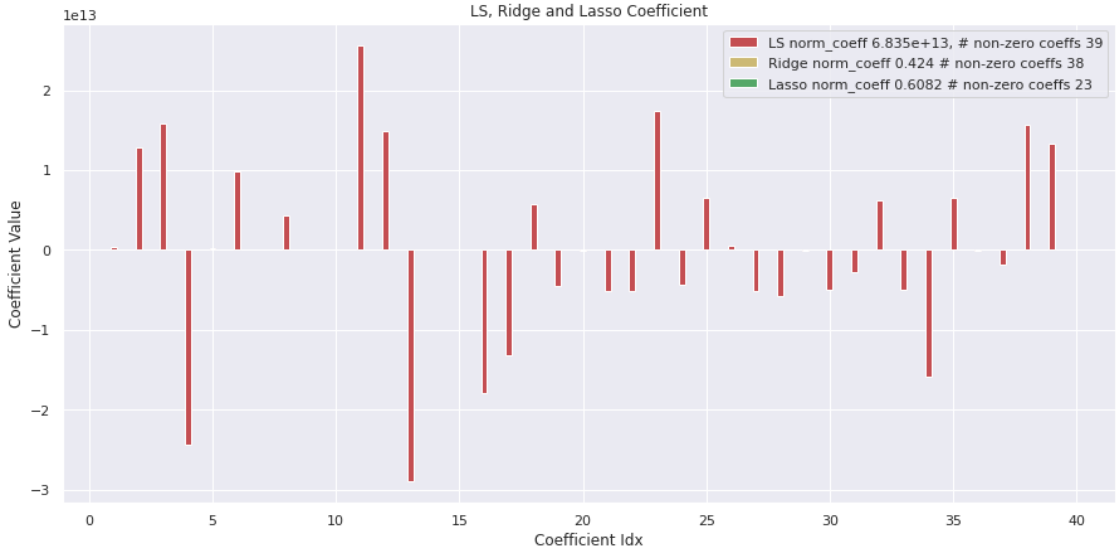
[46]:
```
replicated_x_train, replicated_x_test = replicate_a_random_feature(x_train,␣
 ↪x_test)
added_x_train, added_x_test = add_random_feature(x_train, x_test)

assert (x_train.shape[0], x_train.shape[1] + 1) ==  replicated_x_train.shape
assert (x_test.shape[0], x_test.shape[1] + 1) ==  replicated_x_test.shape

assert (x_train.shape[0], x_train.shape[1] + 1) ==  added_x_train.shape
assert (x_test.shape[0], x_test.shape[1] + 1) ==  added_x_test.shape

assert np.isclose(added_x_train[:,-1].mean(), 0., atol=1e-1)
assert np.isclose(added_x_train[:,-1].var(), 1., atol=0.5)
```

[47]:
```
replicated_x_train, replicated_x_test = replicate_a_random_feature(x_train,␣
 ↪x_test)
num_duplicated_features, checkpoints = 21, [1,2,5,10,20]
results_at_checkpoints, figs_at_checkpoints = [], []
for i in range(1, num_duplicated_features):
    if i in checkpoints:
        results, fig = solve_Ls_Ridge_Lasso(replicated_x_train, y_train,␣
 ↪replicated_x_test, y_test)
        results_at_checkpoints.append(results)
        figs_at_checkpoints.append(fig)
    replicated_x_train, replicated_x_test =␣
 ↪replicate_a_random_feature(replicated_x_train, replicated_x_test)
```


LS, Ridge and Lasso Coefficient
LS norm_coeff 0.697, # non-zero coeffs 20
Ridge norm_coeff 0.523 # non-zero coeffs 20
Lasso norm_coeff 0.6393 # non-zero coeffs 16
Coefficient Value
Coefficient Idx


LS, Ridge and Lasso Coefficient
LS norm_coeff 0.6624, # non-zero coeffs 23
Ridge norm_coeff 0.4739 # non-zero coeffs 23
Lasso norm_coeff 0.6295 # non-zero coeffs 19
Coefficient Value
Coefficient Idx





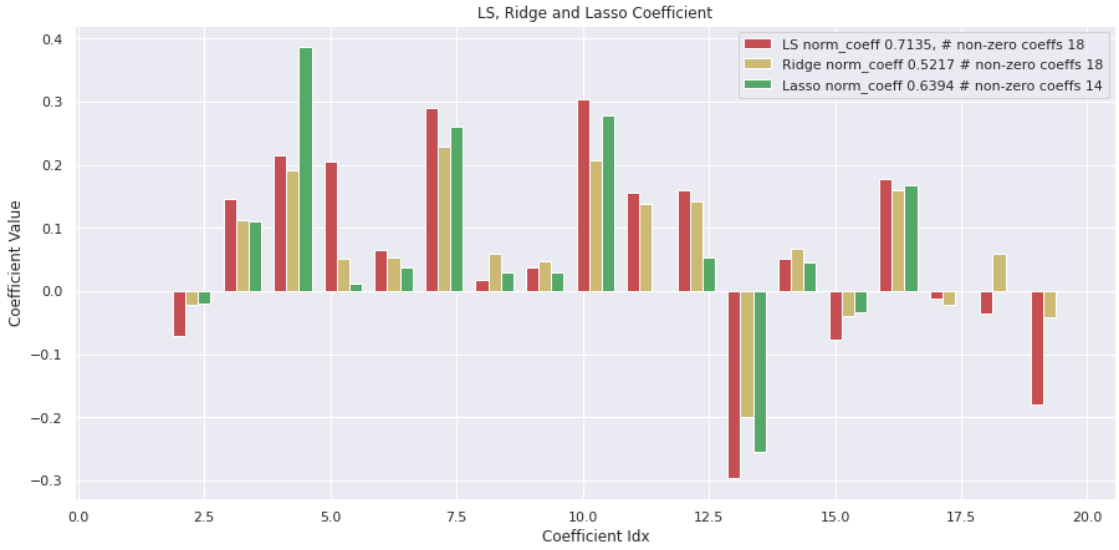
[48]:
```
for checkpoint, results in zip(checkpoints, results_at_checkpoints):
    print(checkpoint, results)
```

```
1            COD_Train  COD_Test   Gen_gap
LS_full     0.757283  0.573130  0.184153
ridge_opt   0.744221  0.677092  0.067130
lasso_opt   0.748509  0.673403  0.075106
2            COD_Train  COD_Test   Gen_gap
LS_full     0.757283  0.573130  0.184153
ridge_opt   0.744336  0.677317  0.067020
```

```
lasso_opt   0.748509  0.673404  0.075106
5            COD_Train  COD_Test   Gen_gap
LS_full     0.757283  0.573130  0.184153
ridge_opt   0.743134  0.678003  0.065131
lasso_opt   0.748507  0.673406  0.075100
10             COD_Train  COD_Test   Gen_gap
LS_full     0.757283  0.573130  0.184153
ridge_opt   0.743007  0.676971  0.066037
lasso_opt   0.748507  0.673409  0.075097
20             COD_Train  COD_Test   Gen_gap
LS_full     0.753236  0.571614  0.181622
ridge_opt   0.743685  0.668329  0.075355
lasso_opt   0.748510  0.673399  0.075111
```

[49]:
```
old_fig
```

[49]:

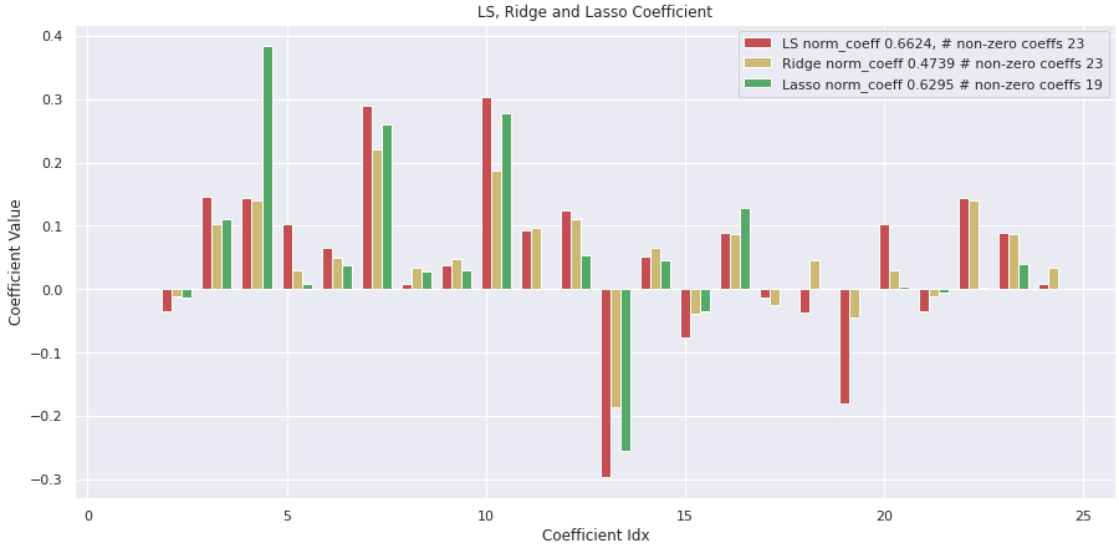


[50]:
```
figs_at_checkpoints[0]
```

[50]:

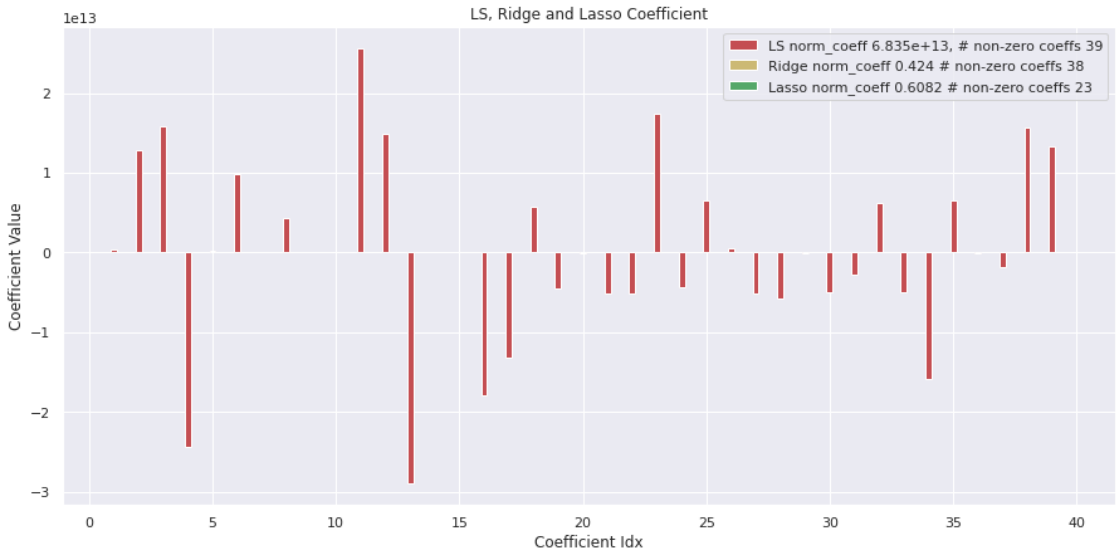
[51]:
```
figs_at_checkpoints[2]
```

[51]:

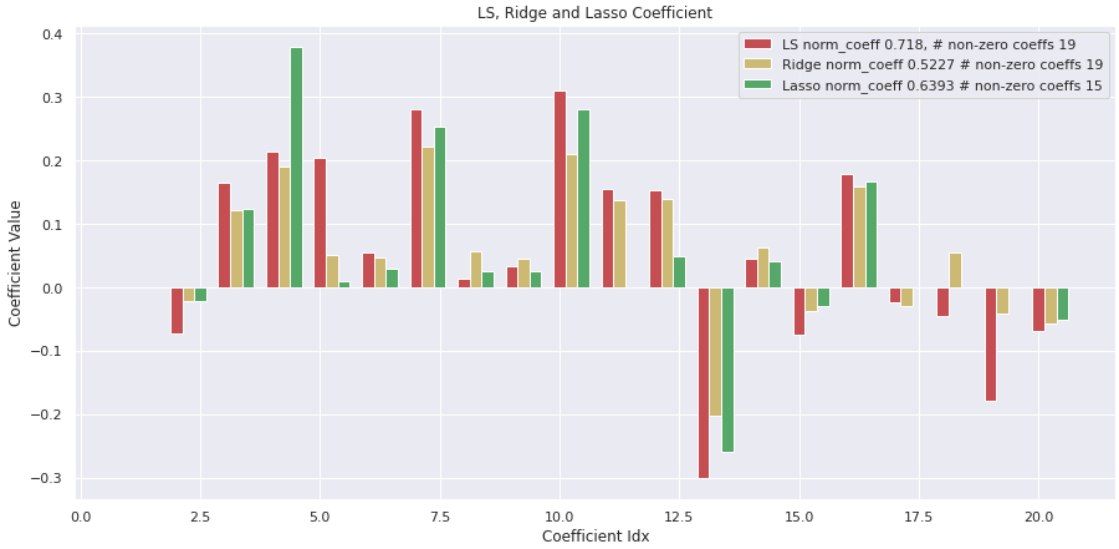


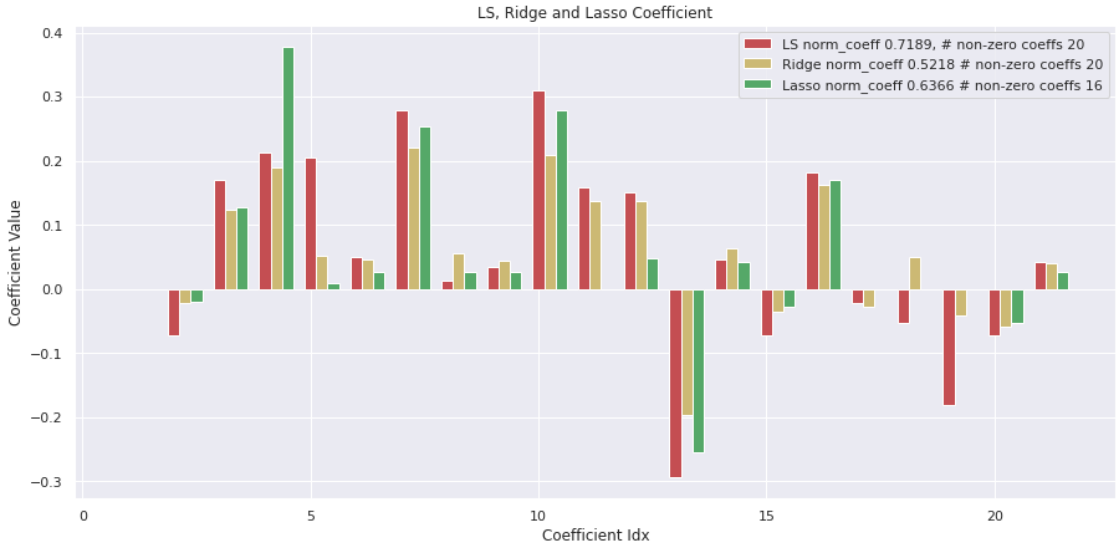
[52]:
```
figs_at_checkpoints[-1]
```

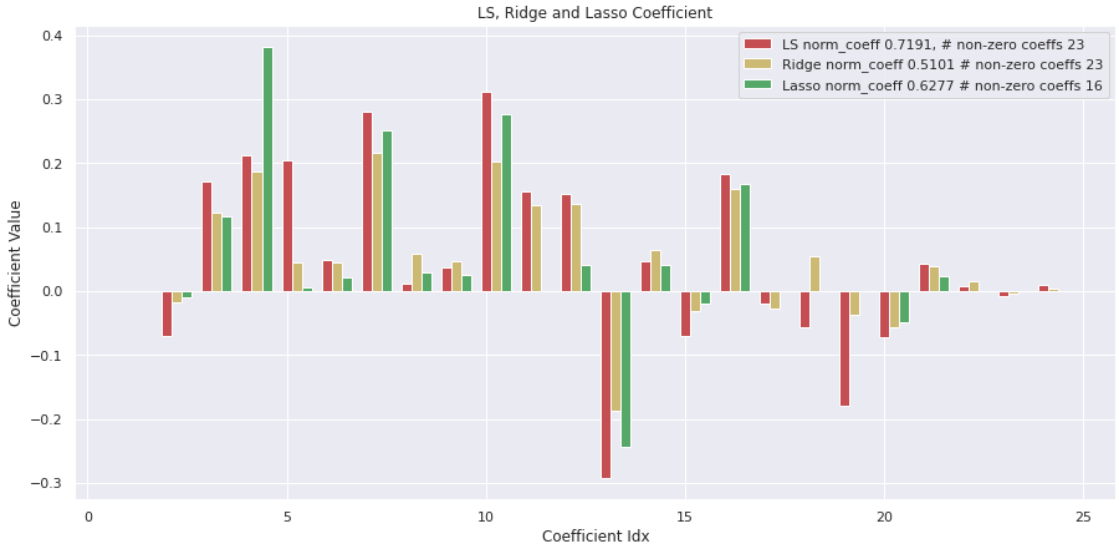
[52]:

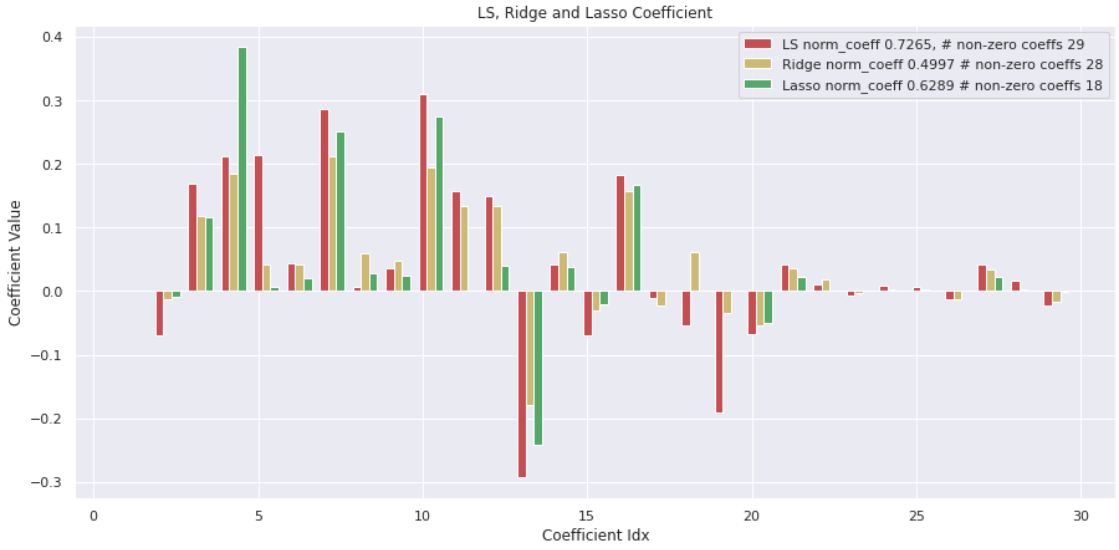


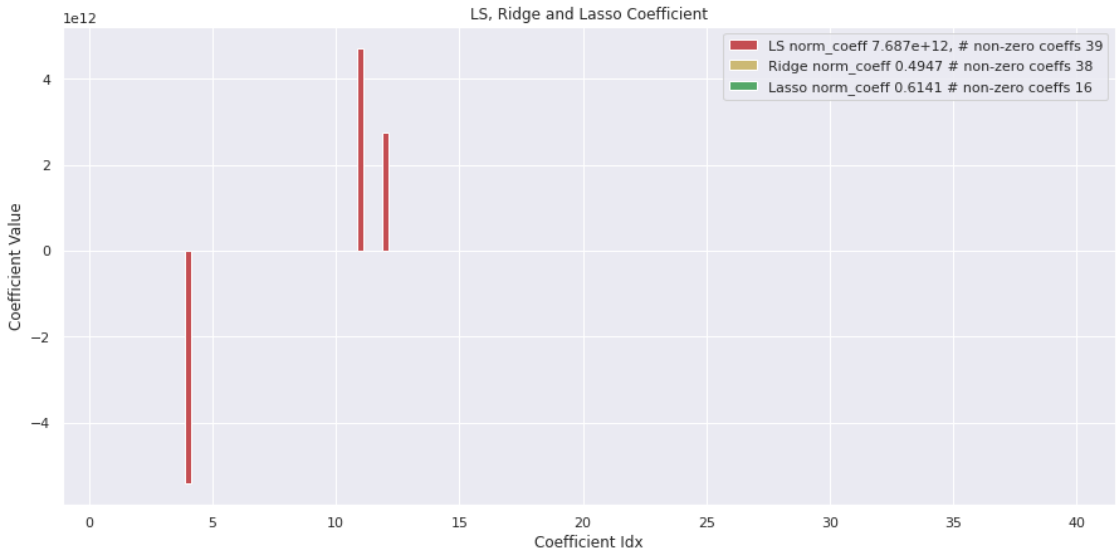
[53]:
```
added_x_train, added_x_test = add_random_feature(x_train, x_test)
num_added_features, checkpoints = 21, [1,2,5,10,20]
added_results_at_checkpoints, added_figs_at_checkpoints = [], []
lambdas = []
for i in range(1, num_added_features):
    if i in checkpoints:
        results, fig = solve_Ls_Ridge_Lasso(added_x_train, y_train,␣
 ↪added_x_test, y_test)
        added_results_at_checkpoints.append(results)
        added_figs_at_checkpoints.append(fig)
        lambdas
    added_x_train, added_x_test = add_random_feature(added_x_train,␣
 ↪added_x_test)
```


LS, Ridge and Lasso Coefficient
LS norm_coeff 0.718, # non-zero coeffs 19
Ridge norm_coeff 0.5227 # non-zero coeffs 19
Lasso norm_coeff 0.6393 # non-zero coeffs 15
Coefficient Value
Coefficient Idx


LS, Ridge and Lasso Coefficient
LS norm_coeff 0.7189, # non-zero coeffs 20
Ridge norm_coeff 0.5218 # non-zero coeffs 20
Lasso norm_coeff 0.6366 # non-zero coeffs 16
Coefficient Value
Coefficient Idx


LS, Ridge and Lasso Coefficient
LS norm_coeff 0.7191, # non-zero coeffs 23
Ridge norm_coeff 0.5101 # non-zero coeffs 23
Lasso norm_coeff 0.6277 # non-zero coeffs 16
Coefficient Value
Coefficient Idx


LS, Ridge and Lasso Coefficient
LS norm_coeff 0.7265, # non-zero coeffs 29
Ridge norm_coeff 0.4997 # non-zero coeffs 28
Lasso norm_coeff 0.6289 # non-zero coeffs 18
Coefficient Value
Coefficient Idx

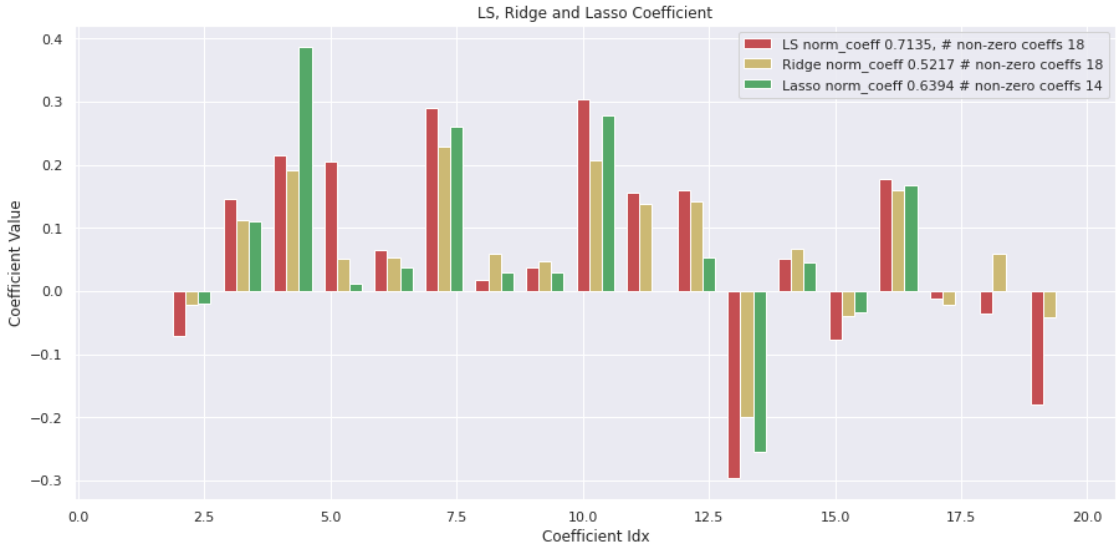


[54]:
```
for checkpoint, results in zip(checkpoints, added_results_at_checkpoints):
    print(checkpoint, results)
```

```
1            COD_Train  COD_Test   Gen_gap
LS_full     0.760761  0.569643  0.191118
ridge_opt   0.746963  0.677414  0.069549
lasso_opt   0.751622  0.670551  0.081071
2            COD_Train  COD_Test   Gen_gap
LS_full     0.762286  0.567859  0.194427
ridge_opt   0.748421  0.676533  0.071888
lasso_opt   0.752752  0.671056  0.081697
5            COD_Train  COD_Test   Gen_gap
LS_full     0.762456  0.566812  0.195644
ridge_opt   0.746437  0.677311  0.069125
lasso_opt   0.750402  0.674037  0.076364
10            COD_Train  COD_Test   Gen_gap
LS_full     0.765059  0.557109  0.207950
ridge_opt   0.746202  0.677267  0.068935
lasso_opt   0.751735  0.672979  0.078756
20            COD_Train  COD_Test   Gen_gap
LS_full     0.776351  0.534932  0.241418
ridge_opt   0.753434  0.673051  0.080383
lasso_opt   0.748916  0.678185  0.070731
```

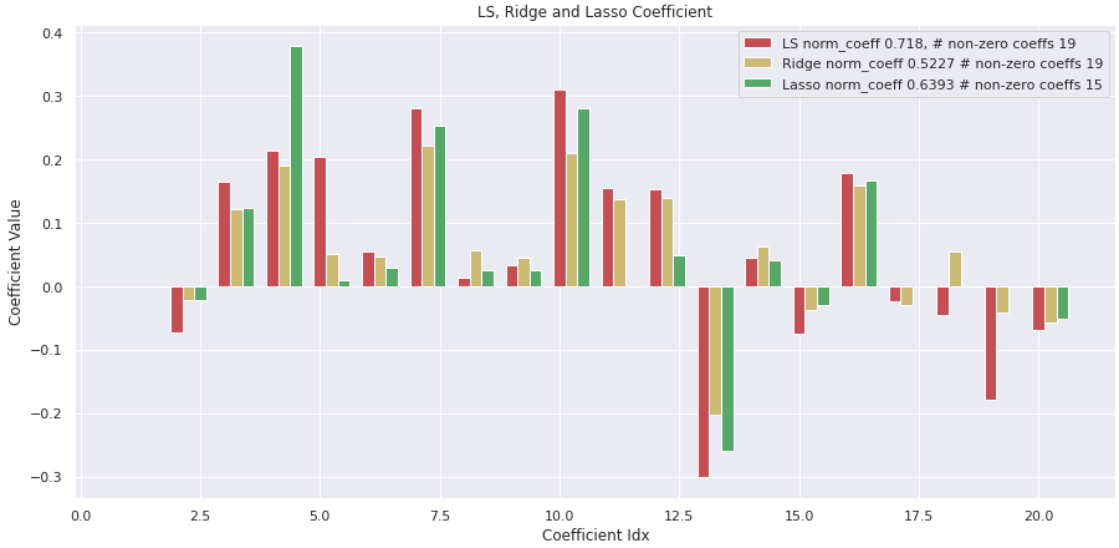
[55]:
```
old_fig
```

[55]:



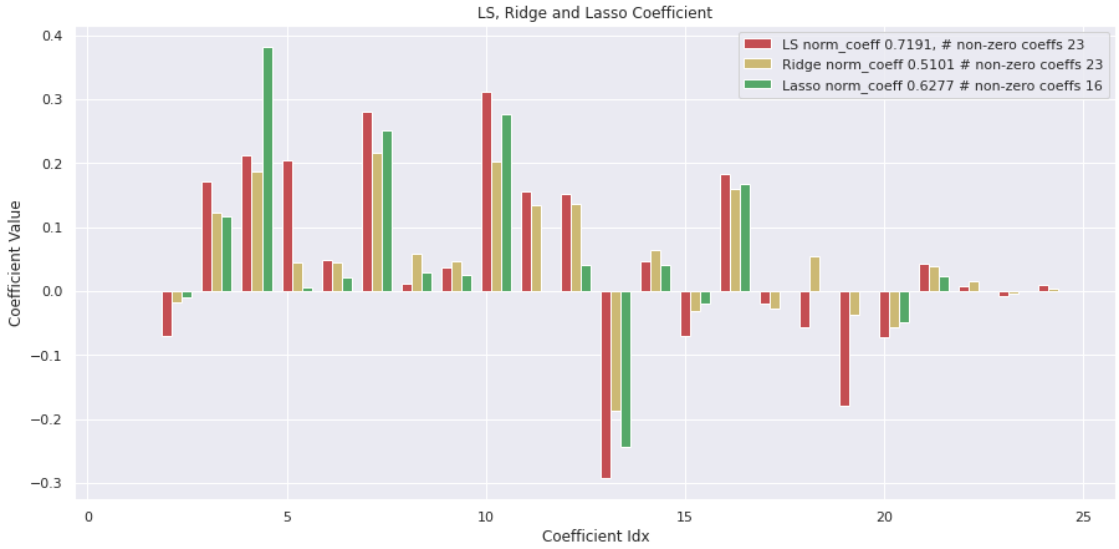
[56]:
```
added_figs_at_checkpoints[0]
```

[56]:



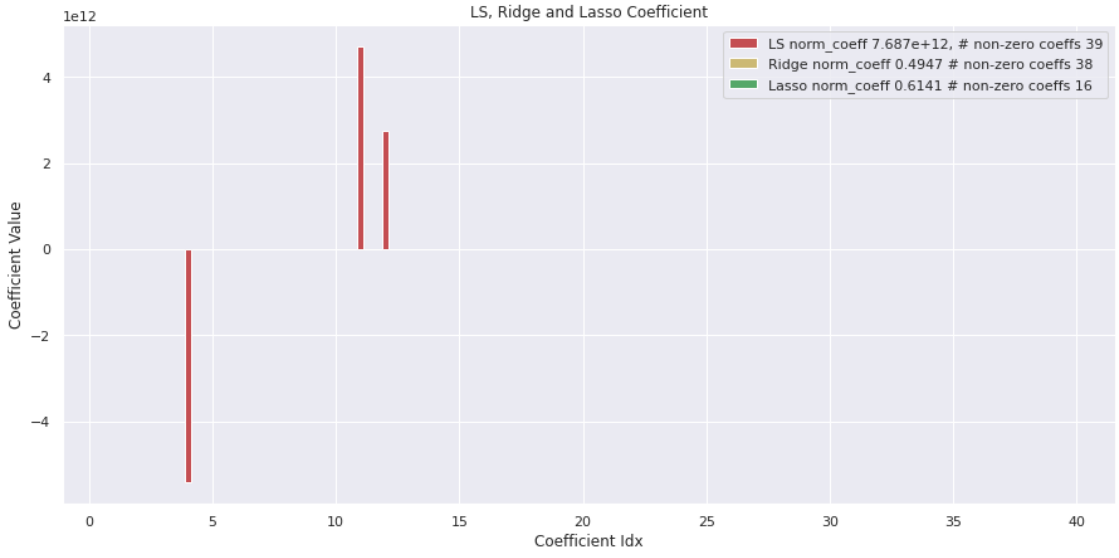
[57]:
```
added_figs_at_checkpoints[2]
```

[57]:



[58]:
```
added_figs_at_checkpoints[-1]
```

[58]:



1.2.6 TODO 13 (you do not need more than 5-7 lines)

What do you observe from previous cells? Describe what you found (both on the tables and figures): is this what you expected (describe in particular the generalization gap as a function of the number of redundant and random features)?

Answer in the next cell (no need to add any piece of code)

1.2.7 YOUR CODE HERE

- When **duplicating** features we add redundant information to the model. This leads (in the LS case) to what is called Multicollinearity (high sensitivity of the **coefficients** w_i to perturbations). Multicollinearity doesn't affect the precision of the overall model (the **gap** doesn't grow too much) but could cause $||w||_2^2$ to grow a lot (see plots with 20 added features).

- When adding **random** (and hence "useless") features the **LS_gap** tends to **grow**. Such features are in high correlation to each other and adding 20 of them leads to Multicollinearity (high norm of w) once again (see last plot).

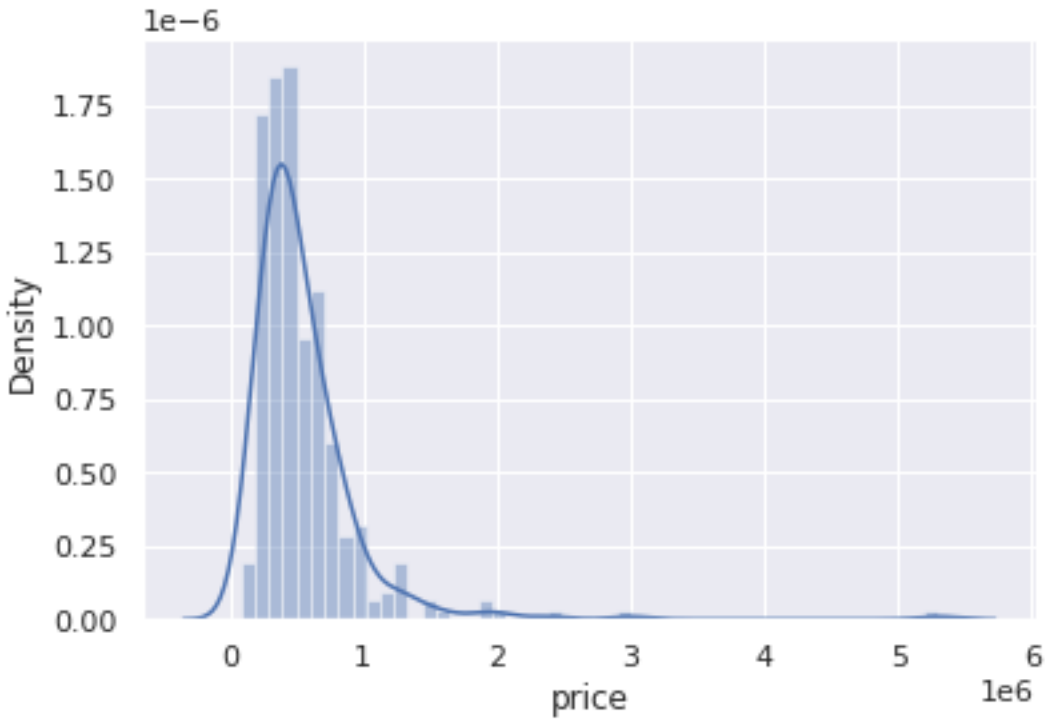
- Regularized models (Ridge and LASSO) are more robust w.r.t. fictitious features:
  1. Model complexity doesn't explode as in LS (see norm_coeff in both cases).
  2. The **gap** is not a monotonic function of the number of added features.

At the beginning of the HW we looked at the distrbutions of the features and their correlation with the regression variable 'price'. We completely overlooked the distribution of 'price' itself. The question now is: - Can we get some insight on the regression problem looking at the distribution of the regression variable? - Can we apply a transformation to the regression variable to make the regression task easier?

Let's have a look at the distribution of the houses price then!

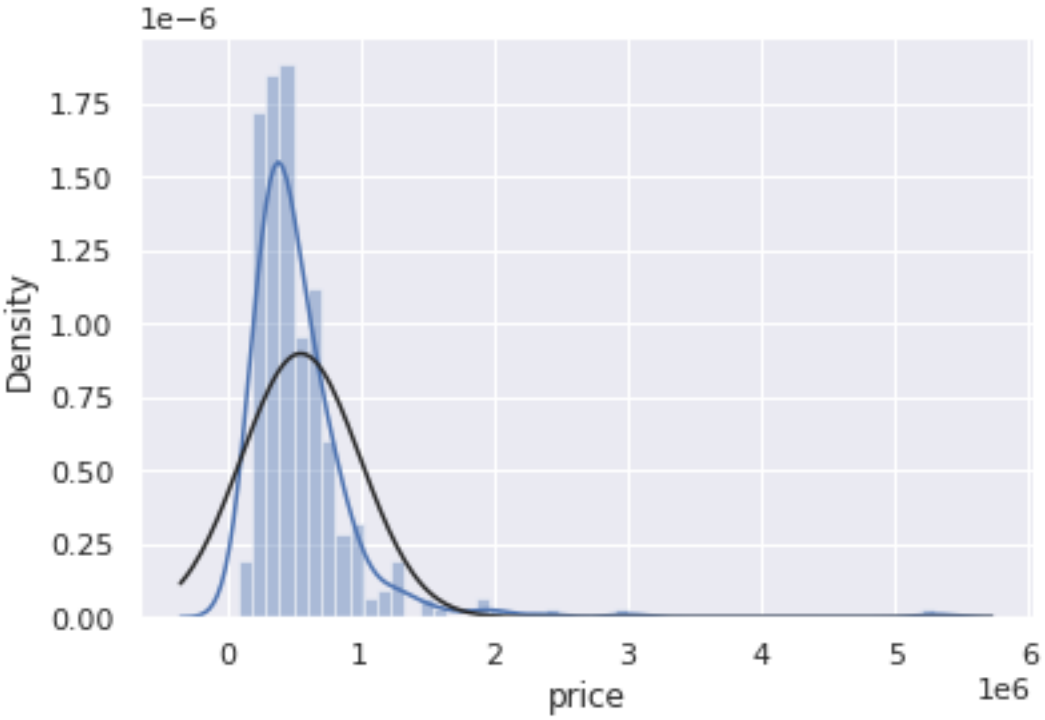
[59]:
```
sns.distplot(train_data['price'])
# Note:
# - This deviates from a normal distribution
# - Positive skewness (tail above the mean value)
# - Peakedness (single peak)

# Print skewness and kurtosis
print("Skewness: %f" % train_data['price'].skew())
print("Kurtosis: %f" % train_data['price'].kurt())

# Note:
# High skewness might make the regression task harder: few data with very high␣
 ↪values, far away from the median
# (but they are not outliers)
# Can we reduce the skewness to make the regression task easier?
```

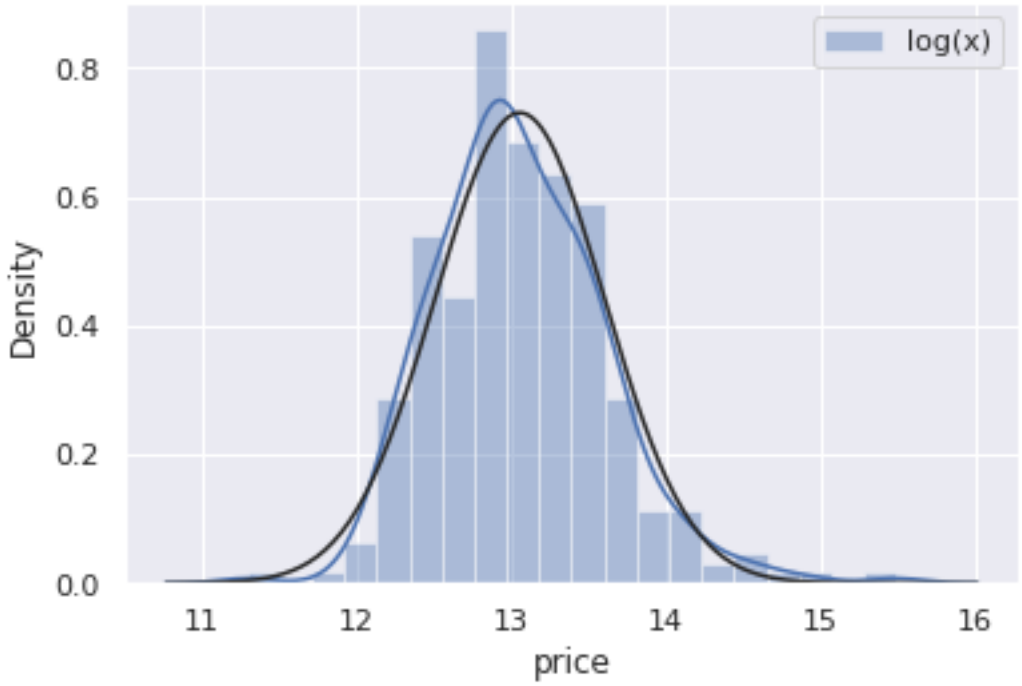
```
Skewness: 5.402253
Kurtosis: 47.043386
```



[60]:
```
# How close is this to gaussianity?
sns.distplot(train_data['price'], fit=stats.norm)
# Not that much, BUT it seems to be peaked
# Can we reduce its (right) skewness? Let's try to apply a logarithm to the␣
 ↪price (we can use log(x) or log(x+1), the
# latter allows x to be equals to zero). In the following we will simply use␣
 ↪log(x) (it's hard to find an house whose
# price is zero!)
plt.figure()
sns.distplot(np.log(train_data['price']), fit=stats.norm, label='log(x)')
plt.legend()
# Great, we know we can make our regression variable resemble a gaussian␣
 ↪distribution if we use a log transform!
# This will greatly simplifies our regression task.
```

[60]:
```
<matplotlib.legend.Legend at 0x7fccaddfb940>
```


1e−6
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
Density
0
1
2
3
4
5
6
1e6
price


log(x)
0.8
0.6
0.4
0.2
0.0
Density
11
12
13
14
15
16
price

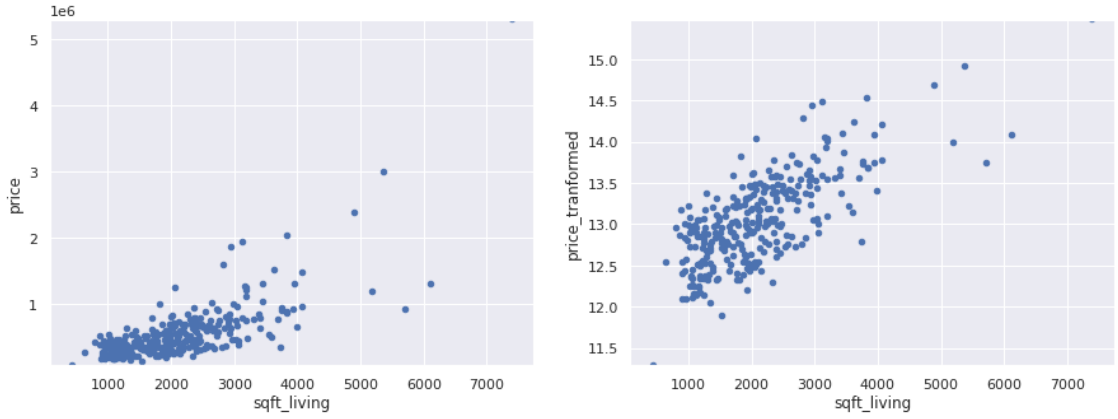
2 Homoschedasticity vs Eteroschedasticity

Look at the scatter plot of 'price' vs 'sqft_living' (generated by the next cell): it seems like the variance of the regression variable ('price') increase as the 'sqft_living' value increases, doesn't it? In other words, the higher the 'sqft_living' the higher the variance on the price. This is a classical example of Eteroschedasticity (variance of the regression variable depends on the independent variable).

It is not trivial to model such kind of variance. Can we transform data such that they closely resemble homoschedasiticty (variance independent on the independent variable)? In other words: can we find a transformation to be applied to the regressed variable such that its scatter plot vs an independent variable does not show a conic shape/diamond?

[61]:
```
fig, axes = plt.subplots(1,2, figsize=(15, 5))
plot_single_feature_vs_y('sqft_living', train_data, ax=axes[0])
# let's try with the log transform which showed to be useful to remove skewness
train_data['price_tranformed'] = np.log(train_data['price'])
plot_single_feature_vs_y('sqft_living', train_data, y='price_tranformed', ax=␣
 ↪axes[1])

# Have a look at the following scatter plots, using the log transform we fixed␣
 ↪(approximately) also the
# Eteroschedasticity issue!
```

```
*c* argument looks like a single numeric RGB or RGBA sequence, which should be
avoided as value-mapping will have precedence in case its length matches with
*x* & *y*.  Please use the *color* keyword-argument or provide a 2D array with a
single row if you intend to specify the same RGB or RGBA value for all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should be
avoided as value-mapping will have precedence in case its length matches with
*x* & *y*.  Please use the *color* keyword-argument or provide a 2D array with a
single row if you intend to specify the same RGB or RGBA value for all points.
```



[62]:
```
# Not compulsory: try to apply the log transform and see whether the regression␣
↪becomes easier.
```

Activating the 7-th cell of this notebook it's possible to apply such transformation. After this the Regression problem does become easier! We can deduce this by looking at the (7x7) scatter plot in cell #14 and also by looking at the values of the output tables #48 and #54 which shows better CODs and better gaps than the ones obtained without the use of the logaritmic transform (in particular the simple LS solution has the best improvement since it's also the most fragile to "bad data").

Note also how such trasformation leads to a dramatic decreasing of the Skewness and of the Kurtosis:

- From:
  - Skewness: 5.402253
  - Kurtosis: 47.043386
- To:
  - Skewness: 0.604337
  - Kurtosis: 1.392035