Universitat Autònoma de Barcelona

Applied Machine Learning: Analysis and Implementation

Research Innovation (Python Report)

Carlos Mougan *Jeremy J. Williams*

October 2018

Abstract

In a world of growing volumes and varieties of data, coding manually can become a problem to achieve relevant insight and knowledge. Using machine learning, we are able develop a solution by understanding the construction of algorithms and applications in the right scope of data sources. And based on the awareness of improvement of computer programs, we can now access data, analyze it and learn from it.

The focus will be to study supervised machine learning algorithms on the process of predicting a continuous variable through machine learning techniques.

An official dataset will be used with python libraries and original implementations will be developed using regression algorithms and models. More specifically, prediction of house prices based on single and multiple features will be conducted.

The regression analysis, study and implementation were explained using selected supervised machine learning algorithmic models. Results of applied machine learning discovered were also deliberated.

Contents

1 Introduction

SINCE the dawn of time, computers were known as a machine used to input data and process it to generate a relevant output. Today, they have made human life stress-free and over time have developed into the key solution to all daily computational tasks. Over the last decade, these computational tasks begin to establish a steadily increasing in the amounts of data that became available for all kinds of business needs. This kind of data growth reduces the amount of knowledge that we need to achieve today´s normal relevant objectives.

It is widely known, that what we lack in knowledge, we make up for in data. This is where we can establish a form of learning, not only for technical valuation and data recovery, but a combination of two capabilities of a computer system to make it perform learning task and make coherent results according to previously observed conditions and previous actions or responses, and not only act according to an immovable strategy. We call this “Machine learning”. A kind of Learning that is imperatively needed when the task to be executed by the computer system or machine is too complicate to be explained in code or within any unknown conditional data mapping.

Within the realm of machine learning, it is very imperative when the task to be executed by the computer system is too complicate a relevant type of machine learning algorithm must be selected.

2 Definitions & Concepts

To conduct our study of applied machine learning, it is important to declare all essential definitions and concepts.

Generally, there are three (3) types of machine Learning algorithms.

They are as followed:

- **Supervised Learning**
- **Unsupervised Learning**
- **Reinforcement Learning**

2.0.1 Supervised Learning

Supervised Learning is a type of machine learning algorithm the consist of a dependent variable which that is to be predicted from a given set of independent variables. Within this set of independent variables, a function must be generated to map inputs to the desired output. This in theory is called "training process". This process continues until the model achieves a successful level of accuracy on the training data. Illustrations of this type of learning algorithm are Regression, Decision Tree, Random Forest, KNN, Logistic Regression etc.

2.0.2 Unsupervised Learning

Unsupervised Learning does not have any dependent variable to predict. It is normally used for clustering analysis and segmentation. Illustrations of this type of learning algorithm are Apriori algorithm, K-means, Mean-Shift, DBSCAN, Expectation–Maximization, Hierarchical Clustering etc.

2.0.3 Reinforcement Learning

Reinforcement Learning is a type of machine learning algorithm that is trained to make specific decisions when a machine is exposed to an environment with continuous training using trial and error. This machine learns previous knowledge and attempts to capture the best possible outcome to create precise business resolutions. Illustrations of this type of learning algorithm are Markov Decision Process, Q-Learning, State-Action-Reward-State-Action, Deep Q Network, Deep Deterministic Policy Gradient etc.

3 Objectives

Our focus will be to study supervised machine learning algorithms on the process of predicting a continuous variable through machine learning techniques. More specifically, we want to predict house prices based on single and multiple features using regression analysis.

We will use the dataset from house sales in King County in Seattle, USA, with programming libraries and original implementations will be developed using regression algorithms and models.

In this report, we will first apply some data analysis techniques to summarize the main characteristics of the dataset. Then we will apply various machine learning algorithms, change some of the tuning parameters to see if we can make an improvement of the code.

4 Algorithmic Methods & Models

Regression analysis provides a "best-fit" mathematical equation for the value of variables. The equation maybe linear (a straight line) or curvilinear, but we will be concentration on the linear type.

The focus of this report is on just two types of variables, "y" and "x". They are called the dependent variable (y) and independent variable (x), since the typical purpose of this type of analysis is to estimate or predict what "y" will be a given value or values of "x".

We will use the following six (6) methods and/or models:

- **Simple Regression**
- **Multiple (Polynomial) Regression**
- **Ridge Regression and Gradient Descent**
- **Lasso and Coordinate Descent**
- **K-Nearest Neighbors**

We will now define our selected supervised machine learning algorithmic models.

4.1 Simple Regression

The simple regression model is a linear equation having a y-intercept and a slope with approximations of these population parameters based on sample data and determined by standard formulas.

The formula for the simple regression model is:

$$y_j = \beta_o + \beta_1 x_i + \varepsilon_i \tag{1}$$

where

y_j = a value of the dependent variable, y,

x_i = a value of the independent variable, x,

β_o = the y-intercept of the regression line,

β_1 = the slope of the regression line and

ε_i = random error, or residual

4.2 Multiple (Polynomial) Regression

The multiple regression model is an extension of the simple linear regression model. However, there are two or more independent variables instead of just one. It is sometimes called *"polynomial regression model"*. As before, estimates of the population parameters in the model are made on the basis of sample data.

The formula for the multiple regression model is:

$$y_j = \beta_o + \beta_1 x_{1i} + \beta_2 x_{2i} + ... + \beta_k x_{ki} + \varepsilon_i \tag{2}$$

where
y_j = a value of the dependent variable, y,
$x_{1i} + x_{2i} + ... + \beta_k x_{ki}$ = a value of the independent variable, x,
β_o = a constant,
$x_{1i} + x_{2i} + ... + x_{ki}$ = the slope of the regression line,
$\beta_1 + \beta_2 + ... + \beta_k$ = partial regression coefficients for the independent variables, $x_{1i} + x_{2i} + ... + x_{ki}$ and
ε_i = random error, or residual

4.3 Ridge Regression and Gradient Descent

4.3.1 Ridge Regression

Ridge regression is type of shrinkage methods used to construct simple models with excessive descriptive extrapolative power. Such simple models explain data with minimum number of parameters or predictor variables. Within the process of ridge regression, it is normally used to create these simple model when the data set has multicollinearity or correlation exist between predictor variables.
The formula used for the ridge regression is:

$$\sum_{i=1}^{n}\left(y_i - \beta_0 + \sum_{j=1}^{p}\beta_i x_{ij}\right)^2 + \lambda\sum_{j=1}^{p}\beta_j^2 = RSS + \lambda\sum_{j=1}^{p}\beta_j^2 \tag{3}$$

where
RSS = Residual Sum Of Squares, which is a statistical procedure used to quantify the volume of variance in a data set that is not described by a regression model and
$\lambda\sum_{j=1}^{p}\beta_j^2$ = the shrinkage penalty with $\lambda \geq 0$ as a tuning parameter.

4.3.2 Gradient Descent

Gradient descent is used to minimize some function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient. Gradient descent is to update the parameters of our model.
The gradient is the direction of increase and therefore the negative gradient is the direction of decrease and we're trying to minimum the number of parameters or predictor variables.
The move in the negative gradient direction is called the *'step size'*.

4.4 Lasso and Coordinate Descent

4.4.1 Lasso

When using the ridge regression, there exist some disadvantages located in the penalty (3). The idea of the shrinking process is to shrink all coefficients to zero however, not all will be set to exactly zero. This will only happen when $\lambda = \infty$. This leads to the need of the *Lasso*, which is an alternative version to ridge regression that is used to correct this disadvantage.

The formula used for the lasso is:

$$\sum_{i=1}^{n}\left(y_i - \beta_0 + \sum_{j=1}^{p}\beta_i x_{ij}\right)^2 + \lambda\sum_{j=1}^{p}|\beta_j| = RSS + \lambda\sum_{j=1}^{p}|\beta_j| \tag{4}$$

where

RSS = Residual Sum Of Squares, which is a statistical procedure used to quantify the volume of variance in a data set that is not described by a regression model and

$\lambda\sum_{j=1}^{p}|\beta_j|$ = the lasso penalty with $\lambda \geq 0$ as a tuning parameter.

In (3), the β_j^2 term in the ridge regression penalty changes to $|\beta_j|$ lasso penalty term to correct the disadvantage in the ridge regression.

4.4.2 Coordinate Descent

Coordinate descent is an optimization algorithm that used to completely minimizes all the coordinate directions to the best minimum of a given function.

The idea is to minimize a function by completely minimizing each of the individual dimension in a cyclic fashion and keeping all other values within the function dimensions fixed. It is sometime referred to as *'cyclic coordinate descent'*.

4.5 K-Nearest Neighbors

k-nearest neighbors or k-NN is an algorithm that classifies an input by using its k nearest neighbors.

k-NN is known as data classification and regression algorithm that tries to govern what collection of data points its in by looking at the data points surrounding it.

Since our focus is on the regression case, in k-NN regression, the output is the property amount for the item. This amount is the mean of the values of its k nearest neighbors.

The k-NN regression method is closely related to the k-NN classifier method and using the formula:

$$\widehat{f}(x_o) = \frac{1}{K} \sum_{x_i \in G_o} \tag{5}$$

where

x_o = a prediction point,

K = given value or amount,

G_o = a group of training responses and

x_i = a trained response

5 Data Overview

As discussed earlier, we use a dataset for the sales coming from an approved public records of home sales in the King County Area, Washington State, USA.

The data set comprises of 21,613 rows.

Each characterizes of a home sold from May 2014 through May 2015.

Below is a breakdown of the variables involved:

- **Id:** Unique ID for each home sold
- **Date:** Date of the home sale
- **Price:** Price of each home sold
- **Bedrooms:** Number of bedrooms
- **Bathrooms:** Number of bathrooms, where .5 accounts for a room with a toilet but no shower
- **Sqft-living:** Square footage of the apartments interior living space
- **Sqft-lot:** Square footage of the land space
- **Floors:** Number of floors
- **Waterfront:** A dummy variable for whether the apartment was overlooking the waterfront or not
- **View:** An index from 0 to 4 of how good the view of the property was
- **Condition:** An index from 1 to 5 on the condition of the apartment
- **Grade:** An index from 1 to 13, where 1-3 falls short of building construction and design, 7 has an average level of construction and design, and 11-13 have a high quality level of construction and design.
- **Sqft-above:** The square footage of the interior housing space that is above ground level
- **Sqft-basement:** The square footage of the interior housing space that is below ground level
- **Yr-built:** The year the house was initially built
- **Yr-renovated:** The year of the house's last renovation
- **Zipcode:** What zipcode area the house is in
- **Lat:** Latitude
- **Long:** Longitude
- **Sqft-living15:** The square footage of interior housing living space for the nearest 15 neighbors
- **Sqft-lot15:** The square footage of the land lots of the nearest 15 neighbors

6 Empirical Analysis & Results

We now presents a thought process of predicting a continuous variable through applied machine learning methods.

More specifically, we want to predict house prices based on single and multiple features using regression analysis.

6.1 Data Characteristics

We now explore the data set and study it's characteristics:

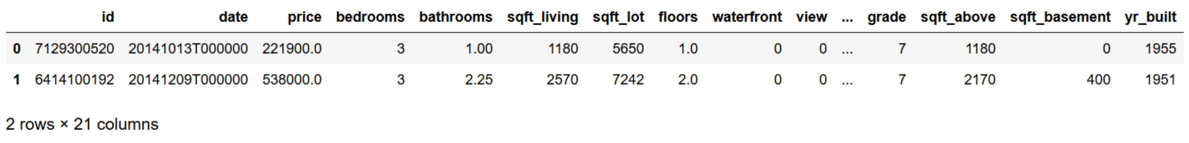
- Column Types: We change the data set columns to categorical data
- Missing Values: We decided to drop them.
- Lengths and shapes: To see what's the amount of data that we are currently managing.

We start our analysis by loading some of the libraries that we will use later.

```

import pandas as pd
import numpy as np
%matplotlib inline
from scipy import stats
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
import time
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge
from sklearn.preprocessing import MinMaxScaler
from sklearn.linear_model import Lasso
import seaborn as sns
from sklearn.preprocessing import PolynomialFeatures
from sklearn.svm import SVC
from sklearn.svm import LinearSVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVR
from sklearn.dummy import DummyRegressor
from sklearn import metrics
from sklearn.metrics import r2_score
import warnings
warnings.filterwarnings('ignore')

#We read the data set
df=pd.read_csv('kc_house_data.csv')
```

Lets convert features to categorical type

```

df['waterfront'] = df['waterfront'].astype('category',ordered=True)
df['view'] = df['view'].astype('category',ordered=True)
df['condition'] = df['condition'].astype('category',ordered=True)
df['grade'] = df['grade'].astype('category',ordered=False)
df['zipcode'] = df['zipcode'].astype(str)
df.head(2) # Show the first 2 lines
```



	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	...	grade	sqft_above	sqft_basement	yr_built
0	7129300520	20141013T000000	221900.0	3	1.00	1180	5650	1.0	0	0	...	7	1180	0	1955
1	6414100192	20141209T000000	538000.0	3	2.25	2570	7242	2.0	0	0	...	7	2170	400	1951

2 rows × 21 columns

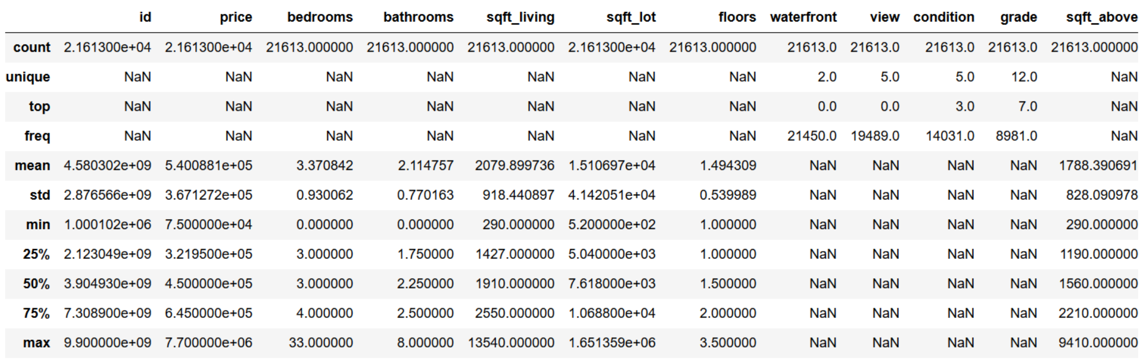
Figure 1: Converted features to categorical showing first 2 lines

We format the date

```

df.drop(columns=['date'],inplace=True)
```

Descriptive Analysis

```

df.describe(include='all')
```



	id	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	condition	grade	sqft_above
count	2.161300e+04	2.161300e+04	21613.000000	21613.000000	21613.000000	2.161300e+04	21613.000000	21613.0	21613.0	21613.0	21613.0	21613.000000
unique	NaN	NaN	NaN	NaN	NaN	NaN	NaN	2.0	5.0	5.0	12.0	NaN
top	NaN	NaN	NaN	NaN	NaN	NaN	NaN	0.0	0.0	3.0	7.0	NaN
freq	NaN	NaN	NaN	NaN	NaN	NaN	NaN	21450.0	19489.0	14031.0	8981.0	NaN
mean	4.580302e+09	5.400881e+05	3.370842	2.114757	2079.899736	1.510697e+04	1.494309	NaN	NaN	NaN	NaN	1788.390691
std	2.876566e+09	3.671272e+05	0.930062	0.770163	918.440897	4.142051e+04	0.539989	NaN	NaN	NaN	NaN	828.090978
min	1.000102e+06	7.500000e+04	0.000000	0.000000	290.000000	5.200000e+02	1.000000	NaN	NaN	NaN	NaN	290.000000
25%	2.123049e+09	3.219500e+05	3.000000	1.750000	1427.000000	5.040000e+03	1.000000	NaN	NaN	NaN	NaN	1190.000000
50%	3.904930e+09	4.500000e+05	3.000000	2.250000	1910.000000	7.618000e+03	1.500000	NaN	NaN	NaN	NaN	1560.000000
75%	7.308900e+09	6.450000e+05	4.000000	2.500000	2550.000000	1.068800e+04	2.000000	NaN	NaN	NaN	NaN	2210.000000
max	9.900000e+09	7.700000e+06	33.000000	8.000000	13540.000000	1.651359e+06	3.500000	NaN	NaN	NaN	NaN	9410.000000

Figure 2: Description of the dataset

With a simple correlation, we can see that there are some variables with a higher correlation like the $sqft_living$ and some others like $zipcode$ or id or $longitude(earth)$ have lower correlation.

Data Map Visualization

```

sns.heatmap(df.corr(),center=True,square=True)
```

We now look for the variables with the strongest correlation with price

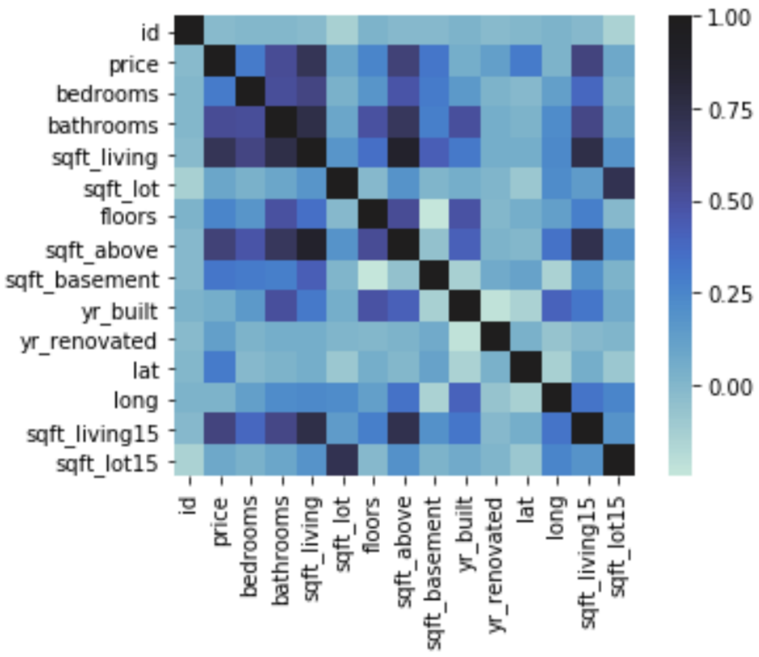


Figure 3: Heat Map of the correlations of some of the variables of the data set

```python

correlation=df.corr()
correlation.sort_values(by='price',inplace=True)
price_correlation=pd.DataFrame(correlation['price'])
```

We now study a boxplot of the dataset.

```python

fig, ax = plt.subplots(figsize=(12,4))
sns.boxplot(x = 'price', data = df, orient = 'h', width = 0.8,
                  fliersize = 3, showmeans=True, ax = ax)
plt.show()
```

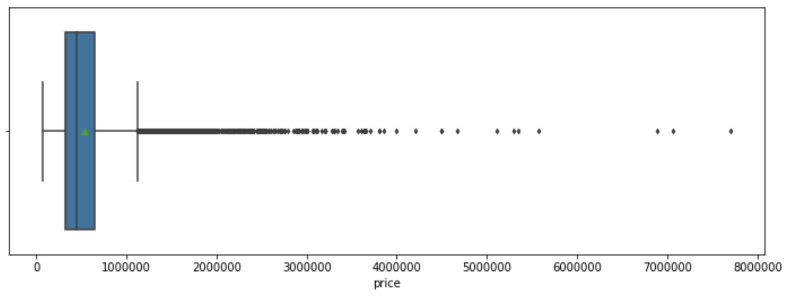


Figure 4: Box Plot of price in the data set.

There seems to be a lot of outliers at the top of the distribution, with a few houses above the 5000000 value.

If we ignore outliers, the range is illustrated by the distance between the opposite ends of the whiskers (1.5 IQR) - about 1000000 here.

Also, we can see that the right whisker is slightly longer than the left whisker and that the median line is gravitating towards the left of the box. The distribution is therefore slightly skewed to the right.

In figure 5, we can see the bi-variate relation of price in the data set.

```

sns.jointplot(x="sqft_living", y="price", data=df, kind = 'reg', size = 7)
plt.show()
```

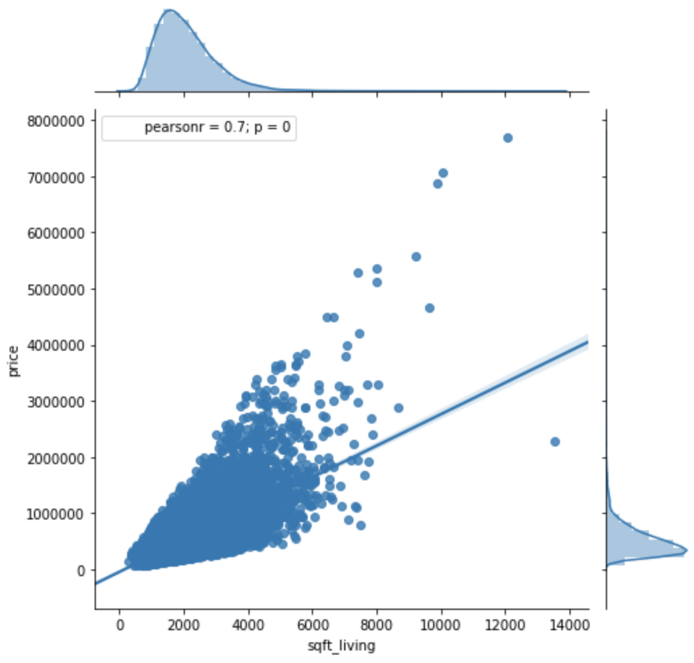


Figure 5: A joint plot of the bivariate distribution of our higher related variable

6.2 Regression Models & Preparation

We now will apply multiple regression models.

We will study the model complexity and try to make selections of the best predictive model using different model tuning variables, a validation set or cross-validation techniques.

We first start to split the model in two different sets: training and testing.

```
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test =
train_test_split(df_2, df_y, random_state=0)
```

We select Random State instance, random state is the random number generator; If random_state=None, the random number generator is the Random State instance used by np.random. But we select random_state=0 to be able to replicate the training of our models.

The test size is 0.25 by default and we let it be like that.

6.3 Models

Now we start to train the different skclearn models with the training and scoring them with both splits trainning and testing.

6.3.1 Linear Models for Regression

In this section, we are going to use Liner Regression, Ridge's Regression and Lasso Regression, we study the difference when we applying the cost function.

The relevant formulas are as followed:

$$J_{(\theta)} = \frac{1}{n}\sum_{i=1}^{n}(y_i - y_i^{\wedge})^2 \tag{6}$$

Linear cost function

$$J_{(\theta)} = \frac{1}{n}\sum_{i=1}^{n}(y_i - y_i^{\wedge})^2 + \lambda\sum_{i=1}^{n} w_i^2 \tag{7}$$

Rigde Cost Function

$$J_{(\theta)} = \frac{1}{n}\sum_{i=1}^{n}(y_i - y_i^{\wedge})^2 + \lambda\sum_{i=1}^{n} |w_i| \tag{8}$$

Lasso Cost Function

We now study the regression model with one feature.

```

linreg = LinearRegression().fit(X_train, y_train)

print('linear␣model␣coeff␣(w):␣{}'
     .format(linreg.coef_))
print('linear␣model␣intercept␣(b):␣{:.3f}'
     .format(linreg.intercept_))
print('R-squared␣score␣(training):␣{:.3f}'
```

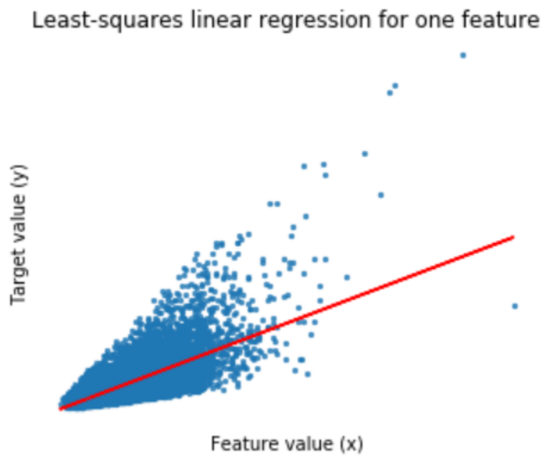


Figure 6: Graph of the linear regression for one feature

```
    .format(linreg.score(X_train, y_train)))
print('R-squared␣score␣(test):␣{:.3f}'
    .format(linreg.score(X_test, y_test)))

#Linear regression one feature
linreg_one_feature = LinearRegression().fit(X_train_one_feature,
    ↪ y_train_one_feature)

print('R-squared␣score␣(training):␣{:.3f}'
     .format(linreg_one_feature.score(X_train_one_feature, y_train_one_feature)))
print('R-squared␣score␣(test):␣{:.3f}'
     .format(linreg_one_feature.score(X_test_one_feature, y_test_one_feature)))

plt.figure(figsize=(5,4))
plt.scatter(df_3, df_y, marker= 'o', s=5, alpha=0.8)
df_3=pd.DataFrame(df_3)
b=linreg_one_feature.coef_ *df_3 + linreg_one_feature.intercept_
plt.plot(df_3,b, 'r-')
plt.title('Least-squares␣linear␣regression␣for␣one␣feature')
plt.xlabel('Feature␣value␣(x)')
plt.ylabel('Target␣value␣(y)')
#plt.ylim((0,600))
plt.tick_params(top=False, bottom=False, left=False, right=False,
    ↪ labelleft=False, labelbottom=False)
plt.gca().spines['top'].set_visible(False)
plt.gca().spines['right'].set_visible(False)
plt.gca().spines['left'].set_visible(False)
plt.gca().spines['bottom'].set_visible(False)
plt.show()
```

According to Table 1 & Figure 6, we can see that the best fit for our model is normal lineal regression with all features.

After applying the algorithm to predict instead of classify we obtain that R^2 is 0.537 (all

Model	R^2 Test	R^2 Test
LinReg - One Feature	0.490	0.483
LinReg - All Features	0.703	0.69
Ridge Regr	0.702	0.691
Ridge Regr Scaled	0.696	0.681
Lasso Reg	0.703	0.69

Table 1: The results obtained with this cost functions applied to this models are seen in the data table

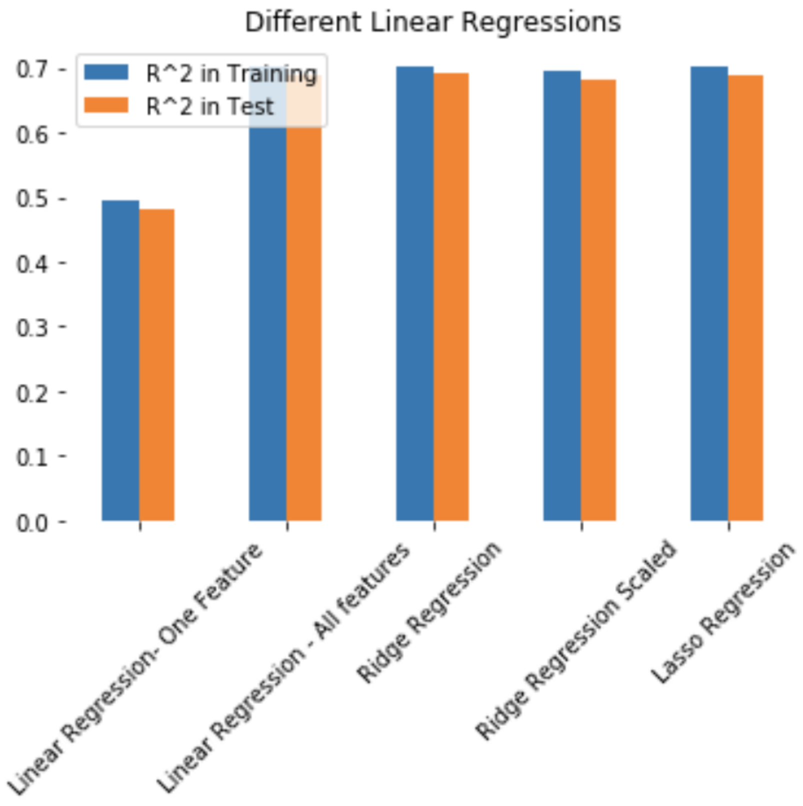


Figure 7: A bar representation of the scores of the different linear regression models

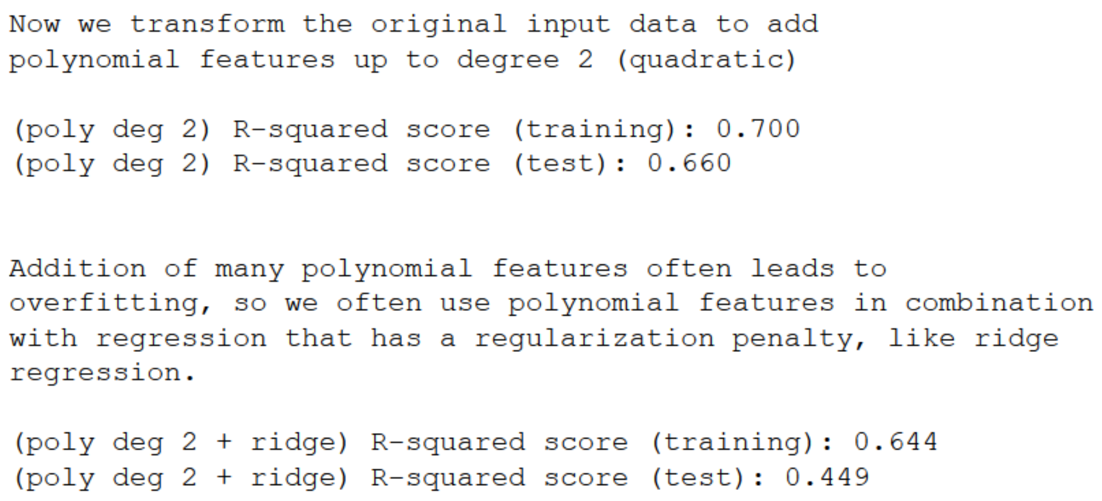
features) and 0.462 for just one feature ($sqft_living$)

6.3.2 Multiple (Polynomial) Regression

In this section, we now study the data set to confirm if it fits better in a nonlinear function.

Degree = 2

```

print('\nNow␣we␣transform␣the␣original␣input␣data␣to␣add\n\
polynomial␣features␣up␣to␣degree␣2␣(quadratic)\n')
poly = PolynomialFeatures(degree=2)
X_F1_poly = poly.fit_transform(df_2)
X_train, X_test, y_train, y_test = train_test_split(X_F1_poly, df_y,
                                                    random_state = 0)
linreg = LinearRegression().fit(X_train, y_train)


print('(poly␣deg␣2)␣R-squared␣score␣(training):␣{:.3f}'
     .format(linreg.score(X_train, y_train)))
print('(poly␣deg␣2)␣R-squared␣score␣(test):␣{:.3f}\n'
     .format(linreg.score(X_test, y_test)))

print('\nAddition␣of␣many␣polynomial␣features␣often␣leads␣to\n\
overfitting,␣so␣we␣often␣use␣polynomial␣features␣in␣combination\n\
with␣regression␣that␣has␣a␣regularization␣penalty,␣like␣ridge\n\
regression.\n')

X_train, X_test, y_train, y_test = train_test_split(X_F1_poly, df_y,
                                                    random_state = 0)
linreg = Ridge().fit(X_train, y_train)


print('(poly␣deg␣2␣+␣ridge)␣R-squared␣score␣(training):␣{:.3f}'
     .format(linreg.score(X_train, y_train)))
print('(poly␣deg␣2␣+␣ridge)␣R-squared␣score␣(test):␣{:.3f}'
     .format(linreg.score(X_test, y_test)))
```



```
Now we transform the original input data to add
polynomial features up to degree 2 (quadratic)

(poly deg 2) R-squared score (training): 0.700
(poly deg 2) R-squared score (test): 0.660


Addition of many polynomial features often leads to
overfitting, so we often use polynomial features in combination
with regression that has a regularization penalty, like ridge
regression.

(poly deg 2 + ridge) R-squared score (training): 0.644
(poly deg 2 + ridge) R-squared score (test): 0.449
```

Figure 8: Results with degree = 2

Degree = 3

```

print('\nNow we transform the original input data to add\n\
polynomial features up to degree 3 (cubic)\n')
poly = PolynomialFeatures(degree=3)
X_F1_poly = poly.fit_transform(df_2)
X_train, X_test, y_train, y_test = train_test_split(X_F1_poly, df_y,
                                                   random_state = 0)
linreg = LinearRegression().fit(X_train, y_train)


print('(poly deg 3) R-squared score (training): {:.3f}'
     .format(linreg.score(X_train, y_train)))
print('(poly deg 3) R-squared score (test): {:.3f}\n'
     .format(linreg.score(X_test, y_test)))

print('\nAddition of many polynomial features often leads to\n\
overfitting, so we often use polynomial features in combination\n\
with regression that has a regularization penalty, like ridge\n\
regression.\n')

X_train, X_test, y_train, y_test = train_test_split(X_F1_poly, df_y,
                                                   random_state = 0)
linreg = Ridge().fit(X_train, y_train)


print('(poly deg 3 + ridge) R-squared score (training): {:.3f}'
     .format(linreg.score(X_train, y_train)))
print('(poly deg 3 + ridge) R-squared score (test): {:.3f}'
     .format(linreg.score(X_test, y_test)))
```

Model	R^2 Test	R^2 Test
LinReg	0.703	0.69
Pol Reg =2	0.700	0.660
Pol Reg=3	0.280	0.240
Pol Reg=2 + Ridge	0.644	0.449

Table 2: The results obtained with this cost functions applied to this models are seen in the data table.

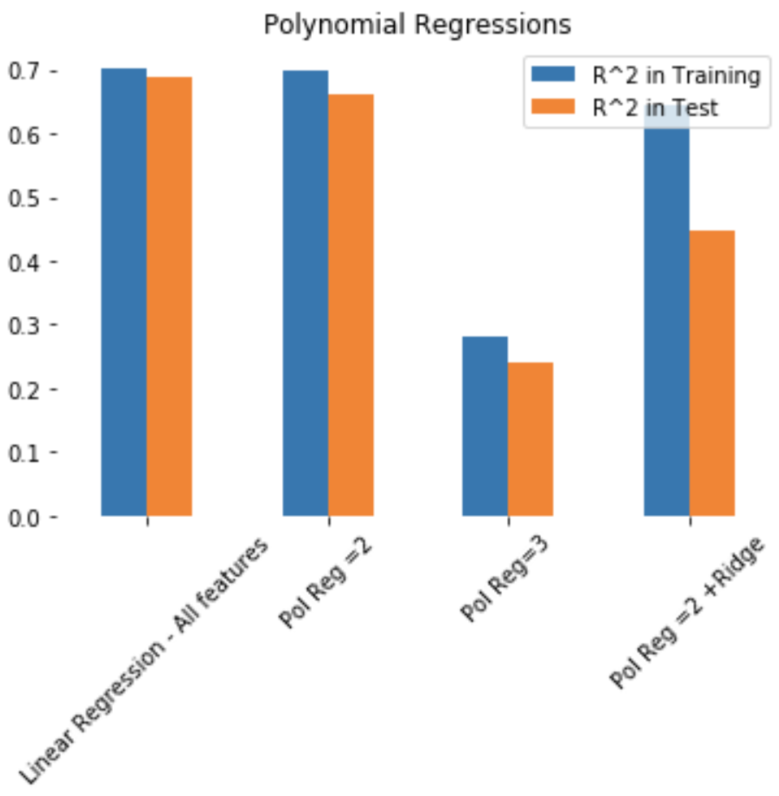


Figure 10: A bar representation of the scores of the different linear regression models

6.4 Ridge Regression and Lasso Alpha Parameter Tuning

Now, we will try to score the best options with different alpha parameters for Ridge and lasso.

```
scores=pd.read_excel('alpha.xlsx')
scores.iloc[0][1]=0.005
scores.iloc[2][0]=0.450
scores.iloc[3][0]=0.41
scores.iloc[7][1]=0.681

sns.set(style="white")
ridge=pd.DataFrame([scores.iloc[8],scores.iloc[9],scores.iloc[10],
scores.iloc[11],scores.iloc[12],scores.iloc[13],
scores.iloc[14],scores.iloc[15]])
g=sns.lineplot(hue="coherence", style="choice",data=ridge)
g.set_xticklabels(g.get_xticklabels(),rotation=90)
g.set_title('Ridge␣Regression␣for␣different␣Alpha')
sns.despine(fig=None, ax=None,
top=True, right=True, left=True,
bottom=True, offset=None, trim=False)
```

6.4.1 Ridge Regression

```
ridge=pd.DataFrame([scores.iloc[8],scores.iloc[9],scores.iloc[10],
                    scores.iloc[11],scores.iloc[12],scores.iloc[13],
                    scores.iloc[14],scores.iloc[15]])

g=ridge.plot()
g.set_title('Ridge␣Regression␣for␣different␣Alpha')
g.set_xticklabels(ridge.index,rotation=45)
g.spines["top"].set_visible(False)
g.spines["bottom"].set_visible(False)
g.spines["left"].set_visible(False)
g.spines["right"].set_visible(False)
```

Model	R^2 Test	R^2 Test
Ridge Alpha=0.5	0.7	0.69
Ridge Alpha=1	0.7	0.69
Ridge Alpha=10	0.7	0.685
Ridge Alpha=20	0.7	0.681
Ridge Alpha=50	0.69	0.67
Ridge Alpha=100	0.67	0.652
Ridge Alpha=1000	0.44	0.432

Table 3: Results obtained for Ridge Alpha data table

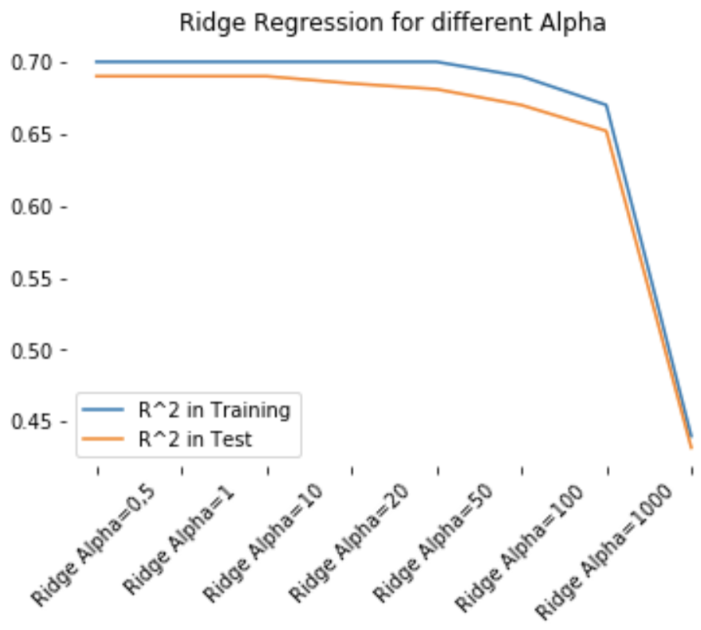


Figure 11: A simple plot of the score of Ridge's regression with different alpha parameters

6.4.2 Lasso

```
lasso=pd.DataFrame([scores.iloc[17],
scores.iloc[18],
scores.iloc[19],scores.iloc[20],
scores.iloc[21],scores.iloc[22],
scores.iloc[23]])

g=lasso.plot()
g.set_title('Lasso␣Regression␣for␣different␣Alpha')
g.set_xticklabels(lasso.index,rotation=45)
g.spines["top"].set_visible(False)
g.spines["bottom"].set_visible(False)
g.spines["left"].set_visible(False)
g.spines["right"].set_visible(False)
```

Model	R^2 Test	R^2 Test
Lasso Alpha=10	0.7	0.69
Lasso Alpha=100	0.7	0.69
Lasso Alpha=500	0.69	0.68
Lasso Alpha=1000	0.69	0.68
Lasso Alpha=5000	0.61	0.61
Lasso Alpha=10000	0.48	0.48

Table 4: Results obtained for Lasso Alpha data table

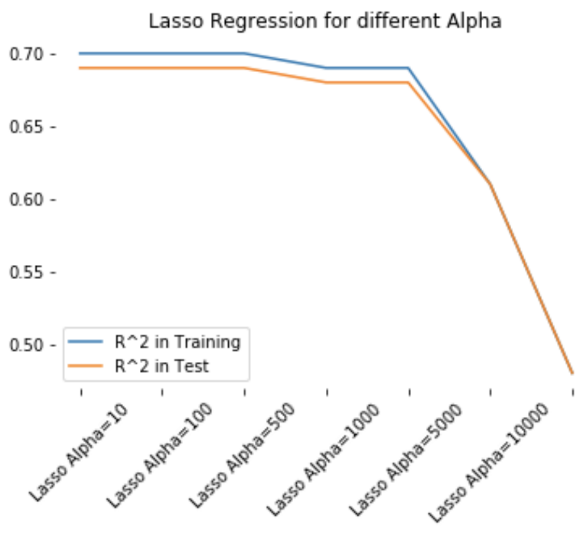


Figure 12: A simple plot of the score of Lasso's regression with different alpha parameters

6.4.3 k-Nearest Neighbors

We now start loading the model from the sklearn library to study K Nearest Neighbors of our data set.

```
from sklearn.neighbors import KNeighborsClassifier
```

We have trained 4 different models (classifier one feature, classifier all feature, regression one feature, regression all features)

	R^2 Test	R^2 Test		
Class - One Feature	0.04	0.005		
Class - All Features	0.18	0.01		
Regr - One Feature	0.45	0.458		
Regr - All features	0.410	0.357		

Table 5: Results obtained for k-Nearest Neighbors

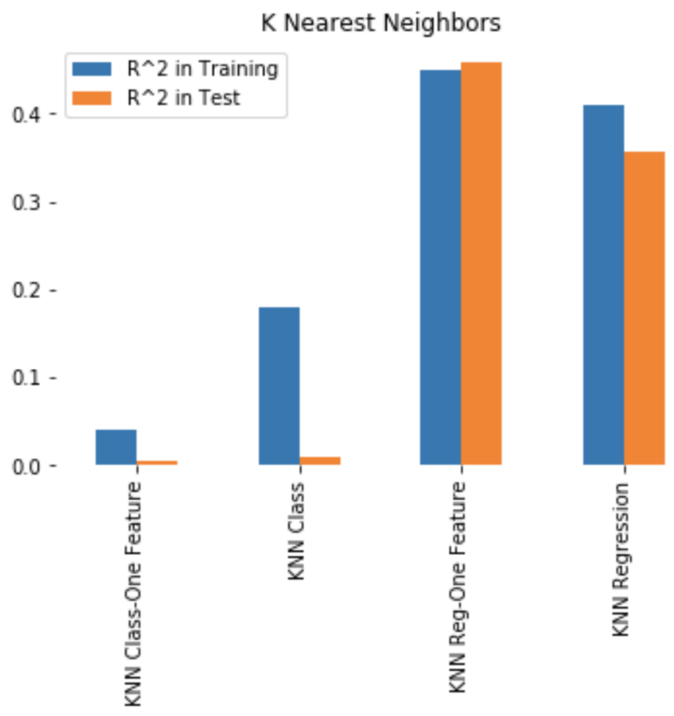


Figure 13: A bar representation of the scores of the different k-NN models

7 Discussion and Conclusions

7.1 Linear and Multiple (Polynomial) Regression

We have applied the algorithm to predict instead of classify, we have obtain $R^2 = 0.537$ (all features) and 0.462 for just one feature (sqft-living).

We have applying the polynomial regressions of degree 2 and 3, we can see that the performance of the model is better with degree = 2 than degree = 3.

Applying the k-NN class, k-NN regression, k-NN regression one feature, linear, linear ridge, linear lasso, linear one feature, linear ridge one feature, square regression, square with ridge, cubic regression and cubic regression with ridge, we can see that the best model is linear regression with all features.

Using regression implementation, the (predicted price) estimated price for a house with 3500 square-feet is 939739.86. Also, with the inverse regression estimate, the estimated square-feet for a house worth 5500000.00 is 19673 square-feet. This provides satisfactory results.

7.2 Ridge Regression

We have executed the following 3 different models:

- **Ridge Linear Model:** all features $R^2 = 0.537$
- **Ridge Linear Model:** all features scaled $R^2 = -2.6$
- **Ridge Linear Model:** all features with different alphas, we obtain the best model with an alpha parameterization between 0.5-1 $R^2 = 0.540$

The result shows that the Ridge regression doesn't work much better than linear regression for this case.

In the implementation of the ridge regression using the gradient, we have predicted the house price for the **1st** house in the test set using the no regularization and high regularization models.

- **The Predicted 1st house (No Regularization):** 387465.47605823533
- **The Predicted 1st house (High Regularization):** 270453.53032194055
- **Actual house price:** 310000.0

We have also predicted the house price for the **11th** house in the test set using the no regularization and high regularization models.

- **The Predicted 11th house (No Regularization):** 478551.3475892041
- **The Predicted 11th house (High Regularization):** 315939.5228610413
- **The Actual house price:** 349000.0

As we study figure 14 & 15, it shows a very similar learned weights for both no regularization and high regularization using the RSS on the test data.

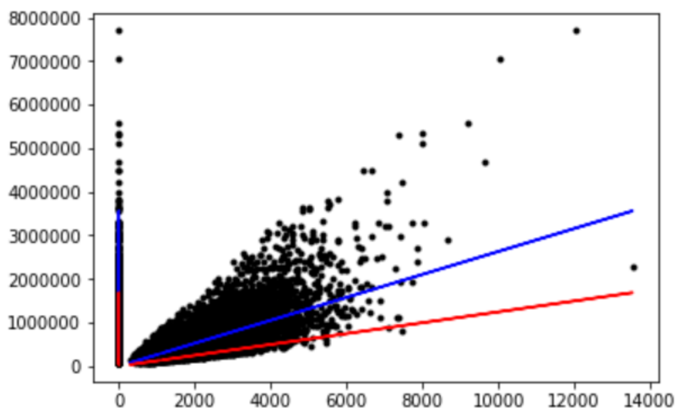


Figure 14: Graphical representation of the ridge regression using gradient descent implementation with 1 feature where blue is for no regularization and red is for high regularization

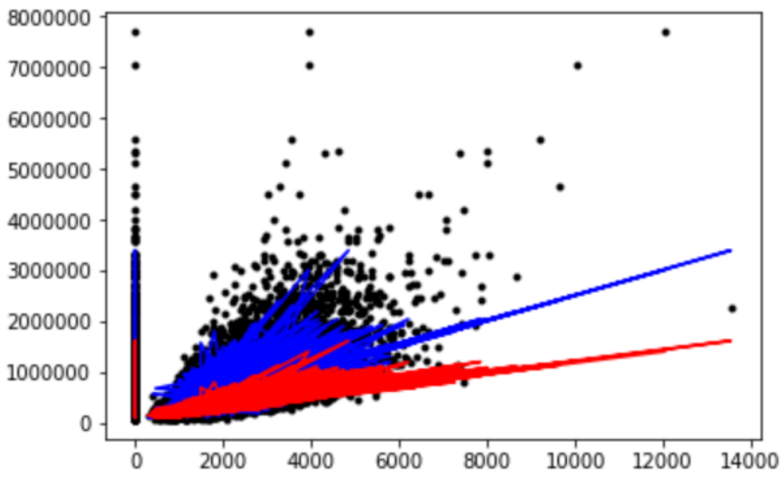


Figure 15: Graphical representation of the ridge regression using gradient descent implementation with 2 features where blue is for no regularization and red is for high regularization

7.3 Lasso and k-Nearest Neighbor

We have obtained results for Lasso (regression) of $R^2 = 0.69$. And the best alpha parameter is between $1 - 500$. This show that the low alphas the results are better. For instance, when the alpha value is 0, Lasso (regression) produces the same coefficients as a linear regression. When alpha is very large, all coefficients are zero.

So, the Lasso (regression) is not working better than linear regression.

We have used the k-NN algorithm to predict instead of classify. This process obtained $R^2 = 0.36$ (all features) and 0.458 for just one feature (sqft-living). It works better with just one feature. We have calculated almost 50 percent of predicting prices with just one variable in k-NN regression.

And after studying the graphical representation, we can see that the best k-NN neighbors parameter is $k = 3$ with we obtain a $R^2 = 0.536$.

7.4 Conclusion

Within this report, we have conducted an empirical study to give on overview of regression methods using the given data set.

We have studied the Simple Regression, Multiple (Polynomial) Regression, Ridge Regression, Gradient Descent, Lasso, Coordinate Descent and k-Nearest Neighbors algorithms and models.

There is many more algorithms and/or models to used i.e. Elastic Net, Kernel Regression, Bayesian Regression, Support Vector Machine (using classification and/or regression analysis), Artificial Neural Network, Convolutional Neural Network, Random Forest, Decision Tree etc.

Based on the study of the python implementations developed, it is recommended to continue our experiments for auxiliary forthcoming progressive exploration and/or investigations.

However, at this moment, we are satisfied with our overall results to have achieved $R^2 = 0.700$ from our test data set without applying any extremely difficult and/or complex Artificial Intelligence algorithm.

References

[1] T. Hastie, R. Tibshirani and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition*. Springer-Verlag New York, 2009.

[2] V. K. Ayyadevara. *Pro Machine Learning Algorithms: A Hands-On Approach to Implementing Algorithms in Python and R*. Apress, 1st edition, 2018.

[3] G. James et al. *An Introduction to Statistical Learning : with Applications in R*. New York :Springer, 2013.

[4] J. Williams. Machine Learning: Regression - Implementation License QMG2VNAPF56C. 9/16/2018. URL: `https://www.coursera.org/account/accomplishments/verify/QMG2VNAPF56C`.

[5] J. Williams. Machine Learning Specialization: Implementation License 2BAL5JR9R5DS. 9/23/2018. URL: `https://www.coursera.org/account/accomplishments/specialization/2BAL5JR9R5DS`.

[6] M. Swamynathan. *Mastering Machine Learning with Python in Six Steps: A Practical Implementation Guide to Predictive Data Analytics Using Python*. Apress, 1st edition, 2017.

[7] P.-N. Tan, M. Steinbach and V. Kumar. *Introduction to Data Mining*. New York: Addison-Vesley, 1st edition, 2004.

[8] T. Mitchell. *Machine Learning*. New York: Mc Graw-Hill, 1997.

[9] R. Garreta and G. Moncecchi. *Learning scikit-learn: Machine Learning in Python*. Packt Publishing Limited, 2013.

[10] P. K. Janert. *Data Analysis with Open Source Tools: A Hands-On Guide for Programmers and Data Scientists*. O'Reilly Media, 1st edition, 2010.

Appendices

A: Regression Algorithms & Model Implementations

In this section, a presentation of an explanation of the *python* code used for *Simple Regression*, *Multiple (Polynomial) Regression*, *Ridge Regression with Gradient Descent* and *Lasso with Coordinate Descent*. Each code implementation are structured in different sections (called *blocks*) that was developed [**4**;**5**].

A1 Simple Regression Code

A1.1 Block 1

Package declaration and initialization, where the packages to be used are loaded to python interpreter.

```
import pandas as pd
import numpy as np
```

A1.2 Block 2

Use the split data - training, testing and data description of sales

```
train_data = pd.read_csv('kc_house_train_data.csv')
test_data = pd.read_csv('kc_house_test_data.csv')
sales.describe()
sales.dtypes
```

A1.3 Block 3

Summary functions - Compute Sum and Arithmetic Average (mean)

```
# Let's compute the mean of the House Prices in King County in 2 different ways.
kcprices = sales['price'] # extract the price column of the sales

# recall that the arithmetic average (the mean) is the sum of the prices divided
    ↪ by the total number of houses:
sum_kcprices = kcprices.sum()
num_kchouses = kcprices.size # when prices is an Series .size returns its length
avg_kcprice_1 = sum_kcprices/num_kchouses
avg_kcprice_2 = kcprices.mean() # if you just want the average, the .mean()
    ↪ function
print "average␣price␣via␣method␣1:␣" + str(avg_kcprice_1)
print "average␣price␣via␣method␣2:␣" + str(avg_kcprice_2)

```

```python
# if we want to multiply every price by 0.5 it's a simple as:
half_kcprices = 0.5*kcprices

# Let's compute the sum of squares of price by multiply with (star*)

kcprices_squared = kcprices*kcprices
sum_kcprices_squared = kcprices_squared.sum() # price_squared is the squares and we want to add them up.
print "the sum of price squared is: " + str(sum_kcprices_squared)
```

A1.4 Block 4

Simple linear regression function

We can compute the slope and intercept for a simple linear regression on observations stored: input-feature, output.

```python

# Now computing the simple linear regression slope and intercept:

def simple_linear_regression(input_feature, output):
    # compute the sum of input_feature and output
    N = input_feature.size
    sum_x = input_feature.sum()
    sum_y = output.sum()
    # compute the product of the output and the input_feature and its sum
    sum_xy = np.dot(input_feature,output).sum()
    # compute the squared value of the input_feature and its sum
    sum_x2 = np.square(input_feature).sum()
    # use the formula for the slope
    slope = (sum_xy - (sum_y*sum_x)/float(N))/(sum_x2 - (sum_x*sum_x)/float(N))
    # use the formula for the intercept
    intercept = output.mean() - slope * input_feature.mean()
    return (intercept, slope)

test_feature = pd.Series(range(5))
test_output = pd.Series(1 + 1*test_feature)
(test_intercept, test_slope) = simple_linear_regression(test_feature, test_output)
print "Intercept: " + str(test_intercept)
print "Slope: " + str(test_slope)

sqft_intercept, sqft_slope = simple_linear_regression(train_data['sqft_living'],
                       train_data['price'])

print "Intercept: " + str(sqft_intercept)
print "Slope: " + str(sqft_slope)
```

A1.5 Block 5

Predicting Values

```

# Now that we have the model parameters: intercept & slope we can make
    ↪ predictions.

def get_regression_predictions(input_feature, intercept, slope):
    # calculate the predicted values:
    predicted_values = input_feature * slope + intercept
    return predicted_values

# We now predicted price for a house with 3500 sqft.

my_house_sqft = 3500
estimated_price = get_regression_predictions(my_house_sqft,
                              sqft_intercept, sqft_slope)
print "The␣estimated␣price␣for␣a␣house␣with␣%d␣squarefeet␣is␣$%.2f" %
    ↪ (my_house_sqft, estimated_price)
```

A1.6 Block 6

Residual Sum of Squares (RSS)

```

# We compute the RSS of a simple linear regression model given the input_feature,
    ↪ output, intercept and slope:

def get_residual_sum_of_squares(input_feature, output,
                              intercept, slope):
    # First get the predictions
    predictions = get_regression_predictions(input_feature,
                                          intercept,slope)
    # then compute the residuals (since we are squaring it doesn't matter which
        ↪ order you subtract)
    residuals = predictions - output
    # square the residuals and add them up
    RSS = np.square(residuals).sum()
    return(RSS)

#We will use the get_residual_sum_of_squares function using #our test model where
    ↪ the data lie exactly on a line.

print get_residual_sum_of_squares(test_feature, test_output,
                                test_intercept, test_slope) # should be 0.0
#We use the simple linear regression using square feet to predict prices on
    ↪ TRAINING data.

rss_kcprices_on_sqft = get_residual_sum_of_squares(train_data['sqft_living'],
                         train_data['price'],
                         sqft_intercept, sqft_slope)
print 'The␣RSS␣of␣predicting␣Prices␣based␣on␣Square␣Feet␣is␣:␣' +
    ↪ str(rss_kcprices_on_sqft)
```

A1.7 Block 7

Predict the sqft. given the price (using the inverse regression estimate)

```
def inverse_regression_predictions(output, intercept, slope):
    # solve output = intercept + slope*input_feature for input_feature.
    # Use this equation to compute the inverse predictions:
    estimated_feature = (output - intercept)/slope
    return estimated_feature

# Now we estimated square-feet for a house costing $5,500,000?

my_house_price = 5500000
estimated_sqft = inverse_regression_predictions(my_house_price,
                             sqft_intercept, sqft_slope)
print "The␣estimated␣squarefeet␣for␣a␣house␣worth␣$%.2f␣is␣%d" % (my_house_price,
    ↪ estimated_sqft)
```

A1.8 Block 8

2nd Model: Estimate prices from bedrooms (using the training data)

```

# Estimate the slope and intercept for predicting 'price' based on 'bedrooms'
bedrooms_intercept, bedrooms_slope =
    ↪ simple_linear_regression(train_data['bedrooms'],
                       train_data['price'])

print "Intercept:␣" + str(sqft_intercept)
print "Slope:␣" + str(sqft_slope)
```

A1.9 Block 9

LR Algorithm Test for both models (sqft & bedrooms)

```
# We will now use the RSS from predicting prices using bedrooms and from
    ↪ predicting prices using square feet.

# RSS when using bedrooms on TEST data:
RSS_prices_on_sqft = get_residual_sum_of_squares(test_data['sqft_living'],
                          test_data['price'],
                          sqft_intercept, sqft_slope)
print 'Predicted␣Prices␣based␣on␣Square␣Feet:␣' + str(RSS_prices_on_sqft)

# RSS when using squarefeet on TEST data:
RSS_prices_on_bedrooms = get_residual_sum_of_squares(test_data['bedrooms'],
                          test_data['price'],
                          bedrooms_intercept,
                          bedrooms_slope)
print 'Predicted␣Prices␣based␣on␣Bedrooms:␣' + str(RSS_prices_on_bedrooms)
```

A2 Multiple (Polynomial) Regression Code

A2.1 Block 1

Package declaration and initialization, where the packages to be used are loaded to python interpreter.

```
import pandas as pd
import numpy as np
```

A2.2 Block 2

Use the split data - training, testing and data description of sales

```
train_data = pd.read_csv('kc_house_train_data.csv')
test_data = pd.read_csv('kc_house_test_data.csv')
```

A2.3 Block 3

Learning: Multiple regression model

```

# We prepare a multiple regression model to predict 'price' based on the
    ↪ following features: ML_features = ['sqft_living', 'bedrooms', 'bathrooms']
    ↪ on training data:

from sklearn.linear_model import LinearRegression
ML_features = ['sqft_living', 'bedrooms', 'bathrooms']
ML_model = LinearRegression()
ML_model.fit(train_data[ML_features],train_data['price'])

# Extract the regression weights (coefficients):
intercept = ML_model.intercept_
ML_coef = ML_model.coef_

pd.DataFrame({'Name':['intercept']+ML_features,'Value':[intercept]+list(ML_coef)})
```

A2.4 Block 4

Our Predictions

```

# We can use the .predict() function to find the predicted values for data we
    ↪ pass.

ML_predictions = ML_model.predict(train_data[ML_features])
print ML_predictions[0]

```

```
# Compute RSS
# Calculate RSS given the model, data, and the outcome.

def get_residual_sum_of_squares(model, data, outcome,features=['sqft_living',
                 'bedrooms', 'bathrooms']):
    # First get the predictions
    predictions = model.predict(data[features])
    # Then compute the residuals/errors
    errors = predictions - outcome
    # Then square and add them up
    RSS = np.square(errors).sum()
    return(RSS)

# Computing the RSS on TEST data for the example model:
rss_ML_train = get_residual_sum_of_squares(ML_model, test_data,
    ↪ test_data['price'])
print 'rss_ML_train:␣' + str(rss_ML_train)
# should be 2.7376153833e+14
```

A2.5 Block 5

Create some new features using log

```
# We use logarithm function to create a new feature.

from math import log

# We now create 4 new features for both TEST and TRAIN data:

# bedrooms_squared = bedrooms*bedrooms
# bed_bath_rooms = bedrooms*bathrooms
# log_sqft_living = log(sqft_living)
# lat_plus_long = lat + long

train_data['bedrooms_squared'] = train_data['bedrooms'].apply(lambda x: x**2)
test_data['bedrooms_squared'] = test_data['bedrooms'].apply(lambda x: x**2)

# create the remaining 3 features in both TEST and TRAIN data
train_data['bed_bath_rooms'] = train_data['bedrooms'] * train_data['bathrooms']
train_data['log_sqft_living'] = train_data['sqft_living'].apply(lambda x:log(x))
train_data['lat_plus_long'] = train_data['lat'] + train_data['long']
test_data['bed_bath_rooms'] = test_data['bedrooms'] * test_data['bathrooms']
test_data['log_sqft_living'] = test_data['sqft_living'].apply(lambda x:log(x))
test_data['lat_plus_long'] = test_data['lat'] + train_data['long']

# We now calculate the mean value of the new 4 features on test data.

test_data[['bedrooms_squared','bed_bath_rooms',
          'log_sqft_living','lat_plus_long']].mean()
```

A2.6 Block 6

Multiple Models Learning

```python

    # Model 1: squarefeet, # bedrooms, # bathrooms, latitude & longitude
    # Model 2: add bedrooms*bathrooms
    # Model 3: Add log squarefeet, bedrooms squared, and the (nonsensical)
        ↪ latitude + longitude

ML_1_features = ['sqft_living', 'bedrooms',
                'bathrooms', 'lat', 'long']
ML_2_features = ML_features + ['bed_bath_rooms']
ML_3_features = ML_features + ['bedrooms_squared',
                             'log_sqft_living', 'lat_plus_long']

# Now we study the value of the weights/coefficients of each model:

# Learn the three models: (don't forget to set validation_set = None)
ML_1 = LinearRegression()
ML_1.fit(train_data[ML_1_features],train_data['price'])
ML_2 = LinearRegression()
ML_2.fit(train_data[ML_2_features],train_data['price'])
ML_3 = LinearRegression()
ML_3.fit(train_data[ML_3_features],train_data['price'])

# Examine/extract each model's coefficients:
for model,features in
    ↪ zip([ML_1,ML_2,ML_3],[ML_1_features,ML_2_features,ML_3_features]):
    intercept = model.intercept_
    coef = model.coef_
    print
        ↪ pd.DataFrame({'Name':['intercepte']+features,'Value':[intercept]+list(coef)})
```

A2.7 Block 7

Multiple Models Comparison

```python
# Find the RSS on TRAINING data for of each models:
for model,features in zip([ML_1,ML_2,ML_3],
                        [ML_1_features, ML_2_features,ML_3_features]):
    print get_residual_sum_of_squares(model,train_data,
                             train_data['price'],features)

# Find the RSS on TESTING data for of each models:
for model,features in zip([ML_1,ML_2,ML_3],
                        [ML_1_features,ML_2_features,ML_3_features]):
    print get_residual_sum_of_squares(model,test_data,
                             test_data['price'],features)
```

A3 Ridge Regression with Gradient Descent Code

A3.1 Block 1

Package declaration and initialization, where the packages to be used are loaded to python interpreter.

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

A3.2 Block 2

Load the house sales and plot house prices from the dataset

```python
dtype_dict = {'bathrooms':float, 'waterfront':int,
             'sqft_above':int, 'sqft_living15':float,
             'grade':int, 'yr_renovated':int, 'price':float,
             'bedrooms':float, 'zipcode':str, 'long':float,
             'sqft_lot15':float, 'sqft_living':float, 'floors':str,
             'condition':int, 'lat':float, 'date':str,
             'sqft_basement':int, 'yr_built':int, 'id':str,
             'sqft_lot':int, 'view':int}

sales = pd.read_csv('kc_house_data.csv',dtype=dtype_dict)

sales.dtypes

sales['price'].plot()
```

A3.3 Block 3

Import functions

```python
def get_numpy_data(data_sframe, features, output):
    data_sframe['constant'] = 1 # this is how you add a constant column to an SFrame
    # add the column 'constant' to the front of
    # the features list so that we can extract it along with the others:
    features = ['constant'] + features # this is how you combine two lists
    # select the columns of data_SFrame given
    # by the features list into the SFrame features_sframe (now including constant):
    features_sframe = data_sframe[features]
    # the following line will convert the features_SFrame into a numpy matrix:
    feature_matrix = features_sframe.as_matrix()
    # assign the column of data_sframe associated with the output to the SArray output_sarray
    output_sarray = data_sframe[output]
    # the following will convert the SArray into a numpy array by first converting
        ↪ it to a list
    output_array = output_sarray.as_matrix()
    return(feature_matrix, output_array)

# The predict_output() function to compute the predictions for an entire matrix
    ↪ of features given the matrix and the weights:

def predict_output(feature_matrix, weights):
    # assume feature_matrix is a numpy matrix containing
    # the features as columns and weights is a corresponding numpy array
    # create the predictions vector by using np.dot()
    predictions = np.dot(feature_matrix,weights)
    return(predictions)
```

A3.4 Block 4

Compute the Derivative

```

# The cost function is the sum over the data points of the squared difference
    ↪ between an observed output and a predicted output, plus the L2 penalty term.

# Cost(wt)
= SUM[ (prediction - output)^2 ]
+ l2_penalty*(wt[0]^2 + wt[1]^2 + ... + wt[k]^2).

# We now add the derivative of the regularization part the derivative of the RSS
    ↪ with respect to wt[i] can be written as:

# 2*SUM[ error*[feature_i] ].

# The derivative of the regularization term with respect to wt[i] is:

# 2*l2_penalty*wt[i].

# Adding both, we get

# 2*SUM[ error*[feature_i] ] + 2*l2_penalty*wt[i].

# In other words, the derivative for the weight for feature i is the sum (over
    ↪ data points) of 2 times the product of the error and the feature itself,
    ↪ plus 2*l2_penalty*wt[i].

# We will not regularize the constant.
# Thus, in the case of the constant, the derivative is just twice the sum of the
    ↪ errors (without the 2*l2_penalty*wt[0] term).

def feature_derivative_ridge(errors, feature,weight,
                             l2_penalty, feature_is_constant):
    # If feature_is_constant is True,
    # derivative is twice the dot product of errors and feature
    if feature_is_constant:
        derivative = 2*np.dot(errors,feature)
    else:
        derivative = 2*np.dot(errors,feature) + 2*l2_penalty*weight
    # Otherwise, derivative is twice the dot product plus 2*l2_penalty*weight

    return derivative

# Testing the feature derivative:

(ML_features, ML_output) = get_numpy_data(sales, ['sqft_living'], 'price')
ML_weights = np.array([1., 10.])
test_predictions = predict_output(ML_features, ML_weights)
errors = test_predictions - ML_output # prediction errors

# next two lines should print the same values
print feature_derivative_ridge(errors, ML_features[:,1],
                               ML_weights[1], 1, False)
print np.sum(errors*ML_features[:,1])*2+20.
print ''

# next two lines should print the same values
print feature_derivative_ridge(errors, ML_features[:,0],
                               ML_weights[0], 1, True)
print np.sum(errors)*2.
```

A3.5 Block 5

Gradient Descent

```
def ridge_regression_gradient_descent(feature_matrix, output, initial_weights,
    ↪ step_size,
                                      l2_penalty, max_iterations=100):
    print 'Starting␣gradient␣descent␣with␣l2_penalty␣=␣' + str(l2_penalty)

    weights = np.array(initial_weights)
    iteration = 0 # iteration counter
    print_frequency = 1 # for adjusting frequency of debugging output

    #while not reached maximum number of iterations:
    while iteration < max_iterations:
        iteration += 1 # increment iteration counter
        ### === code section for adjusting frequency of debugging output. ===
        if iteration == 10:
            print_frequency = 10
        if iteration == 100:
            print_frequency = 100
        if iteration%print_frequency==0:
            print('Iteration␣=␣' + str(iteration))
        ### === end code section ===

        # compute the predictions based on feature_matrix
        # and weights using your predict_output() function
        predictions = predict_output(feature_matrix,weights)
        # compute the errors as predictions - output
        errors = predictions - output
        # from time to time, print the value of the cost function
        if iteration%print_frequency==0:
            print 'Cost␣function␣=␣', str(np.dot(errors,errors) +
                                      l2_penalty*(np.dot(weights,weights)
                                                - weights[0]**2))

        for i in xrange(len(weights)): # loop over each weight
            # Recall that feature_matrix[:,i] is the feature column
            # associated with weights[i]
            # compute the derivative for weight[i].
            #(Remember: when i=0, you are computing the derivative of the constant!)
            if i == 0 :
                derivative = feature_derivative_ridge(errors,
                                                  feature_matrix[:,i],
                                                  weights[i], l2_penalty, True)
            else:
                derivative = feature_derivative_ridge(errors,
                                                  feature_matrix[:,i],
                                                  weights[i], l2_penalty, False)

            # subtract the step size times the derivative from the current weight
            weights[i] -= (step_size*derivative)
    print 'gradient␣descent␣iteration␣completed', iteration
    print 'Learned␣weights␣=␣', str(weights)
    return weights
```

A3.6 Block 6

```

# We now study how the large weights get penalized and consider ML1 with 1
    ↪ feature:

ML1_features = ['sqft_living']
ML1_output = 'price'

# Let us split the dataset into training set and test set. Make sure to use
    ↪ seed=0:

train_data = pd.read_csv('kc_house_train_data.csv',dtype = dtype_dict)
test_data = pd.read_csv('kc_house_test_data.csv',dtype=dtype_dict)

# In this part, we will use only 'sqft_living' to predict 'price'for both the
    ↪ train_data and the test_data.

(ML1_feature_matrix, output) = get_numpy_data(train_data,
                                              ML1_features, ML1_output)
(ML1_test_feature_matrix, test_output) = get_numpy_data(test_data,
                                                        ML1_features, ML1_output)

# Now let us view the parameters for our optimization:

initial_weights = np.array([0., 0.])
step_size = 1e-12
max_iterations=1000

# First, let's consider no regularization where the l2_penalty to 0.0

ML1_weights_0_penalty=ridge_regression_gradient_descent(ML1_feature_matrix,
                                                        output, initial_weights,
                                                        step_size, 0.0, max_iterations)

# Now we consider high regularization with the l2_penalty to 1e11

ML1_weights_high_penalty=ridge_regression_gradient_descent(ML1_feature_matrix,
                                                           output, initial_weights,
                                                           step_size, 1e11,
                                                               ↪ max_iterations)

# We now plot the two learned models with the following:

# Blue - no regularization
# Red - high regularization

import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(ML1_feature_matrix,output,'k.',
        ML1_feature_matrix,predict_output(ML1_feature_matrix,
                                          ML1_weights_0_penalty),'b-',
       ML1_feature_matrix,predict_output(ML1_feature_matrix,
                                         ML1_weights_high_penalty),'r-')

# We now compare weights learned with no regularization and high regularization
    ↪ using the RSS on the TEST data.

ML_RSS1 = sum((-test_output)**2)
ML_RSS2 = ((predict_output(ML1_test_feature_matrix,
                          ML1_weights_0_penalty)-test_output)**2).sum()
ML_RSS3 = sum((predict_output(ML1_test_feature_matrix,
                             ML1_weights_high_penalty)-test_output)**2)
print "RSS1:␣" + str(ML_RSS1)
print "RSS2:␣" + str(ML_RSS2)
print "RSS3:␣" + str(ML_RSS3)

print "No␣Regulaization␣with␣value␣1:␣" + str(ML1_weights_0_penalty[1])
print "High␣Regularization␣value␣1:␣" + str(ML1_weights_high_penalty[1])
```

A3.7 Block 7

Multiple regression with L2 penalty

```

sales.dtypes

# Now we will study a model with 2 features: ['sqft_living', 'sqft_living15'].

# First, we using the training and test data with these two features.

ML2_features = ['sqft_living', 'sqft_living15'] # sqft_living15 is the
                                            # average squarefeet
                                            # for the nearest 15 neighbors.
ML2_output = 'price'
(ML2_feature_matrix, output) = get_numpy_data(train_data,
                                          ML2_features,
                                          ML2_output)
(ML2_test_feature_matrix, test_output) = get_numpy_data(test_data,
                                                   ML2_features,
                                                   ML2_output)

# We need to re-inialize the weights, set the step size to 1e-12 and maximum
    ↪ number of iterations 1000 .

initial_weights = np.array([0.0,0.0,0.0])
step_size = 1e-12
max_iterations = 1000

# First, let's consider no regularization where the l2_penalty to 0.0

ML2_weights_0_penalty = ridge_regression_gradient_descent(ML2_feature_matrix,
                                                     output, initial_weights,
                                                     step_size, 0.0,
                                                         ↪ max_iterations)

# Now we consider high regularization with the l2_penalty to 1e11

ML2_weights_high_penalty = ridge_regression_gradient_descent(ML2_feature_matrix,
                                                        output, initial_weights,
                                                        step_size, 1e11,
                                                            ↪ max_iterations)

# We now plot the two learned models with the following:

# Blue - no regularization
# Red - high regularization

import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(ML2_feature_matrix,output,'k.',
        ML2_feature_matrix,predict_output(ML2_feature_matrix,
                                       ML2_weights_0_penalty),'b-',
          ML2_feature_matrix,predict_output(ML2_feature_matrix,
                                          ML2_weights_high_penalty),'r-')

# We now compare weights learned with no regularization and high regularization using the RSS on the TEST data.

ML_RSS1 = sum((-test_output)**2)
ML_RSS2 = ((predict_output(ML2_test_feature_matrix,
                           ML2_weights_0_penalty)-test_output)**2).sum()
ML_RSS3 = sum((predict_output(ML2_test_feature_matrix,
                              ML2_weights_high_penalty)-test_output)**2)
print "RSS1:␣" + str(ML_RSS1)
print "RSS2:␣" + str(ML_RSS2)
print "RSS3:␣" + str(ML_RSS3)

# We now predict the house price for the 1st house in the test set using the no regularization and high regularization models.

print "Predicted␣1st␣house␣(No␣Regularization):␣" + str(predict_output(ML2_test_feature_matrix,
                        ML2_weights_0_penalty)[0])
print "Predicted␣1st␣house␣(High␣Regularization):␣" + str(predict_output(ML2_test_feature_matrix,
                      ML2_weights_high_penalty)[0])
print "Actual␣house␣price:␣" + str(test_output[0])

# We now predict the house price for the 11th house in the test set using the no regularization and high regularization models.

print "Predicted␣11th␣house␣(No␣Regularization):␣" + str(predict_output(ML2_test_feature_matrix,
                     ML2_weights_0_penalty)[10])
print "Predicted␣11th␣house␣(High␣Regularization):␣" + str(predict_output(ML2_test_feature_matrix,
                    ML2_weights_high_penalty)[10])
print "Actual␣house␣price:␣" + str(test_output[10])
```

A4 Lasso with Coordinate Descent Code

A4.1 Block 1

Package declaration and initialization, where the packages to be used are loaded to python interpreter.

```
import pandas as pd
import numpy as np
```

A4.2 Block 2

Load the house sales and plot house prices from the dataset

```python


dtype_dict = {'bathrooms':float, 'waterfront':int,
             'sqft_above':int, 'sqft_living15':float,
             'grade':int, 'yr_renovated':int, 'price':float,
             'bedrooms':float, 'zipcode':str, 'long':float,
             'sqft_lot15':float, 'sqft_living':float, 'floors':str,
             'condition':int, 'lat':float, 'date':str,
             'sqft_basement':int, 'yr_built':int, 'id':str,
             'sqft_lot':int, 'view':int}

sales = pd.read_csv('kc_house_data.csv',dtype=dtype_dict)
sales['floors'] = sales['floors'].astype(float).astype(int)

sales.dtypes

sales['price'].plot()
```

A4.3 Block 3

Import functions

```python

def get_numpy_data(data_sframe, features, output):
    data_sframe['constant'] = 1 # this is how you add a constant column to an
        ↪ SFrame
    # add the column 'constant' to the front of the features list
    # so that we can extract it along with the others:
    features = ['constant'] + features # this is how you combine two lists
    # select the columns of data_SFrame given by the features list
    # into the SFrame features_sframe (now including constant):
    features_sframe = data_sframe[features]
    # the following line will convert the features_SFrame into a numpy matrix:
    feature_matrix = features_sframe.as_matrix()
    # assign the column of data_sframe associated with the output to the SArray
        ↪ output_sarray
    output_sarray = data_sframe[output]
    # the following will convert the SArray into a numpy array by first converting
        ↪ it to a list
    output_array = output_sarray.as_matrix()
    return(feature_matrix, output_array)

# The predict_output() function to compute the predictions for an entire matrix
    ↪ of features given the matrix and the weights:

def predict_output(feature_matrix, weights):
    # assume feature_matrix is a numpy matrix containing
    # the features as columns and weights is a corresponding numpy array
    # create the predictions vector by using np.dot()
    predictions = np.dot(feature_matrix,weights)
    return(predictions)
```

A4.4 Block 4

Normalize features

We now normalize features by dividing each feature by its norm 2 so, that the transformed feature has norm 1.

```
# Let's us first consider a small matrix.

X = np.array([[20.,24.,40.],[22.,28.,44.]])
print X

norms = np.linalg.norm(X, axis=0) # gives [norm(X[:,0]), norm(X[:,1]),
    ↪ norm(X[:,2])]
print norms

# To normalize, apply element-wise division:

print X / norms # gives [X[:,0]/norm(X[:,0]), X[:,1]/norm(X[:,1]),
    ↪ X[:,2]/norm(X[:,2])]

# Now, we will use these norms to normalize the test data in the same way as we
    ↪ normalized the training data.

def normalize_features(feature_matrix):
    norms = np.linalg.norm(feature_matrix,axis = 0)
    normalized_features = feature_matrix/norms
    return normalized_features,norms

# Now testing the function:

features, norms = normalize_features(np.array([[20.,24.,40.],[22.,28.,44.]]))
print "Features:␣" + str(features)

print "Norms:␣" + str(norms)
```

A4.5 Block 5

Developing the coordinate descent with normalized features

```

# We would like to minimize the LASSO cost function

# SUM[ (prediction - output)^2 ] + lambda*( |wt[1]| + ... + |wt[k]|).

# We do not want to push the intercept to zero so, wt[0] is not included in the
    ↪ L1 penalty term.

# The coordinate descent will be used at each iteration

# We will fix all weights but weight i

# Then find the value of weight i that minimizes the objective.

# In other words, we seek:

# argmin_{wt[i]} [ SUM[ (prediction - output)^2 ] + lambda*( |wt[1]| + ... +
    ↪ |wt[k]|) ]

# where all weights other than wt[i] are held to be constant.

# We will also optimize one wt[i] at a time, and move through the computation
    ↪ with the weights multiple times.

# For this process:

# We use the cyclical coordinate descent with normalized features, where we cycle
    ↪ through coordinates 0 to (d-1) in order, and assume the features were
    ↪ normalized as discussed above.

# We optimize each coordinate using the following formula:

# when wt[i] = [ (Zo[i] + lambda/2), if Zo[i] < -lambda/2
# or
# when wt[i] = [ 0, if -lambda/2 <= Zo[i] <= lambda/2
# or
# when wt[i] = [ (Zo[i] - lambda/2), if Zo[i] > lambda/2

# where

# Zo[i] = SUM[ [feature_i]*(output - prediction + wt[i]*[feature_i])].

# We do no regularize the weight of the constant feature (intercept) wt[0].

# With that said, we updated with:

# wt[0] = Zo[i]
```

A4.6 Block 6

Studying the L1 penalty

```

# Let us consider a simple model with 2 features:

ML1_features = ['sqft_living', 'bedrooms']
ML1_output = 'price'
(ML1_feature_matrix, output) = get_numpy_data(sales, ML1_features,
                                              ML1_output)

# Normalizing the features:

ML1_feature_matrix, norms = normalize_features(ML1_feature_matrix)
```

```

# We assign some random set of initial weights and
inspect the values of Zo[i]:

weights = np.array([2., 8., 6.])

Use predict_output() to make predictions on this data.

prediction = predict_output(ML1_feature_matrix,weights)

# We compute the values of Zo[i] for each feature in this simple model:

# Zo[i] = SUM[ [feature_i]*(output - prediction + wt[i]*[feature_i]) ]

Zo = list()
print weights
for i in range(0,len(ML1_features)+1):
    print i
    feature_i = ML1_feature_matrix[:,i]
    Zo.append( sum((feature_i)*(output - prediction + weights[i]*feature_i) ))
Zo
```

A4.7 Block 7

Single Coordinate Descent Step

```

# We now minimize the cost function over a single feature i with the coordinate
    ↪ descent step.

# To optimize over, the function should accept feature matrix, output, current
    ↪ weights, l1 penalty and index of the feature.

# And the function should return new weight for feature i.

def lasso_coordinate_descent_step(i, feature_matrix, output, weights, l1_penalty):
    # compute prediction
    prediction = predict_output(feature_matrix,weights)
    # compute Zo[i] = SUM[ [feature_i]*(output - prediction +
        ↪ weight[i]*[feature_i]) ]
    feature_i = feature_matrix[:,i]
    Zo_i = sum((feature_i)*(output - prediction + weights[i]*feature_i) )

    if i == 0: # intercept -- do not regularize
        new_weight_i = Zo_i
    elif Zo_i < -l1_penalty/2.:
        new_weight_i = Zo_i + l1_penalty/2
    elif Zo_i > l1_penalty/2.:
        new_weight_i = Zo_i - l1_penalty/2
    else:
        new_weight_i = 0.

    return new_weight_i

# To test the function using the following:

import math
print lasso_coordinate_descent_step(1, np.array([[3./math.sqrt(13),1./math.sqrt(10)],
               [2./math.sqrt(13),3./math.sqrt(10)]]),
                np.array([1., 1.]), np.array([1., 4.]), 0.1)
```

A4.8 Block 8

Cyclical coordinate descent

```

# We now implement a cyclical coordinate descent where we optimize coordinates 0, 1, ..., (d-1) in order and repeat.

# We will measure the change in weight for each coordinate.

# If no coordinate changes by more than a specified threshold, we stop.

# For each iteration:

# As we loop over features in order and perform coordinate descent, measure how much each coordinate changes.

# After the loop, if the maximum change across all coordinates is falls below the tolerance, stop.

# Otherwise, go back to step 1.

# Return weights

def lasso_cyclical_coordinate_descent(feature_matrix, output,
                                      initial_weights, l1_penalty,
                                      tolerance):
    while True:
        maxstep = -1
        maxi = 0
        weights = np.zeros(len(initial_weights))
        for i in range(len(initial_weights)):
            weights[i] = lasso_coordinate_descent_step(i,feature_matrix,
                                                       output,initial_weights,
                                                       l1_penalty)
            step = abs(weights[i]-initial_weights[i])
            if(step>maxstep):
                maxstep = step
                maxi = i
        initial_weights[maxi]=weights[maxi]
        #print maxi,maxstep
        if maxstep < tolerance:
            return initial_weights

# Using the following parameters, learn the weights on the sales dataset.

sales.dtypes

ML1_features = ['sqft_living', 'sqft_living15']
ML1_output = 'price'
initial_weights = np.zeros(3)
l1_penalty = 1e7
tolerance = 1

# We normalized the feature matrix:

(ML1_feature_matrix, output) = get_numpy_data(sales,
                                          ML1_features, ML1_output)
(normalized_ML1_feature_matrix, ML1_norms) =
    ↪ normalize_features(ML1_feature_matrix) # normalize features

# We normalized the feature matrix:

(ML1_feature_matrix, output) = get_numpy_data(sales,
                                          ML1_features, ML1_output)
(normalized_ML1_feature_matrix, ML1_norms) =
    ↪ normalize_features(ML1_feature_matrix) # normalize features

# Now we execute the LASSO coordinate descent implementation:

weights = lasso_cyclical_coordinate_descent(normalized_ML1_feature_matrix, output,
                                        initial_weights, l1_penalty, tolerance)
RSS = sum((predict_output(normalized_ML1_feature_matrix,weights)-output)**2)
print 'RSS:␣' + str(RSS)
print 'Weights:␣' + str(weights)
```

A4.9 Block 9

Study the LASSO fit with more features

```
# Let us split the sales dataset into training and test sets.

train_data = pd.read_csv('kc_house_train_data.csv',dtype=dtype_dict)
test_data = pd.read_csv('kc_house_test_data.csv',dtype=dtype_dict)
train_data['floors'] = train_data['floors'].astype(float).astype(int)
test_data['floors'] = test_data['floors'].astype(float).astype(int)

# Let us consider the following set of features.

more_features = ['bedrooms',
               'bathrooms',
                'sqft_living',
                'sqft_lot',
                'floors',
                'waterfront',
                'view',
                'condition',
                'grade',
                'sqft_above',
                'sqft_basement',
                'yr_built',
                'yr_renovated']

# Normalize the feature matrix from the TRAINING data with these features:

(feature_matrix, output) = get_numpy_data(train_data,
                                        more_features, 'price')
normalized_training,norms = normalize_features(feature_matrix)

# Now we learn the weights with l1_penalty=1e7, on the training data.

# Initialize weights to all zeros, and set the tolerance=1.

# Call resulting weights wts_1e7.

initial_weights = np.zeros(14)
wts_1e7 = lasso_cyclical_coordinate_descent(normalized_training, output,
    ↪ initial_weights, 1e7, 1)
a7 = wts_1e7
a7

print wts_1e7
for i in range(0,len(wts_1e7)):
    if wts_1e7[i]!=0 and i==0:
        print 0,'constant'
    elif wts_1e7[i]!=0:
        print i,more_features[i-1]

# Now we learn the weights with l1_penalty=1e8, on the training data.

# Initialize weights to all zeros, and set the tolerance=1.

# Call resulting weights wts_1e8.

initial_weights = np.zeros(14)
wts_1e8 = lasso_cyclical_coordinate_descent(normalized_training, output,
                                          initial_weights, 1e8, 1)

for i in range(0,len(wts_1e8)):
    if wts_1e8[i]!=0:
        if i == 0:
            print 0,'constant'
        else:
            print i,more_features[i-1]
a8 = wts_1e8

# Finally, we learn the weights with l1_penalty=1e4, on the training data.

# Initialize weights to all zeros, and set the tolerance=5e5.

# Call resulting weights wts_1e4.

initial_weights = np.zeros(14)
wts_1e4 = lasso_cyclical_coordinate_descent(normalized_training,
    ↪ output,initial_weights, 1e4, 5e5)
for i in range(0,len(wts_1e4)):
    if wts_1e4[i]!=0:
        if i ==0:
            print 0,'constant'
        else:
            print i,more_features[i-1]
a4 = wts_1e4
```

A4.10 Block 10

Rescaling learned weights

```

# We can rescale the learned weights to include the normalization.

# This will help us to not worry about normalizing the test data:

# With that said, we will scale our weights to make predictions with original
    ↪ features:

# features, norms = normalize_features(features)
# weights_normalized = weights / norms

# Now, we can use the test data, without normalizing it!

# We apply to the weights learned above. (wt_1e4, wt_1e7, wt_1e8).

#features, norms = normalize_features(more_features)
normalized_wt_1e4=a4/norms
normalized_wt_1e7=a7/norms
normalized_wt_1e8=a8/norms

# To check your results, if you call normalized_wt_1e7 the normalized version of
    ↪ wt_1e7, then:

# print normalized_wt_1e7[3]

# should return 161.3174...

```

```
print normalized_wt_1e7[3]
```

A4.11 Block 11

Study the learned models on the test data

```

# We will now evaluate the three models on the test data:

(ML1_feature_matrix, ML1_output) = get_numpy_data(test_data,
                                                  more_features, 'price')

# The RSS for all three normalized weights on the (unnormalized)
    ↪ ML1_feature_matrix:

RSS_wt_4 = sum((predict_output(ML1_feature_matrix,
                              normalized_wt_1e4)-ML1_output)**2)
RSS_wt_7 = sum((predict_output(ML1_feature_matrix,
                              normalized_wt_1e7)-ML1_output)**2)
RSS_wt_8 = sum((predict_output(ML1_feature_matrix,
                              normalized_wt_1e8)-ML1_output)**2)

# The RSS for all three normalized weights on the (unnormalized)
    ↪ ML1_feature_matrix:

RSS_wt_4 = sum((predict_output(ML1_feature_matrix,
                              normalized_wt_1e4)-ML1_output)**2)
RSS_wt_7 = sum((predict_output(ML1_feature_matrix,
                              normalized_wt_1e7)-ML1_output)**2)
RSS_wt_8 = sum((predict_output(ML1_feature_matrix,
                              normalized_wt_1e8)-ML1_output)**2)

print 'RSS for wt_1e4: ' + str(RSS_wt_4)
print 'RSS for wt_1e7: ' + str(RSS_wt_7)
print 'RSS for wt_1e8: ' + str(RSS_wt_8)
```