chapter-4-0

January 23, 2022

Run in Google Colab

1 Supervised Models: Linear Regression

1.1 Linear Regression

Despite its simplicity, a good understanding of linear regression is prerequisite for understanding how more advanced models work. Generally speaking, a linear model makes a prediction by simply computing a weighted sum of the input features, plus a constant called the *bias* term (also called the *intercept* term):

$$Y = a + b_1 X_1 + b_2 X_2 + \cdots + b_m X_m + \epsilon \tag{1}$$

where:

- Y is the predicted value (the value of the target);
- m is the number of features;
- X_i is the i^{th} feature value that are used to predict Y;
- a and b_j are the j^{th} model parameters (a being the bias term and b_i the weights)
- ϵ is the predicton error.

As usual the parameters a and b_i are chosen to minimize the mean squared error over the training data set.

This means that the task in linear regression is to find values for a and b_i that minimize

$$\frac{1}{n}\sum_{i=1}^{n}\left(Y - a - b_1 X_{i1} - b_2 X_{i2} - \cdots - b_m X_{im}\right)^2 \tag{2}$$

where n is the size of the training set.

Training a model means setting its parameters so that the model best fits the training set. For this purpose, we first need a measure of how well (or poorly) the model fits the training data. The most common performance measure of a regression model is the Root Mean Square Error (RMSE), therefore, to train a Linear Regression model, you need to find the value of θ that minimizes the RMSE. In practice, it is simpler to minimize the Mean Square Error (MSE) than the RMSE, and it leads to the same result.

1.2 Validation and Testing

As we have already said, when data is used for forecasting there is a danger that the machine learning model will work very well for data, but will not generalize well to other data. An obvious point is that it is important that the data used in a machine learning model be representative of the situations to which the model is to be applied. It is also important to test a model out-of-sample, by this we mean that the model should be tested on data that is different from the sample data used to determine the parameters of the model.

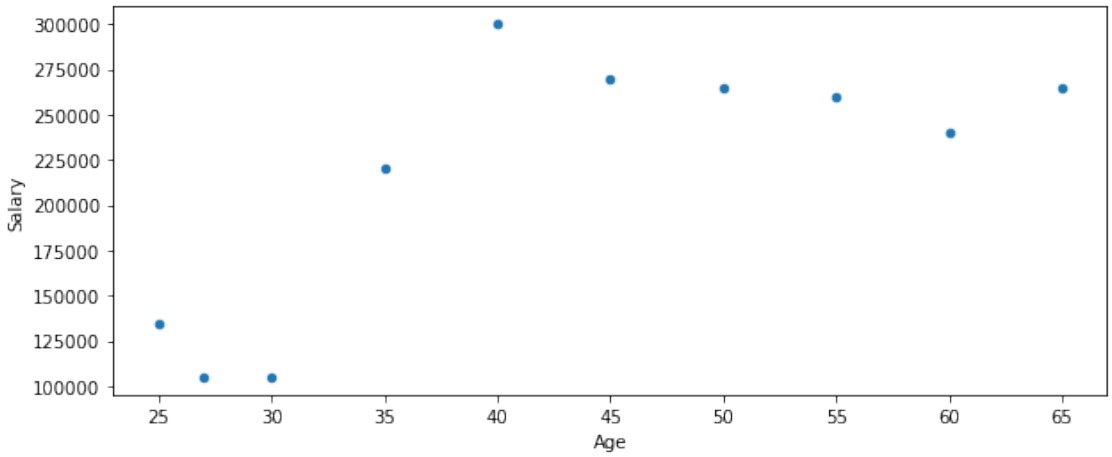
Data scientist refer to the sample data as the **training set** and the data used to determine the accuracy of the model as the **test set**, often a **validation set** is used as well as we explain later;

[12]:
```
#
# Here we have to load the file 'salary_vs_age_1.csv'
#
if 'google.colab' in str(get_ipython()):
    from google.colab import files
    uploaded = files.upload()
    path = ''
else:
    path = './data/'
```

[13]:
```
# Load the Pandas libraries with alias 'pd'
import pandas as pd
# Read data from file 'salary_vs_age_1.csv'
# (in the same directory that your python process is based)
# Control delimiters, with read_table
df1 = pd.read_table(path + "salary_vs_age_1.csv", sep=";")
# Preview the first 5 lines of the loaded data
print(df1.head())
```

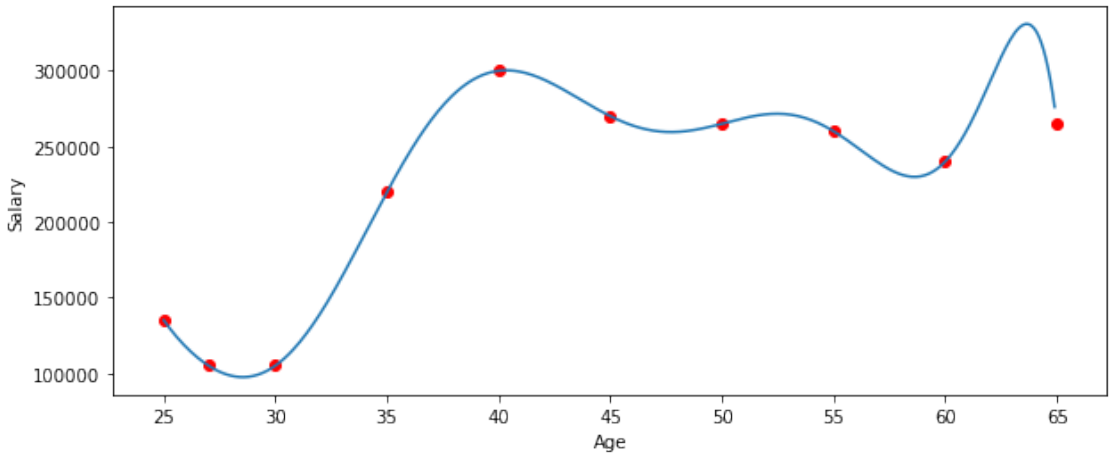
```
   Age  Salary
0   25  135000
1   27  105000
2   30  105000
3   35  220000
4   40  300000
```

[14]:
```
import matplotlib.pyplot as plt

plt.rcParams['figure.figsize'] = [10, 4]
ax=plt.gca()

df1.plot(x ='Age', y='Salary', kind = 'scatter', ax=ax)
plt.show()
```



polynomial fitting with pandas

[15]:
```
import numpy as np

x1 = df1['Age']
y1 = df1['Salary']

n = len(x1)

degree = 9

weights = np.polyfit(x1, y1, degree)
model   = np.poly1d(weights)

xx1 = np.arange(x1[0], x1[n-1], 0.1)
plt.plot(xx1, model(xx1))
plt.xlabel("Age")
plt.ylabel("Salary")
plt.scatter(x1,y1, color='red')
plt.show()
```



[16]:
```
y1  = np.array(y1)
yy1 = np.array(model(x1))

rmse = np.sqrt(np.sum((y1-yy1)**2)/(n-1))

print('Root Mean Square Error:')
print(rmse)
```

```
Root Mean Square Error:
0.0007184405154295555
```

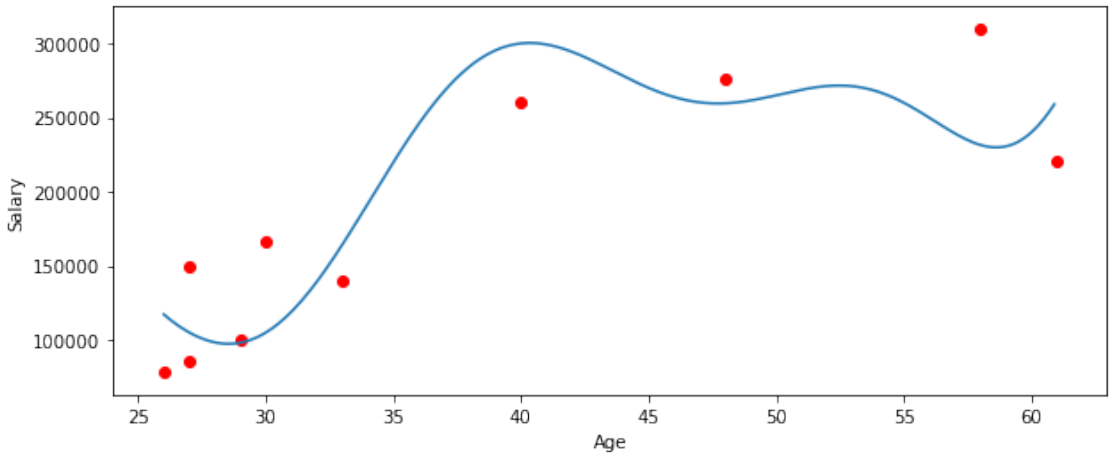
[17]:
```
if 'google.colab' in str(get_ipython()):
    from google.colab import files
    uploaded = files.upload()
    path = ''
else:
    path = './data/'
```

[18]:
```
df2 = pd.read_table(path + "salary_vs_age_2.csv", sep=";")
x2 = df2['Age']
y2 = df2['Salary']
n  = len(x2)

y2  = np.array(y2)
yy2 = np.array(model(x2))

rmse = np.sqrt(np.sum((y2-yy2)**2)/(n-1))

print('Root Mean Square Error:')
print(rmse)
```

```
Root Mean Square Error:
```

```
44726.74305949611
```

[19]:
```
xx2 = np.arange(x2[0], x2[n-1], 0.1)
plt.plot(xx2, model(xx2))
plt.xlabel("Age")
plt.ylabel("Salary")
plt.scatter(x2,y2, color='red')
plt.show()
```



- The root mean squared error (rmse) for the training data set is $12,902
- The rmse for the test data set is $38,794

We conclude that the model overfits the data. The complexity of the model should be increased only until out-of-sample tests indicate that it does not generalize well.

1.3 Example 1 - Predicting Iowa House Prices (from Kaggle)

[20]:
```
# loading packages

import os

import pandas as pd
import numpy as np

# plotting packages
%matplotlib inline
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import matplotlib.colors as clrs

# Kmeans algorithm from scikit-learn
```

```
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_samples, silhouette_score
```

The Problem The objective is to predict the prices of house in Iowa from features. We have 800 observations in training set, 600 in validation set, and 508 in test set

Categorical Features (INSERIRE RIMANDO ALLA PRESENTAZIONE DEL PROBLEMA Ch 2-1) Categorical features are features where there are a number of non-numerical alternatives. We can define a dummy variable for each alternative. The variable equals 1 if the alternative is true and zero otherwise. This is known as **one-hot encoding**. But sometimes we do not have to do this because there is a natural ordering of variables. For example in this problem one of the categorical features is concerned with the basement quality as indicated by the ceiling height. The categories are:

- *Excellent (< 100 inches)*
- *Good (90-99 inches)*
- *Typical (80-89 inches)*
- *Fair (70-79 inches)*
- *Poor (< 70 inches)*
- *No Basement*

This is an example of a categorical variable where *there is* a natural ordering. We created a new variable that had a values of 5, 4, 3, 2, 1 and 0 for the above six categories respectively.

The other categorical features specifies the location of the house as in one of 25 neighborhoods. We introduce 25 dummy variables with a one-hot encoding. The dummy variable equals one for an observation if the neighborhood is that in which the house is located and zero otherwise.

Loading data (J. C. Hull, 2019, Chapter 3) To illustrate the regression techniques discussed in this chapter we will use a total of 48 feature. 21 are numerical and two are categorical and to this we had, as discussed above, 25 categorical variables for the neighborhoods.

[21]:
```
if 'google.colab' in str(get_ipython()):
    from google.colab import files
    uploaded = files.upload()
    path = ''
else:
    path = './data/'
```

[22]:
```
# Both features and target have already been scaled: mean = 0; SD = 1
data = pd.read_csv(path + 'Houseprice_data_scaled.csv')
data.head()
```

[22]:

	LotArea	OverallQual	OverallCond	YearBuilt	YearRemodAdd	BsmtFinSF1	\
0	-0.199572	0.652747	-0.512407	1.038851	0.875754	0.597837	
1	-0.072005	-0.072527	2.189741	0.136810	-0.432225	1.218528	
2	0.111026	0.652747	-0.512407	0.972033	0.827310	0.095808	
3	-0.077551	0.652747	-0.512407	-1.901135	-0.722887	-0.520319	

```
4  0.444919     1.378022    -0.512407   0.938624      0.730423    0.481458

   BsmtUnfSF  TotalBsmtSF  1stFlrSF  2ndFlrSF  ...   OLDTown     SWISU  \
0  -0.937245    -0.482464 -0.808820  1.203988  ... -0.286942 -0.136621
1  -0.635042     0.490326  0.276358 -0.789421  ... -0.286942 -0.136621
2  -0.296754    -0.329118 -0.637758  1.231999  ... -0.286942 -0.136621
3  -0.057698    -0.722067 -0.528171  0.975236  ... -0.286942 -0.136621
4  -0.170461     0.209990 -0.036366  1.668495  ... -0.286942 -0.136621

   Sawyer   SawyerW   Somerst   StoneBr    Timber    Veenker  Bsmt Qual  \
0 -0.2253 -0.214192 -0.268378 -0.127929 -0.152629  -0.091644   0.584308
1 -0.2253 -0.214192 -0.268378 -0.127929 -0.152629  10.905682   0.584308
2 -0.2253 -0.214192 -0.268378 -0.127929 -0.152629  -0.091644   0.584308
3 -0.2253 -0.214192 -0.268378 -0.127929 -0.152629  -0.091644  -0.577852
4 -0.2253 -0.214192 -0.268378 -0.127929 -0.152629  -0.091644   0.584308

   Sale Price
0    0.358489
1    0.008849
2    0.552733
3   -0.528560
4    0.895898

[5 rows x 48 columns]
```

First of all check how many records we have

[23]:
```
print("Number of available data = "  + str(len(data.index)))
```

```
Number of available data = 2908
```

Before starting we emphasize the need to divide all available data into three parts: a **training set**, a **validation set** and a **test set**. The training set is used to determine parameters for trial models. The validation set is used to determine the extent to chich the models created from the training set generalize to new data. Finally, the test set is used as a final estimate of the accuracy of the chosen model.

We had 2908 observations. We split this as follows: 1800 in the training set, 600 in the validation set and 508 in the test set.

[24]:
```
# First 1800 data items are training set; the next 600 are the validation set
train = data.iloc[:1800]
val = data.iloc[1800:2400]
```

We now procede to create **labels** and **features**. As we have already said, the labels are the values of the target that is to be predicted, in this case the 'Sale Price', and we indicate that whit 'y':

[25]:
```
y_train, y_val = train[['Sale Price']], val[['Sale Price']]
```

The features and dummy variables were scaled using the Z-score method. Also the target values

(i.e. the house prices) have been scaled with the Z-score method. The features are the variables from which the predictions are to be made and, in this case, can be obtained simply dropping the column 'Sale Price' from our dataset:

[26]:
```
X_train, X_val = train.drop('Sale Price', axis=1), val.drop('Sale Price', axis=1)
```

[27]:
```
X_train.columns
```

[27]:
```
Index(['LotArea', 'OverallQual', 'OverallCond', 'YearBuilt', 'YearRemodAdd',
       'BsmtFinSF1', 'BsmtUnfSF', 'TotalBsmtSF', '1stFlrSF', '2ndFlrSF',
       'GrLivArea', 'FullBath', 'HalfBath', 'BedroomAbvGr', 'TotRmsAbvGrd',
       'Fireplaces', 'GarageCars', 'GarageArea', 'WoodDeckSF', 'OpenPorchSF',
       'EnclosedPorch', 'Blmngtn', 'Blueste', 'BrDale', 'BrkSide', 'ClearCr',
       'CollgCr', 'Crawfor', 'Edwards', 'Gilbert', 'IDOTRR', 'MeadowV',
       'Mitchel', 'Names', 'NoRidge', 'NPkVill', 'NriddgHt', 'NWAmes',
       'OLDTown', 'SWISU', 'Sawyer', 'SawyerW', 'Somerst', 'StoneBr', 'Timber',
       'Veenker', 'Bsmt Qual'],
      dtype='object')
```

Linear Regression with sklearn

[28]:
```
# Importing models
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error as mse
```

[29]:
```
lr = LinearRegression()
lr.fit(X_train,y_train)
```

[29]:
```
LinearRegression()
```

[30]:
```
lr.intercept_
```

[30]:
```
array([-3.06335941e-11])
```

[31]:
```
coeffs = pd.DataFrame(
    [
        ['intercept'] + list(X_train.columns),
        list(lr.intercept_) + list(lr.coef_[0])
    ]
)
coeffs
```

[31]:

	0	1	2	3	4	5	\
0	intercept	LotArea	OverallQual	OverallCond	YearBuilt	YearRemodAdd	
1	-3.06336e-11	0.0789996	0.214395	0.0964787	0.160799	0.0253524	

	6	7	8	9	...	38	39	\
0	BsmtFinSF1	BsmtUnfSF	TotalBsmtSF	1stFlrSF	...	NWAmes	OLDTown	
1	0.0914664	-0.0330798	0.138199	0.152786	...	-0.0517591	-0.026499	

```
            40         41         42         43         44          45  \
0        SWISU      Sawyer    SawyerW    Somerst    StoneBr      Timber
1 -0.00414298 -0.0181341 -0.0282754  0.0275063  0.0630586 -0.00276173

            46         47
0     Veenker  Bsmt Qual
1  0.00240311  0.0113115

[2 rows x 48 columns]
```

[32]:
```
# Create dataFrame with corresponding feature and its respective coefficients
coeffs = pd.DataFrame(
    [
        ['intercept'] + list(X_train.columns),
        list(lr.intercept_) + list(lr.coef_[0])
    ]
).transpose().set_index(0)
coeffs
```

[32]:
```
                         1
0
intercept     -3.06336e-11
LotArea          0.0789996
OverallQual       0.214395
OverallCond      0.0964787
YearBuilt         0.160799
YearRemodAdd     0.0253524
BsmtFinSF1       0.0914664
BsmtUnfSF       -0.0330798
TotalBsmtSF       0.138199
1stFlrSF          0.152786
2ndFlrSF          0.132765
GrLivArea         0.161303
FullBath        -0.0208076
HalfBath         0.0171941
BedroomAbvGr    -0.0835202
TotRmsAbvGrd     0.0832203
Fireplaces       0.0282578
GarageCars       0.0379971
GarageArea       0.0518093
WoodDeckSF       0.0208337
OpenPorchSF      0.0340982
EnclosedPorch   0.00682223
Blmngtn         -0.0184305
Blueste         -0.0129214
BrDale          -0.0246262
```

```
BrkSide          0.0207618
ClearCr         -0.00737828
CollgCr         -0.00675362
Crawfor           0.0363235
Edwards        -0.000690065
Gilbert         -0.00834022
IDOTRR          -0.00153683
MeadowV           -0.016418
Mitchel         -0.0284821
Names           -0.0385057
NoRidge           0.0515626
NPkVill         -0.0219519
NriddgHt            0.12399
NWAmes          -0.0517591
OLDTown          -0.026499
SWISU           -0.00414298
Sawyer          -0.0181341
SawyerW         -0.0282754
Somerst           0.0275063
StoneBr           0.0630586
Timber          -0.00276173
Veenker          0.00240311
Bsmt Qual         0.0113115
```

[33]:
```
len(coeffs.index)
```

[33]:
```
48
```

[34]:
```
pred_t=lr.predict(X_train)
mse(y_train,pred_t)
```

[34]:
```
0.11401526431246334
```

[35]:
```
pred_v=lr.predict(X_val)
mse(y_val,pred_v)
```

[35]:
```
0.11702499460121657
```

For the data we are considering it turns out that this regression model generalizes well. The mean squared error for the validation set was only a little higher than that for the training set. However linear regression with no regularization leads to some strange results because of the correlation between features. For example it makes no sense that the weights for number of full bathrooms and number of bedrooms are negative!

[36]:
```
x1 = X_train['GrLivArea']
x2 = X_train['BedroomAbvGr']
x1.corr(x2)
```

[36]:
```
0.5347396038733939
```

Ridge Regression

[37]:
```
# Importing Ridge
from sklearn.linear_model import Ridge
```

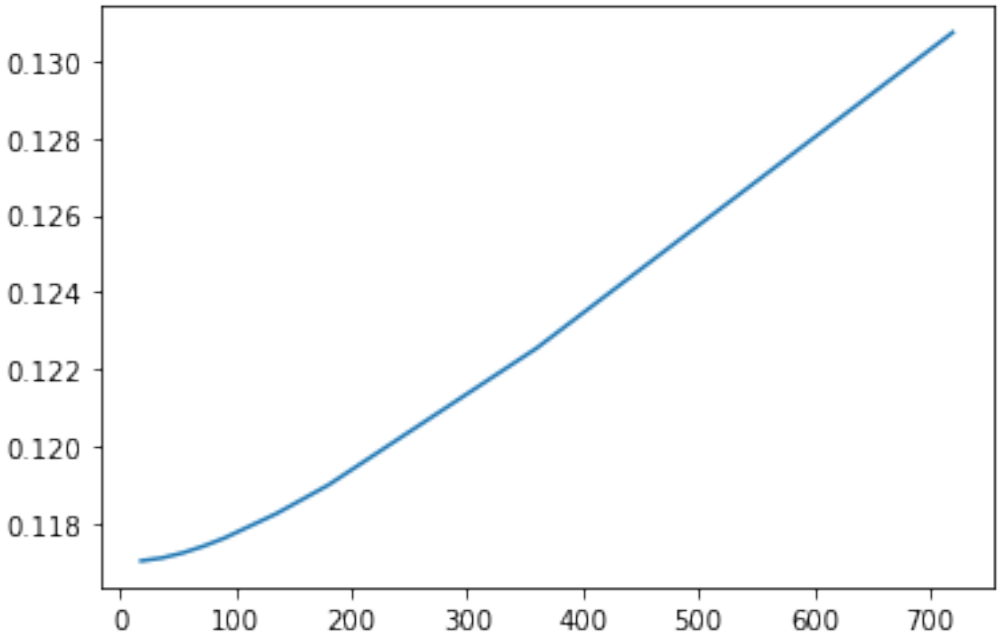
We try using Ridge regression with different values of the hyperparameter λ. The following code shows the effect of this parameter on the prediction error.

[38]:
```
# The alpha used by Python's ridge should be the lambda in Hull's book times the␣
 ↪number of observations
alphas=[0.01*1800, 0.02*1800, 0.03*1800, 0.04*1800, 0.05*1800, 0.075*1800,0.
 ↪1*1800,0.2*1800, 0.4*1800]
mses=[]
for alpha in alphas:
    ridge=Ridge(alpha=alpha)
    ridge.fit(X_train,y_train)
    pred=ridge.predict(X_val)
    mses.append(mse(y_val,pred))
    print(mse(y_val,pred))
```

```
0.11703284346091342
0.11710797319752984
0.11723952924901117
0.11741457158889518
0.1176238406871145
0.11825709631198021
0.11900057469147927
0.1225464999629295
0.13073599680747128
```

[39]:
```
plt.plot(alphas, mses)
```

[39]:
```
[<matplotlib.lines.Line2D at 0x17a4f841808>]
```



As expected the prediction error increases as λ increases. Values of λ in the range 0 to 0.1 might be reasonably be considered because prediction errors increases only slightly when λ is in this range. However it turns out that the improvement in the model is quite small for these values of λ.

Lasso

[21]:
```
# Import Lasso
from sklearn.linear_model import Lasso
```

[22]:
```
# Here we produce results for alpha=0.05 which corresponds to lambda=0.1 in␣
 ↪Hull's book
lasso = Lasso(alpha=0.05)
lasso.fit(X_train, y_train)
```

[22]:
```
Lasso(alpha=0.05)
```

[23]:
```
# DataFrame with corresponding feature and its respective coefficients
coeffs = pd.DataFrame(
    [
        ['intercept'] + list(X_train.columns),
        list(lasso.intercept_) + list(lasso.coef_)
    ]
).transpose().set_index(0)
coeffs
```

[23]:

	1
0	
intercept	-1.25303e-11
LotArea	0.0443042
OverallQual	0.298079
OverallCond	0
YearBuilt	0.0520907
YearRemodAdd	0.0644712
BsmtFinSF1	0.115875
BsmtUnfSF	-0
TotalBsmtSF	0.10312
1stFlrSF	0.0322946
2ndFlrSF	0
GrLivArea	0.297065
FullBath	0
HalfBath	0
BedroomAbvGr	-0
TotRmsAbvGrd	0
Fireplaces	0.0204043
GarageCars	0.027512
GarageArea	0.0664096
WoodDeckSF	0.00102883
OpenPorchSF	0.00215018
EnclosedPorch	-0
Blmngtn	-0
Blueste	-0
BrDale	-0
BrkSide	0
ClearCr	0
CollgCr	-0
Crawfor	0
Edwards	-0
Gilbert	0
IDOTRR	-0
MeadowV	-0
Mitchel	-0
Names	-0
NoRidge	0.013209
NPkVill	-0
NriddgHt	0.0842993
NWAmes	-0
OLDTown	-0
SWISU	-0
Sawyer	-0
SawyerW	-0
Somerst	0
StoneBr	0.0168153

```
Timber                    0
Veenker                   0
Bsmt Qual         0.0202754
```

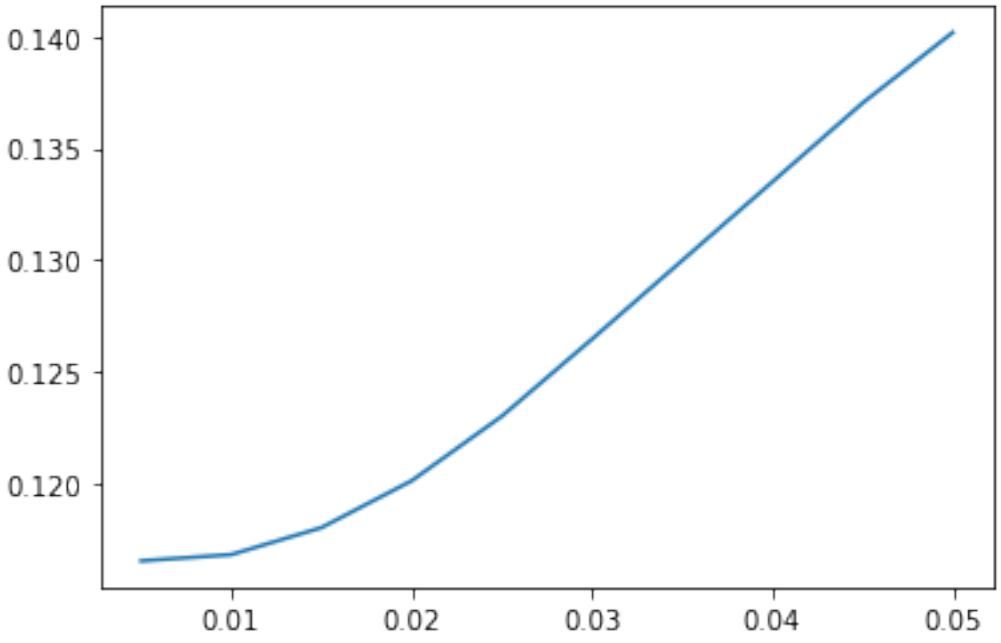
Lasso with different levels of alpha and its mse

[24]:
```
# We now consider different lambda values. The alphas are half the lambdas
alphas=[0.01/2, 0.02/2, 0.03/2, 0.04/2, 0.05/2, 0.06/2, 0.08/2, 0.09/2, 0.1/2]
mses=[]
for alpha in alphas:
    lasso=Lasso(alpha=alpha)
    lasso.fit(X_train,y_train)
    pred=lasso.predict(X_val)
    mses.append(mse(y_val,pred))
    print("lambda = " + '{:<05}'.format(alpha) + " - mse = " +␣
 ↪str(round(mse(y_val, pred),6)))
```

```
lambda = 0.005 - mse = 0.116548
lambda = 0.010 - mse = 0.116827
lambda = 0.015 - mse = 0.118033
lambda = 0.020 - mse = 0.120128
lambda = 0.025 - mse = 0.123015
lambda = 0.030 - mse = 0.126462
lambda = 0.040 - mse = 0.133492
lambda = 0.045 - mse = 0.137016
lambda = 0.050 - mse = 0.140172
```

[25]:
```
plt.plot(alphas, mses)
```

[25]:
```
[<matplotlib.lines.Line2D at 0x193b3f7fc08>]
```



Lasso regression leads to more interesting results. In the plot above you can see how the error in the validation set changes as tha value of the lasso λ increases. For small values of λ the error is actually less than when $\lambda = 0$ but as λ increases beyond about 0.03 the error starts to increase. A value of $\lambda = 0.04$ could be chosen.

1.4 References

John C. Hull, **Machine Learning in Business: An Introduction to the World of Data Science**, Amazon, 2019.

Paul Wilmott, **Machine Learning: An Applied Mathematics Introduction**, Panda Ohana Publishing, 2019.