Santiago_Gabriela

March 12, 2022

[1]:
```
#This jupyter notebook was prepared by "Gabriela Santiago".
```

[2]:
```
###Load Data and perform general EDA
```

[3]:
```
##import libraries: pandas, numpy, matplotlib (set %matplotlib inline),␣
↪matplotlib's pyplot, seaborn, missingno, scipy's stats, sklearn (1 pt)
```

[4]:
```
%matplotlib inline
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import scipy.stats as st
import sklearn
import missingno as msno
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error
from sklearn.datasets import make_regression
from sklearn.linear_model import SGDRegressor
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import Pipeline
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso
from sklearn.linear_model import ElasticNet
```

[5]:
```
##import the data to a dataframe and show the count of rows and columns
```

[6]:
```
df = pd.read_csv('ecommarce.csv')
print("Rows: ", len(df))
print("Columns: ", len(df.columns))
```

```
Rows:  500
Columns:  9
```

```
[7]: ##Show the top 5 and last 5 rows (1 pt)
```

```
[8]: print("Top 5: \n", df.head())
```

```
Top 5:
    Unnamed: 0         Email                                          Address  \
0           0  adkv@ota.com               89280 Mark Lane\nNew John, MN 16131
1           1  gjun@syj.com  363 Amanda Cliff Apt. 638\nWest Angela, KS 31437
2           2  qjyr@pkk.com           62008 Adam Lodge\nLake Pamela, NY 30677
3           3  jkiu@xsb.com         950 Tami Island\nLake Aimeeview, MT 93614
4           4  stvb@niy.com       08254 Kelly Squares\nNorth Lauren, AR 78382

           Credit Card  Avg. Session Length  Time on App  Time on Website  \
0     3544288738428794            35.497268    13.655651        40.577668
1     6546228325389133            32.926272    12.109461        38.268959
2  4406395951712628314            34.000915    12.330278        38.110597
3       30334036663133            35.305557    14.717514        37.721283
4     3582080469154498            34.330673    13.795189        38.536653

   Length of Membership  Yearly Amount Spent
0              4.582621           588.951054
1              3.164034           393.204933
2              4.604543           488.547505
3              3.620179           582.852344
4              4.946308           600.406092
```

```
[9]: print("Last 5: \n", df.tail())
```

```
Last 5:
      Unnamed: 0         Email  \
495         495  xskz@gwj.com
496         496  awrc@iok.com
497         497  pndt@jyr.com
498         498  zvtz@onj.com
499         499  phqb@nlg.com

                                               Address          Credit Card  \
495            7083 Wallace Rest\nNew Trevor, NM 70240       30206742023085
496  663 Christopher Garden\nLake Carrieberg, PA 70796     6011536844623717
497            1555 Chen Road\nBergerchester, NH 46418  4086276267550896697
498  5568 Robert Station Apt. 030\nTurnerstad, GA 9…       36218092488069
499       424 Mark Junctions\nDarrellchester, TX 09088     5427200269739116

     Avg. Session Length  Time on App  Time on Website  Length of Membership  \
495            34.237660    14.566160        37.417985              4.246573
496            35.702529    12.695736        38.190268              4.076526
497            33.646777    12.499409        39.332576              5.458264
```

```
498          34.322501   13.391423      37.840086             2.836485
499          34.715981   13.418808      36.771016             3.235160

     Yearly Amount Spent
495           574.847438
496           530.049004
497           552.620145
498           457.469510
499           498.778642
```

[10]:
```
##call the describe method of dataframe to see some summary statistics of the
↪numerical columns. (1 pt)
```

[11]:
```
df.describe()
```

[11]:
```
       Unnamed: 0   Credit Card  Avg. Session Length  Time on App  \
count  500.000000  5.000000e+02           500.000000   500.000000
mean   249.500000  3.706324e+17            34.053194    13.052488
std    144.481833  1.235588e+18             0.992563     0.994216
min      0.000000  5.018057e+11            30.532429     9.508152
25%    124.750000  3.683275e+13            33.341822    12.388153
50%    249.500000  3.513612e+15            34.082008    12.983231
75%    374.250000  4.777131e+15            34.711985    13.753850
max    499.000000  4.959148e+18            37.139662    16.126994

       Time on Website  Length of Membership  Yearly Amount Spent
count       500.000000            500.000000           500.000000
mean         38.060445              4.033462           500.314038
std           1.010489              0.999278            79.314782
min          34.913847              0.769901           257.670582
25%          37.349257              3.430450           446.038277
50%          38.069367              4.033975           499.887875
75%          38.716432              4.626502           550.313828
max          41.005182              7.422689           766.518462
```

[12]:
```
#Explain in words about the description of any two variables (1 pt)
```

Answer: Looks like this function is useful for the numerical values such as avg session length, time spent on app/website, length of membership and yearly amount spent.

Answer: The credit card number description information should be useless as it is unique for each person which means there will be no correlation with other variables.

[13]:
```
##Show any missing value analysis  (1 pt)
```

[14]:
```
nulls = df.isnull().sum().to_frame('nulls')
nulls.sort_values("nulls", inplace = True, ascending = False)
for index, row in nulls.iterrows():
    print(index, row[0])
```

```
Unnamed: 0 0
Email 0
Address 0
Credit Card 0
Avg. Session Length 0
Time on App 0
Time on Website 0
Length of Membership 0
Yearly Amount Spent 0
```

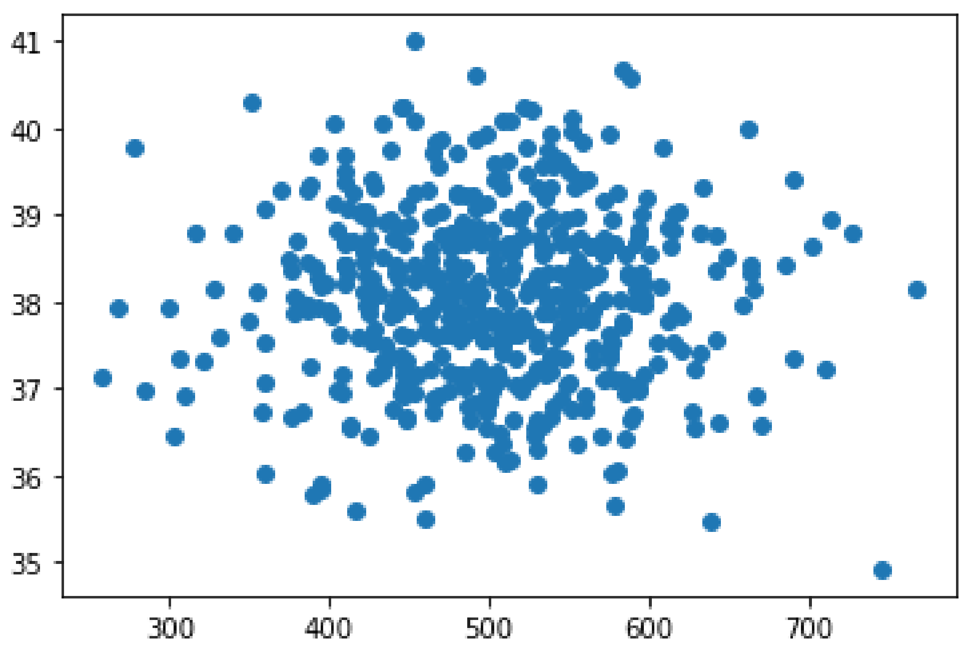
Answer: There doesn't seem to be any missing values in this dataset

[15]:
```
##Plot various scatter plots to understand the data:
```

[16]:
```
#Yearly amount Spent vs Time on Website
```

[17]:
```
plt.scatter(data = df, x = 'Yearly Amount Spent', y = 'Time on Website')
```

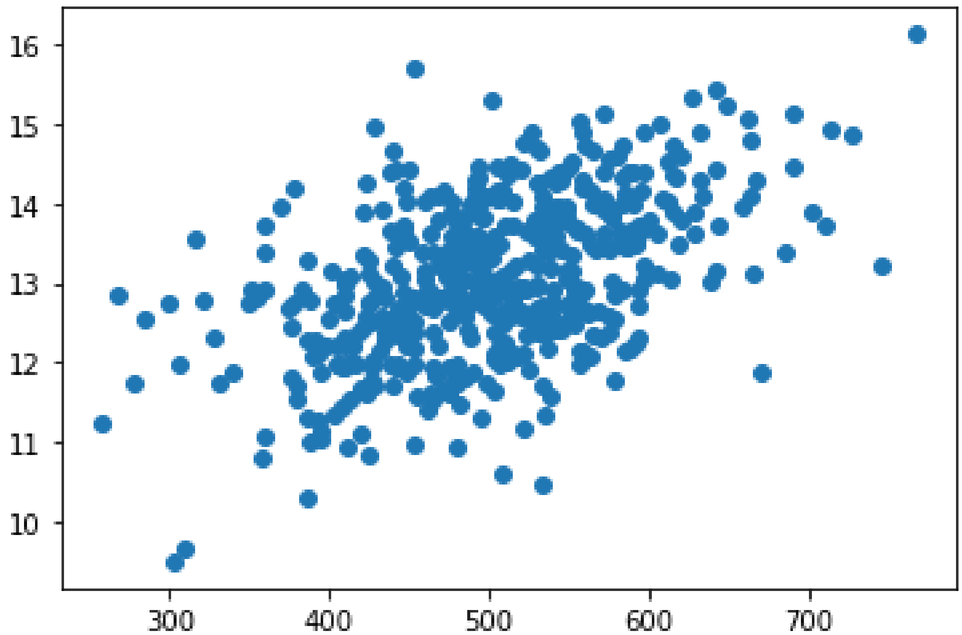
[17]: <matplotlib.collections.PathCollection at 0x7f962b26a1f0>



[18]:
```
#Yearly amount Spent vs Time on App
```

[19]:
```
plt.scatter(data = df, x = 'Yearly Amount Spent', y = 'Time on App')
```

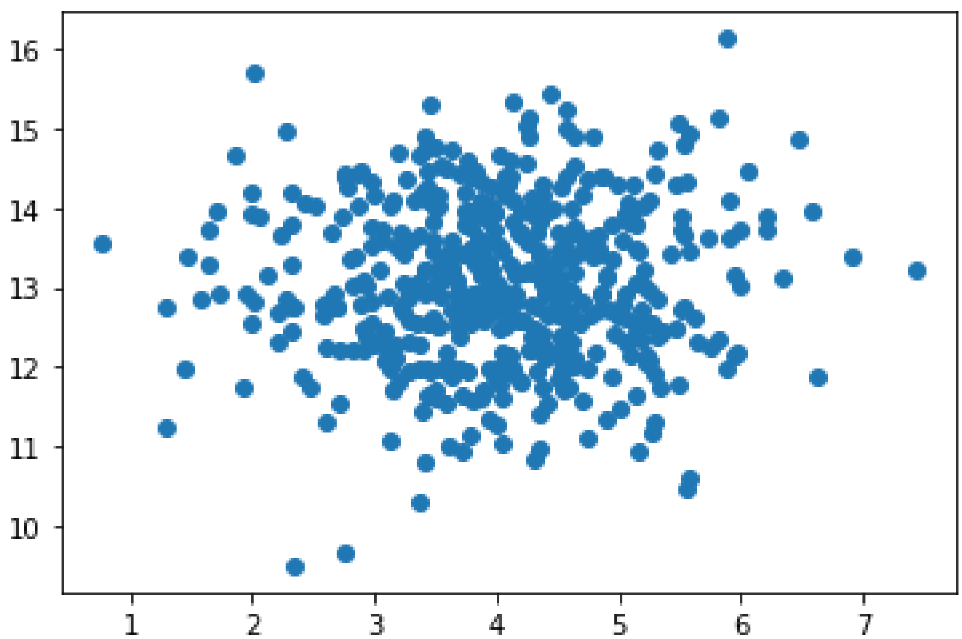
[19]: <matplotlib.collections.PathCollection at 0x7f962b395820>



[20]:
```
#Length of membership vs Time on App
```

[21]:
```
plt.scatter(data = df, x = 'Length of Membership', y = 'Time on App')
```

[21]: <matplotlib.collections.PathCollection at 0x7f962b4f5d00>

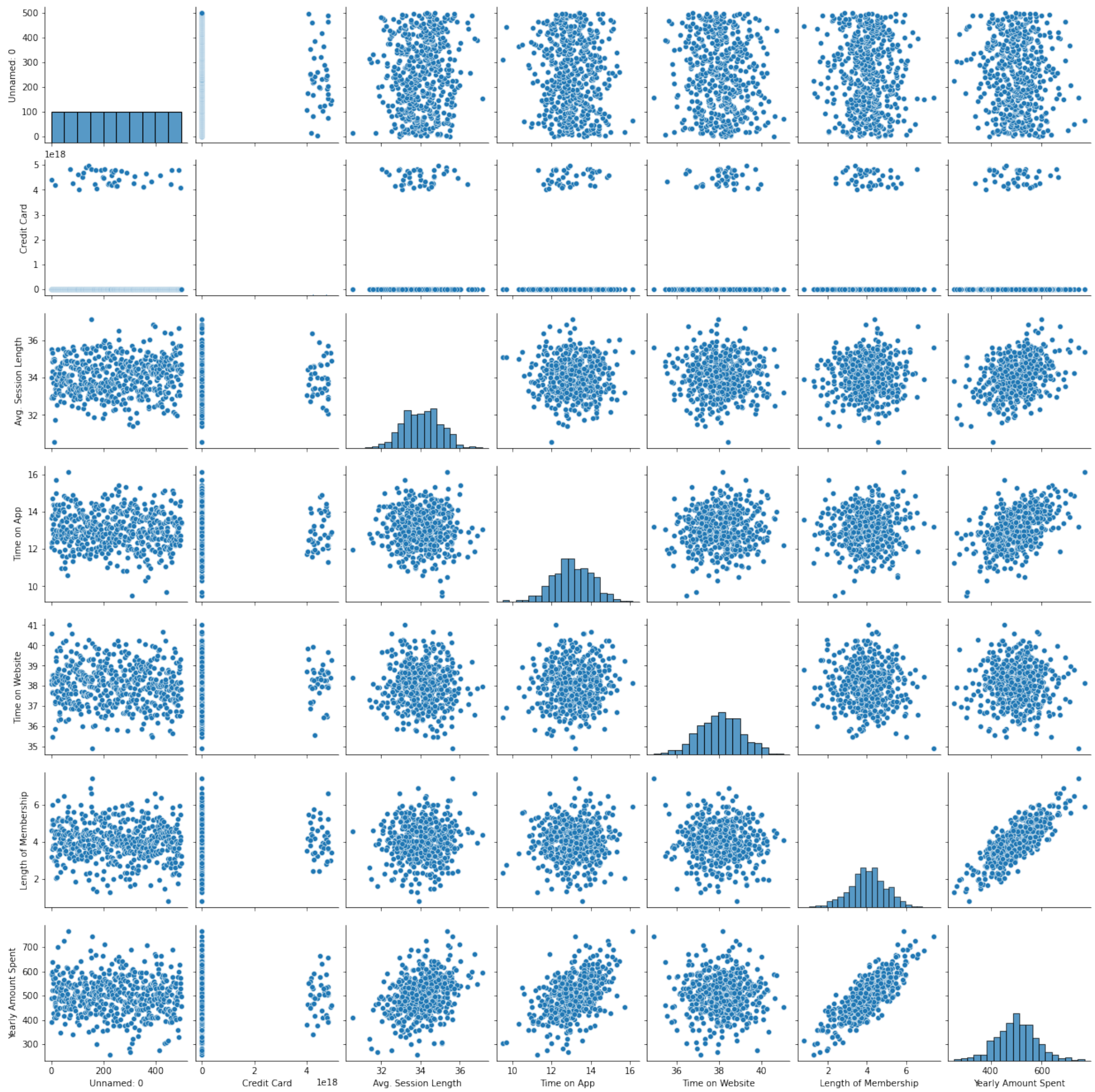


[22]:

```python
#Generate sns pairplot. Based on the plots, what feature is mostly correlated with the yearly amount spent?
```

[23]:

```python
pairPlot = sns.pairplot(df)
```

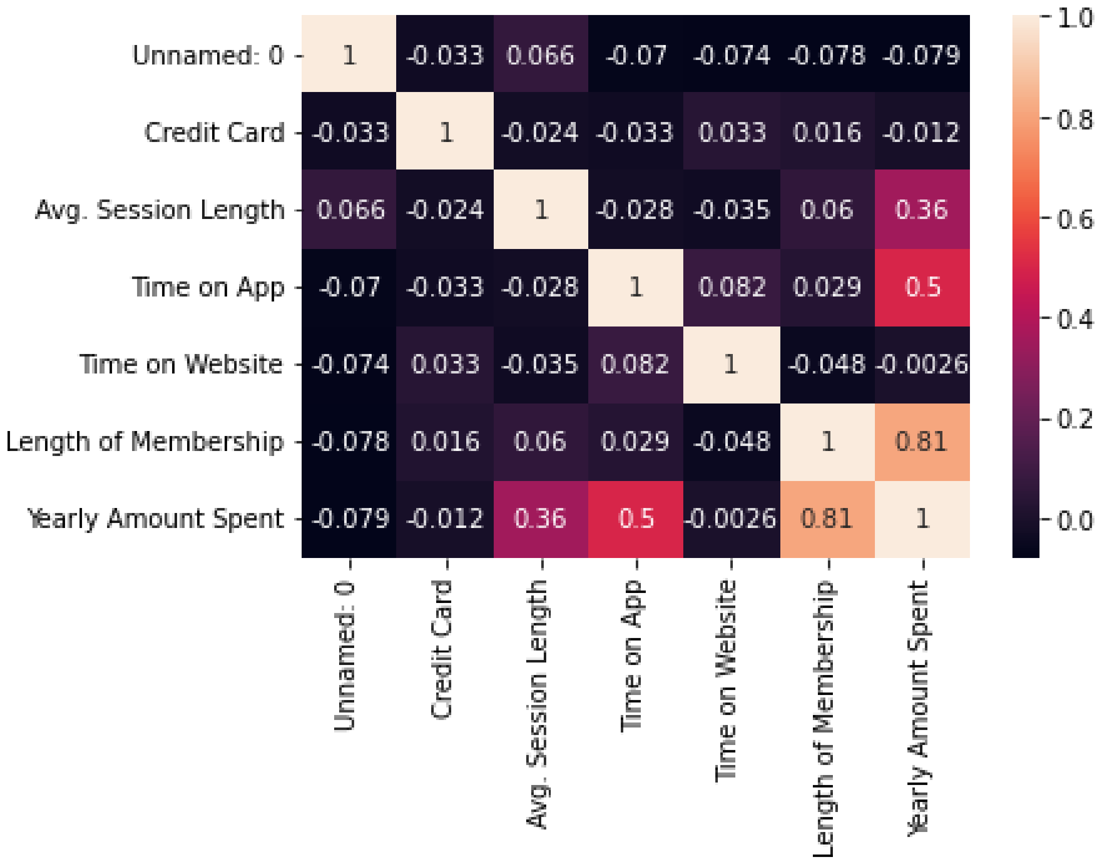


Answer: By far, the 'length of membership' feature is the most correlated with the 'yearly amount spent' feature as they both increase as the other increases

[24]:

```python
#Also, plot sns heatmap based on correlation with annot=True and discuss which columns must be removed based on that and which column is mostly interesting and related to Yearly Amount Spent?
```

[25]:
```
numerical = df.select_dtypes(include=[np.number])
correlation = numerical.corr()
ax = sns.heatmap(correlation, annot = True)
```



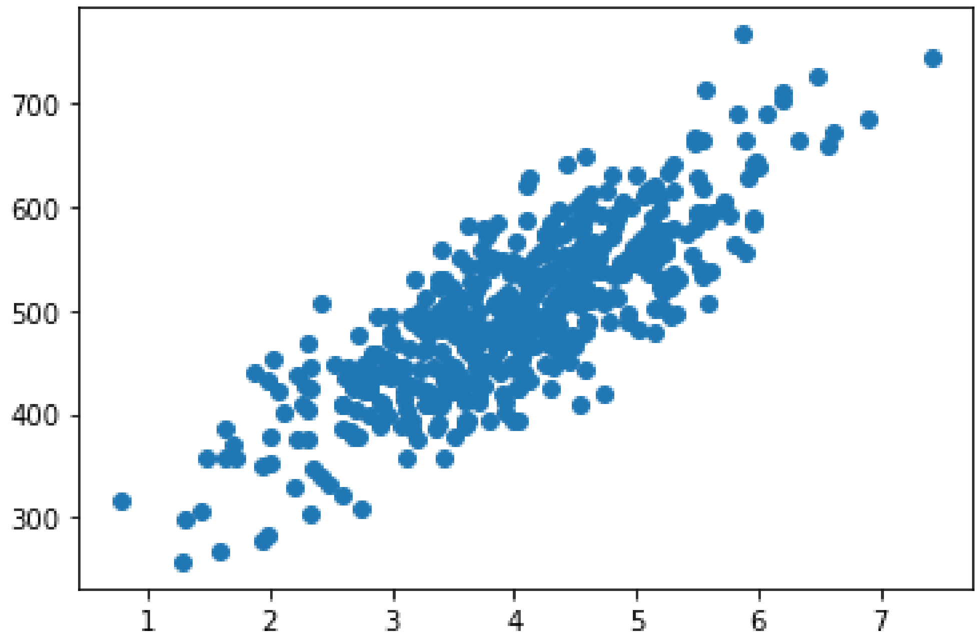
Answer: Columns that should be removed are 'unnamed' and 'Credit Card'. This is because they have little to no correlation with any other features of the dataset, especially the target value. I will also be removing 'email' as it has no correlation, but that is unrelated to the heatmap.

Answer: The most interesting feature with the highest correlation to the yearly amount spent is the 'Length of Membership' feature with a 0.81 score for correlation

[26]:
```
#Generate a scatter plot with the interesting column you found in the last step␣
↪against the Yearly Amount Spent
```

[27]:
```
plt.scatter(data = df, x = 'Length of Membership', y = 'Yearly Amount Spent')
```

[27]: <matplotlib.collections.PathCollection at 0x7f9611c4f910>



[28]:
```
###Feature Selection and Pre-processing
```

[29]:
```
##Based on the EDA and null analysis, drop the unnecessary columns for the␣
↪regression
```

[30]:
```
df = df.drop(columns = ['Credit Card', 'Unnamed: 0', 'Email'])
```

[31]:
```
###X/Y and Training/Test Split
```

[32]:
```
##Use sklearn's train_test_split to split the data set into training and test␣
↪sets. There should be 30% records in the test set. The random_stat should be␣
↪101
```

[33]:
```
numerical = df.select_dtypes(include=[np.number])
X = numerical
X = X.drop(columns = ['Yearly Amount Spent'])
y = df['Yearly Amount Spent']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3,␣
↪random_state = 101)
yte = y_test.to_numpy()
```

[34]:
```
##As we will be doing gradient descent as well as some other regression␣
↪technique, scaling the data set is important. So, use sklearn's␣
↪StandardScaler for scalling the X of training and test sets. But don't do it␣
↪for y(target) train and test.
```

[35]:
```
scaler = StandardScaler()
SX_train = scaler.fit_transform(X_train)
SX_test = scaler.fit_transform(X_test)
```

[36]:
```
###Training Linear Model using SKLearn's LinearRegression
```

[37]:
```
##Train a linear model using Sklearn's LinearRegression
```

[38]:
```
lin_reg = LinearRegression()
lin_reg.fit(SX_train, y_train)
```

[38]: LinearRegression()

[39]:
```
xt = SX_train
xta = np.array(xt)
xta1 = np.c_[np.ones(len(xta)), xta]
theta_best_svd, residuals, rank, s = np.linalg.lstsq(xta1, y_train, rcond=1e-6)
```

[40]:
```
##After training, show the coefficients and intercept
```

[41]:
```
print("Intercept: ", lin_reg.intercept_)
print("Coefficiant: ", lin_reg.coef_)
```

```
Intercept:  499.7231164913073
Coefficiant:  [26.04265125 36.67425683  0.18503853 60.20236045]
```

[42]:
```
print("Least squares thetas: ")
print(theta_best_svd.reshape(5,1))
```

```
Least squares thetas:
[[4.99723116e+02]
 [2.60426512e+01]
 [3.66742568e+01]
 [1.85038527e-01]
 [6.02023604e+01]]
```

[43]:
```
##Predict for the test data
```

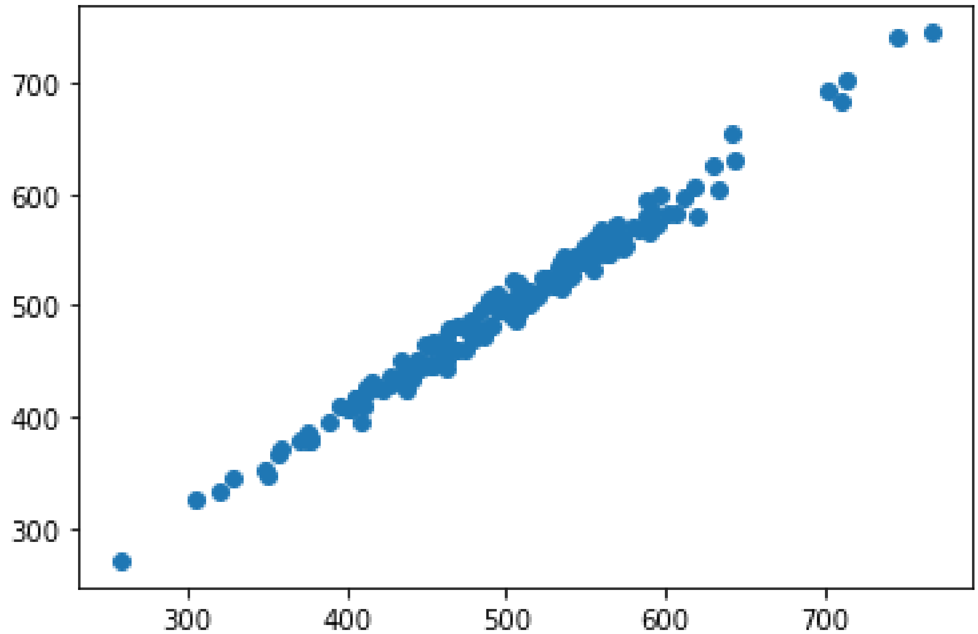
[44]:
```
y_pred = lin_reg.predict(SX_test)
print("Predicted: ", y_pred[5])
print("Actual: ", yte[5])
```

```
Predicted:  543.4923417319907
Actual:  536.4807751896418
```

[45]:
```
##Generate a scatter plot that shows the Y test on x-axis and y predicted in␣
↪y-axis
```

[46]:
```
plt.scatter(x = yte, y = y_pred)
```

[46]: <matplotlib.collections.PathCollection at 0x7f9611c90df0>



[47]:
```
##Use sklearn's metrics to print the value of MAE, MSE, RMSE, and R^2
```

[48]:
```
print("MAE: %.2f" % mean_absolute_error(y_test, y_pred))
print("MSE: %.2f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.2f" % mean_squared_error(y_test, y_pred, squared = False))
print("R^2: %.2f" % r2_score(y_test, y_pred))
```

```
MAE: 9.37
MSE: 133.27
RMSE: 11.54
R^2: 0.98
```

[49]:
```
##Interpretation: Interpret the coefficient and which coefficient belongs to␣
↪which feature and based on that explain any strategy that should help the␣
↪business
```

Answer: The first theta belongs to 'Avg session length', the second 'Time on App', third 'Time on website', and the fourth 'Length of membership'

The coefficients prove that the fourth feature, 'Length of membership' has the biggest effect on the target value, as it is very high, and the third feature, 'Time on website', has almost no correlation with the target value as it's coefficient is close to zero. As a business, is is good to know that the length of membership plays a strong part in the yearly amount spent at your business.

[50]:
```
###Normal Equation
```

[51]:
```
##Implement Normal Equation and find best_theta values based on the training set
```

[52]:
```
yt = y_train
yta = np.array(yt)
yta = yta.reshape(350, 1)
best_theta = np.linalg.inv(xta1.T.dot(xta1)).dot(xta1.T).dot(yta)
```

[53]:
```
##Display the theta values. Are they very close to the sklearn's linear␣
↪regression?
```

[54]:
```
print(best_theta)
```

```
[[4.99723116e+02]
 [2.60426512e+01]
 [3.66742568e+01]
 [1.85038527e-01]
 [6.02023604e+01]]
```

Answer: Very close to the least squares portion of the linear regression section

[55]:
```
##Prepare the test set before prediction
```

[56]:
```
xtest = SX_test
xtest = np.array(xtest)
x_new_b = np.c_[np.ones(len(xtest)), xtest]
```

[57]:
```
##Perform prediction for the test set
```

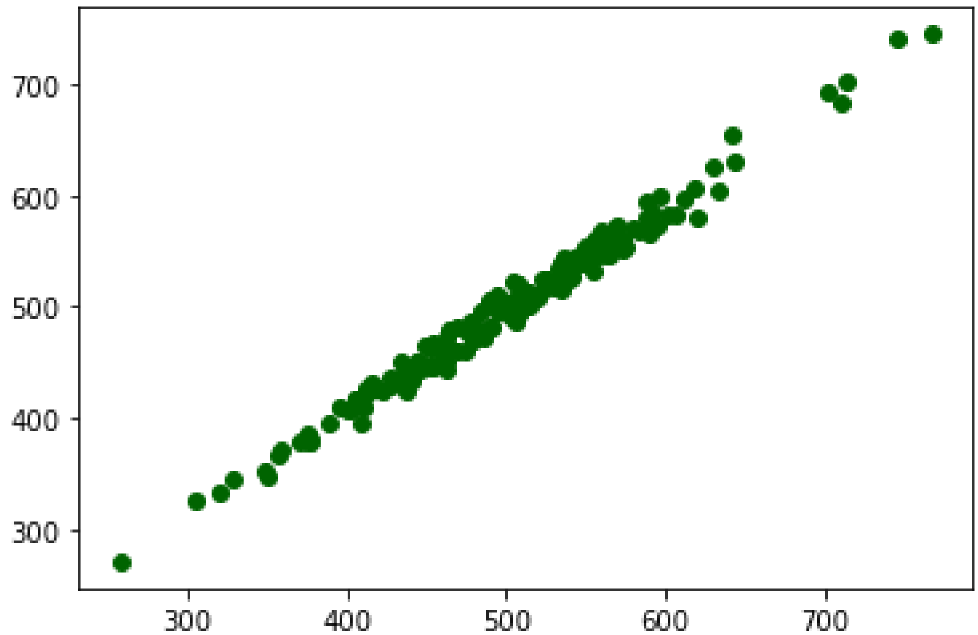
[58]:
```
predict_value = x_new_b.dot(best_theta)
y_pred2 = predict_value
print("Predicted: ", y_pred2[2])
print("Actual: ", yte[2])
```

```
Predicted:  [411.28893669]
Actual:  411.06961105998295
```

[59]:
```
##Generate a scatter plot that shows the Y test on x-axis and y predicted in␣
↪y-axis
```

[60]:
```
plt.scatter(x = yte, y = y_pred2, c = 'darkgreen')
```

[60]:
```
<matplotlib.collections.PathCollection at 0x7f9611b3b910>
```



[61]:
```
##Use sklearn's metrics to print the value of MAE, MSE, RMSE, and R^2
```

[62]:
```
print("MAE: %.2f" % mean_absolute_error(y_test, y_pred2))
print("MSE: %.2f" % mean_squared_error(y_test, y_pred2))
print("RMSE: %.2f" % mean_squared_error(y_test, y_pred2, squared = False))
print("R^2: %.2f" % r2_score(y_test, y_pred2))
```

```
MAE: 9.37
MSE: 133.27
RMSE: 11.54
R^2: 0.98
```

[63]:
```
##What is the limitation of using the Normal equation for regression?
```

Answer: The larger the dataset or the more features the dataset has, the slower and more expensive it will be to use this method for regression, as computing the inverse of X transpose X takes longer the bigger the dataset is.

[64]:
```
###Batch Gradient Descent
```

[65]:
```
##Implement Batch Gradient Descent based on the way we have learned in the␣
↪class. You can playwith eta and n_iterations and should set to reasonable␣
↪eta and number of iterations so that you can get the thetas close to Normal␣
↪equation's theta
```

[66]:
```
cost_list = []
epoch_list = []
predicted_list = []


n_iterations = 100
m = 100
eta = 0.2

theta2 = np.random.randn(5,1)
for iteration in range(n_iterations):
    gradients = 2/m * xta1.T.dot(xta1.dot(theta2) - yta)
    theta2 = theta2 - eta * gradients

    y_predicted = np.dot(theta2.T, xta1.T)
    cost = np.mean(np.square(yta-y_predicted))

    if iteration%10==0:
      cost_list.append(cost)
      epoch_list.append(iteration)
```

[67]:
```
##Display the theta values. Are they very close to the sklearn's linear␣
↪regression?
```

[68]:
```
print(theta2)
```

```
[[4.99723116e+02]
 [2.60426512e+01]
 [3.66742568e+01]
 [1.85038527e-01]
 [6.02023604e+01]]
```

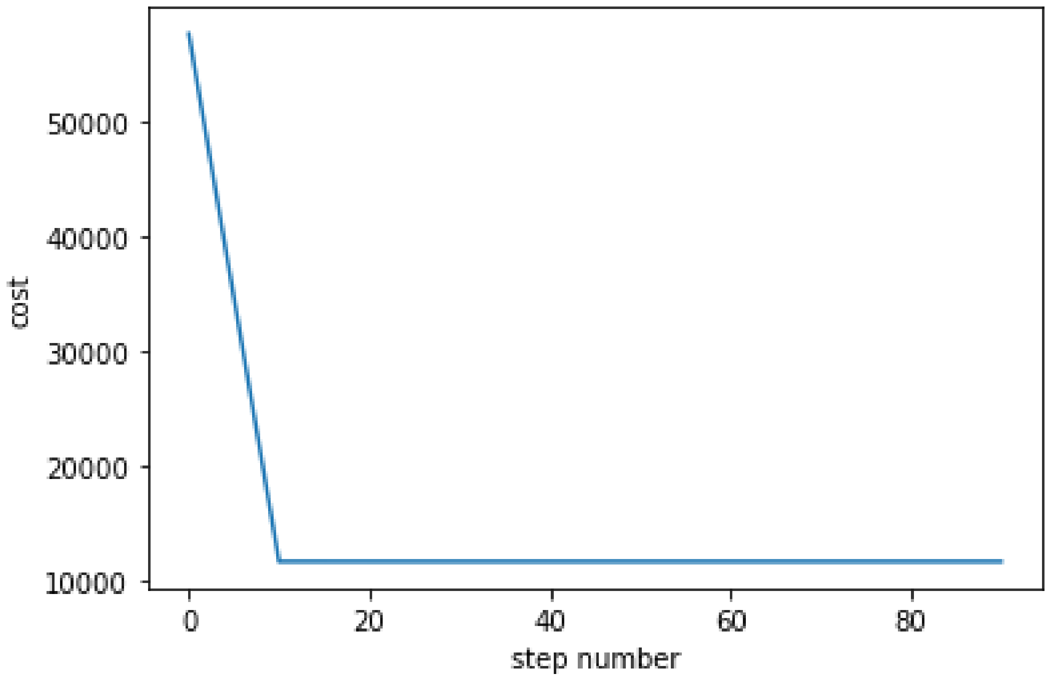
Answer: They are similar to the normal equation's thetas and the least squares portion of the linear regression section.

[69]:
```
##Also plot step number (in x-axis) against the cost(y axis).
```

[70]:
```
import matplotlib.pyplot as plt

plt.xlabel("step number")
plt.ylabel("cost")
plt.plot(epoch_list,cost_list)
```

[70]:
```
[<matplotlib.lines.Line2D at 0x7f96131f2ca0>]
```



[71]:
```
##Perform prediction for the test set
```

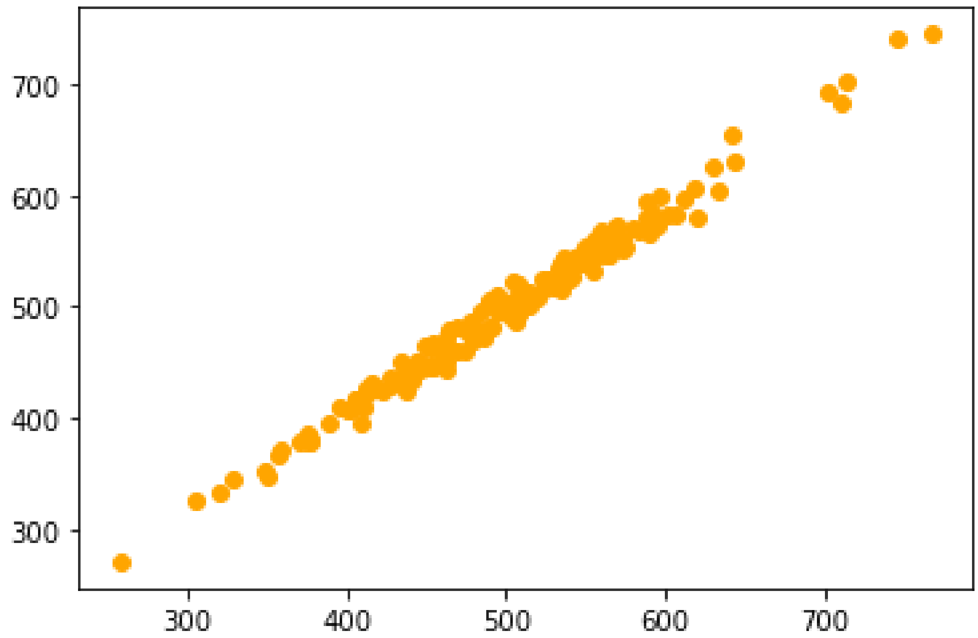
[72]:
```
y_pred3 = x_new_b.dot(theta2)
print("Predicted: ", y_pred3[120])
print("Actual: ", yte[120])
```

```
Predicted:  [346.76957459]
Actual:  351.0582001638451
```

[73]:
```
##Generate a scatter plot that shows the Y test on the x-axis and y predicted␣
↪in the y-axis
```

[74]:
```
plt.scatter(x = yte, y = y_pred3, c = 'orange')
```

[74]:
```
<matplotlib.collections.PathCollection at 0x7f96134cbcd0>
```



[75]:
```
##Use sklearn's metrics to print the value of MAE, MSE, RMSE, and R^2
```

[76]:
```
print("MAE: %.2f" % mean_absolute_error(y_test, y_pred3))
print("MSE: %.2f" % mean_squared_error(y_test, y_pred3))
print("RMSE: %.2f" % mean_squared_error(y_test, y_pred3, squared = False))
print("R^2: %.2f" % r2_score(y_test, y_pred3))
```

```
MAE: 9.37
MSE: 133.27
RMSE: 11.54
R^2: 0.98
```

[77]:
```
##Short Question: How do derivatives help in the process of gradient descent?
```

Answer: It helps to determine the slope of the curve. Since this slope and eta will be multiplied, the smaller the slope, the smaller the steps forward until the minimum is found.

[78]:
```
##Short Question: What are the benefits and the limitations of using batch␣
↪gradient descent?
```

Answer:

Limitations: It can sometimes get snagged onto a local minima. Also, you might need additional memory using this method if the dataset is too big, as it processes the entire batch at once.

Benefits: More efficient than stochastic gradient descent.

[79]:
```
###Stochastic Gradient Descent
```

[80]:
```
##Implement Stochastic Gradient Descent and train our data set. You must have␣
↪to use learning_schedule
```

[81]:
```
theta_path_sgd = []
m = len(xta1)
np.random.seed(42)
```

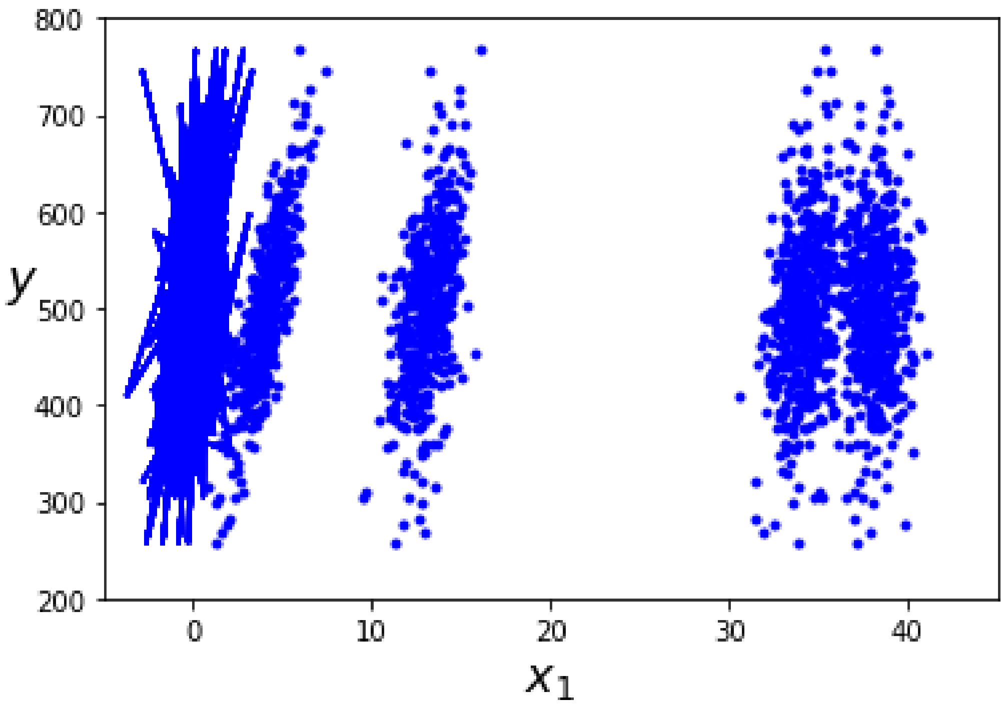
[82]:
```
n_epochs = 200
t0, t1 = 5, 50

def learning_schedule(t):
    return t0 / (t + t1)

theta3 = np.random.randn(5,1)

for epoch in range(n_epochs):
    for i in range(m):
        if epoch == 0 and i < 20:
            y_predict = x_new_b.dot(theta3)
            style = "b-" if i > 0 else "r--"
            plt.plot(xtest, yte, style)
        random_index = np.random.randint(m)
        xi = xta1[random_index:random_index+1]
        yi = yta[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta3) - yi)
        eta = learning_schedule(epoch * m + i)
        theta3 = theta3 - eta * gradients
        theta_path_sgd.append(theta3)

plt.plot(X, y, "b.")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([-5, 45, 200, 800])
plt.show()
```



[83]:
```
##Display the theta values. Are they very close to the sklearn's linear␣
↪regression?
```

[84]:
```
print(theta3)
```

```
[[4.99746434e+02]
 [2.60127085e+01]
 [3.66293647e+01]
 [3.27374021e-01]
 [6.02296448e+01]]
```

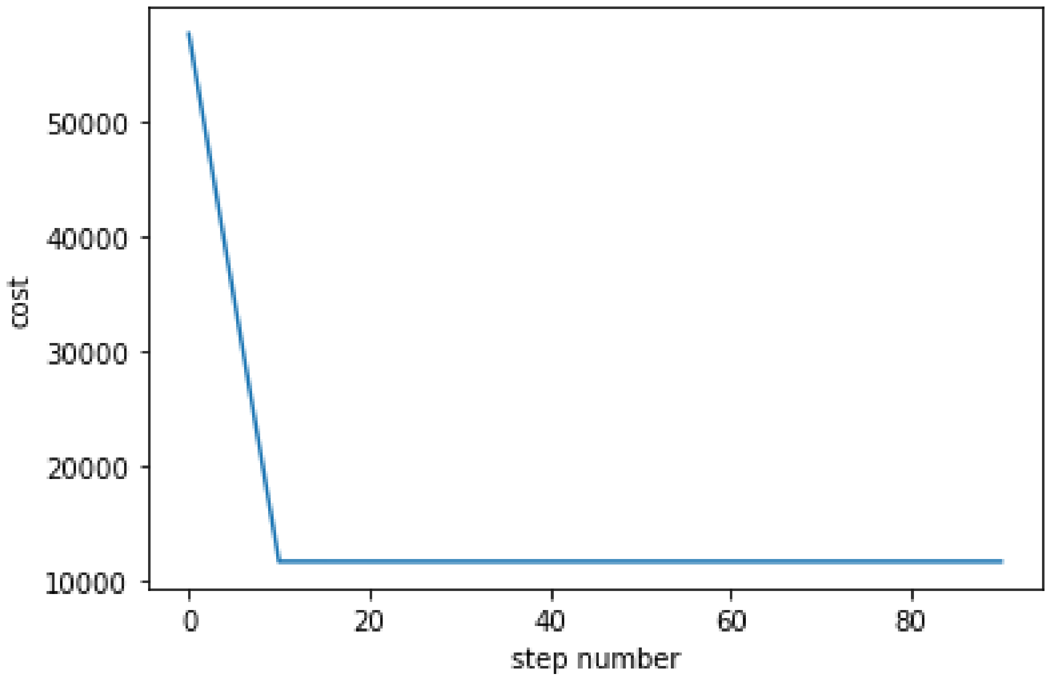
Answer: Pretty close to the normal equation, batch gradient descent, and the least squares portion of the linear regression section.

[85]:
```
##Also plot step number (in x-axis) against cost(y-axis).
```

[86]:
```
import matplotlib.pyplot as plt

plt.xlabel("step number")
plt.ylabel("cost")
plt.plot(epoch_list,cost_list)
```

[86]: `[<matplotlib.lines.Line2D at 0x7f96144e93a0>]`



[87]:
```
##Perform Prediction for the test set
```

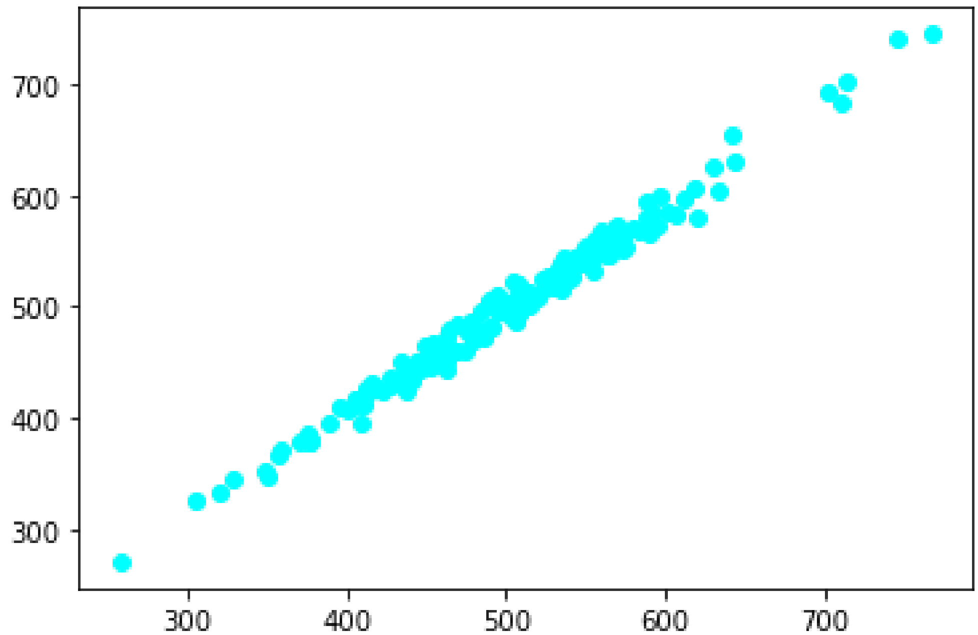
[88]:
```
y_pred4 = x_new_b.dot(theta3)
print("Predicted: ", y_pred4[3])
print("Actual: ", yte[3])
```

```
Predicted:  [583.81664507]
Actual:  600.4060920457634
```

[89]:
```
##Generate a scatter plot that shows the Y test on the x-axis and y predicted␣
↪in the y-axis
```

[90]:
```
plt.scatter(x = y_test, y = y_pred4, c = 'cyan')
```

[90]:
```
<matplotlib.collections.PathCollection at 0x7f9614636fa0>
```




```
[91]: ##Use sklearn's metrics to print the value of MAE, MSE, RMSE, and R^2
```

```
[92]: print("MAE: %.2f" % mean_absolute_error(y_test, y_pred4))
      print("MSE: %.2f" % mean_squared_error(y_test, y_pred4))
      print("RMSE: %.2f" % mean_squared_error(y_test, y_pred4, squared = False))
      print("R^2: %.2f" % r2_score(y_test, y_pred4))
```

```
MAE: 9.37
MSE: 133.12
RMSE: 11.54
R^2: 0.98
```

```
[93]: ##Short Question: What are the benefits and the limitations of using Stochastic␣
      ↪gradient descent?
```

Answer:

Benefits: The noisy update process allows the model to go staight for the global minimum and avoid local minima.

Limitations: Frequent updating of the model can be expensive, big datasets can take a lot of time to process.

```
[94]: ###SGDRegressor from sklearn
```

```
[95]: ##Use sklearn's SGDRegressor to train a model for our data set. Put a␣
      ↪reasonable iteration and tolerance and learning steps so that we can get␣
      ↪coefficients close to normal equation
```

[96]:
```
sgd_reg = SGDRegressor(max_iter=100, tol=1e-3, penalty=None, eta0=0.1,␣
↪random_state=42)
sgd_reg.fit(xta, yta.ravel())
```

[96]:
```
SGDRegressor(eta0=0.1, max_iter=100, penalty=None, random_state=42)
```

[97]:
```
##Display the theta values. Are they very close to sklearn's linear regression?
```

[98]:
```
print("Intercept: ", sgd_reg.intercept_)
print("Coefficient: ", sgd_reg.coef_)
```

```
Intercept:  [498.70720946]
Coefficient:  [26.33410089 35.71407738  0.3486048  59.40269987]
```

Answer: Yes very similar to the linear regression but not the normal equation. I'd appreciate any insight as to why the linreg and the normal equation ended up so different for me when they both predict perfectly fine.

[99]:
```
##Predict for the test data
```

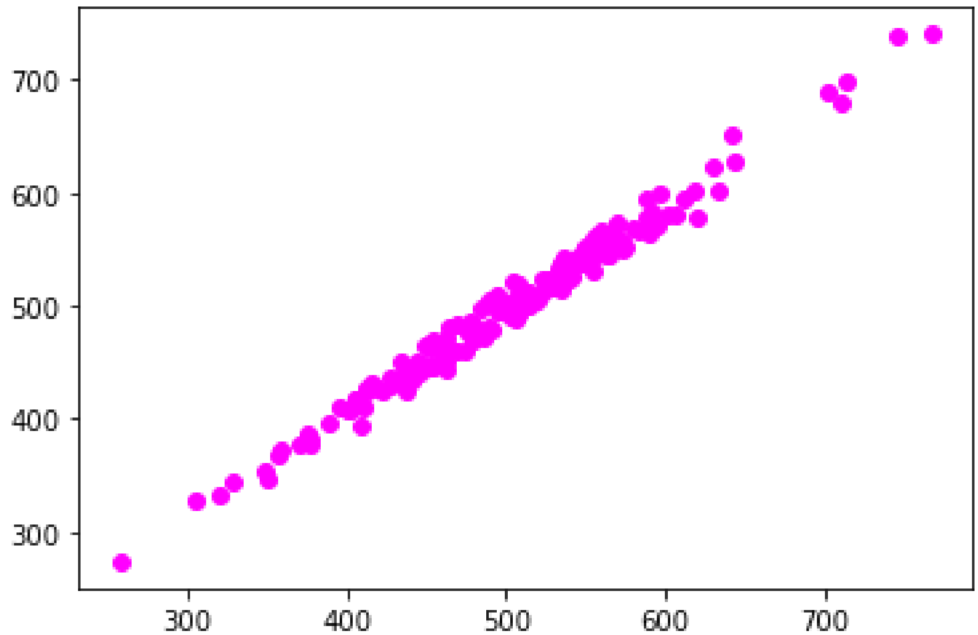
[100]:
```
y_pred5 = sgd_reg.predict(SX_test)
print("Predicted: ", y_pred5[1])
print("Actual: ", yte[1])
```

```
Predicted:  409.12627802054214
Actual:  402.0331352191061
```

[101]:
```
##Generate a scatter plot that shows the Y test on the x-axis and y predicted␣
↪in the y-axis
```

[102]:
```
plt.scatter(x = yte, y = y_pred5, c = 'magenta')
```

[102]:
```
<matplotlib.collections.PathCollection at 0x7f9614749250>
```



[103]:
```
##Use sklearn's metrics to print the value of MAE, MSE, RMSE, and R^2
```

[104]:
```
print("MAE: %.2f" % mean_absolute_error(y_test, y_pred5))
print("MSE: %.2f" % mean_squared_error(y_test, y_pred5))
print("RMSE: %.2f" % mean_squared_error(y_test, y_pred5, squared = False))
print("R^2: %.2f" % r2_score(y_test, y_pred5))
```

```
MAE: 10.18
MSE: 157.71
RMSE: 12.56
R^2: 0.98
```

[105]:
```
###Mini-batch Gradient Descent
```

[106]:
```
##Briefly explain how mini-batch can overcome the limitations of Batch gradient␣
↪descent and SGD.
```

Answer: Mini-batch is both efficient and robust as the model update frequency is higher, allowing for a more robust convergence, and it's smaller batches allow for the process to be far more efficient.

[107]:
```
###Polynomial of degree 2
```

[108]:
```
##Use sklearn's Polynomial features to degree = 2 on our training and test set
```

[109]:
```
poly_features = PolynomialFeatures(degree = 2, include_bias = False)
X_polytr = poly_features.fit_transform(SX_train)
X_polyte = poly_features.fit_transform(SX_test)
```

[110]:
```
##Use linearRegression on the new polynomial features
```

[111]:
```
lin_reg.fit(X_polytr, yta)
lin_reg.intercept_, lin_reg.coef_
```

[111]:
```
(array([499.68658573]),
 array([[ 2.59548427e+01,  3.67216721e+01,  1.62102271e-01,
          6.02168680e+01, -8.56590069e-01, -1.56166087e-01,
         -6.78266356e-02,  1.52385983e-01,  3.69981677e-01,
         -2.06815454e-01,  3.94911158e-02,  5.53534986e-01,
         -3.47157889e-01, -4.14356771e-02]]))
```

[112]:
```
##Predict for test set
```

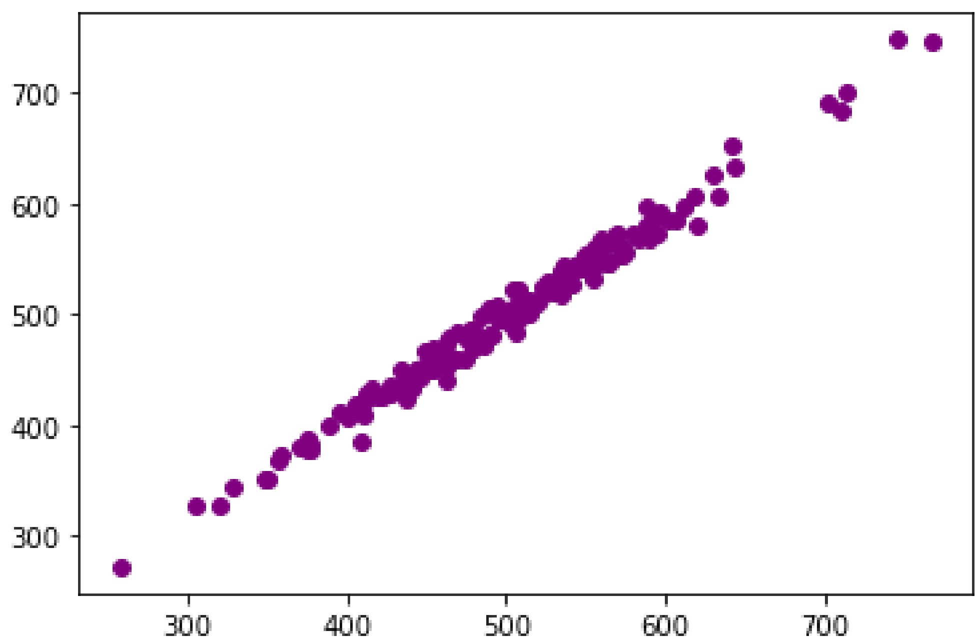
[113]:
```
y_pred6 = lin_reg.predict(X_polyte)
print("Predicted: ", y_pred6[2])
print("Actual: ", yte[2])
```

```
Predicted:  [409.6083498]
Actual:  411.06961105998295
```

[114]:
```
##Generate a scatter plot that shows the Y test on the x-axis and y predicted␣
↪in the y-axis
```

[115]:
```
plt.scatter(x = yte, y = y_pred6, c = 'purple')
```

[115]:
```
<matplotlib.collections.PathCollection at 0x7f96148333d0>
```



[116]:
```
##Use sklearn's metrics to print the value of MAE, MSE, RMSE, and R^2
```

[117]:
```
print("MAE: %.2f" % mean_absolute_error(y_test, y_pred6))
print("MSE: %.2f" % mean_squared_error(y_test, y_pred6))
print("RMSE: %.2f" % mean_squared_error(y_test, y_pred6, squared = False))
print("R^2: %.2f" % r2_score(y_test, y_pred6))
```

```
MAE: 9.44
MSE: 137.68
RMSE: 11.73
R^2: 0.98
```

[118]:
```
###Polynomial of degree 3
```

[119]:
```
##Use sklearn's Polynomial features to degree = 3 on our training and test set
```

[120]:
```
poly_features2 = PolynomialFeatures(degree = 3, include_bias = False)
X_polytr2 = poly_features2.fit_transform(SX_train)
X_polyte2 = poly_features2.fit_transform(SX_test)
```

[121]:
```
##Use linearRegression on the new polynomial features
```

[122]:
```
lin_reg.fit(X_polytr2, yta)
lin_reg.intercept_, lin_reg.coef_
```

[122]:
```
(array([499.76435904]),
 array([[25.99640364, 36.09811543,  1.67232402, 60.53178834, -0.92108551,
         -0.17770036,  0.14643857,  0.60308084,  0.19406201, -0.25124219,
         -0.20345443,  0.56307476, -0.66408957,  0.07280879,  0.08639436,
         -0.34320665, -0.10776856,  0.41106478,  0.12227022, -0.59613933,
          0.27396066, -0.26736033,  0.63518173, -0.35425134, -0.08194705,
         -0.43076588, -0.25491937,  1.03317254,  0.50573609,  0.437486  ,
         -0.30896836, -0.97687431, -0.38467864,  0.14980536]]))
```

[123]:
```
##Predict for test set
```

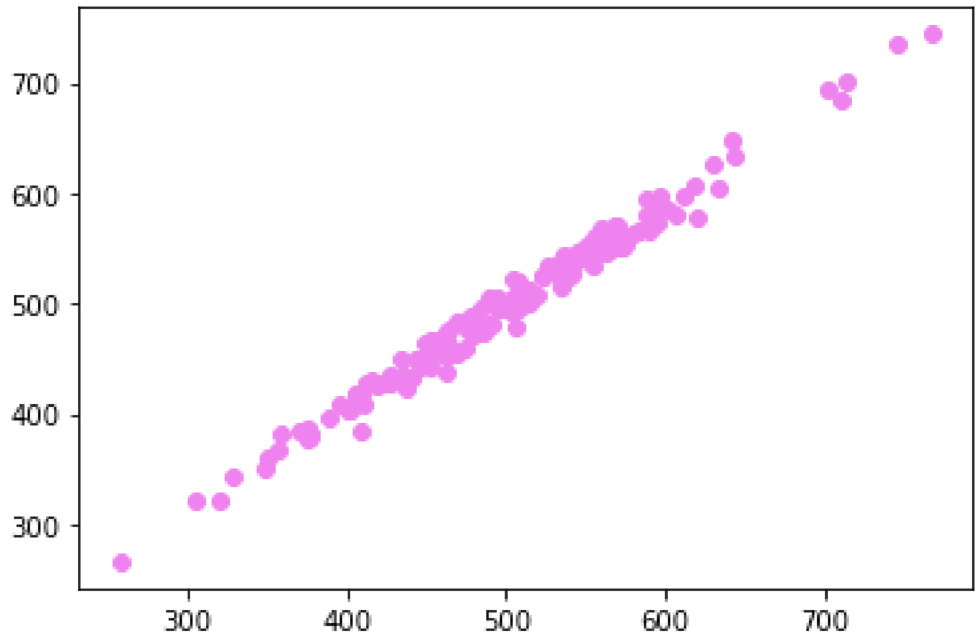
[124]:
```
y_pred7 = lin_reg.predict(X_polyte2)
print("Predicted: ", y_pred7[11])
print("Actual: ", yte[11])
```

```
Predicted:  [494.9003322]
Actual:  493.55683370047706
```

[125]:
```
##Generate a scatter plot that shows the Y test on the x-axis and y predicted␣
↪in the y-axis
```

[126]:
```
plt.scatter(x = yte, y = y_pred7, c = 'violet')
```

[126]:
```
<matplotlib.collections.PathCollection at 0x7f961511cfa0>
```



[127]:
```
##Use sklearn's metrics to print the value of MAE, MSE, RMSE, and R^2
```

[128]:
```
print("MAE: %.2f" % mean_absolute_error(y_test, y_pred7))
print("MSE: %.2f" % mean_squared_error(y_test, y_pred7))
print("RMSE: %.2f" % mean_squared_error(y_test, y_pred7, squared = False))
print("R^2: %.2f" % r2_score(y_test, y_pred7))
```

```
MAE: 9.49
MSE: 141.74
RMSE: 11.91
R^2: 0.98
```

[129]:
```
###Learning Curve
```

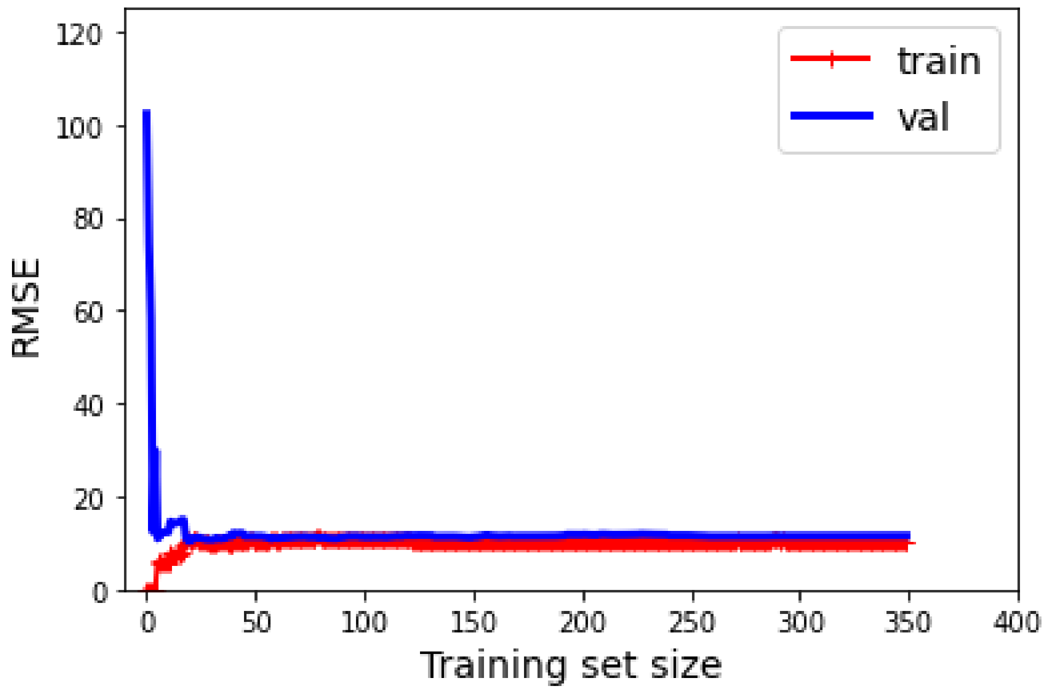
[130]:
```
def plot_learning_curves(model, X, y):
    train_errors, val_errors = [], []
    for m in range(1, len(SX_train) + 1):
        model.fit(SX_train[:m], y_train[:m])
        y_train_predict = model.predict(SX_train[:m])
        y_val_predict = model.predict(SX_test)
        train_errors.append(mean_squared_error(y_train[:m], y_train_predict))
        val_errors.append(mean_squared_error(yte, y_val_predict))

    plt.plot(np.sqrt(train_errors), "r-+", linewidth=2, label="train")
    plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="val")
    plt.legend(loc="upper right", fontsize=14)
```

```
    plt.xlabel("Training set size", fontsize=14)
    plt.ylabel("RMSE", fontsize=14)
```

[131]:
```
##Generate learning curve with linearRegression
```

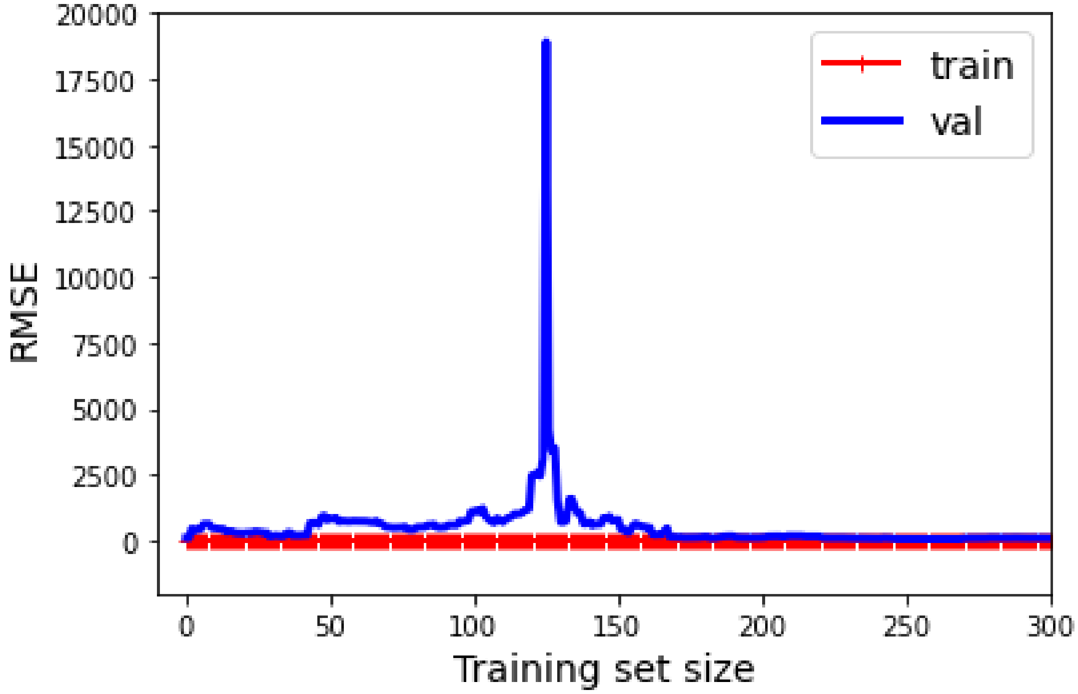
[132]:
```
lin_reg = LinearRegression()
plot_learning_curves(lin_reg, SX_train, yta)
plt.axis([-10, 400, 0, 125])
plt.show()
```



[133]:
```
##Generate learning curve with polynomial regression with degree = 5
```

[134]:
```
polynomial_regression = Pipeline([
        ("poly_features", PolynomialFeatures(degree = 5, include_bias = False)),
        ("lin_reg", LinearRegression()),
    ])

plot_learning_curves(polynomial_regression, SX_train, yta)
plt.axis([-10, 300, -2000, 20000])
plt.show()
```



[135]:
```
##Interpret the result
```

Answer: By the time it trains the 175th or so training data, the predictions become as good as they're going to get by the looks of it as the line's slope reaches zero

[136]:
```
###Regularization
```

[137]:
```
##Explain the purpose of regularization
##For the following Regularization methods (number 14, 15 16, 17)
```

Answer: Regularization reduces the number of polynomial degrees in order to avoid overfitting the data. There's ridge regression, the SGD regressor, lasso, and elastic net. Each are slightly different, though elastic net is the generally preferred of the four.

[138]:
```
###Ridge Regression
```

[139]:
```
##Use sklearn's Ridge to train the data set
```

[140]:
```
ridge_reg = Ridge(alpha=1, solver="cholesky", random_state=42)
ridge_reg.fit(X_polytr2, y_train)
```

[140]:
```
Ridge(alpha=1, random_state=42, solver='cholesky')
```

[141]:
```
##Predict for test set
```

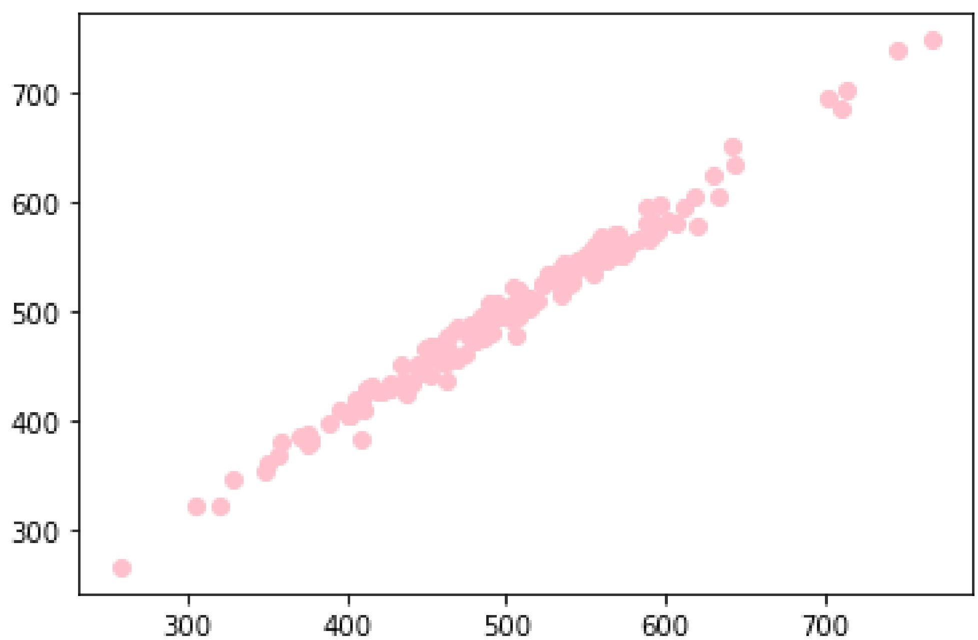
[142]:
```
y_pred8 = ridge_reg.predict(X_polyte2)
print("Predicted: ", y_pred8[11])
print("Actual: ", yte[11])
```

```
Predicted:  494.8527744067474
Actual:  493.55683370047706
```

[143]:
```
##Generate a scatter plot that shows the Y test on the x-axis and y predicted␣
↪in the y-axis
```

[144]:
```
plt.scatter(x = yte, y = y_pred8, c = 'pink')
```

[144]: <matplotlib.collections.PathCollection at 0x7f9616334580>



[145]:
```
##Use sklearn's metrics to print the value of MAE, MSE, RMSE, and R^2
```

[146]:
```
print("MAE: %.2f" % mean_absolute_error(y_test, y_pred8))
print("MSE: %.2f" % mean_squared_error(y_test, y_pred8))
print("RMSE: %.2f" % mean_squared_error(y_test, y_pred8, squared = False))
print("R^2: %.2f" % r2_score(y_test, y_pred8))
```

```
MAE: 9.63
MSE: 146.19
RMSE: 12.09
R^2: 0.98
```

[147]:
```
###SGDRegressor for Ridge
```

[148]:
```
##Use sklearn's SGDRegressor for Ridge Regression
```

[149]:
```
sgd_reg2 = SGDRegressor(penalty="l2", max_iter = 50, tol=1e-3, random_state=42)
sgd_reg2.fit(xta, yta.ravel())
```

[149]:
```
SGDRegressor(max_iter=50, random_state=42)
```

[150]:
```
print("Intercept: ", sgd_reg2.intercept_)
print("Coefficient: ", sgd_reg2.coef_)
```

```
Intercept:  [499.71309449]
Coefficient:  [26.04508977 36.67741432  0.18954917 60.18760439]
```

[151]:
```
##Predict for test set
```

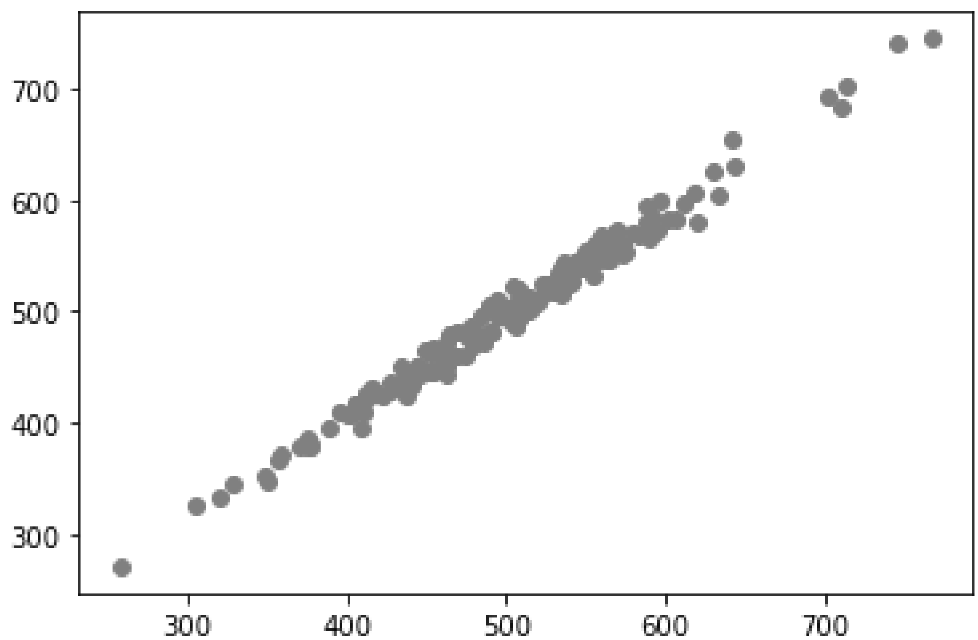
[152]:
```
y_pred9 = sgd_reg2.predict(SX_test)
print("Predicted: ", y_pred9[56])
print("Actual: ", yte[56])
```

```
Predicted:  498.51730971112295
Actual:  508.39006178986654
```

[153]:
```
##Generate a scatter plot that shows the Y test on the x-axis and y predicted␣
↪in the y-axis
```

[154]:
```
plt.scatter(x = yte, y = y_pred9, c = 'grey')
```

[154]:
```
<matplotlib.collections.PathCollection at 0x7f961623a370>
```



[155]:
```
##Use sklearn's metrics to print the value of MAE, MSE, RMSE, and R^2  (see␣
↪documentation of sklearn's metrics)
```

[156]:
```
print("MAE: %.2f" % mean_absolute_error(y_test, y_pred9))
print("MSE: %.2f" % mean_squared_error(y_test, y_pred9))
print("RMSE: %.2f" % mean_squared_error(y_test, y_pred9, squared = False))
print("R^2: %.2f" % r2_score(y_test, y_pred9))
```

```
MAE: 9.37
MSE: 133.39
RMSE: 11.55
R^2: 0.98
```

[157]:
```
###Lasso Regression
```

[158]:
```
##Use sklearn's Lasso
```

[159]:
```
lasso_reg = Lasso(alpha = 0.1)
lasso_reg.fit(SX_train, y_train)
```

[159]:
```
Lasso(alpha=0.1)
```

[160]:
```
##Predict for test set
```

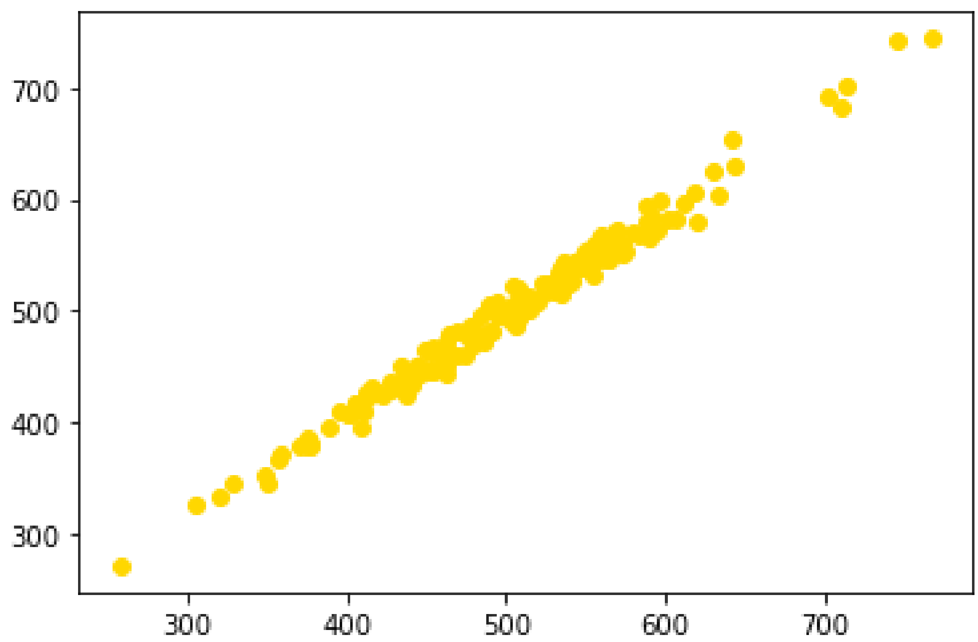
[161]:
```
y_pred10 = lasso_reg.predict(SX_test)
print("Predicted: ", y_pred10[11])
print("Actual: ", yte[11])
```

```
Predicted:  496.0827999605675
Actual:  493.55683370047706
```

[162]:
```
##Generate a scatter plot that shows the Y test on the x-axis and y predicted␣
↪in the y-axis
```

[163]:
```
plt.scatter(x = yte, y = y_pred10, c = 'gold')
```

[163]:
```
<matplotlib.collections.PathCollection at 0x7f9616459b80>
```



[164]:
```
##Use sklearn's metrics to print the value of MAE, MSE, RMSE, and R^2
```

[165]:
```
print("MAE: %.2f" % mean_absolute_error(y_test, y_pred10))
print("MSE: %.2f" % mean_squared_error(y_test, y_pred10))
print("RMSE: %.2f" % mean_squared_error(y_test, y_pred10, squared = False))
print("R^2: %.2f" % r2_score(y_test, y_pred10))
```

```
MAE: 9.44
MSE: 135.18
RMSE: 11.63
R^2: 0.98
```

[166]:
```
##How Lasso perform the regularization and how does that affect the thetas?
```

[167]:
```
print("Intercept: ", lasso_reg.intercept_)
print("Coefficiant: ", lasso_reg.coef_)
```

```
Intercept:  499.7231164913073
Coefficiant:  [25.94175202 36.5798387   0.09165954 60.10343669]
```

Answer: Lasso performs feature selection and outputs a sparse model that tries to eliminate the least important features. In this particular case, it made my least important feature/the third coefficient smaller and closer to zero.

[168]:
```
###Elastic Net
```

[169]:
```
##Use sklearn's ElasticNet
```

[170]:
```
elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5, random_state=42)
elastic_net.fit(SX_train, y_train)
```

[170]:
```
ElasticNet(alpha=0.1, random_state=42)
```

[171]:
```
##Predict for test set
```

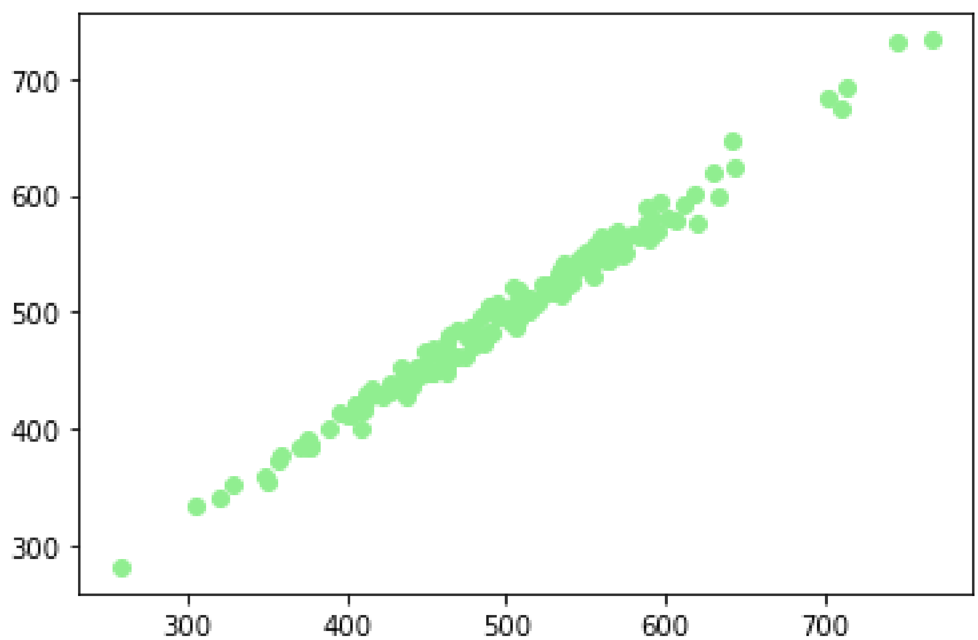
[172]:
```
y_pred11 = elastic_net.predict(SX_test)
print("Predicted: ", y_pred11[11])
print("Actual: ", yte[11])
```

```
Predicted:  496.18142459400735
Actual:  493.55683370047706
```

[173]:
```
##Generate a scatter plot that shows the Y test in x axis and y predicted in y␣
↪axis
```

[174]:
```
plt.scatter(x = yte, y = y_pred11, c = 'lightgreen')
```

[174]:
```
<matplotlib.collections.PathCollection at 0x7f961636f970>
```



[175]:
```
##Use sklearn's metrics to print the value of MAE, MSE, RMSE and R^2
```

[176]:
```
print("MAE: %.2f" % mean_absolute_error(y_test, y_pred11))
print("MSE: %.2f" % mean_squared_error(y_test, y_pred11))
print("RMSE: %.2f" % mean_squared_error(y_test, y_pred11, squared = False))
```

```
print("R^2: %.2f" % r2_score(y_test, y_pred11))
```

```
MAE: 11.01
MSE: 188.17
RMSE: 13.72
R^2: 0.97
```

[177]:
```
##How ElasticNet different compared to Lasso and RIDGE perform the␣
↪regularization and how does that affect the thetas?
```

[178]:
```
print("Intercept: ", elastic_net.intercept_)
print("Coefficiant: ", elastic_net.coef_)
```

```
Intercept:  499.7231164913073
Coefficiant:  [24.72092914 34.88365993  0.18937494 57.37547163]
```

Answer: ElasticNet is similar to lasso in that it will also try to eliminate insignificant features, but it is preferable to lasso in that it won't behave strangely when two features are very highly correlated.