Linear_Ridge_Regression

May 10, 2019

In [1]:
```
import pandas as pd                    # for working with data in Python
import numpy as np
import matplotlib.pyplot as plt        # for visualization
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn import linear_model

# use Pandas to read in csv files. The pd.read_csv() method creates a DataFrame from a
train = pd.read_csv('train.csv')
test = pd.read_csv('test.csv')

print("1 \n")

# check out the size of the data
print("Train data shape:", train.shape)
print("Test data shape:", test.shape)



print("2 \n")

# look at a few rows using the DataFrame.head() method
# train.head()
print(train.head())
```

```
1

Train data shape: (1460, 81)
Test data shape: (1459, 80)
2

   Id  MSSubClass MSZoning  LotFrontage  LotArea Street Alley LotShape  \
0   1          60       RL         65.0     8450   Pave   NaN      Reg
1   2          20       RL         80.0     9600   Pave   NaN      Reg
2   3          60       RL         68.0    11250   Pave   NaN      IR1
3   4          70       RL         60.0     9550   Pave   NaN      IR1
4   5          60       RL         84.0    14260   Pave   NaN      IR1
```

```
  LandContour Utilities  ... PoolArea PoolQC Fence MiscFeature MiscVal MoSold  \
0         Lvl    AllPub  ...        0    NaN   NaN         NaN       0      2
1         Lvl    AllPub  ...        0    NaN   NaN         NaN       0      5
2         Lvl    AllPub  ...        0    NaN   NaN         NaN       0      9
3         Lvl    AllPub  ...        0    NaN   NaN         NaN       0      2
4         Lvl    AllPub  ...        0    NaN   NaN         NaN       0     12

  YrSold  SaleType  SaleCondition  SalePrice
0   2008        WD         Normal     208500
1   2007        WD         Normal     181500
2   2008        WD         Normal     223500
3   2006        WD        Abnorml     140000
4   2008        WD         Normal     250000

[5 rows x 81 columns]
```

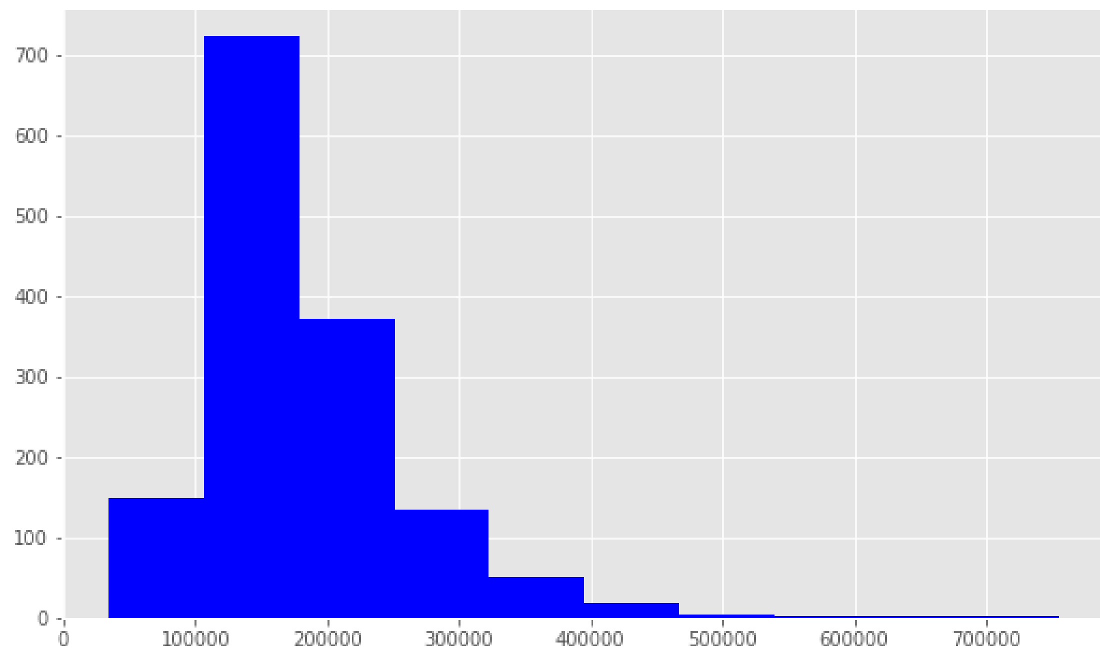
```
In [3]: plt.style.use(style='ggplot')
        plt.rcParams['figure.figsize'] = (10, 6)


        #######################################################
        #  2. Explore the data and engineer Features          ###
        #######################################################

        print("3 \n")
```

```
3
```

```
In [4]: # to get more information like count, mean, std, min, max etc
        # train.SalePrice.describe()
        print (train.SalePrice.describe())

        print("4 \n")

        # to plot a histogram of SalePrice
        print ("Skew is:", train.SalePrice.skew())
        plt.hist(train.SalePrice, color='blue')
        plt.show()

        print("5 \n")
```

```
count      1460.000000
mean     180921.195890
std       79442.502883
min       34900.000000
```

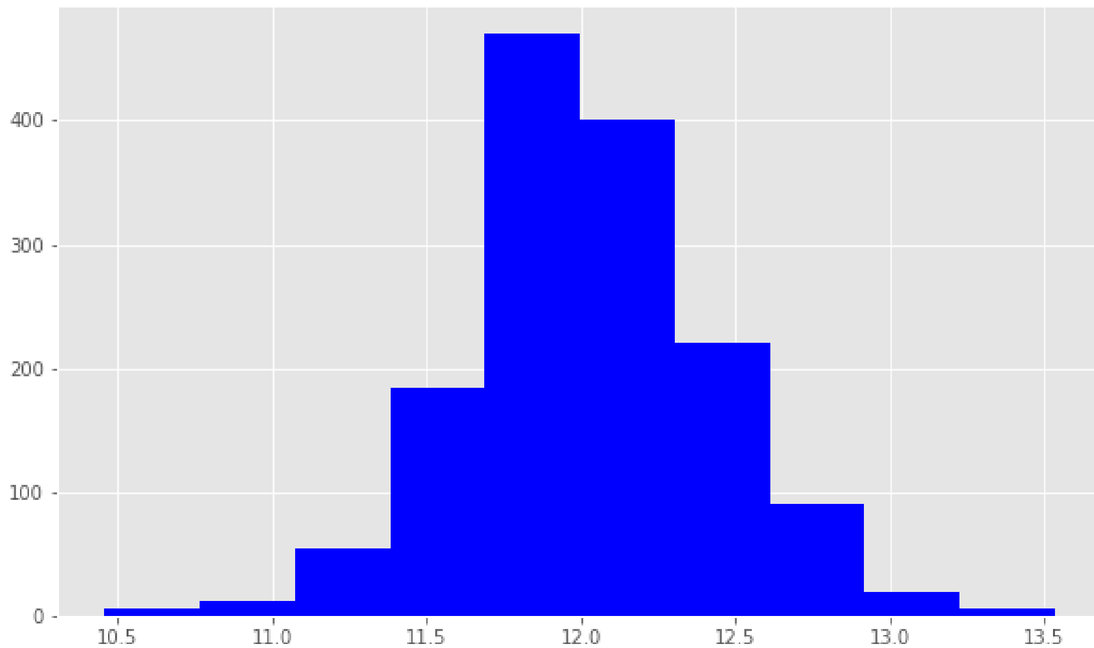
```
25%       129975.000000
50%       163000.000000
75%       214000.000000
max       755000.000000
Name: SalePrice, dtype: float64
4

Skew is: 1.8828757597682129
```



5

In [5]:
```
# use np.log() to transform train.SalePric and calculate the skewness a second time, a
target = np.log(train.SalePrice)
print ("\n Skew is:", target.skew())
plt.hist(target, color='blue')
plt.show()
```

```
 Skew is: 0.12133506220520406
```



In [6]:
```
# return a subset of columns matching the specified data types
numeric_features = train.select_dtypes(include=[np.number])
# numeric_features.dtypes
print(numeric_features.dtypes)
```

```
Id                 int64
MSSubClass         int64
LotFrontage      float64
LotArea            int64
OverallQual        int64
OverallCond        int64
YearBuilt          int64
YearRemodAdd       int64
MasVnrArea       float64
BsmtFinSF1         int64
BsmtFinSF2         int64
BsmtUnfSF          int64
TotalBsmtSF        int64
1stFlrSF           int64
2ndFlrSF           int64
LowQualFinSF       int64
GrLivArea          int64
BsmtFullBath       int64
BsmtHalfBath       int64
FullBath           int64
HalfBath           int64
```

```
BedroomAbvGr       int64
KitchenAbvGr       int64
TotRmsAbvGrd       int64
Fireplaces         int64
GarageYrBlt      float64
GarageCars         int64
GarageArea         int64
WoodDeckSF         int64
OpenPorchSF        int64
EnclosedPorch      int64
3SsnPorch          int64
ScreenPorch        int64
PoolArea           int64
MiscVal            int64
MoSold             int64
YrSold             int64
SalePrice          int64
dtype: object
```

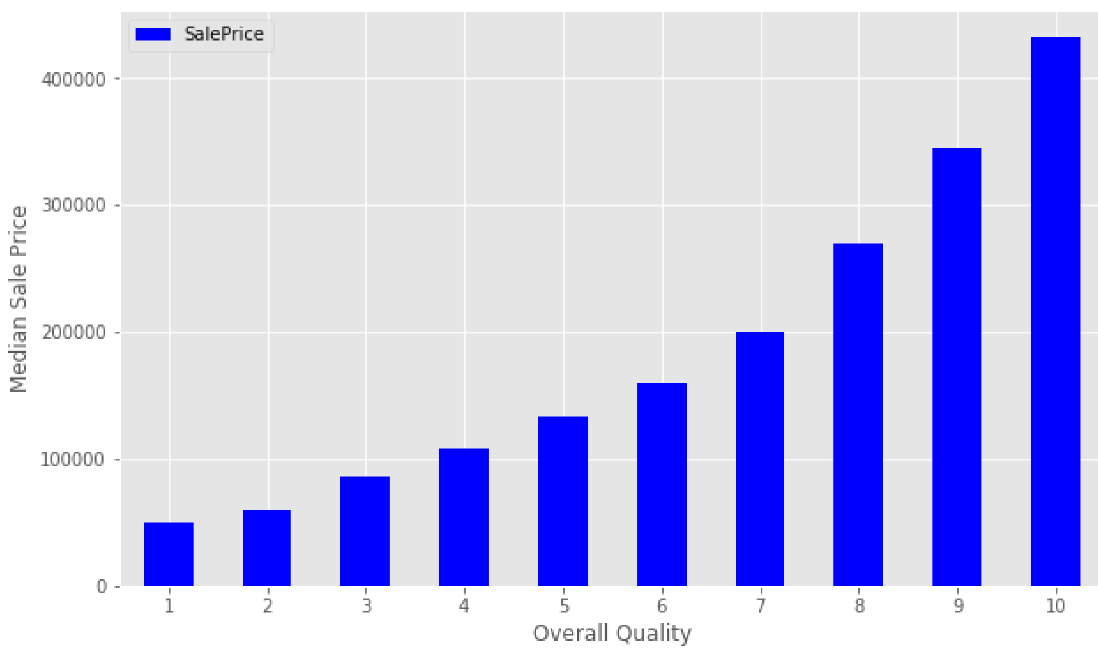
```
In [7]: corr = numeric_features.corr()

        # The first five features are the most positively correlated with SalePrice, while the
        print (corr['SalePrice'].sort_values(ascending=False)[:5], '\n')
        print (corr['SalePrice'].sort_values(ascending=False)[-5:])
```

```
SalePrice      1.000000
OverallQual    0.790982
GrLivArea      0.708624
GarageCars     0.640409
GarageArea     0.623431
Name: SalePrice, dtype: float64

YrSold          -0.028923
OverallCond     -0.077856
MSSubClass      -0.084284
EnclosedPorch   -0.128578
KitchenAbvGr    -0.135907
Name: SalePrice, dtype: float64
```

```
In [8]: print(train.OverallQual.unique())
        """
        print("9 \n")
        """
        #investigate the relationship between OverallQual and SalePrice.
        #We set index='OverallQual' and values='SalePrice'. We chose to look at the median her
        quality_pivot = train.pivot_table(index='OverallQual', values='SalePrice', aggfunc=np.r
        print(quality_pivot)
```

```
[ 7  6  8  5  9  4 10  3  1  2]
             SalePrice
OverallQual
1                50150
2                60000
3                86250
4               108000
5               133000
6               160000
7               200141
8               269750
9               345000
10              432390
```

```
In [11]: #visualize this pivot table more easily, we can create a bar plot
         #Notice that the median sales price strictly increases as Overall Quality increases.
         quality_pivot.plot(kind='bar', color='blue')
         plt.xlabel('Overall Quality')
         plt.ylabel('Median Sale Price')
         plt.xticks(rotation=0)
         plt.show()
```

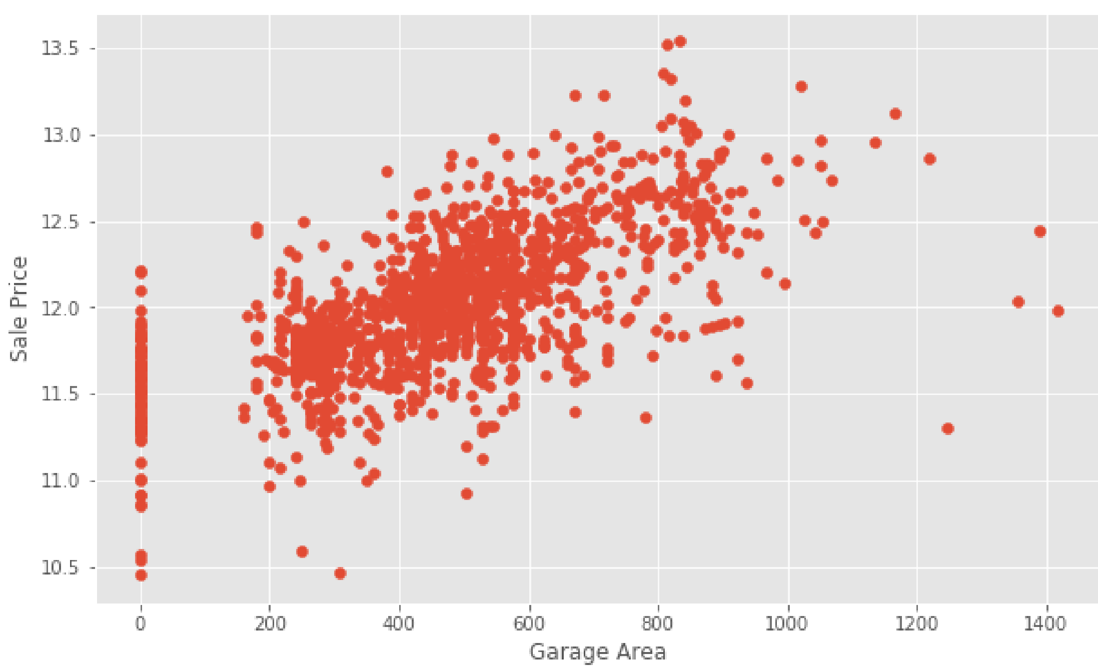



```
In [12]: print("11 \n")
         """
         #to generate some scatter plots and visualize the relationship between the Ground Liv
```

```
plt.scatter(x=train['GrLivArea'], y=target)
plt.ylabel('Sale Price')
plt.xlabel('Above grade (ground) living area square feet')
plt.show()
"""
print("12 \n")

# do the same for GarageArea.
plt.scatter(x=train['GarageArea'], y=target)
plt.ylabel('Sale Price')
plt.xlabel('Garage Area')
plt.show()
```

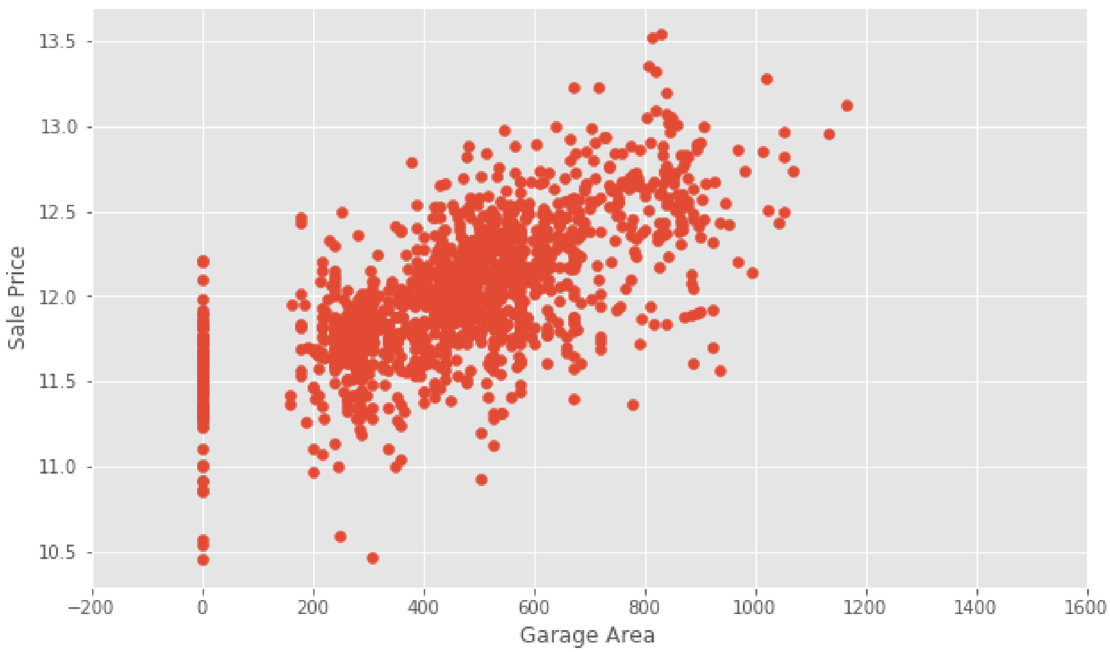
11

12




```
In [13]: # create a new dataframe with some outliers removed
         train = train[train['GarageArea'] < 1200]

         # display the previous graph again without outliers
         plt.scatter(x=train['GarageArea'], y=np.log(train.SalePrice))
         plt.xlim(-200,1600)      # This forces the same scale as before
         plt.ylabel('Sale Price')
```

```
plt.xlabel('Garage Area')
plt.show()
```




```
In [14]: # create a DataFrame to view the top null columns and return the counts of the null v
         nulls = pd.DataFrame(train.isnull().sum().sort_values(ascending=False)[:25])
         nulls.columns = ['Null Count']
         nulls.index.name = 'Feature'
         #nulls
         print(nulls)
```

```
              Null Count
Feature
PoolQC              1449
MiscFeature         1402
Alley               1364
Fence               1174
FireplaceQu          689
LotFrontage          258
GarageCond            81
GarageType            81
GarageYrBlt           81
GarageFinish          81
GarageQual            81
BsmtExposure          38
BsmtFinType2          38
BsmtFinType1          37
```

```
BsmtCond            37
BsmtQual            37
MasVnrArea           8
MasVnrType           8
Electrical           1
Utilities            0
YearRemodAdd         0
MSSubClass           0
Foundation           0
ExterCond            0
ExterQual            0
```

```
In [15]: print("15 \n")
         """
         #to return a list of the unique values
         print ("Unique values are:", train.MiscFeature.unique())
         """

         ######################################################
         #   Wrangling the non-numeric Features             ##
         ######################################################

         print("16 \n")

         # consider the non-numeric features and display details of columns
         categoricals = train.select_dtypes(exclude=[np.number])
         #categoricals.describe()
         print(categoricals.describe())
```

```
15

16

       MSZoning Street Alley LotShape LandContour Utilities LotConfig  \
count      1455   1455    91     1455        1455      1455      1455
unique        5      2     2        4           4         2         5
top          RL   Pave  Grvl      Reg         Lvl    AllPub    Inside
freq       1147   1450    50      921        1309      1454      1048

       LandSlope Neighborhood Condition1  ... GarageType GarageFinish  \
count       1455         1455       1455  ...       1374         1374
unique         3           25          9  ...          6            3
top          Gtl        NAmes       Norm  ...     Attchd          Unf
freq        1378          225       1257  ...        867          605

       GarageQual GarageCond PavedDrive PoolQC  Fence MiscFeature SaleType  \
count        1374       1374       1455      6    281          53     1455
```

```
unique          5          5          3      3      4           4        9
top            TA         TA          Y     Gd  MnPrv        Shed       WD
freq         1306       1321       1335      2    157          48     1266

       SaleCondition
count           1455
unique             6
top           Normal
freq            1196

[4 rows x 43 columns]
```

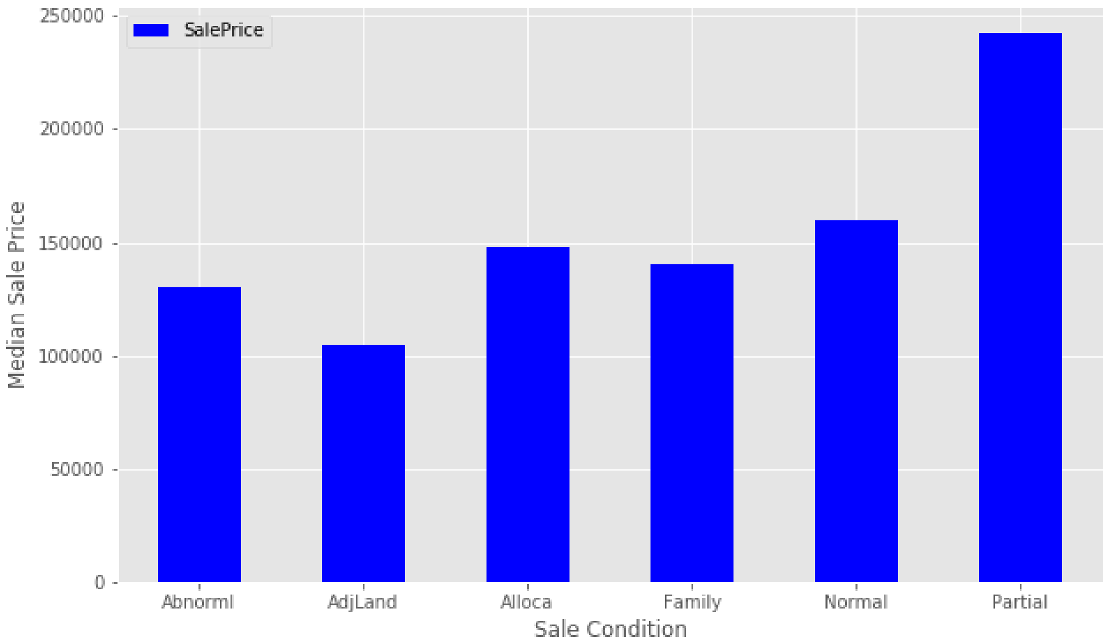
In [16]:
```python
#####################################################
#   Transforming and engineering features           ##
######################################################

print("17 \n")

# When transforming features, it's important to remember that any transformations tha
# fitting the model must be applied to the test data.

#Eg:
print ("Original: \n")
print (train.Street.value_counts(), "\n")
```

```
17

Original:

Pave    1450
Grvl       5
Name: Street, dtype: int64
```

In [17]:
```python
print("18 \n")

# our model needs numerical data, so we will use one-hot encoding to transform the da
# create a new column called enc_street. The pd.get_dummies() method will handle this
train['enc_street'] = pd.get_dummies(train.Street, drop_first=True)
test['enc_street'] = pd.get_dummies(test.Street, drop_first=True)

print ('Encoded: \n')
print (train.enc_street.value_counts())  # Pave and Grvl values converted into 1 and

print("19 \n")
```

```
# look at SaleCondition by constructing and plotting a pivot table, as we did above f
condition_pivot = train.pivot_table(index='SaleCondition', values='SalePrice', aggfunc
condition_pivot.plot(kind='bar', color='blue')
plt.xlabel('Sale Condition')
plt.ylabel('Median Sale Price')
plt.xticks(rotation=0)
plt.show()
```

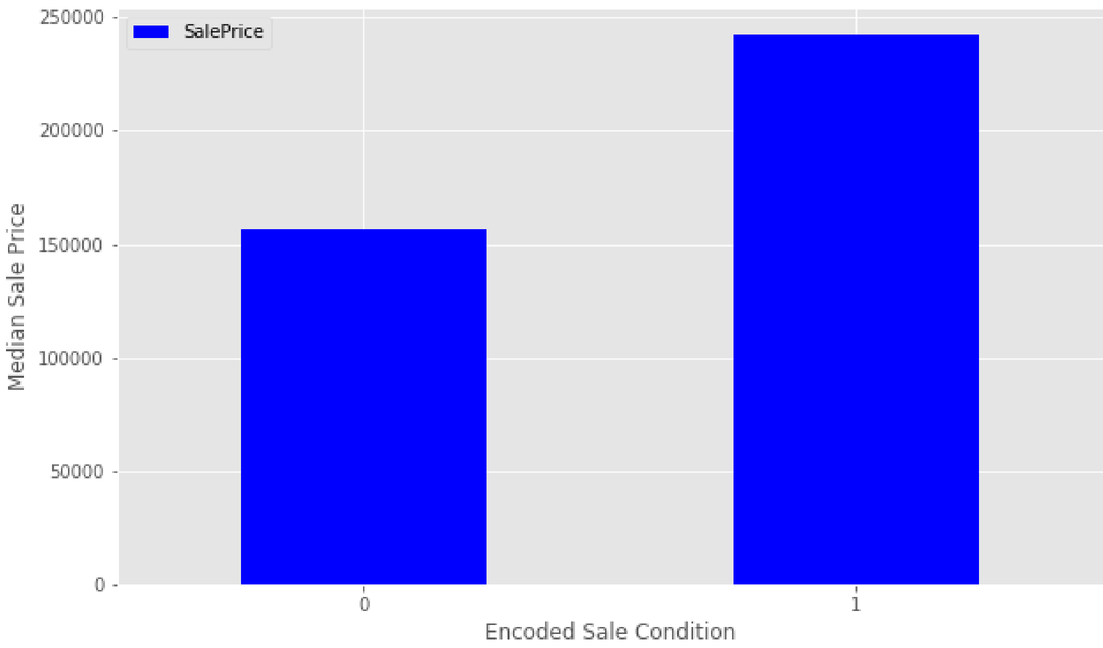
```
18

Encoded:

1    1450
0       5
Name: enc_street, dtype: int64
19
```




```
In [18]: # encode this SaleCondition as a new feature by using a similar method that we used f
         def encode(x): return 1 if x == 'Partial' else 0
         train['enc_condition'] = train.SaleCondition.apply(encode)
         test['enc_condition'] = test.SaleCondition.apply(encode)

         print("20 \n")
```

```
# explore this newly modified feature as a plot.
condition_pivot = train.pivot_table(index='enc_condition', values='SalePrice', aggfunc
condition_pivot.plot(kind='bar', color='blue')
plt.xlabel('Encoded Sale Condition')
plt.ylabel('Median Sale Price')
plt.xticks(rotation=0)
plt.show()
```

20




```
In [19]: ####################################################################################
         #   Dealing with missing values
         #   We'll fill the missing values with an average value and then assign the results t
         #   This is a method of interpolation
         ####################################################################################
         data = train.select_dtypes(include=[np.number]).interpolate().dropna()

         print("21 \n")
         # Check if the all of the columns have 0 null values.
         # sum(data.isnull().sum() != 0)
         print(sum(data.isnull().sum() != 0))

         print("22 \n")
```

21

0
22

In [20]:
```
#####################################################
#  3. Build a linear model                            ##
#####################################################

# separate the features and the target variable for modeling.
# We will assign the features to X and the target variable(Sales Price)to y.

y = np.log(train.SalePrice)
X = data.drop(['SalePrice', 'Id'], axis=1)
# exclude ID from features since Id is just an index with no relationship to SalePric

#======= partition the data ===================================================
#   Partitioning the data in this way allows us to evaluate how our model might perfo
#   If we train the model on all of the test data, it will be difficult to tell if ov
#==============================================================================
# also state how many percentage from train data set, we want to take as test data se
# In this example, about 33% of the data is devoted to the hold-out set.
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42, test_size=
```

In [21]:
```
#========= Begin modelling =======================#
#    Linear Regression Model                      #
#=================================================#

# ---- first create a Linear Regression model.
# First, we instantiate the model.
lr = linear_model.LinearRegression()

# ---- fit the model / Model fitting
# lr.fit() method will fit the linear regression on the features and target variable
model = lr.fit(X_train, y_train)

print("23 \n")
```

23

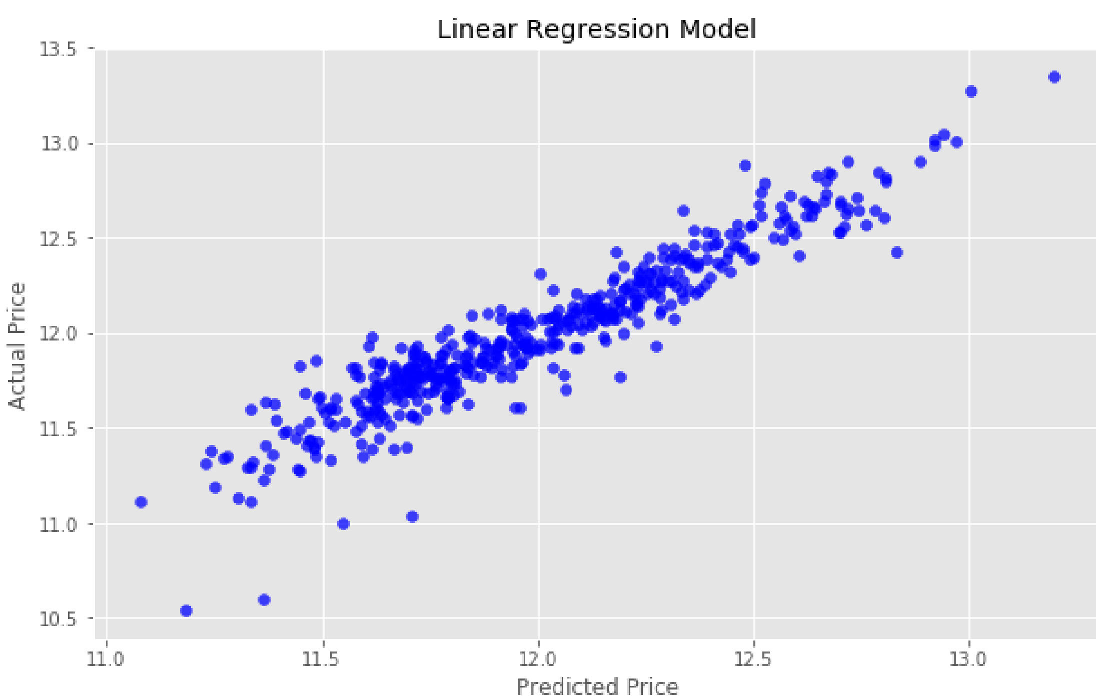
In [22]:
```
# ---- Evaluate the performance and visualize results
# r-squared value is a measure of how close the data are to the fitted regression lin
# a higher r-squared value means a better fit(very close to value 1)
print("R^2 is: \n", model.score(X_test, y_test))
```

```
# use the model we have built to make predictions on the test data set.
predictions = model.predict(X_test)

print("24 \n")
```

```
R^2 is:
 0.8882477709262542
24
```

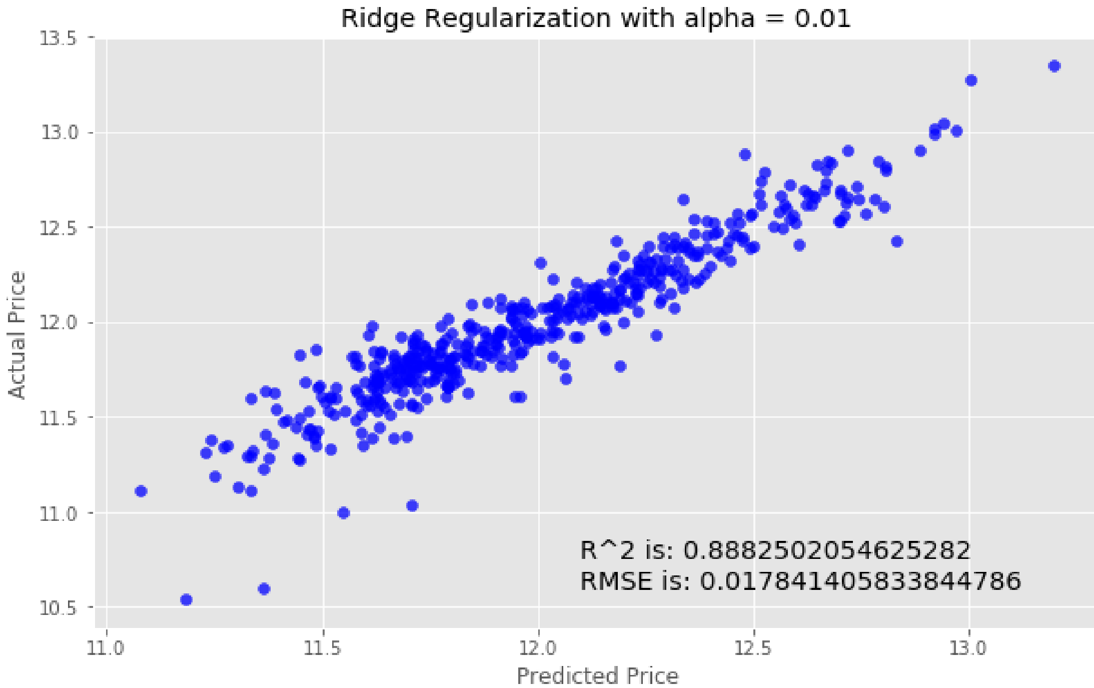
```
In [23]: print('RMSE is: \n', mean_squared_error(y_test, predictions))

         print("25 \n")
         # view this relationship between predictions and actual_values graphically with a sca
         actual_values = y_test
         plt.scatter(predictions, actual_values, alpha=.75,
                     color='b')  # alpha helps to show overlapping data
         plt.xlabel('Predicted Price')
         plt.ylabel('Actual Price')
         plt.title('Linear Regression Model')
         plt.show()
```

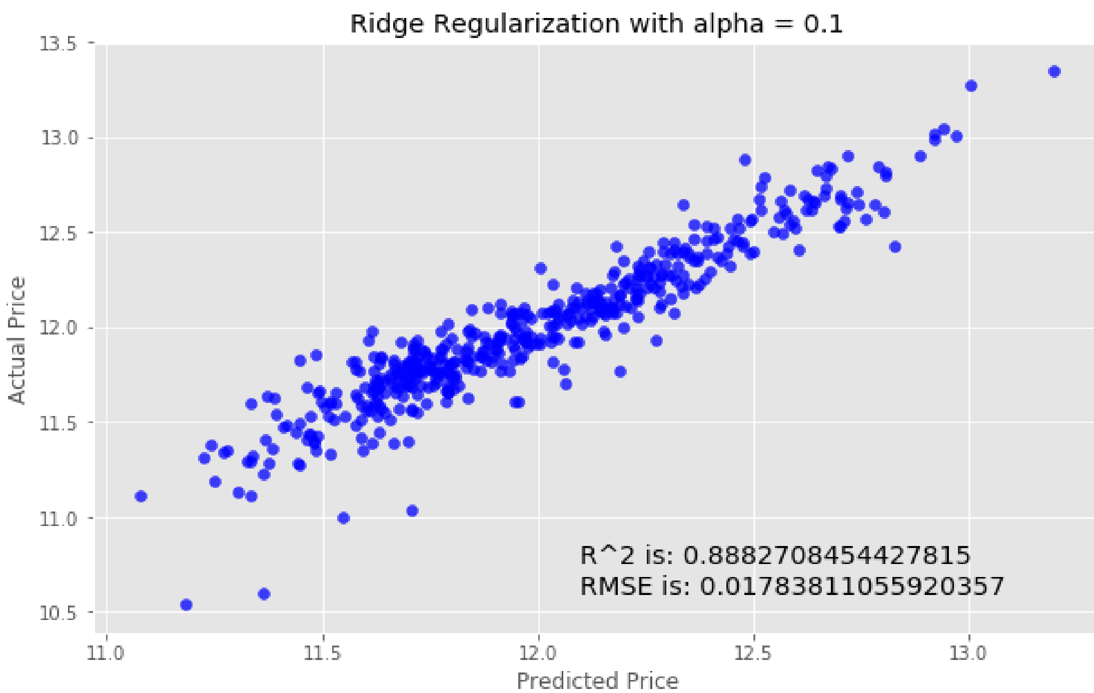
```
RMSE is:
 0.017841794519567734
25
```

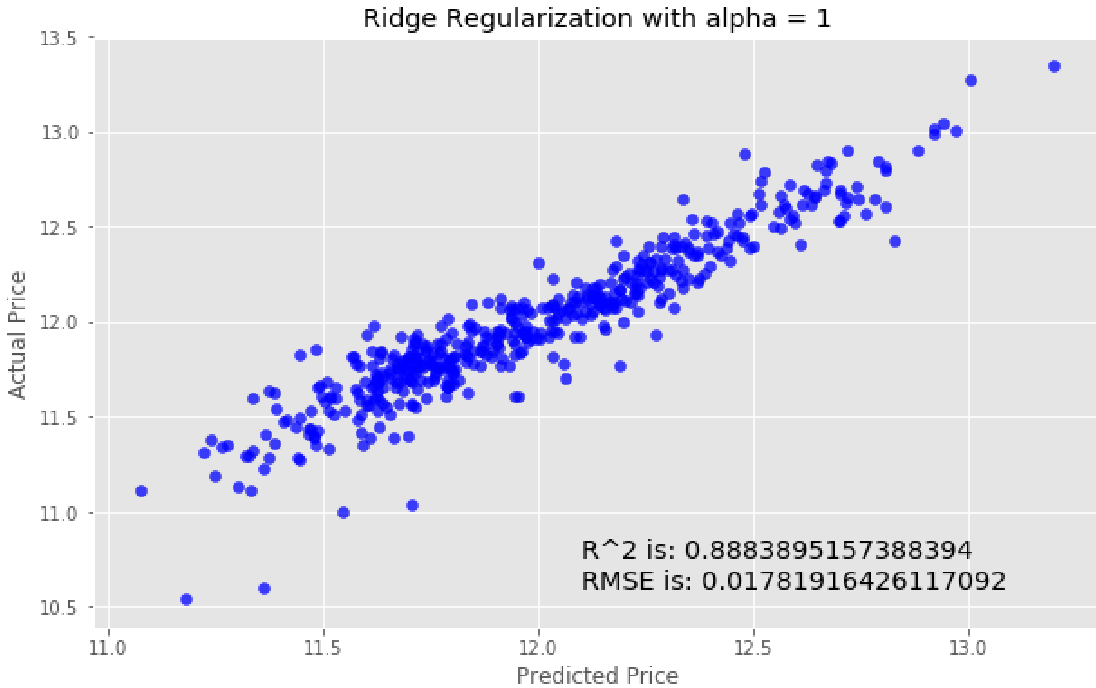


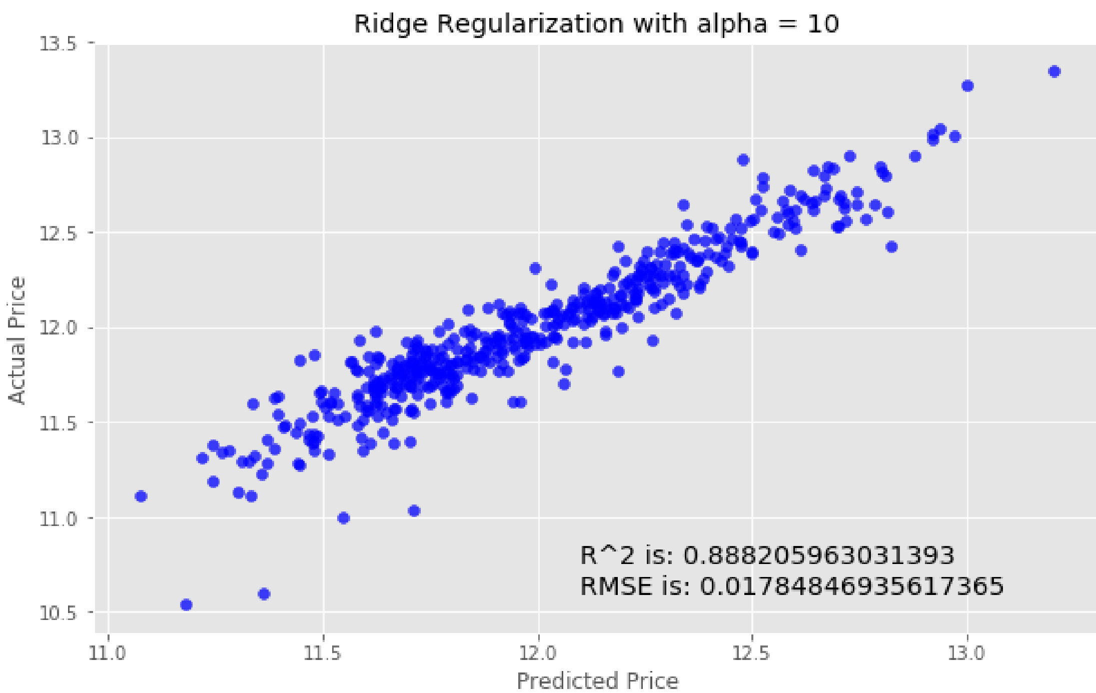
In [24]:
```
#====== improve the model ================================================================
#  try using Ridge Regularization to decrease the influence of less important feature
#========================================================================================

print("26 \n")
# experiment by looping through a few different values of alpha, and see how this cha

for i in range (-2, 3):
    alpha = 10**i
    rm = linear_model.Ridge(alpha=alpha)
    ridge_model = rm.fit(X_train, y_train)
    preds_ridge = ridge_model.predict(X_test)

    plt.scatter(preds_ridge, actual_values, alpha=.75, color='b')
    plt.xlabel('Predicted Price')
    plt.ylabel('Actual Price')
    plt.title('Ridge Regularization with alpha = {}'.format(alpha))
    overlay = 'R^2 is: {}\nRMSE is: {}'.format(
                    ridge_model.score(X_test, y_test),
                    mean_squared_error(y_test, preds_ridge))
    plt.annotate(s=overlay,xy=(12.1,10.6),size='x-large')
    plt.show()

# if you examined the plots you can see these models perform almost identically to th
# In our case, adjusting the alpha did not substantially improve our model.

print("27 \n")
print("R^2 is: \n", model.score(X_test, y_test))
```

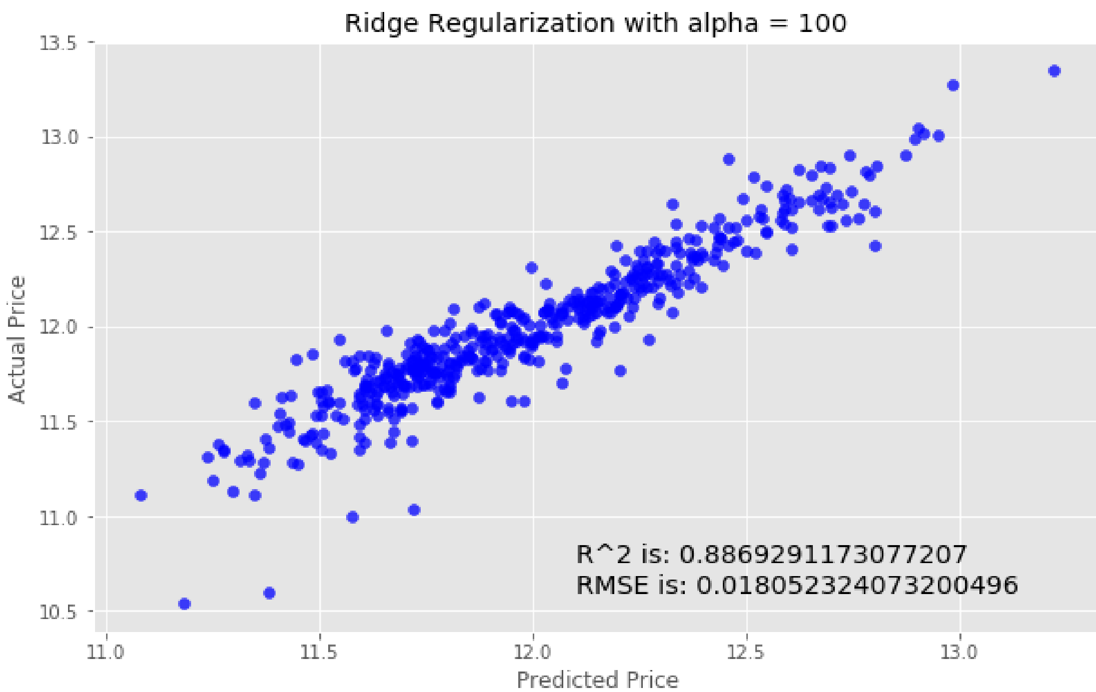
```
26
```


Ridge Regularization with alpha = 0.01
Actual Price
Predicted Price
13.5
13.0
12.5
12.0
11.5
11.0
10.5
11.0
11.5
12.0
12.5
13.0
R^2 is: 0.8882502054625282
RMSE is: 0.017841405833844786


Ridge Regularization with alpha = 0.1
Actual Price
Predicted Price
13.5
13.0
12.5
12.0
11.5
11.0
10.5
11.0
11.5
12.0
12.5
13.0
R^2 is: 0.8882708454427815
RMSE is: 0.01783811055920357


Ridge Regularization with alpha = 1
Actual Price
Predicted Price
R^2 is: 0.8883895157388394
RMSE is: 0.01781916426117092


Ridge Regularization with alpha = 10
Actual Price
Predicted Price
R^2 is: 0.888205963031393
RMSE is: 0.01784846935617365




```
27

R^2 is:
 0.8882477709262542
```

In [25]:
```
##################################################
#     4.  Make a submission                     ##
##################################################

# create a csv that contains the predicted SalePrice for each observation in the test
submission = pd.DataFrame()
# The first column must the contain the ID from the test data.
submission['Id'] = test.Id

# select the features from the test data for the model as we did above.
feats = test.select_dtypes(
    include=[np.number]).drop(['Id'], axis=1).interpolate()

# generate predictions
predictions = model.predict(feats)

# transform the predictions to the correct form
# apply np.exp() to our predictions becasuse we have taken the logarithm(np.log()) pr
final_predictions = np.exp(predictions)
```

```python
print("28 \n")

# check the difference
print("Original predictions are: \n", predictions[:10], "\n")
print("Final predictions are: \n", final_predictions[:10])

print("29 \n")
# assign these predictions and check
submission['SalePrice'] = final_predictions
# submission.head()
print(submission.head())

# export to a .csv file as Kaggle expects.
# pass index=False because Pandas otherwise would create a new index for us.
submission.to_csv('submission1.csv', index=False)


print("\n Finish")
```

```
28

Original predictions are:
 [11.76725362 11.71929504 12.07656074 12.20632678 12.11217655 12.05709882
 12.16036698 12.01665734 12.17126892 11.66318882]

Final predictions are:
 [128959.49172586 122920.74024359 175704.82598102 200050.83263756
 182075.46986405 172318.33397533 191064.62164201 165488.55901671
 193158.99133192 116214.02546462]
29

     Id      SalePrice
0  1461  128959.491726
1  1462  122920.740244
2  1463  175704.825981
3  1464  200050.832638
4  1465  182075.469864

 Finish
```