Prediction of California Median House Prices Using Regression Techniques

Import required libraries

In [42]:
```
%matplotlib inline
import seaborn as sns
import pandas as pd
import numpy as np
import os
import matplotlib.pyplot as plt
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn import tree
from sklearn import preprocessing
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_recall_fscore_support
from sklearn.metrics import accuracy_score
from sklearn import metrics
from sklearn.neighbors import KNeighborsClassifier
from scipy import stats
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression, LogisticRegressionCV
import statsmodels.api as sm
from dmba import classificationSummary, gainsChart, liftChart
from dmba.metric import AIC_score
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
from sklearn.tree import DecisionTreeRegressor
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import accuracy_score, mean_squared_error
from sklearn.preprocessing import MinMaxScaler
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import LinearRegression
import matplotlib.gridspec as gridspec
from scipy.stats import norm
from scipy.cluster import hierarchy as hc
from dmba import backward_elimination, forward_selection, stepwise_selection
from dmba import regressionSummary, exhaustive_search
from sklearn.linear_model import Lasso
from sklearn.linear_model import Ridge
from regressormetricgraphplot import *
from sklearn.datasets import make_regression
from statsmodels.stats.outliers_influence import variance_inflation_factor
import warnings
warnings.filterwarnings('ignore')
```

DATA SPLITTING (training, validation, and test sets)

In [43]:
```
df = pd.read_csv('data/California_Houses.csv')
df
```

Out[43]:

	Median_House_Value	Median_Income	Median_Age	Tot_Rooms	Tot_Bedrooms	Population	Households	Latitude	Longitude	Distance_to_coast	Distance
0	452600.0	8.3252	41	880	129	322	126	37.88	-122.23	9263.040773	556529.
1	358500.0	8.3014	21	7099	1106	2401	1138	37.86	-122.22	10225.733072	554279.
2	352100.0	7.2574	52	1467	190	496	177	37.85	-122.24	8259.085109	554610.
3	341300.0	5.6431	52	1274	235	558	219	37.85	-122.25	7768.086571	555194.
4	342200.0	3.8462	52	1627	280	565	259	37.85	-122.25	7768.086571	555194.
...	...	...	...	...	...	...	...	...	...	...	...
20635	78100.0	1.5603	25	1665	374	845	330	39.48	-121.09	162031.481121	654530.
20636	77100.0	2.5568	18	697	150	356	114	39.49	-121.21	160445.433537	659747.
20637	92300.0	1.7000	17	2254	485	1007	433	39.43	-121.22	153754.341182	654042.
20638	84700.0	1.8672	18	1860	409	741	349	39.43	-121.32	152005.022239	657698.
20639	89400.0	2.3886	16	2785	616	1387	530	39.37	-121.24	146866.196892	648723.

20640 rows × 14 columns

After parsing through the initial data set, we can see there are 20640 entries and 14 data columns. All data represents a block within a neighborhood.

In [44]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 14 columns):
 #   Column                  Non-Null Count  Dtype  
---  ------                  --------------  -----  
 0   Median_House_Value      20640 non-null  float64
 1   Median_Income           20640 non-null  float64
 2   Median_Age              20640 non-null  int64  
 3   Tot_Rooms               20640 non-null  int64  
 4   Tot_Bedrooms            20640 non-null  int64  
 5   Population              20640 non-null  int64  
 6   Households              20640 non-null  int64  
 7   Latitude                20640 non-null  float64
 8   Longitude               20640 non-null  float64
 9   Distance_to_coast       20640 non-null  float64
 10  Distance_to_LA          20640 non-null  float64
 11  Distance_to_SanDiego    20640 non-null  float64
 12  Distance_to_SanJose     20640 non-null  float64
 13  Distance_to_SanFrancisco  20640 non-null  float64
dtypes: float64(9), int64(5)
memory usage: 2.2 MB
```

Initial descriptive statistics of the data. While not particulary useful now, it is important to see variances in the data.

In [45]:
```
df.describe()
```

Out[45]:

	Median_House_Value	Median_Income	Median_Age	Tot_Rooms	Tot_Bedrooms	Population	Households	Latitude	Longitude	Distance_to_c
count	20640.000000	20640.000000	20640.000000	20640.000000	20640.000000	20640.000000	20640.000000	20640.000000	20640.000000	20640.00
mean	206855.816909	3.870671	28.639486	2635.763081	537.898014	1425.476744	499.539680	35.631861	-119.569704	40509.26
std	115395.615874	1.899822	12.585558	2181.615252	421.247906	1132.462122	382.329753	2.135952	2.003532	49140.03
min	14999.000000	0.499900	1.000000	2.000000	1.000000	3.000000	1.000000	32.540000	-124.350000	120.67
25%	119600.000000	2.563400	18.000000	1447.750000	295.000000	787.000000	280.000000	33.930000	-121.800000	9079.75
50%	179700.000000	3.534800	29.000000	2127.000000	435.000000	1166.000000	409.000000	34.260000	-118.490000	20522.01
75%	264725.000000	4.743250	37.000000	3148.000000	647.000000	1725.000000	605.000000	37.710000	-118.010000	49830.41
max	500001.000000	15.000100	52.000000	39320.000000	6445.000000	35682.000000	6082.000000	41.950000	-114.310000	333804.68

Let's dive into the data and look for unique values to ensure there are no duplicates.

In [46]:
```
# Finding out how many unqiue values there are
dict = {}
for col in list(df.columns):
    dict[col] = df[col].value_counts().shape[0]
pd.DataFrame(dict,index=["unique count"]).T
```

Out[46]:

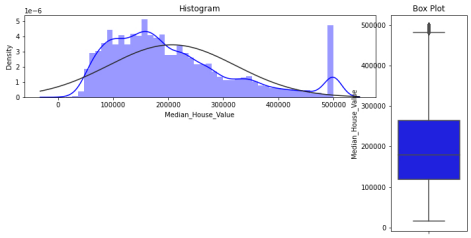
	unique count
Median_House_Value	3842
Median_Income	12928
Median_Age	52
Tot_Rooms	5926
Tot_Bedrooms	1928
Population	3888
Households	1815
Latitude	862
Longitude	844
Distance_to_coast	12590
Distance_to_LA	12590
Distance_to_SanDiego	12590
Distance_to_SanJose	12590
Distance_to_SanFrancisco	12590

For initial pre-processing, let's group by data. Cont_col is all data, Loc_col is geographical and dist_col is distance. The target variable is Median_House_Value.

In [47]:
```
cont_col = ['Median_Income', 'Median_Age', 'Tot_Rooms', 'Tot_Bedrooms', 'Population', 'Households', 'Distance_to_cc
            'Distance_to_LA', 'Distance_to_SanDiego', 'Distance_to_SanJose', 'Distance_to_SanFrancisco']
loc_col = ['Latitude', 'Longitude']
dist_col = ['Distance_to_coast', 'Distance_to_LA', 'Distance_to_SanDiego', 'Distance_to_SanJose', 'Distance_to_SanF
target = 'Median_House_Value'
```

In [48]:
```
def visualize_target(df, feature):
    print('Feature: {}, Skewness: {}, Kurtosis: {}'.format(feature,round(df[feature].skew(),5),round(df[feature].ku

    fig = plt.figure(constrained_layout=True, figsize=(12,6))
    grid = gridspec.GridSpec(ncols=5, nrows=5, figure=fig)

    ax1 = fig.add_subplot(grid[0:2, :4])
    ax1.set_title('Histogram')
    sns.distplot(df.loc[:,feature], norm_hist=True,fit=norm, ax = ax1,color='blue')

    ax3 = fig.add_subplot(grid[:, 4])
    ax3.set_title('Box Plot')
    sns.boxplot(y=df.loc[:,feature], orient='v', ax = ax3,color='blue')

visualize_target(df,target)
```

```
Feature: Median_House_Value, Skewness: 0.97776, Kurtosis: 0.32787
```



You can see the outliers on the higher end of the median house spectrum. We will now remove these according IQR.

In [49]:
```
def remove_outlier(df, col_name):
    q1 = df[col_name].quantile(0.25)
    q3 = df[col_name].quantile(0.75)
    iqr = q3-q1 #Interquartile range
    outlier = []
    for i in df[col_name]:
        if i<(q1 - 1.5 * iqr) or i>(q3 + 1.5 * iqr):
            outlier.append(i)
    outlier = pd.DataFrame(outlier,columns=['outlier'])
    print('{}% Outlier removed in {} according to IQR'.format(round((outlier.shape[0]/df.shape[0])*100,2),col_name)
    fence_low  = q1-1.5*iqr
    fence_high = q3+1.5*iqr
    df = df[(df[col_name] > fence_low) & (df[col_name] < fence_high)]
    return df

df = remove_outlier(df,target)
```

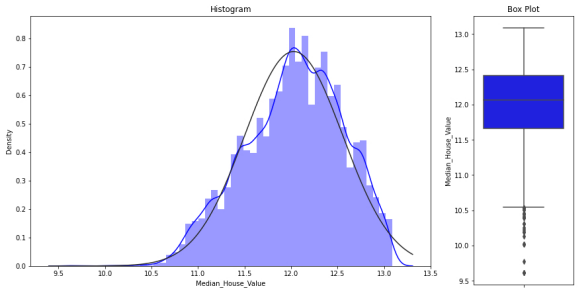
```
5.19% Outlier removed in Median_House_Value according to IQR
```

In [50]:
```
df["Median_House_Value"] = np.log(df["Median_House_Value"])
```

This visualization is to check that all outliers were removed properly.

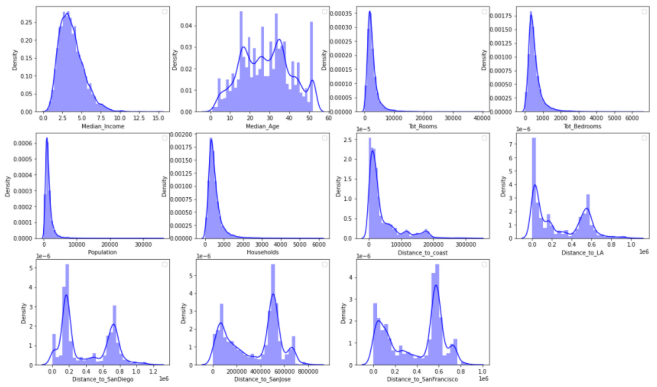
In [51]:
```
visualize_target(df,target)
```

```
Feature: Median_House_Value, Skewness: -0.31629, Kurtosis: -0.36984
```



In [53]:
```
plt.figure(figsize=(20, 12))
for i, column in enumerate(cont_col, 1):
    plt.subplot(3, 4, i)
    sns.distplot(x=df[column],color='blue')
    plt.legend()
    plt.xlabel(column)
```

```
No handles with labels found to put in legend.
No handles with labels found to put in legend.
No handles with labels found to put in legend.
No handles with labels found to put in legend.
No handles with labels found to put in legend.
No handles with labels found to put in legend.
No handles with labels found to put in legend.
No handles with labels found to put in legend.
No handles with labels found to put in legend.
No handles with labels found to put in legend.
No handles with labels found to put in legend.
```

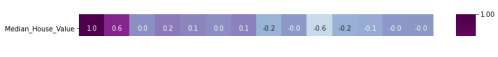


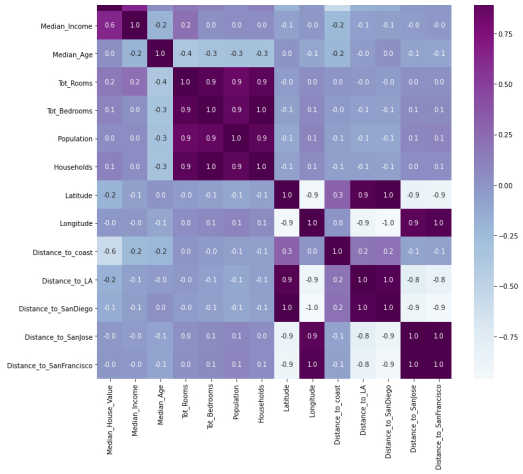
As you can see, all data columns have no extreme outliers. Exploratory Data Analysis can begin now.

Exploratory Data Analysis

In [55]:
```
#Heatmap of a Correlation Table with new data frame
corr = df.corr()
fig, ax = plt.subplots()
fig.set_size_inches(12,11)
sns.heatmap(corr, annot=True, fmt=".1f", cmap="BuPu", center=0, ax=ax)
```

Out[55]: <AxesSubplot:>

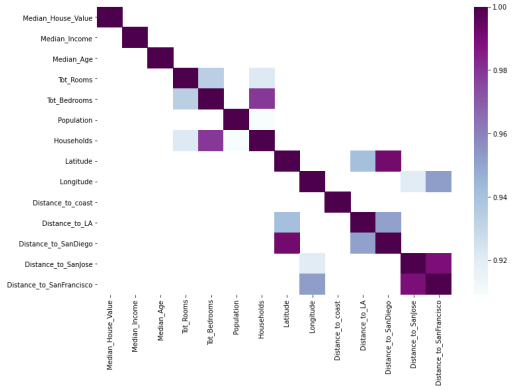




This first correlation is all data columns against each other. We are examining for all variables that are the highest correlation. Below we filter out all variables with greater than 0.9.

In [56]:
```
#Filtering out highly correlated variables (>0.9)with heatmap
corr = df.corr()
kot = corr[corr>=.9]
plt.figure(figsize=(12,8))
sns.heatmap(kot, cmap="BuPu")
```

Out[56]: <AxesSubplot:>



Highly correlated variables (>0.9) were observed among Tot_Rooms,Tot_Bedrooms,Households,and Population. Tot_Room, Tot_Bedrooms, Households will be excluded since Household variable alone can represent them.

Another set of high correlation between geographical variables: Latitude, Longitude, Distance_to_SanDiego, Distance_to_SanJose, and Distance_to_SanFrancisco. Thus, Latitude, Longitude, Distance_to_SanJose, and Distance_to_SanDiego will be dropped. Keeping only Distance_to_LA will cover Distance_to_SanDiego and Distance_to_SanFrancisco will cover Distance_to_SanJose since they are in close proximity.

In [57]:
```
#dropping highly correlated variables
df = df.drop(columns= ['Latitude','Longitude', 'Distance_to_SanJose','Tot_Bedrooms',
                       'Tot_Rooms','Population','Distance_to_SanDiego'])

df #new data frame after removing highly correlated attributes
```

Out[57]:

	Median_House_Value	Median_Income	Median_Age	Households	Distance_to_coast	Distance_to_LA	Distance_to_SanFrancisco
0	13.022764	8.3252	41	126	9263.040773	556529.158342	21250.213767
1	12.789684	8.3014	21	1138	10225.733072	554279.850069	20880.600400
2	12.771671	7.2574	52	177	8259.085109	554610.717069	18811.487450
3	12.740517	5.6431	52	219	7768.086571	555194.266086	18031.047568
4	12.743151	3.8462	52	259	7768.086571	555194.266086	18031.047568
...	...	...	...	...	...	...	...
20635	11.265745	1.5603	25	330	162031.481121	654530.186299	222619.890417
20636	11.252859	2.5568	18	114	160445.433537	659747.068444	218314.424634
20637	11.432799	1.7000	17	433	153754.341182	654042.214020	212097.936232
20638	11.346871	1.8672	18	349	152005.022239	657698.007703	207923.199166
20639	11.400876	2.3886	16	530	146866.196892	648723.337126	205473.376575

19569 rows × 7 columns

DATA SPLITTING (training, validation, and test sets)

There were total records of 19569 and seven attributes after removing the outliers and highly correlated variables. The data set was then partitioned into training, validation, and test data sets (40%, 35%, and 25% respectively). There were 7828 records in training, 6849 records in validation, and 4620 records in test data sets with 7 variables.

In [58]:
```
#Total records after removing outliers = 19569,
#Splitting train = 7828(40%), valid = 6849(35%), test = 4892(25%)

train, temp = train_test_split(df, train_size= 7828, random_state=1)
valid, test = train_test_split(temp, train_size= 6849, random_state=1)
print('Training   : ', train.shape)
print('Validation : ', valid.shape)
print('Test : ', test.shape)
```

```
Training   :  (7828, 7)
Validation :  (6849, 7)
Test :  (4892, 7)
```

MODEL BUILDING STRATEGIES

Regression Analysis is used in many studies including the predictions of house prices. Since the purpose of this study was to predict the numeric and continuous target variable of house prices, Multiple Linear Regression, Regression Tree, Random Forest, and Ridge and Lasso Regression models were selected.

Model will be trained on the train data set and evaluated with validation data set for better performance. The best model will be selected by comparing the metrics of each model on the test data set.

The selected models were built to predict the target outcome of Median House Value by using the independent predictors: 'Median_Income', 'Median_Age', 'Households', 'Distance_to_coast', 'Distance_to_LA', 'Distance_to_SanFrancisco'. The target outcome was the Median House Value.

In [59]:
```
#Assigning predictors and outcome variables
predictors = list(df.columns)
outcome = 'Median_House_Value'
predictors.remove(outcome)
print(predictors)
print(outcome)

#Assigning Xs,ys for train, valid, and test data sets
train_X = train[predictors] #train_X, train_y for train data set
train_y = train[outcome]
valid_X = valid[predictors]  #valid_X, valid_y for valid data set
valid_y = valid[outcome]
X = test[predictors]    #X, y for test data set
```

```
y = test[outcome]
```

```
['Median_Income', 'Median_Age', 'Households', 'Distance_to_coast', 'Distance_to_LA', 'Distance_to_SanFrancisco']
Median_House_Value
```

MULTIPLE LINEAR REGRESSION MODEL

For the Multiple Linear Regression model, the chosen parameter of stepwise regression was used for the best predictors.

In [60]:
```
#Multiple Linear Regression Model with stepwise regression for selecting the best variables

def train_model(variables):
    if len(variables) == 0:
        return None
    model = LinearRegression()
    return model.fit(train_X[variables], train_y)

def score_model(model, variables):
    if len(variables) == 0:
        return mean_squared_error(train_y, [train_y.mean()] * len(train_y), model)
    return mean_squared_error(train_y, model.predict(train_X[variables]))


linear_model, best_variables1 = stepwise_selection(predictors, train_model, score_model,
                                                   direction='stepwise', verbose=True)
print(best_variables1)
linear_predictors = best_variables1

```

```
Variables: Median_Income, Median_Age, Households, Distance_to_coast, Distance_to_LA, Distance_to_SanFrancisco
Start: score=0.28, constant
Step: score=0.16, add Median_Income
Step: score=0.11, add Distance_to_coast
Step: score=0.11, add Households
Step: score=0.11, add Median_Age
Step: score=0.11, add Distance_to_LA
Step: score=0.11, add Distance_to_SanFrancisco
Step: score=0.11, unchanged None
['Median_Income', 'Distance_to_coast', 'Households', 'Median_Age', 'Distance_to_LA', 'Distance_to_SanFrancisco']
```

The best predictor was Median_Income with a score of 01.6, and the rest were Distance_to_coast, Households, Median_Age, Distance_to_LA, Distance_to_San_Francisco with the same score of 0.11. All the predictors were used for training the model since all revealed similar scores.

In [61]:
```
#Regression Statistics by MULTIPLE LINEAR REGRESSION MODEL
print('Train Performance')
regressionSummary(train_y, linear_model.predict(train_X[best_variables1]))

print('Validation Performance')
regressionSummary(valid_y, linear_model.predict(valid_X[best_variables1]))

print('Test Performance')
regressionSummary(y, linear_model.predict(X[best_variables1]))
```

```
Train Performance

Regression statistics

                      Mean Error (ME) : -0.0000
       Root Mean Squared Error (RMSE) : 0.3265
            Mean Absolute Error (MAE) : 0.2516
          Mean Percentage Error (MPE) : -0.0750
Mean Absolute Percentage Error (MAPE) : 2.1019
Validation Performance

Regression statistics

                      Mean Error (ME) : -0.0043
       Root Mean Squared Error (RMSE) : 0.3250
            Mean Absolute Error (MAE) : 0.2511
          Mean Percentage Error (MPE) : -0.1144
Mean Absolute Percentage Error (MAPE) : 2.1027
Test Performance

Regression statistics

                      Mean Error (ME) : 0.0045
       Root Mean Squared Error (RMSE) : 0.3183
            Mean Absolute Error (MAE) : 0.2474
          Mean Percentage Error (MPE) : -0.0365
Mean Absolute Percentage Error (MAPE) : 2.0652
```

REGRESSION TREE MODEL

The Stepwise regression was also used for the Regression Tree model.

In [62]:
```
# regression tree
def train_model(variables):
    if len(variables) == 0:
        return None
    model = DecisionTreeRegressor()
    return model.fit(train_X[variables], train_y)

def score_model(model, variables):
    if len(variables) == 0:
        return mean_squared_error(train_y, [train_y.mean()] * len(train_y), model)
    return mean_squared_error(train_y, model.predict(train_X[variables]))

tree_model, best_variables = stepwise_selection(predictors, train_model, score_model, direction='forward', verbose=
print(best_variables)
```

```
Variables: Median_Income, Median_Age, Households, Distance_to_coast, Distance_to_LA, Distance_to_SanFrancisco
Start: score=0.28, constant
Step: score=0.01, add Distance_to_coast
Step: score=0.00, add Median_Income
Step: score=0.00, unchanged None
['Distance_to_coast', 'Median_Income']
```

The model was built using the best predictors, Distance_to_coast and Median_Income.

In [63]:
```
#Regression Statistics by REGRESSION TREE MODEL
print('Train Performance')
regressionSummary(train_y, tree_model.predict(train_X[best_variables]))
print('Validation performance')
regressionSummary(valid_y, tree_model.predict(valid_X[best_variables]))
print('Test performance')
regressionSummary(y, tree_model.predict(X[best_variables]))
```

```
Train Performance

Regression statistics

                      Mean Error (ME) : 0.0000
       Root Mean Squared Error (RMSE) : 0.0000
            Mean Absolute Error (MAE) : 0.0000
          Mean Percentage Error (MPE) : 0.0000
Mean Absolute Percentage Error (MAPE) : 0.0000
Validation performance

Regression statistics

                      Mean Error (ME) : -0.0180
       Root Mean Squared Error (RMSE) : 0.4373
            Mean Absolute Error (MAE) : 0.3285
          Mean Percentage Error (MPE) : -0.2208
Mean Absolute Percentage Error (MAPE) : 2.7479
Test performance

Regression statistics

                      Mean Error (ME) : -0.0049
       Root Mean Squared Error (RMSE) : 0.4417
            Mean Absolute Error (MAE) : 0.3343
          Mean Percentage Error (MPE) : -0.1088
Mean Absolute Percentage Error (MAPE) : 2.7865
```

Since it was all zero percent error for train data set, overfitting was noted for Regression Tree model.

RANDOM FOREST

Random Forest model was built using n_estimators of 500 and random_state 1. The two most Important features were Distance_to_coast and Median_Income.

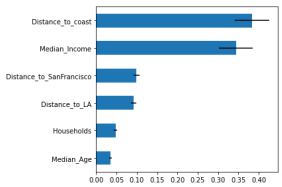
In [64]:
```
rf = RandomForestRegressor(n_estimators=500, random_state=1)
rf.fit(train_X, train_y)
```

Out[64]:
```
RandomForestRegressor(n_estimators=500, random_state=1)
```

In [65]:
```
#To see the most important features

importances = rf.feature_importances_
std = np.std([tree.feature_importances_ for tree in rf.estimators_], axis=0)

df = pd.DataFrame({'feature': train_X.columns, 'importance': importances, 'std': std})
df = df.sort_values('importance')
print(df)

```

```
ax = df.plot(kind='barh', xerr='std', x='feature', legend=False)
ax.set_ylabel('')

plt.tight_layout()
plt.show()
```

```
                   feature  importance       std
1               Median_Age    0.034889  0.003350
2               Households    0.047419  0.004691
4          Distance_to_LA     0.091673  0.006557
5  Distance_to_SanFrancisco    0.099123  0.006975
0            Median_Income    0.343972  0.041653
3         Distance_to_coast    0.382924  0.042084
```



Random Forest model was built using n_estimators of 500 and random_state 1. The two most Important features were Distance_to_coast and Median_Income.

In [66]:
```
#Regression Statistics by RANDOM FOREST MODEL
regressionSummary(train_y, rf.predict(train_X))
regressionSummary(valid_y, rf.predict(valid_X))
regressionSummary(y, rf.predict(X))
```

```
Regression statistics

                    Mean Error (ME) : 0.0006
     Root Mean Squared Error (RMSE) : 0.0889
          Mean Absolute Error (MAE) : 0.0633
        Mean Percentage Error (MPE) : -0.0107
Mean Absolute Percentage Error (MAPE) : 0.5281

Regression statistics

                    Mean Error (ME) : -0.0071
     Root Mean Squared Error (RMSE) : 0.2435
          Mean Absolute Error (MAE) : 0.1716
        Mean Percentage Error (MPE) : -0.1043
Mean Absolute Percentage Error (MAPE) : 1.4368

Regression statistics
```

Lasso Regression

We're going to test Ridge and Lasso regression as they're two simple techniques that help reduce model complexity and prevent over-fitting that might occur from simple linear regression.

In [67]:
```
lasso = Lasso()
lasso.fit(train_X,train_y)
train_score=lasso.score(train_X, train_y)
test_score=lasso.score(valid_X, valid_y)
coeff_used = np.sum(lasso.coef_!=0)

print ("training score:", train_score)
print ("test score: ", test_score)
print ("number of features used: ", coeff_used)
```

```
training score: 0.35446376917787525
test score:  0.3807762875599566
number of features used:  4
```

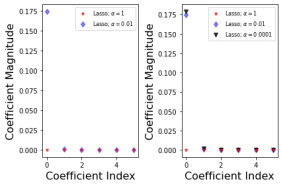
In [68]:
```
# we'll tune lasso to alpha = 0.01

lasso_001 = Lasso(alpha=0.01, max_iter=10e5)
lasso_001.fit(train_X, train_y)

train_score001=lasso_001.score(train_X, train_y)
test_score001=lasso_001.score(valid_X, valid_y)
coeff_used001 = np.sum(lasso_001.coef_!=0)

print ("training score for alpha=0.01:", train_score001)
print ("test score for alpha =0.01: ", test_score001)
print ("number of features used: for alpha =0.01:", coeff_used001)
```

```
training score for alpha=0.01: 0.6160676953562032
test score for alpha =0.01:  0.6307185175715928
number of features used: for alpha =0.01: 6
```

In [69]:
```
# we'll tune lasso to alpha = 0.0001

lasso_00001 = Lasso(alpha=0.0001, max_iter=10e5)
lasso_00001.fit(train_X, train_y)

train_score00001=lasso_00001.score(train_X, train_y)
test_score00001=lasso_00001.score(valid_X, valid_y)
coeff_used00001 = np.sum(lasso_00001.coef_!=0)

print ("training score for alpha=0.0001:", train_score00001)
print ("test score for alpha =0.0001: ", test_score00001)
print ("number of features used: for alpha =0.0001:", coeff_used00001)
```

```
training score for alpha=0.0001: 0.6162504994217306
test score for alpha =0.0001:  0.6312128947118348
number of features used: for alpha =0.0001: 6
```

In [70]:
```
# plot the coefficients

plt.subplot(1,2,1)
plt.plot(lasso.coef_,alpha=0.7,linestyle='none',marker='*',markersize=5,color='red',label=r'Lasso; $\alpha = 1$',zo
plt.plot(lasso_001.coef_,alpha=0.5,linestyle='none',marker='d',markersize=6,color='blue',label=r'Lasso; $\alpha = 0

plt.xlabel('Coefficient Index',fontsize=16)
plt.ylabel('Coefficient Magnitude',fontsize=16)
plt.legend(fontsize=8,loc=1)
plt.subplot(1,2,2)
plt.plot(lasso.coef_,alpha=0.7,linestyle='none',marker='*',markersize=5,color='red',label=r'Lasso; $\alpha = 1$',zo
plt.plot(lasso_001.coef_,alpha=0.5,linestyle='none',marker='d',markersize=6,color='blue',label=r'Lasso; $\alpha = 0
plt.plot(lasso_00001.coef_,alpha=0.8,linestyle='none',marker='v',markersize=6,color='black',label=r'Lasso; $\alpha
plt.xlabel('Coefficient Index',fontsize=16)
plt.ylabel('Coefficient Magnitude',fontsize=16)
plt.legend(fontsize=8,loc=1)
plt.tight_layout()
plt.show()
```



In [71]:
```
#Regression Statistics by Lasso Regression MODEL
regressionSummary(train_y, lasso_00001.predict(train_X))
regressionSummary(valid_y, lasso_00001.predict(valid_X))
regressionSummary(y, lasso_00001.predict(X))
```

```
Regression statistics

                    Mean Error (ME) : 0.0000
     Root Mean Squared Error (RMSE) : 0.3265
          Mean Absolute Error (MAE) : 0.2516
        Mean Percentage Error (MPE) : -0.0750
Mean Absolute Percentage Error (MAPE) : 2.1019

Regression statistics

                    Mean Error (ME) : -0.0043
     Root Mean Squared Error (RMSE) : 0.3250
          Mean Absolute Error (MAE) : 0.2511
        Mean Percentage Error (MPE) : -0.1144
Mean Absolute Percentage Error (MAPE) : 2.1027

Regression statistics
```

```
                    Mean Error (ME) : 0.0045
         Root Mean Squared Error (RMSE) : 0.3183
              Mean Absolute Error (MAE) : 0.2474
            Mean Percentage Error (MPE) : -0.0365
Mean Absolute Percentage Error (MAPE) : 2.0652
```

Ridge Regression

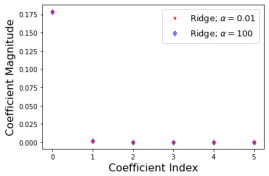
In [72]:
```
# here we'll tune alpha to 0.01 & 100, the higher the alpha, more limitations on the coefficients.

# alpha = 0.1
RR = Ridge(alpha=0.01)
RR.fit(train_X, train_y)

Ridge_train_score=RR.score(train_X, train_y)
Ridge_test_score=RR.score(valid_X, valid_y)
coeff_used = np.sum(lasso.coef_!=0)

print ("training score:", Ridge_train_score)
print ("test score: ", Ridge_test_score)
print ("number of features used: ", coeff_used)
```

```
training score: 0.6162505177038775
test score:  0.6312162077642185
number of features used:  4
```

In [73]:
```
# alpha = 100
rr100 = Ridge(alpha=100) #  comparison with alpha value
rr100.fit(train_X, train_y)

Ridge_train_score100 = rr100.score(train_X, train_y)
Ridge_test_score100 = rr100.score(valid_X, valid_y)
coeff_used = np.sum(lasso.coef_!=0)

print ("training score:", Ridge_train_score100)
print ("test score: ", Ridge_test_score100)
print ("number of features used: ", coeff_used)
```

```
training score: 0.6162419109776894
test score:  0.6311403965344502
number of features used:  4
```

In [74]:
```
# plot the coefficients

plt.plot(RR.coef_,alpha=0.7,linestyle='none',marker='*',markersize=5,color='red',label=r'Ridge; $\alpha = 0.01$',zo
plt.plot(rr100.coef_,alpha=0.5,linestyle='none',marker='d',markersize=6,color='blue',label=r'Ridge; $\alpha = 100$'
plt.xlabel('Coefficient Index',fontsize=16)
plt.ylabel('Coefficient Magnitude',fontsize=16)
plt.legend(fontsize=13,loc=1)
plt.show()
```



In [75]:
```
#Regression Statistics by Ridge Regression MODEL
regressionSummary(train_y, rr100.predict(train_X))
regressionSummary(valid_y, rr100.predict(valid_X))
regressionSummary(y, rr100.predict(X))
```

```
Regression statistics

                    Mean Error (ME) : 0.0000
         Root Mean Squared Error (RMSE) : 0.3265
              Mean Absolute Error (MAE) : 0.2516
            Mean Percentage Error (MPE) : -0.0752
Mean Absolute Percentage Error (MAPE) : 2.1019

Regression statistics

                    Mean Error (ME) : -0.0043
         Root Mean Squared Error (RMSE) : 0.3250
              Mean Absolute Error (MAE) : 0.2511
            Mean Percentage Error (MPE) : -0.1150
Mean Absolute Percentage Error (MAPE) : 2.1030

Regression statistics

                    Mean Error (ME) : 0.0045
         Root Mean Squared Error (RMSE) : 0.3183
              Mean Absolute Error (MAE) : 0.2474
            Mean Percentage Error (MPE) : -0.0369
Mean Absolute Percentage Error (MAPE) : 2.0653
```

RESULTS AND FINAL MODEL SELECTION

In [76]:
```
#REGRESSION STATISTICS OF DIFFERNT MODELS (BASED ON THE TEST DATA)

print("MULTIPLE LINEAR REGRESSION:")
regressionSummary(y,linear_model.predict(X[best_variables1]))

print ("LASSO:")
regressionSummary(y,lasso_00001.predict(X))

print ("RIDGE:")
regressionSummary(y,rr100.predict(X))

print("REGRESSION TREE MODEl:")
regressionSummary(y, tree_model.predict(X[best_variables]))

print("RANDOM FOREST:")
regressionSummary(y, rf.predict(X))

#According to Regression Statistics and Residual Errors' plot, RANDOM FOREST DEEMS THE BEST MODEL.
```

```
MULTIPLE LINEAR REGRESSION:

Regression statistics

                    Mean Error (ME) : 0.0045
         Root Mean Squared Error (RMSE) : 0.3183
              Mean Absolute Error (MAE) : 0.2474
            Mean Percentage Error (MPE) : -0.0365
Mean Absolute Percentage Error (MAPE) : 2.0652
LASSO:

Regression statistics

                    Mean Error (ME) : 0.0045
         Root Mean Squared Error (RMSE) : 0.3183
              Mean Absolute Error (MAE) : 0.2474
            Mean Percentage Error (MPE) : -0.0365
Mean Absolute Percentage Error (MAPE) : 2.0652
RIDGE:

Regression statistics

                    Mean Error (ME) : 0.0045
         Root Mean Squared Error (RMSE) : 0.3183
              Mean Absolute Error (MAE) : 0.2474
            Mean Percentage Error (MPE) : -0.0369
Mean Absolute Percentage Error (MAPE) : 2.0653
REGRESSION TREE MODEl:

Regression statistics

                    Mean Error (ME) : -0.0049
         Root Mean Squared Error (RMSE) : 0.4417
              Mean Absolute Error (MAE) : 0.3343
            Mean Percentage Error (MPE) : -0.1088
Mean Absolute Percentage Error (MAPE) : 2.7865
RANDOM FOREST:

Regression statistics

                    Mean Error (ME) : 0.0007
         Root Mean Squared Error (RMSE) : 0.2300
              Mean Absolute Error (MAE) : 0.1647
            Mean Percentage Error (MPE) : -0.0341
Mean Absolute Percentage Error (MAPE) : 1.3736
```

In [77]:
```
#Printing R2 and RMSE metrics from each model

pred_test = linear_model.predict(X[best_variables1])
pred_RT_test = tree_model.predict(X[best_variables])
pred_RF_test = rf.predict(X)

print('Multiple Linear Regression:', CompareModels.R2AndRMSE(y_test=y, y_pred=pred_test))
```

```
print('Random Forest:', CompareModels.R2AndRMSE(y_test=y, y_pred= rf.predict(X)))

print('Regression Tree:', CompareModels.R2AndRMSE(y_test=y, y_pred=pred_RT_test))

print ('Lasso Regression:', CompareModels.R2AndRMSE(y_test=y, y_pred = lasso_00001.predict(X)))

print ('Ridge Regression:', CompareModels.R2AndRMSE(y_test =y, y_pred = rr100.predict(X)))
```

```
Multiple Linear Regression: (0.6282688626237674, 0.31829977423969413)
Random Forest: (0.8058548489465904, 0.23003034691641602)
Regression Tree: (0.2843100307821140 3, 0.44165619395979 83)
Lasso Regression: (0.6282666086798013, 0.3183007392231 1347)
Ridge Regression: (0.6282273903822614, 0.3183175293261262)
```

In [78]:

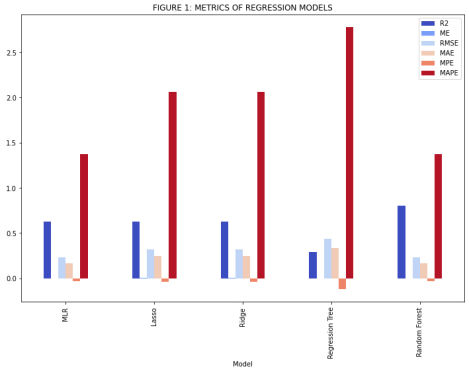
```
#Creating Summary Metrics Table of each model on test data set

data = {'Model': ['MLR', 'Lasso', 'Ridge', 'Regression Tree', 'Random Forest'],
        'R2': [0.6283, 0.6283, 0.6282, 0.2886, 0.8059],
        'ME': [0.0007,0.0045,0.0045,-0.0059,0.0007],
        'RMSE':[0.2300,0.3183,0.3183, 0.4403,0.2300],
        'MAE': [0.1647,0.2474,0.2474,0.3330,0.1647],
        'MPE': [-0.0341,-0.0365,-0.0369,-0.1172,-0.0341],
        'MAPE':[1.3736,2.0652,2.0653,2.7772,1.3736],
        }
Metric_Table = pd.DataFrame(data)
Metric_Table
```

Out[78]:

	Model	R2	ME	RMSE	MAE	MPE	MAPE
0	MLR	0.6283	0.0007	0.2300	0.1647	-0.0341	1.3736
1	Lasso	0.6283	0.0045	0.3183	0.2474	-0.0365	2.0652
2	Ridge	0.6282	0.0045	0.3183	0.2474	-0.0369	2.0653
3	Regression Tree	0.2886	-0.0059	0.4403	0.3330	-0.1172	2.7772
4	Random Forest	0.8059	0.0007	0.2300	0.1647	-0.0341	1.3736

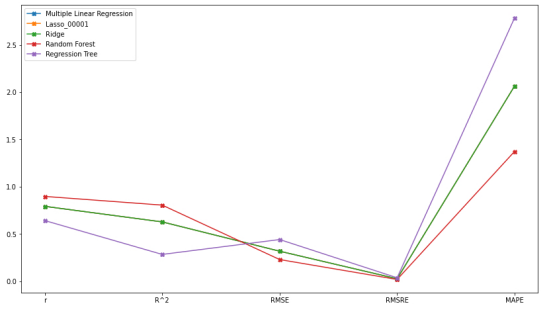
In [79]:

```
#Bargraph for Comparison of Metrics for Each Model

Metric_Table.plot(x="Model", y=["R2", "ME","RMSE","MAE","MPE",'MAPE'], kind="bar", figsize=(12,8), colormap='coolwa
plt.title('FIGURE 1: METRICS OF REGRESSION MODELS')
plt.show()
```



In [80]:

```
#Plotting graphs to show the best model and similarity between MLR, LASSO, and RIDGE.

plot = CompareModels()

plot.add(model_name = 'Multiple Linear Regression', y_test=y, y_pred= pred_test)

plot.add(model_name = 'Lasso_00001', y_test =y, y_pred = lasso_00001.predict(X))

plot.add(model_name = 'Ridge', y_test =y, y_pred = rr100.predict(X))

plot.add(model_name='Random Forest', y_test=y, y_pred=pred_RF_test)

plot.add(model_name = 'Regression Tree', y_test=y, y_pred=pred_RT_test)


plot.show(figsize=(14,8))

#RANDOM FOREST PERFORMED THE BEST.
#THE METRICS FOR MLR, RIDGE, AND LASSO WERE ALMOST IDENTICAL AND OVERLAPPED IN THE GRAPH.
#REGRESSION TREE YIELED THE LEAST R2 VALUE AND HIGHEST ERROR RATES.
```



From the Metric Table, metrics of Multiple Linear Regression, Ridge,and Lasso were almost identical with R2 value of 0.6282 (62.82%). Thus, in the above graph, all three overlap together and doesn't show any difference between these three models.

The metrics of Regression Tree for the train data set showed zero percent error revealing the overfitting behavior and the R2 value of 28.86%, the lowest score out of all the models.

Among them, Random Forest scored the highest R2 value of 80.59% best fitting the data set. The metrics for the Random Forest also ranked the lowest percentage errors of ME, RMSE, MAE, MPE, and MAPE as shown in the above figures. Thus, Random Forest deemed the best model for the prediction of California house prices.

Recommended Next Steps

The Random Forest scored the highest R2 value of 80.59% best fitting the data set. The metrics for the Random Forest also ranked the lowest percentage errors of ME, RMSE, MAE, MPE, and MAPE. Therefore, we would use this algorithm for all machine learning processes. The application of this research can be applied to applications across the real estate industry. The models had high precision outputs and could be utilized to create deliverables that can identify real estate evaluation to answer questions such as: Which house should I buy or build to maximize my return? Where or when should I do so? What is its optimum rent or sale price?

In [ ]:

```
```