House Price Prediction using Regression (Lasso and Ridge Regularization) and Random Forest

Dataset Link: https://www.kaggle.com/c/house-prices-advanced-regression-techniques/

In [ ]:
```
%matplotlib inline
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import cross_val_score, train_test_split, GridS
from sklearn.preprocessing import PolynomialFeatures, StandardScaler, MinMax
from sklearn.metrics import mean_squared_error
```

In [ ]:
```
df = pd.read_csv('/Users/gamingspectrum24/Documents/University Coursework/6t
print(df.shape)
```

```
(1460, 81)
```

Cleaning Data

In [ ]:
```
df.head()
```

Out[ ]:

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandCo
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	

5 rows × 81 columns

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

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1460 entries, 0 to 1459
Data columns (total 81 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   Id             1460 non-null   int64  
 1   MSSubClass     1460 non-null   int64  
 2   MSZoning       1460 non-null   object 
 3   LotFrontage    1201 non-null   float64
 4   LotArea        1460 non-null   int64  
 5   Street         1460 non-null   object 
 6   Alley          91 non-null     object 
 7   LotShape       1460 non-null   object 
 8   LandContour    1460 non-null   object 
 9   Utilities      1460 non-null   object 
 10  LotConfig      1460 non-null   object 
 11  LandSlope      1460 non-null   object 
 12  Neighborhood   1460 non-null   object 
 13  Condition1     1460 non-null   object 
 14  Condition2     1460 non-null   object 
 15  BldgType       1460 non-null   object 
 16  HouseStyle     1460 non-null   object 
 17  OverallQual    1460 non-null   int64  
 18  OverallCond    1460 non-null   int64  
 19  YearBuilt      1460 non-null   int64  
 20  YearRemodAdd   1460 non-null   int64  
 21  RoofStyle      1460 non-null   object 
 22  RoofMatl       1460 non-null   object 
 23  Exterior1st    1460 non-null   object 
 24  Exterior2nd    1460 non-null   object 
 25  MasVnrType     588 non-null    object 
 26  MasVnrArea     1452 non-null   float64
 27  ExterQual      1460 non-null   object 
 28  ExterCond      1460 non-null   object 
 29  Foundation     1460 non-null   object 
 30  BsmtQual       1423 non-null   object 
 31  BsmtCond       1423 non-null   object 
 32  BsmtExposure   1422 non-null   object 
 33  BsmtFinType1   1423 non-null   object 
 34  BsmtFinSF1     1460 non-null   int64  
 35  BsmtFinType2   1422 non-null   object 
 36  BsmtFinSF2     1460 non-null   int64  
 37  BsmtUnfSF      1460 non-null   int64  
 38  TotalBsmtSF    1460 non-null   int64  
 39  Heating        1460 non-null   object 
 40  HeatingQC      1460 non-null   object 
 41  CentralAir     1460 non-null   object 
 42  Electrical     1459 non-null   object 
 43  1stFlrSF       1460 non-null   int64  
 44  2ndFlrSF       1460 non-null   int64  
 45  LowQualFinSF   1460 non-null   int64  
 46  GrLivArea      1460 non-null   int64  
 47  BsmtFullBath   1460 non-null   int64  
 48  BsmtHalfBath   1460 non-null   int64  
 49  FullBath       1460 non-null   int64  
 50  HalfBath       1460 non-null   int64  
```

```
 51  BedroomAbvGr   1460 non-null   int64
 52  KitchenAbvGr   1460 non-null   int64
 53  KitchenQual    1460 non-null   object
 54  TotRmsAbvGrd   1460 non-null   int64
 55  Functional     1460 non-null   object
 56  Fireplaces     1460 non-null   int64
 57  FireplaceQu    770 non-null    object
 58  GarageType     1379 non-null   object
 59  GarageYrBlt    1379 non-null   float64
 60  GarageFinish   1379 non-null   object
 61  GarageCars     1460 non-null   int64
 62  GarageArea     1460 non-null   int64
 63  GarageQual     1379 non-null   object
 64  GarageCond     1379 non-null   object
 65  PavedDrive     1460 non-null   object
 66  WoodDeckSF     1460 non-null   int64
 67  OpenPorchSF    1460 non-null   int64
 68  EnclosedPorch  1460 non-null   int64
 69  3SsnPorch      1460 non-null   int64
 70  ScreenPorch    1460 non-null   int64
 71  PoolArea       1460 non-null   int64
 72  PoolQC         7 non-null      object
 73  Fence          281 non-null    object
 74  MiscFeature    54 non-null     object
 75  MiscVal        1460 non-null   int64
 76  MoSold         1460 non-null   int64
 77  YrSold         1460 non-null   int64
 78  SaleType       1460 non-null   object
 79  SaleCondition  1460 non-null   object
 80  SalePrice      1460 non-null   int64
dtypes: float64(3), int64(35), object(43)
memory usage: 924.0+ KB
```

We can see some features are numeric while others are text. There are also missing values in the dataset.

In [ ]:

```
df.isnull().sum()
miss_val = df.isnull().sum().sort_values(ascending=False)
miss_val = pd.DataFrame(data=df.isnull().sum().sort_values(ascending=False),

# Add a new column to the dataframe and fill it with the percentage of missi
miss_val['Percent'] = miss_val.MissvalCount.apply(lambda x : '{:.2f}'.format
miss_val = miss_val[miss_val.MissvalCount > 0]
miss_val
```

Out[ ]:

	MissvalCount	Percent
PoolQC	1453	99.52
MiscFeature	1406	96.30
Alley	1369	93.77
Fence	1179	80.75
MasVnrType	872	59.73
FireplaceQu	690	47.26
LotFrontage	259	17.74
GarageYrBlt	81	5.55
GarageCond	81	5.55
GarageType	81	5.55
GarageFinish	81	5.55
GarageQual	81	5.55
BsmtFinType2	38	2.60
BsmtExposure	38	2.60
BsmtQual	37	2.53
BsmtCond	37	2.53
BsmtFinType1	37	2.53
MasVnrArea	8	0.55
Electrical	1	0.07

We'll remove those features with a high percent of missing values such as PoolQC, MiscFeature, Alley, Fence, and FireplaceQu. Note that the LotFrontage feature has only 16% missing. This is relatively low so we can choose to replace the NaN values with the imputed mean of the column. We will remove the remainder rows with missing values.

In [ ]:
```
# replace NaNs in the column with the imputed mean of that column
#df['LotFrontage'].fillna(df['LotFrontage'].mean(), inplace=True)
```

In [ ]:
```
# drop columns with high missing values
df = df.drop(['Fence', 'MiscFeature', 'PoolQC','FireplaceQu','Alley'], axis=
```

In [ ]:
```
# drop rows with any missing values
df.dropna(inplace=True)
```

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

In [ ]:
```
# Check the dimension of the dataset
df.shape
```

Out[ ]: (455, 76)

The dataset is cleaned. It now has 1094 observations and 76 features.

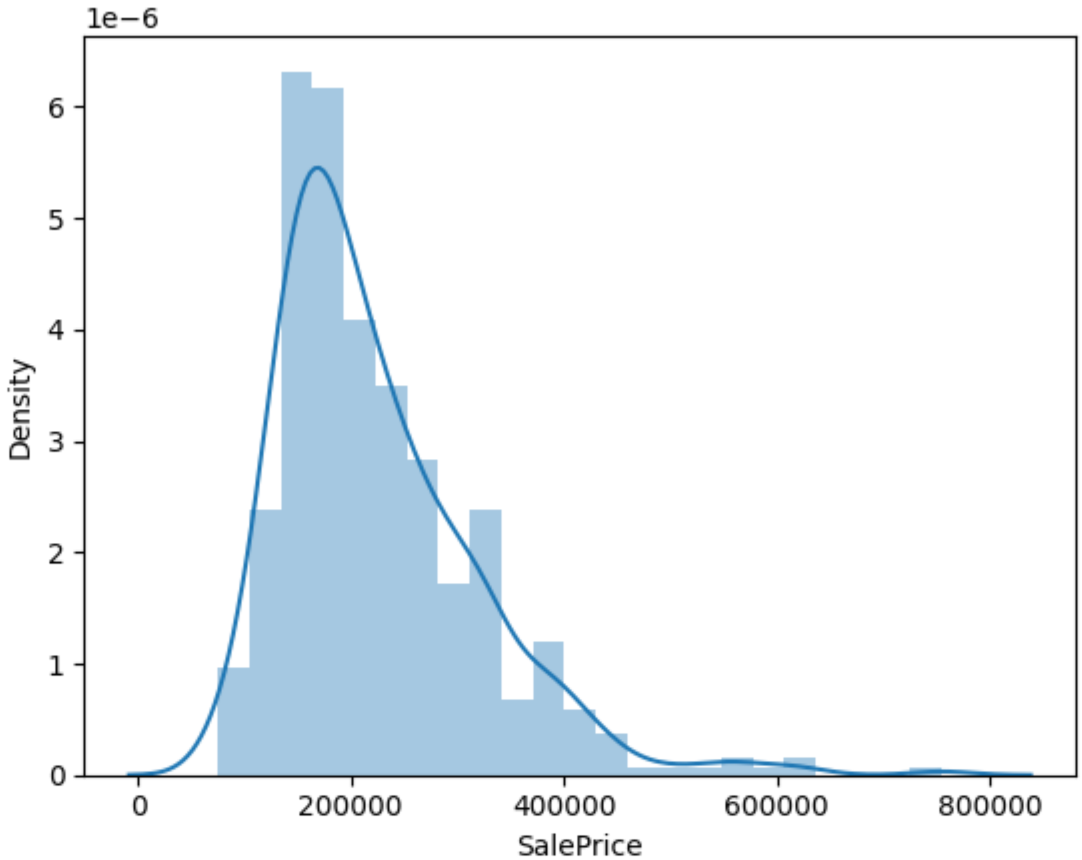
Explore data

Let's examine the data distributions of the features. We will start with the target variable, SalesPrice, to make sure it's normal distributed. This is important because most machine learning algorithms make the assumption that the data is normal distributed. When data fits a normal distribution, we can make statements about the population using analytical techniques.

In [ ]:
```
# Check distribution of target variable
sns.distplot(df.SalePrice)
```

```
/var/folders/9_/z2nkxcrx2zl6t6hr07fp81840000gn/T/ipykernel_49981/154314342
7.py:2: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(df.SalePrice)
```

Out[ ]: <Axes: xlabel='SalePrice', ylabel='Density'>

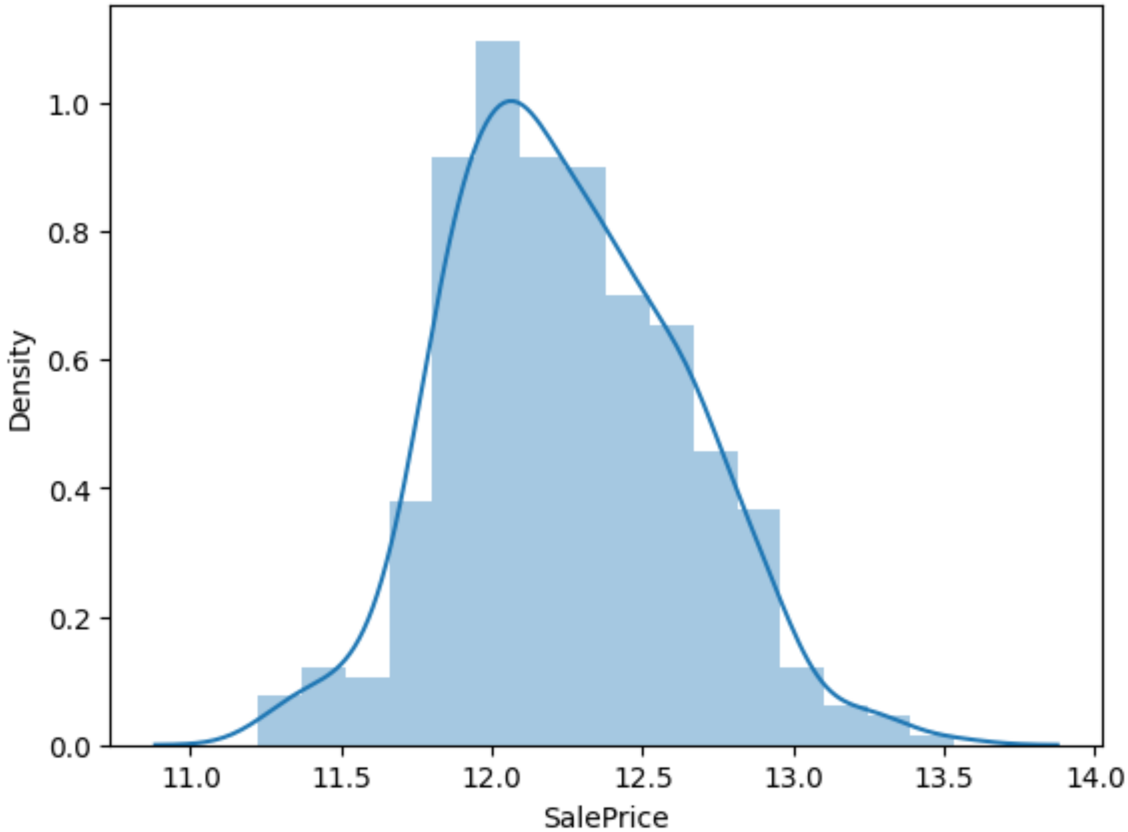


We can see the SalePrice distribution is skewed to the right. Let's transform it so that it follows a gaussian normal distribution.

In [ ]:
```
# Transform the target variable
sns.distplot(np.log(df.SalePrice))
```

```
/var/folders/9_/z2nkxcrx2zl6t6hr07fp81840000gn/T/ipykernel_49981/414447981
1.py:2: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(np.log(df.SalePrice))
```

Out[ ]: <Axes: xlabel='SalePrice', ylabel='Density'>



The data is now more normal distributed. We will use this transformed data in the dataframe and remove the skewed distribution:

In [ ]:
```
df['LogOfPrice'] = np.log(df.SalePrice)
df.drop(["SalePrice"], axis=1, inplace=True)
```

Let's check the skewness of the input feature.

In [ ]:
```
numeric_df = df.select_dtypes(include=['number'])
skewness = numeric_df.skew().sort_values(ascending=False)
print(skewness)
```

```
MiscVal          13.062759
PoolArea         12.559283
3SsnPorch         7.997668
BsmtFinSF2        6.127774
KitchenAbvGr      5.678190
EnclosedPorch     4.848662
BsmtHalfBath      4.290351
LotArea           4.082474
ScreenPorch       3.662176
TotalBsmtSF       2.592187
BsmtFinSF1        2.146508
MasVnrArea        2.128411
GrLivArea         1.895301
OpenPorchSF       1.667232
LotFrontage       1.623360
OverallCond       1.372570
1stFlrSF          1.343134
MSSubClass        1.106013
WoodDeckSF        0.857639
BsmtUnfSF         0.855804
2ndFlrSF          0.852999
TotRmsAbvGrd      0.736877
GarageArea        0.659486
HalfBath          0.472357
Fireplaces        0.335822
LogOfPrice        0.261798
BedroomAbvGr      0.252570
YrSold            0.158343
OverallQual       0.157532
BsmtFullBath      0.151391
MoSold            0.138804
Id                0.055842
LowQualFinSF      0.000000
GarageCars       -0.076837
FullBath         -0.508108
YearBuilt        -0.736565
GarageYrBlt      -0.764004
YearRemodAdd     -0.954359
dtype: float64
```

Values closer to zero are less skewed. The results show some features having a positive (right-tailed) or negative (left-tailed) skew. We can see YearBuilt is slightly skewed to the left but pretty much normal distributed while LotArea and PoolArea are highly skewed to the right. Highly skewed distributions in the dataset may benefit from data transforms in some way to improve our prediction accuracy.

Train-Test Split dataset

Before we can start modeling the data, we need to split the dataset into training and test sets. We will train the models with the training set and cross-validate with the test set. Recall we have lots of features in the dataset that are text. Most machine learning models require numerical input features. Since the process of converting text features to a numeric representation an involved task, we will only use the numeric features in our price prediction (for simplicity sake).

In [ ]:
```
# set the target and predictors
y = df.LogOfPrice  # target

# use only those input features with numeric data type
df_temp = df.select_dtypes(include=["int64","float64"])
X = df_temp.drop(["LogOfPrice"],axis=1)  # predictors
```

To split the dataset, we will use random sampling with 75/25 train-test split; that is, we'll use 75% of the dataset for training and set aside 25% for testing:

In [ ]:
```
# split the dataset into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = .25, r
```

Modeling

We will build four models and evaluate their performances with R-squared metric. Additionally, we will gain insights on the features that are strong predictors of house prices.

Linear Regression

In [ ]:
```
lr = LinearRegression()
# fit optimal linear regression line on training data, this performs gradien
lr.fit(X_train, y_train)
```

Out[ ]:
```
▾ LinearRegression

LinearRegression()
```

In [ ]:
```
# given our model and our fit, predict y_values using X_test set
yr_hat = lr.predict(X_test)
```

In [ ]:
```
# evaluate the algorithm with a test set
lr_score = lr.score(X_test, y_test)  # train test
print("Accuracy: ", lr_score)
```

```
Accuracy:  0.8583743620607989
```

Let's see how well the train-test split method performed. We will do cross-validation to see whether the model is over-fitting the data:

In [ ]:
```
# cross validation to find 'validate' score across multiple samples, automat
lr_cv = cross_val_score(lr, X, y, cv = 5, scoring= 'r2')
print("Cross-validation results: ", lr_cv)
print("R2: ", lr_cv.mean())
```

```
Cross-validation results:  [0.86179324 0.75396783 0.69743321 0.86432327 0.27
489253]
R2:  0.6904820157282376
```

It doesn't appear that for this train-test dataset, the model is not over-fitting the data (the cross-validation performance is very close in value). It may be a slightly over-fitted but we can't really tell by the R-squared metric alone. If it is over-fitted, we can do some data transforms or feature engineering to improve its performance. But our main objective initially is to spot-check a few algorithms and fine tune the model later on.

To help prevent over-fitting in which may result from simple linear regression, we can use regression models with regularization. Let's look at ridge and lasso next.

Regularization

The alpha parameter in ridge and lasso regularizes the regression model. The regression algorithms with regularization differ from linear regression in that they try to penalize those features that are not significant in our prediction. Ridge will try to reduce their effects (i.e., shrink their coeffients) in order to optimize all the input features. Lasso will try to remove the not-significant features by making their coefficients zero. In short, Lasso (L1 regularization) can eliminate the not-significant features, thus performing feature selection while Ridge (L2 regularization) cannot.

Ridge Regression

In [ ]:
```
ridge = Ridge(alpha = 1)  # sets alpha to a default value as baseline
ridge.fit(X_train, y_train)

ridge_cv = cross_val_score(ridge, X, y, cv = 5, scoring = 'r2')
print ("Cross-validation results: ", ridge_cv)
print ("R2: ", ridge_cv.mean())
```

```
Cross-validation results:  [0.86157021 0.75386273 0.69706725 0.86647877 0.27
327086]
R2:  0.6904499618990533
```

Lasso Regression

In [ ]:
```
lasso = Lasso(alpha = 0.001)  # sets alpha to almost zero as baseline
lasso.fit(X_train, y_train)

lasso_cv = cross_val_score(lasso, X, y, cv = 5, scoring = 'r2')
print ("Cross-validation results: ", lasso_cv)
print ("R2: ", lasso_cv.mean())
```

```
Cross-validation results:  [0.86170089 0.7493793  0.6949451  0.87115145 0.25
108923]
R2:  0.6856531942354757
```

Note: Alpha is the regularization parameter. The alpha values choosen for ridge and lasso serve as a starting point and are not likely the best. To determine the best alpha for the model, we can use GridSearch. We would feed GridSearch a range of alpha values and it will try them all in cross-validation to output the best one for the model.

Random Forest

In [ ]:
```
#rfr = RandomForestRegressor(n_estimators = 100, max_depth = 5, min_samples_
rfr = RandomForestRegressor()
rfr.fit(X_train, y_train) # gets the parameters for the rfr model
rfr_cv = cross_val_score(rfr,X, y, cv = 5, scoring = 'r2')
print("Cross Validation Score: ", rfr_cv)
print("R2: ", rfr_cv.mean())
```

```
Cross Validation Score:  [0.87526781 0.85520351 0.82641925 0.91666508 0.7660
2481]
R2:  0.8479160933435621
```

Random forest is an advanced decision tree based machine learning. It has a classification and a regression random forest algorithm. Its performance is slightly better than regression. Like regularization, we can optimize the model parameters for best performance using gridsearch.

In [ ]:
```
# Random forest determined feature importances
rfr.feature_importances_
```

Out[ ]:
```
array([6.32244756e-03, 4.61464072e-03, 1.68201420e-02, 3.71964796e-02,
       5.71886123e-01, 4.04399311e-03, 3.85718849e-02, 8.73575322e-03,
       1.44268931e-02, 2.22362976e-02, 2.69866813e-04, 1.19993209e-02,
       3.15063199e-02, 1.86424508e-02, 1.24669945e-02, 0.00000000e+00,
       7.60247090e-02, 1.32022804e-03, 1.61992611e-04, 1.98429147e-02,
       1.61517530e-03, 1.85198701e-03, 1.74767231e-04, 6.17775702e-03,
       6.90099883e-03, 4.64693485e-03, 2.85105079e-02, 2.07067759e-02,
       6.52205453e-03, 1.41087427e-02, 3.54858452e-04, 1.26413197e-04,
       5.07983536e-04, 2.83279114e-04, 9.00001198e-06, 7.90306878e-03,
       2.51024340e-03])
```

Plotting the Feature Importance

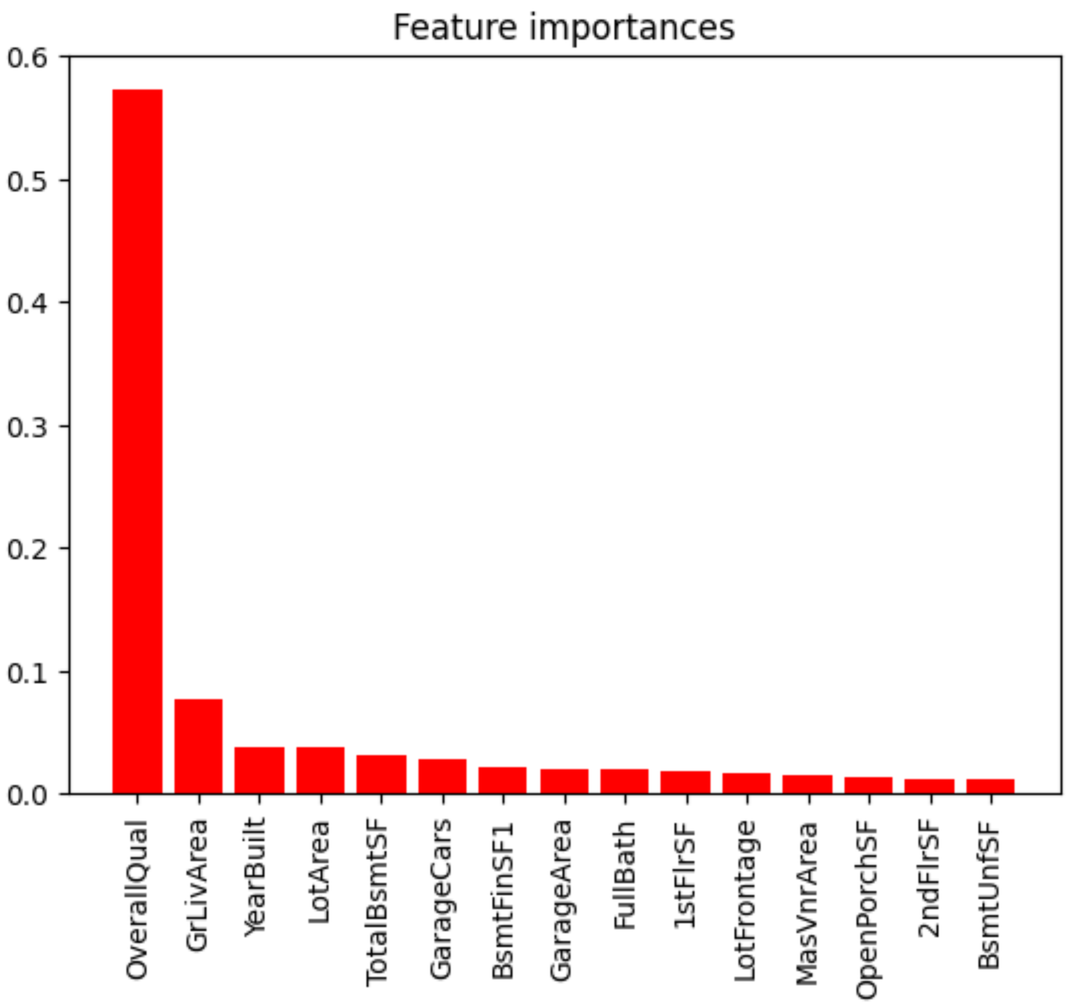
Let's see the features that are the most promising predictors:

In [ ]:
```
importance = rfr.feature_importances_

# map feature importance values to the features
feature_importances = zip(importance, X.columns)
#list(feature_importances)

sorted_feature_importances = sorted(feature_importances, reverse = True)
#print(sorted_feature_importances)
```

```
top_15_predictors = sorted_feature_importances[0:15]
values = [value for value, predictors in top_15_predictors]
predictors = [predictors for value, predictors in top_15_predictors]
print(predictors)
```

```
['OverallQual', 'GrLivArea', 'YearBuilt', 'LotArea', 'TotalBsmtSF', 'GarageC
ars', 'BsmtFinSF1', 'GarageArea', 'FullBath', '1stFlrSF', 'LotFrontage', 'Ma
sVnrArea', 'OpenPorchSF', '2ndFlrSF', 'BsmtUnfSF']
```

In [ ]:
```
# Plot the feature importances of the forest
plt.figure()
plt.title("Feature importances")
plt.bar(range(len(predictors)), values,color="r", align="center");
plt.xticks(range(len(predictors)), predictors, rotation=90);
```



Conclusion

Random Forest is the most accurate model for predicting the house price. It scored an estimated accuracy of 85%, out performing the regression models (linear, ridge, and lasso) by about 2%. Random Forest determined the overall quality of a home is by far the most important predictor. Following are the size of above grade (ground) living area and the size of total basement square footage. Surprisingly, the lot area did not rank as high as I had expected.