# Predicting House Prices with Machine Learning (Top 4% of Kaggle)

Eric Kim, B.A.

Silicon Valley, CA

September 7, 2017

eric@kimanalytics.com
https://www.kimanalytics.com
https://github.com/kimanalytics/Predicting-House-Prices-with-Machine-Learning

# Table of Contents

**Introduction** **3**

**Part 1 - Understanding the Client and their Problem** **4**

**Part 2 - Load Data and Packages** **4**

2.1 Load Packages 4

1.2 Load Data 5

**Part 3 - Analyzing the Test Variable (Sale Price)** **6**

3.1 Description 6

3.2 Histogram 6

**Part 4 - Multivariable Analysis** **7**

4.1 Categorical Data 7

4.2 Numerical Data 8

4.3 Correlated Features with Sale Price 8

**Part 5 - Impute Missing Data and Clean Data** **11**

5.1 Missing Ratio 11

5.2 Imputing Missing Values 13

**Part 6 - Feature Transformation/Engineering** **15**

**Part 7 - Modeling and Predictions** **18**

7.1 Cross-validation with k-folds 18

7.2 Modeling 18

**Conclusions and Recommendations for House Prices** **21**

House Buyers 21

House Sellers 21

Discussion 21

**Improvements to Study** **22**

## Introduction

Ask a home buyer to describe their dream house, and they probably won't begin with the height of the basement ceiling or the proximity to an east-west railroad. But this Kaggle competition dataset proves that much more influences price negotiations than the number of bedrooms or a white-picket fence.

With 79 explanatory variables describing (almost) every aspect of residential homes in Ames, Iowa, this competition challenges data scientists to predict the final price of each home.

A house value is simply more than location and square footage. Like the features that make up a person, an educated party would want to know all aspects that give a house its value. We are going to take advantage of all of the feature variables available to use and use it to analyze and predict house prices. We are going to break everything into logical steps that allow us to ensure the cleanest, most realistic data for our model to make accurate predictions from.

- Understanding the Client and their Problem
- Load Data and Packages
- Analyzing the Test Variable (Sale Price)
- Multivariable Analysis
- Impute Missing Data and Clean Data
- Feature Transformation/Engineering
- Modeling and Predictions

## Part 1 - Understanding the Client and their Problem

A benefit to this study is that we can have two clients at the same time! (Think of being a divorce lawyer for both interested parties) However, in this case, we can have both clients with no conflict of interest!

Client Housebuyer: This client wants to find their next dream home with a reasonable price tag. They have their locations of interest ready. Now, they want to know if the house price matches the house value. With this study, they can understand which features (ex. Number of bathrooms, location, etc.) influence the final price of the house. If all matches, they can ensure that they are getting a fair price.

Client Houseseller: Think of the average house-flipper. This client wants to take advantage of the features that influence a house price the most. They typically want to buy a house at a low price and invest on the features that will give the highest return. For example, buying a house at a good location but small square footage. The client will invest on making rooms at a small cost to get a large return.

# Part 2 - Load Data and Packages

## 2.1 Load Packages

For our packages, we will use pandas, numpy, scipy, seaborn, and matplotlib for data visualization. We will use scikit-learn with xgboost and lightgbm for machine learning.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib
import matplotlib.pyplot as plt
import warnings
import xgboost as xgb
import lightgbm as lgb
from scipy.stats import skew
from scipy import stats
from scipy.stats.stats import pearsonr
from scipy.stats import norm
from collections import Counter
from sklearn.linear_model import LinearRegression,LassoCV, Ridge, LassoLarsCV,ElasticNetCV
from sklearn.model_selection import GridSearchCV, cross_val_score, learning_curve
from sklearn.ensemble import RandomForestRegressor, AdaBoostRegressor, ExtraTreesRegressor,
GradientBoostingRegressor
from sklearn.preprocessing import StandardScaler, Normalizer, RobustScaler
warnings.filterwarnings('ignore')
sns.set(style='white', context='notebook', palette='deep')
%config InlineBackend.figure_format = 'retina' #set 'png' here when working on notebook
%matplotlib inline
```

## 1.2 Load Data

Here, we load up our train and test set. After, we take a look at the amount of samples and features.

```
# Load train and Test set
train = pd.read_csv("../input/train.csv")
test = pd.read_csv("../input/test.csv")

# Check the numbers of samples and features
print("The train data size before dropping Id feature is : {} ".format(train.shape))
print("The test data size before dropping Id feature is : {} ".format(test.shape))
```

```
# Save the 'Id' column
train_ID = train['Id']
test_ID = test['Id']

# Now drop the 'Id' column since it's unnecessary for the prediction process.
train.drop("Id", axis = 1, inplace = True)
test.drop("Id", axis = 1, inplace = True)

# Check data size after dropping the 'Id' variable
print("\nThe train data size after dropping Id feature is : {} ".format(train.shape))
print("The test data size after dropping Id feature is : {} ".format(test.shape))

The train data size before dropping Id feature is : (1460, 81)
The test data size before dropping Id feature is : (1459, 80)
The train data size after dropping Id feature is : (1460, 80)
The test data size after dropping Id feature is : (1459, 79)
```

Although we don't have the largest dataset, we do have an immense amount of features. Working with 80 features will be very interesting!

## Part 3 - Analyzing the Test Variable (Sale Price)

Let's check out the most interesting feature in this study: Sale Price. Important Note: This data is from Ames, Iowa. The location is extremely correlated with Sale Price. (I had to take a double-take at a point, since I consider myself a house-browsing enthusiast)

### 3.1 Description

We get a brief description of the 'Sale Price' since it is the variable that we want to predict.

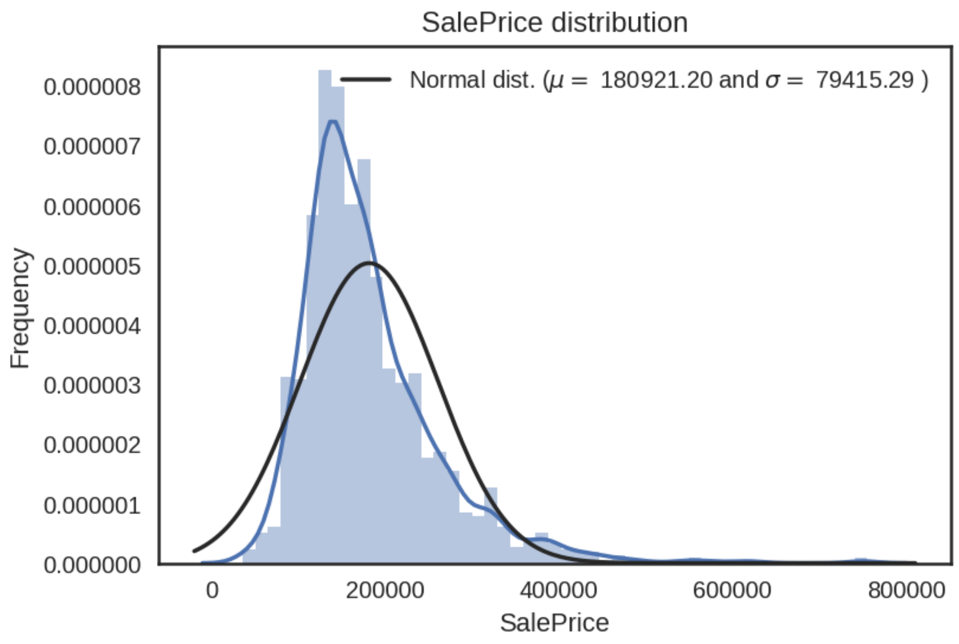
```
# Getting Description
train['SalePrice'].describe()

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
```

With an average house price of $180921, it seems like I should relocated to Iowa! Let's go deeper with a visualization.

## 3.2 Histogram

Here we get a histogram of our ‘Sale Price’ distribution.

```
# Plot Histogram
sns.distplot(train['SalePrice'] , fit=norm);

# Get the fitted parameters used by the function
(mu, sigma) = norm.fit(train['SalePrice'])
print( '\n mu = {:.2f} and sigma = {:.2f}\n'.format(mu, sigma))
plt.legend(['Normal dist. ($\mu=$ {:.2f} and $\sigma=$ {:.2f} )'.format(mu, sigma)],
            loc='best')
plt.ylabel('Frequency')
plt.title('SalePrice distribution')

fig = plt.figure()
res = stats.probplot(train['SalePrice'], plot=plt)
plt.show()

print("Skewness: %f" % train['SalePrice'].skew())
print("Kurtosis: %f" % train['SalePrice'].kurt())

Skewness: 1.882876
Kurtosis: 6.536282
```



Looks like a normal distribution? Not quite! Looking at the kurtosis score, we can see that there is a very nice peak. However, looking at the skewness score, we can see that the sale prices deviate from the normal distribution. Going to have to fix this later! We want our data to be as "normal" as possible.

## Part 4 - Multivariable Analysis

Let's check out all the variables! There are two types of features in housing data, categorical and numerical.

### 4.1 Categorical Data

Categorical data is just like it sounds. It is in categories. It isn't necessarily linear, but it follows some kind of pattern. For example, take a feature of "Downtown". The response is either "Near", "Far", "Yes", and "No". Back then, living in downtown usually meant that you couldn't afford to live in uptown. Thus, it could be implied that downtown establishments cost less to live in. However, today, that is not the case. (Thank you, hipsters!) So we can't really establish any particular order of response to be "better" or "worse" than the other.

```
# Checking Categorical Data
train.select_dtypes(include=['object']).columns

Index(['MSZoning', 'Street', 'Alley', 'LotShape', 'LandContour', 'Utilities',
       'LotConfig', 'LandSlope', 'Neighborhood', 'Condition1', 'Condition2',
       'BldgType', 'HouseStyle', 'RoofStyle', 'RoofMatl', 'Exterior1st',
       'Exterior2nd', 'MasVnrType', 'ExterQual', 'ExterCond', 'Foundation',
       'BsmtQual', 'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinType2',
       'Heating', 'HeatingQC', 'CentralAir', 'Electrical', 'KitchenQual',
       'Functional', 'FireplaceQu', 'GarageType', 'GarageFinish', 'GarageQual',
       'GarageCond', 'PavedDrive', 'PoolQC', 'Fence', 'MiscFeature',
       'SaleType', 'SaleCondition'],
      dtype='object')
```

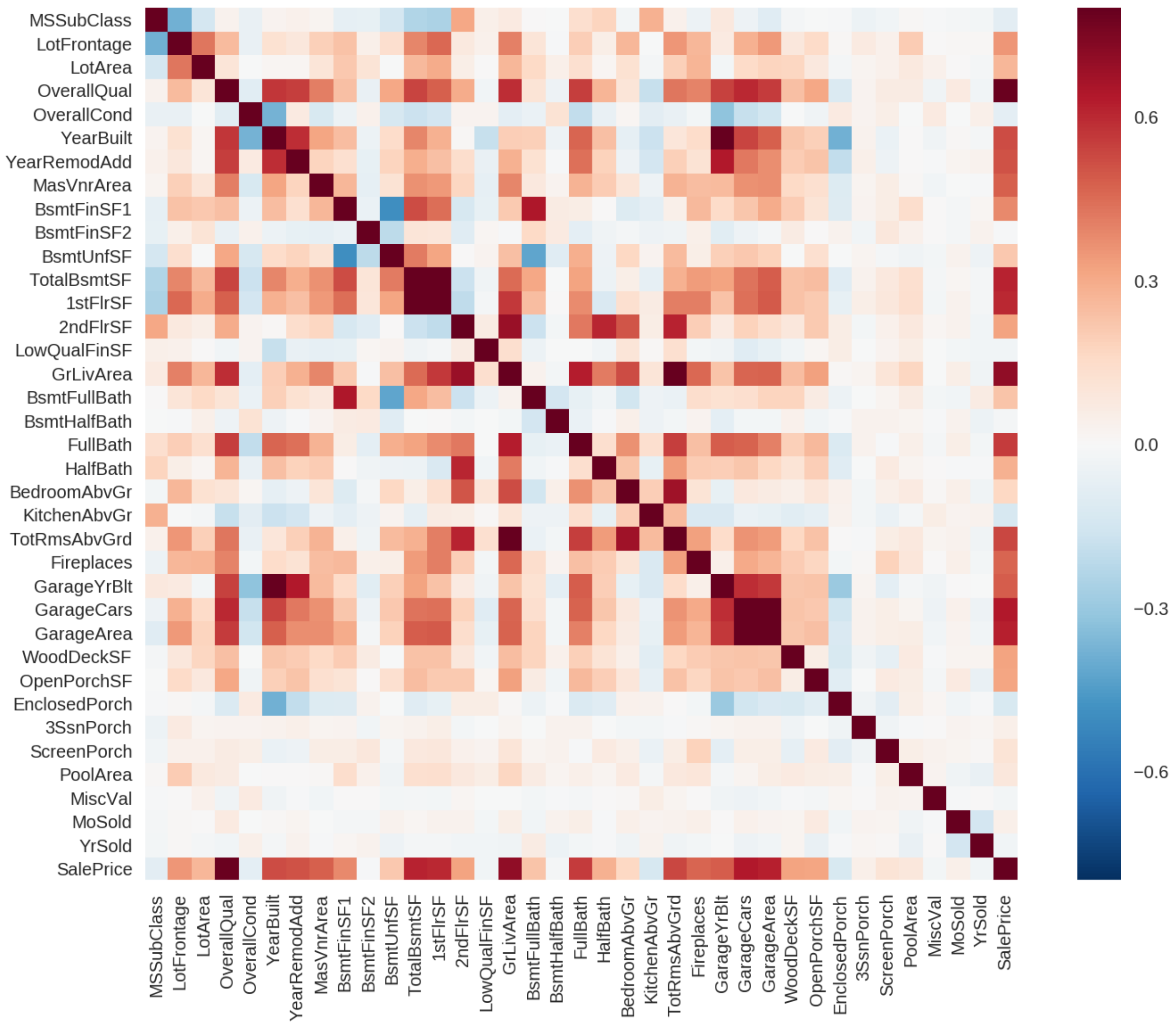
### 4.2 Numerical Data

Numerical data is data in number form. (Who could have thought!) These features are in a linear relationship with each other. For example, a 2,000 square foot place is 2 times "bigger" than a 1,000 square foot place. Plain and simple. Simple and clean.

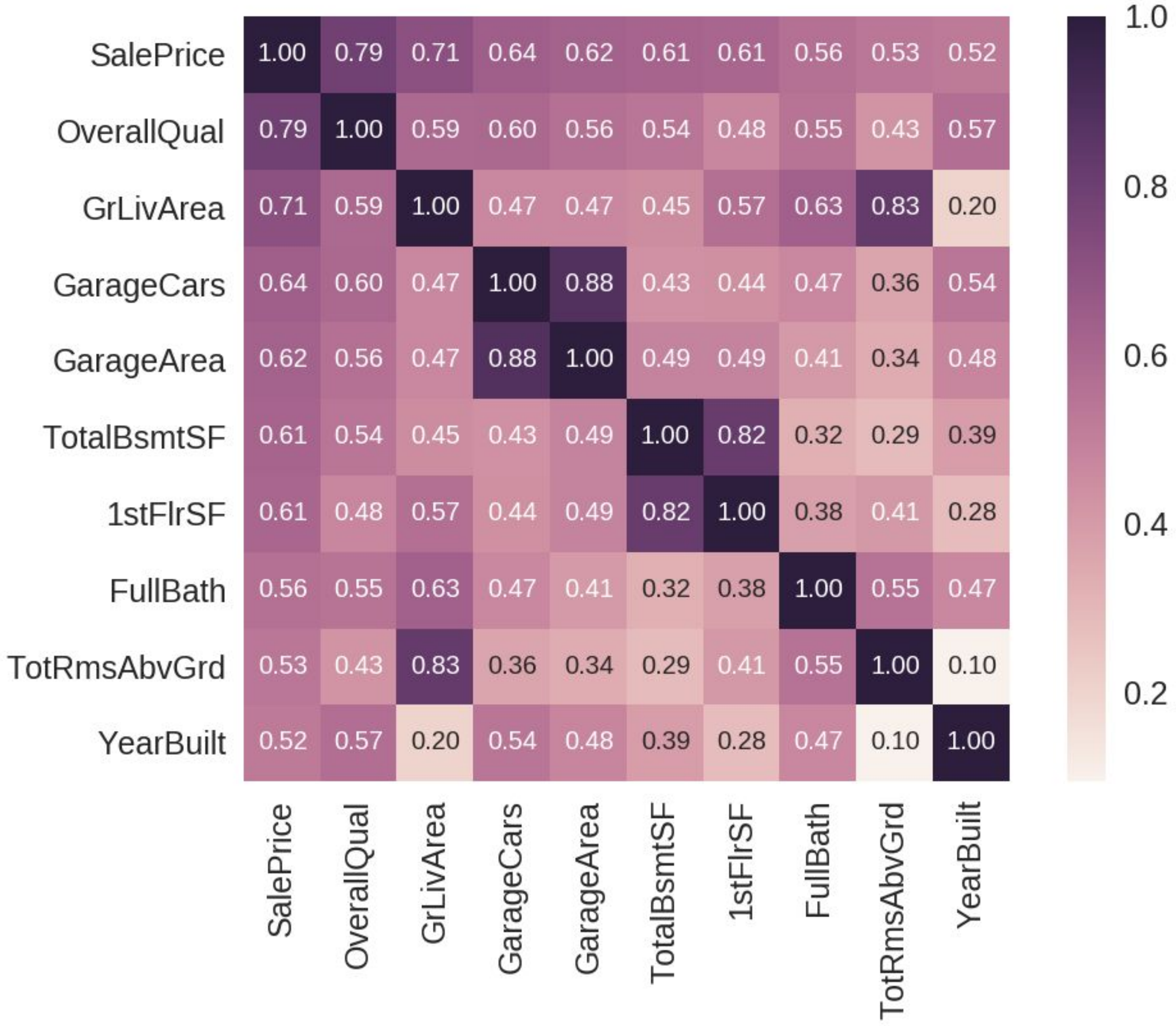
```
# Checking Numerical Data
train.select_dtypes(include=['int64','float64']).columns

Index(['MSSubClass', 'LotFrontage', 'LotArea', 'OverallQual', 'OverallCond',
       'YearBuilt', 'YearRemodAdd', 'MasVnrArea', 'BsmtFinSF1', 'BsmtFinSF2',
       'BsmtUnfSF', 'TotalBsmtSF', '1stFlrSF', '2ndFlrSF', 'LowQualFinSF',
       'GrLivArea', 'BsmtFullBath', 'BsmtHalfBath', 'FullBath', 'HalfBath',
       'BedroomAbvGr', 'KitchenAbvGr', 'TotRmsAbvGrd', 'Fireplaces',
       'GarageYrBlt', 'GarageCars', 'GarageArea', 'WoodDeckSF', 'OpenPorchSF',
       'EnclosedPorch', '3SsnPorch', 'ScreenPorch', 'PoolArea', 'MiscVal',
       'MoSold', 'YrSold', 'SalePrice'],
      dtype='object')
```

## 4.3 Correlated Features with Sale Price

With 81 features, how could we possibly tell which feature is most related to house prices? Good thing we have a correlation matrix.



It's a nice overview, but oh man is that a lot of data to look at. Let's zoom into the top 10 features most related to Sale Price.



| | SalePrice | OverallQual | GrLivArea | GarageCars | GarageArea | TotalBsmtSF | 1stFlrSF | FullBath | TotRmsAbvGrd | YearBuilt |
|---|---|---|---|---|---|---|---|---|---|---|
| SalePrice | 1.00 | 0.79 | 0.71 | 0.64 | 0.62 | 0.61 | 0.61 | 0.56 | 0.53 | 0.52 |
| OverallQual | 0.79 | 1.00 | 0.59 | 0.60 | 0.56 | 0.54 | 0.48 | 0.55 | 0.43 | 0.57 |
| GrLivArea | 0.71 | 0.59 | 1.00 | 0.47 | 0.47 | 0.45 | 0.57 | 0.63 | 0.83 | 0.20 |
| GarageCars | 0.64 | 0.60 | 0.47 | 1.00 | 0.88 | 0.43 | 0.44 | 0.47 | 0.36 | 0.54 |
| GarageArea | 0.62 | 0.56 | 0.47 | 0.88 | 1.00 | 0.49 | 0.49 | 0.41 | 0.34 | 0.48 |
| TotalBsmtSF | 0.61 | 0.54 | 0.45 | 0.43 | 0.49 | 1.00 | 0.82 | 0.32 | 0.29 | 0.39 |
| 1stFlrSF | 0.61 | 0.48 | 0.57 | 0.44 | 0.49 | 0.82 | 1.00 | 0.38 | 0.41 | 0.28 |
| FullBath | 0.56 | 0.55 | 0.63 | 0.47 | 0.41 | 0.32 | 0.38 | 1.00 | 0.55 | 0.47 |
| TotRmsAbvGrd | 0.53 | 0.43 | 0.83 | 0.36 | 0.34 | 0.29 | 0.41 | 0.55 | 1.00 | 0.10 |
| YearBuilt | 0.52 | 0.57 | 0.20 | 0.54 | 0.48 | 0.39 | 0.28 | 0.47 | 0.10 | 1.00 |

Now let us get a descending list of most correlated features.

```
# Most correlated features list
most_corr = pd.DataFrame(cols)
most_corr.columns = ['Most Correlated Features']
most_corr

Most Correlated Features
0   SalePrice
1   OverallQual
2   GrLivArea
3   GarageCars
4   GarageArea
5   TotalBsmtSF
6   1stFlrSF
7   FullBath
```

```
8   TotRmsAbvGrd
9   YearBuilt
```

Well, the most correlated feature to Sale Price is... Sale Price?!? Of course. For the other 9, they are as listed. Here is a short description of each. (Thank you, data_description.txt!)

OverallQual: Rates the overall material and finish of the house (1 = Very Poor, 10 = Very Excellent)
GrLivArea: Above grade (ground) living area square feet
GarageCars: Size of garage in car capacity
GarageArea: Size of garage in square feet
TotalBsmtSF: Total square feet of basement area
1stFlrSF: First Floor square feet
FullBath: Full bathrooms above grade
TotRmsAbvGrd: Total rooms above grade (does not include bathrooms)
YearBuilt: Original construction date

## Part 5 - Impute Missing Data and Clean Data

Important questions when thinking about missing data:

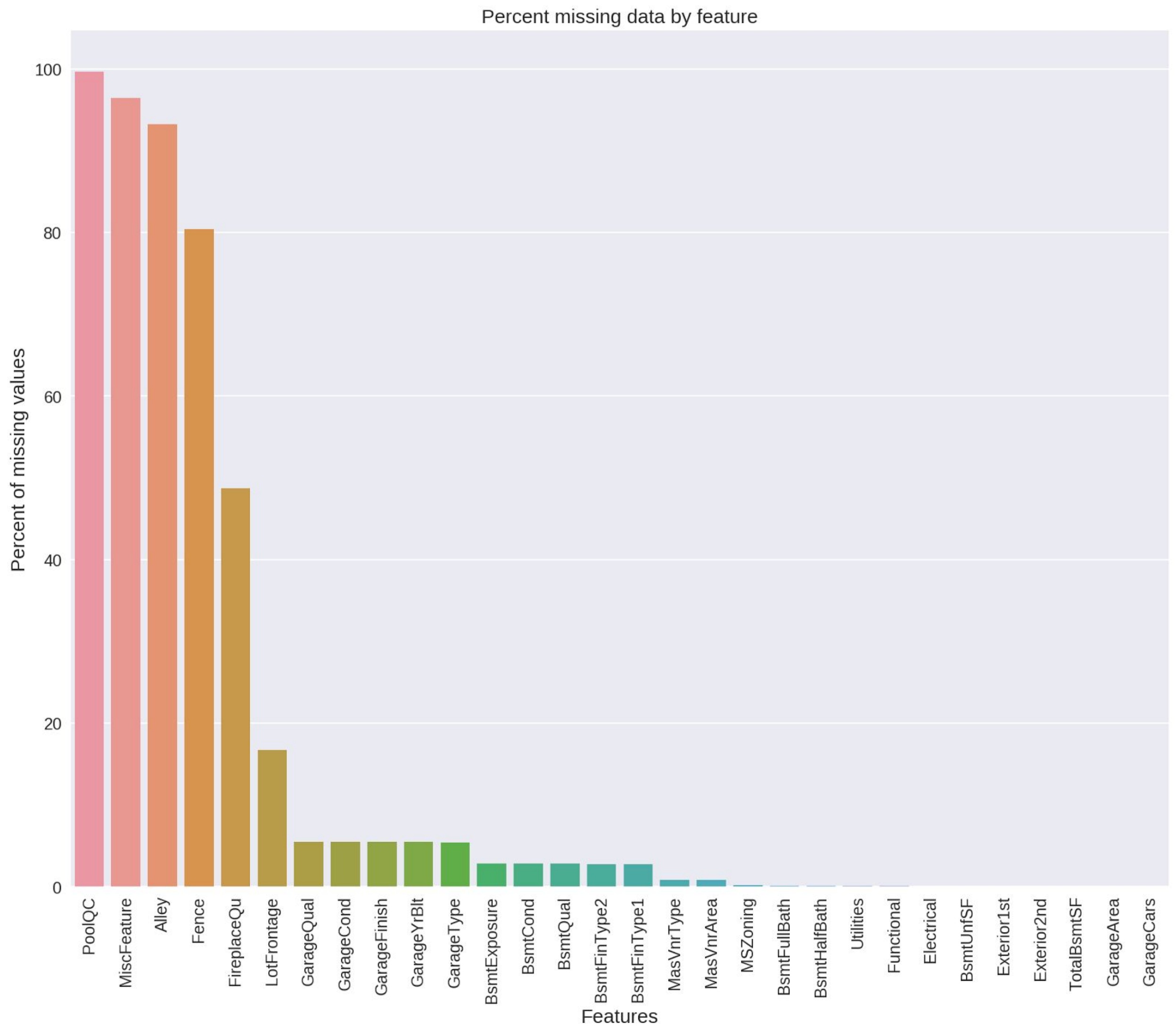
How prevalent is the missing data?
Is missing data random or does it have a pattern?
The answer to these questions is important for practical reasons because missing data can imply a reduction of the sample size. This can prevent us from proceeding with the analysis. Moreover, from a substantive perspective, we need to ensure that the missing data process is not biased and hiding an inconvenient truth.

### 5.1 Missing Ratio

Let's combine both training and test data into one dataset to impute missing values and do some cleaning.

```python
# Combining Datasets
ntrain = train.shape[0]
ntest = test.shape[0]
y_train = train.SalePrice.values
all_data = pd.concat((train, test)).reset_index(drop=True)
all_data.drop(['SalePrice'], axis=1, inplace=True)
print("Train data size is : {}".format(train.shape))
print("Test data size is : {}".format(test.shape))
print("Combined dataset size is : {}".format(all_data.shape))
```

```
Train data size is : (1448, 80)
Test data size is : (1459, 79)
Combined dataset size is : (2907, 79)

# Find Missing Ratio of Dataset
all_data_na = (all_data.isnull().sum() / len(all_data)) * 100
all_data_na = all_data_na.drop(all_data_na[all_data_na ==
0].index).sort_values(ascending=False)[:30]
missing_data = pd.DataFrame({'Missing Ratio' :all_data_na})
missing_data

Missing       Ratio
PoolQC        99.690402
MiscFeature   96.422429
Alley         93.223254
Fence         80.392157
FireplaceQu   48.710010
LotFrontage   16.683867
GarageQual    5.469556
GarageCond    5.469556
GarageFinish  5.469556
GarageYrBlt   5.469556
GarageType    5.400757
BsmtExposure  2.820777
BsmtCond      2.820777
BsmtQual      2.786378
BsmtFinType2  2.751978
BsmtFinType1  2.717578
MasVnrType    0.825593
MasVnrArea    0.791194
MSZoning      0.137599
BsmtFullBath  0.068799
BsmtHalfBath  0.068799
Utilities    0.068799
Functional   0.068799
Electrical   0.034400
BsmtUnfSF    0.034400
Exterior1st  0.034400
Exterior2nd  0.034400
TotalBsmtSF  0.034400
GarageArea   0.034400
GarageCars   0.034400
```



## 5.2 Imputing Missing Values

- PoolQC : data description says NA means "No Pool"
- MiscFeature : data description says NA means "no misc feature"
- Alley : data description says NA means "no alley access"
- Fence : data description says NA means "no fence"
- FireplaceQu : data description says NA means "no fireplace"
- LotFrontage : Since the area of each street connected to the house property most likely have a similar area to other houses in its neighborhood , we can fill in missing values by the median LotFrontage of the neighborhood.
- GarageType, GarageFinish, GarageQual and GarageCond : Replacing missing data with "None".

- GarageYrBlt, GarageArea and GarageCars : Replacing missing data with 0.
- BsmtFinSF1, BsmtFinSF2, BsmtUnfSF, TotalBsmtSF, BsmtFullBath and BsmtHalfBath: Replacing missing data with 0.
- BsmtQual, BsmtCond, BsmtExposure, BsmtFinType1 and BsmtFinType2 : For all these categorical basement-related features, NaN means that there isn't a basement.
- MasVnrArea and MasVnrType : NA most likely means no masonry veneer for these houses. We can fill 0 for the area and None for the type.
- MSZoning (The general zoning classification) : 'RL' is by far the most common value. So we can fill in missing values with 'RL'.
- Utilities : For this categorical feature all records are "AllPub", except for one "NoSeWa" and 2 NA . Since the house with 'NoSewa' is in the training set, this feature won't help in predictive modelling. We can then safely remove it.
- Functional : data description says NA means typical.
- Electrical : It has one NA value. Since this feature has mostly 'SBrkr', we can set that for the missing value.
- KitchenQual: Only one NA value, and same as Electrical, we set 'TA' (which is the most frequent) for the missing value in KitchenQual.
- Exterior1st and Exterior2nd : Both Exterior 1 & 2 have only one missing value. We will just substitute in the most common string
- SaleType : Fill in again with most frequent which is "WD"
- MSSubClass : Na most likely means No building class. We can replace missing values with None

```
all_data["PoolQC"] = all_data["PoolQC"].fillna("None")
all_data["MiscFeature"] = all_data["MiscFeature"].fillna("None")
all_data["Alley"] = all_data["Alley"].fillna("None")
all_data["Fence"] = all_data["Fence"].fillna("None")
all_data["FireplaceQu"] = all_data["FireplaceQu"].fillna("None")
all_data["LotFrontage"] = all_data.groupby("Neighborhood")["LotFrontage"].transform(lambda x:
x.fillna(x.median()))
for col in ('GarageType', 'GarageFinish', 'GarageQual', 'GarageCond'):
    all_data[col] = all_data[col].fillna('None')
for col in ('GarageYrBlt', 'GarageArea', 'GarageCars'):
    all_data[col] = all_data[col].fillna(0)
for col in ('BsmtFinSF1', 'BsmtFinSF2', 'BsmtUnfSF','TotalBsmtSF', 'BsmtFullBath',
'BsmtHalfBath'):
    all_data[col] = all_data[col].fillna(0)
for col in ('BsmtQual', 'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinType2'):
    all_data[col] = all_data[col].fillna('None')
all_data["MasVnrType"] = all_data["MasVnrType"].fillna("None")
all_data["MasVnrArea"] = all_data["MasVnrArea"].fillna(0)
all_data['MSZoning'] = all_data['MSZoning'].fillna(all_data['MSZoning'].mode()[0])
all_data = all_data.drop(['Utilities'], axis=1)
all_data["Functional"] = all_data["Functional"].fillna("Typ")
all_data['Electrical'] = all_data['Electrical'].fillna(all_data['Electrical'].mode()[0])
```

```
all_data['KitchenQual'] = all_data['KitchenQual'].fillna(all_data['KitchenQual'].mode()[0])
all_data['Exterior1st'] = all_data['Exterior1st'].fillna(all_data['Exterior1st'].mode()[0])
all_data['Exterior2nd'] = all_data['Exterior2nd'].fillna(all_data['Exterior2nd'].mode()[0])
all_data['SaleType'] = all_data['SaleType'].fillna(all_data['SaleType'].mode()[0])
all_data['MSSubClass'] = all_data['MSSubClass'].fillna("None")

# Check if there are any missing values left
all_data_na = (all_data.isnull().sum() / len(all_data)) * 100
all_data_na = all_data_na.drop(all_data_na[all_data_na ==
0].index).sort_values(ascending=False)
missing_data = pd.DataFrame({'Missing Ratio' :all_data_na})
missing_data.head()

Missing  Ratio
[None]   [None]
```

## Part 6 - Feature Transformation/Engineering

Let's take a look at some features that may be misinterpreted to represent something it's not.

- MSSubClass: Identifies the type of dwelling involved in the sale.

- 20 1-STORY 1946 & NEWER ALL STYLES
- 30 1-STORY 1945 & OLDER
- 40 1-STORY W/FINISHED ATTIC ALL AGES
- 45 1-1/2 STORY - UNFINISHED ALL AGES
- 50 1-1/2 STORY FINISHED ALL AGES
- 60 2-STORY 1946 & NEWER
- 70 2-STORY 1945 & OLDER
- 75 2-1/2 STORY ALL AGES
- 80 SPLIT OR MULTI-LEVEL
- 85 SPLIT FOYER
- 90 DUPLEX - ALL STYLES AND AGES
- 120 1-STORY PUD (Planned Unit Development) - 1946 & NEWER
- 150 1-1/2 STORY PUD - ALL AGES
- 160 2-STORY PUD - 1946 & NEWER
- 180 PUD - MULTILEVEL - INCL SPLIT LEV/FOYER
- 190 2 FAMILY CONVERSION - ALL STYLES AND AGES

```
all_data['MSSubClass'].describe()

count    2907.000000
mean       57.094943
std        42.510238
min        20.000000
25%        20.000000
```

```
50%         50.000000
75%         70.000000
max        190.000000
Name: MSSubClass, dtype: float64
```

So, the average is a 57 type. What does that mean? Is a 90 type 3 times better than a 30 type? This feature was interpreted as numerical when it is actually categorical. The types listed here are codes, not values. Thus, we need to feature transformation with this and many other features.

```
#MSSubClass =The building class

all_data['MSSubClass'] = all_data['MSSubClass'].apply(str)

#Changing OverallCond into a categorical variable

all_data['OverallCond'] = all_data['OverallCond'].astype(str)

#Year and month sold are transformed into categorical features.

all_data['YrSold'] = all_data['YrSold'].astype(str)

all_data['MoSold'] = all_data['MoSold'].astype(str)
```

In our previous example, we could tell that our categories don't follow a particular order. What about categories that do? Let's take a look at "Kitchen Quality".

```
all_data['KitchenQual'].unique()

array(['Gd', 'TA', 'Ex', 'Fa'], dtype=object)
```

Here, data_description.txt comes to the rescue again!

Kitchen Quality:

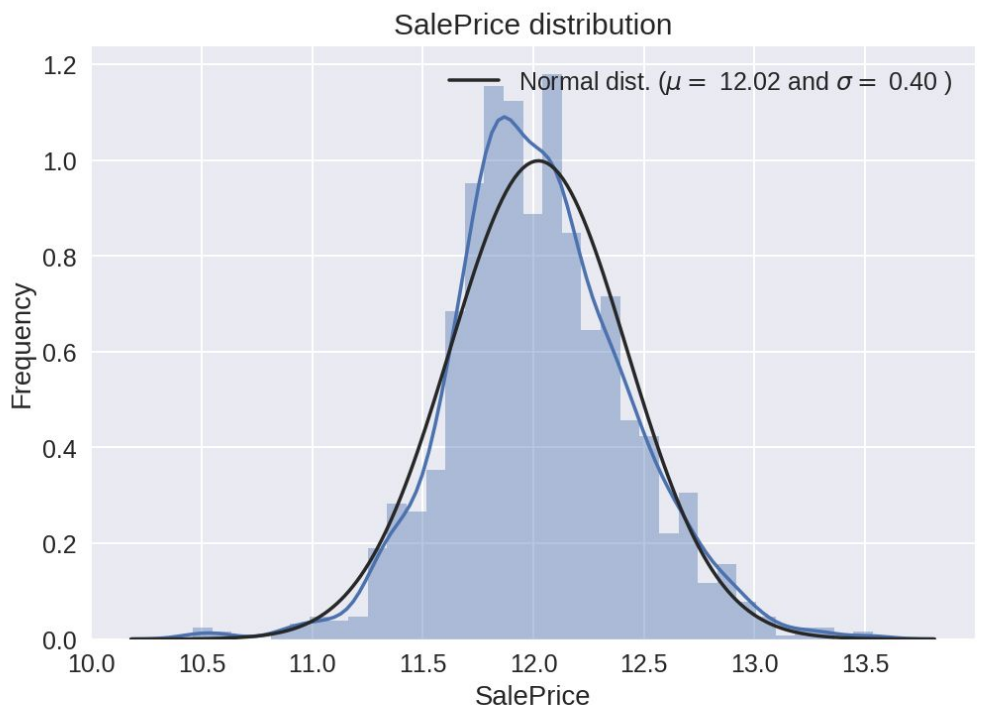
- Ex: Excellent
- Gd: Good
- TA: Typical/Average
- Fa: Fair
- Po: Poor

Is a score of “Gd” better than “TA” but worse than “Ex”? I think so, let’s encode these labels to give meaning to their specific orders.

```
from sklearn.preprocessing import LabelEncoder
cols = ('FireplaceQu', 'BsmtQual', 'BsmtCond', 'GarageQual', 'GarageCond',
        'ExterQual', 'ExterCond','HeatingQC', 'PoolQC', 'KitchenQual', 'BsmtFinType1',
        'BsmtFinType2', 'Functional', 'Fence', 'BsmtExposure', 'GarageFinish', 'LandSlope',
        'LotShape', 'PavedDrive', 'Street', 'Alley', 'CentralAir', 'MSSubClass',
'OverallCond',
        'YrSold', 'MoSold')

# Process columns and apply LabelEncoder to categorical features
for c in cols:
    lbl = LabelEncoder()
    lbl.fit(list(all_data[c].values))
    all_data[c] = lbl.transform(list(all_data[c].values))
```

Now we finally fix our skewed features.

```
# We use the numpy fuction log1p which  applies log(1+x) to all elements of the column
train["SalePrice"] = np.log1p(train["SalePrice"])

#Check the new distribution
sns.distplot(train['SalePrice'] , fit=norm);

# Get the fitted parameters used by the function
(mu, sigma) = norm.fit(train['SalePrice'])
print( '\n mu = {:.2f} and sigma = {:.2f}\n'.format(mu, sigma))
plt.legend(['Normal dist. ($\mu=$ {:.2f} and $\sigma=$ {:.2f} )'.format(mu, sigma)],
            loc='best')
plt.ylabel('Frequency')
plt.title('SalePrice distribution')

fig = plt.figure()
res = stats.probplot(train['SalePrice'], plot=plt)
plt.show()

y_train = train.SalePrice.values

print("Skewness: %f" % train['SalePrice'].skew())
print("Kurtosis: %f" % train['SalePrice'].kurt())
```



# Part 7 - Modeling and Predictions

## 7.1 Cross-validation with k-folds

First we do cross-validation with k-folds to get the best split in test/training data.

```
# Cross-validation with k-folds
n_folds = 5

def rmsle_cv(model):
    kf = KFold(n_folds, shuffle=True, random_state=42).get_n_splits(train.values)
    rmse= np.sqrt(-cross_val_score(model, train.values, y_train,
scoring="neg_mean_squared_error", cv = kf))
    return(rmse)
```

## 7.2 Modeling

For our models, we are going to use lasso, elastic net, kernel ridge, gradient boosting, XGBoost, and LightGBM regression.

```
lasso = make_pipeline(RobustScaler(), Lasso(alpha =0.0005, random_state=1))
ENet = make_pipeline(RobustScaler(), ElasticNet(alpha=0.0005, l1_ratio=.9, random_state=3))
KRR = KernelRidge(alpha=0.6, kernel='polynomial', degree=2, coef0=2.5)
GBoost = GradientBoostingRegressor(n_estimators=3000, learning_rate=0.05,
                                   max_depth=4, max_features='sqrt',
                                   min_samples_leaf=15, min_samples_split=10,
                                   loss='huber', random_state =5)
model_xgb = xgb.XGBRegressor(colsample_bytree=0.2, gamma=0.0,
                             learning_rate=0.05, max_depth=6,
                             min_child_weight=1.5, n_estimators=7200,
                             reg_alpha=0.9, reg_lambda=0.6,
                             subsample=0.2,seed=42, silent=1,
                             random_state =7)
model_lgb = lgb.LGBMRegressor(objective='regression',num_leaves=5,
                              learning_rate=0.05, n_estimators=720,
                              max_bin = 55, bagging_fraction = 0.8,
                              bagging_freq = 5, feature_fraction = 0.2319,
                              feature_fraction_seed=9, bagging_seed=9,
                              min_data_in_leaf =6, min_sum_hessian_in_leaf = 11)
```

Checking performance of base models by evaluating the cross-validation RMSLE error.

```
score = rmsle_cv(lasso)
print("\nLasso score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))
score = rmsle_cv(ENet)
print("ElasticNet score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))
score = rmsle_cv(KRR)
print("Kernel Ridge score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))
score = rmsle_cv(GBoost)
print("Gradient Boosting score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))
score = rmsle_cv(model_xgb)
print("Xgboost score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))
score = rmsle_cv(model_lgb)
print("LGBM score: {:.4f} ({:.4f})\n" .format(score.mean(), score.std()))


Lasso score: 0.1111 (0.0071)

ElasticNet score: 0.1111 (0.0072)

Kernel Ridge score: 0.1148 (0.0075)

Gradient Boosting score: 0.1177 (0.0079)

Xgboost score: 0.1177 (0.0048)

LGBM score: 0.1159 (0.0059)
```

Here, we stack the models to average their scores.

```
class AveragingModels(BaseEstimator, RegressorMixin, TransformerMixin):
    def __init__(self, models):
        self.models = models
    # we define clones of the original models to fit the data in
    def fit(self, X, y):
        self.models_ = [clone(x) for x in self.models]

        # Train cloned base models
        for model in self.models_:
            model.fit(X, y)

        return self

    #Now we do the predictions for cloned models and average them
    def predict(self, X):
        predictions = np.column_stack([
            model.predict(X) for model in self.models_
        ])
        return np.mean(predictions, axis=1)
```

Now we get the root mean squared log errors for Stacked Models, XGBoost, and LightGBM

```
stacked_averaged_models.fit(train.values, y_train)
stacked_train_pred = stacked_averaged_models.predict(train.values)
stacked_pred = np.expm1(stacked_averaged_models.predict(test.values))
print(rmsle(y_train, stacked_train_pred))

0.0818432096134

model_xgb.fit(train, y_train)
xgb_train_pred = model_xgb.predict(train)
xgb_pred = np.expm1(model_xgb.predict(test))
print(rmsle(y_train, xgb_train_pred))

0.0416468753988

model_lgb.fit(train, y_train)
lgb_train_pred = model_lgb.predict(train)
lgb_pred = np.expm1(model_lgb.predict(test.values))
print(rmsle(y_train, lgb_train_pred))

0.0719898901013
```

Finally, we do ensemble prediction for our submission.

```
''''RMSE on the entire Train data when averaging'''

print('RMSLE score on train data:')
print(rmsle(y_train,stacked_train_pred*Stacked +
               xgb_train_pred*XGBoost + lgb_train_pred*LGBM))

RMSLE score on train data:
```

```
0.0626295977484
```

```
ensemble = stacked_pred*Stacked + xgb_pred*XGBoost + lgb_pred*LGBM
```

Our submission in a csv file.

```
sub = pd.DataFrame()
sub['Id'] = test_ID
sub['SalePrice'] = ensemble
sub.to_csv('submission.csv',index=False)
```

# Conclusions and Recommendations for House Prices

With this study, there are some conclusions that could help out house buyers and house sellers.

## House Buyers

The ultimate factor in house prices is overall quality and square footage. Since it is a bit more difficult to control square footage, house buyers should compensate on overall quality if they want the best value out of their purchase. Home improvements could always be done outside of the house purchase while adjusting square footage is more difficult.

## House Sellers

Knowing that overall quality and square footage are the best feature for higher house prices, house sellers should find the maximum square footage house then improve its quality in order to flip house effectively.

## Discussion

Although this study does an amazing performance in predicting house prices, it does not do much for interested clients. House buyers/sellers are more likely to have this knowledge before reading this study. However, with the caveats of the recommendations, we can improve this study even further by adding additional goals.

## Improvements to Study

A major downfall to this study is that it is focused in one place, Iowa. Something simple we could do is make it cover the entire nation. That way, we can analyze different prices per location. We could also record prices per season each year to have data on price fluctuations. That way, we could do a time-series analysis to predict when is the best time to buy a house. Finally, we could move on to making a recurrent neural network due to the increased amount of data available and also the fact that we have time-series data to work with.

This challenge is available on Kaggle as the “Zillow’s Home Value Prediction (Zestimate)”. The grand prize includes $1,000,000 for the best submission.