# Housing Prices Machine Learning Project

Predicting house prices is crucial for everyone involved in the real estate industry, from property owners and investors to buyers and sellers. In this project I am given housing data describing many aspects of residential homes in Ames, Iowa. This data includes 79 explanatory variables to help characterize houses including the size the house, year the house was built, the number of bedrooms, the number of bathrooms and the number of kitchens just to name a few. For this project I will be utilizing these 79 explanatory variables and the power of machine learning, **Random Forest Regression** and **XGBoost** algorithms, to predict what the final price should be for a set of given homes.

## Loading The Data

In this project, I am given 2 datasets with housing information. The dataset **train.csv** comprises of data on 1460 houses and includes the target variable *SalePrice* indicating the property's sale price in dollars. The dataset **test.csv** on the otherhand contains data on a seperate 1459 houses, but does not contain a variable that indicates the sales price of the house. I will analyze and use Machine Learning techniques on the **train.csv** data in order to create a model. This model will then be used to predict the house prices of those in the **test.csv** dataset. This project is a "competition project" on the Kaggle site, so I will not know what the real housing prices are for those in the **test.csv** dataset. Instead I will submit my guess of the sales price for those 1459 houses using my model and will receive a score based on the accuracy.

I will begin by importing packages that I will use throughout this project. I will also import the datasets from the Kaggle site.

In [1]:
```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder
from sklearn.ensemble import RandomForestRegressor
from xgboost import XGBRegressor
from sklearn.metrics import mean_absolute_error


import os
if not os.path.exists("../input/train.csv"):
    os.symlink("../input/home-data-for-ml-course/train.csv", "../input/train.csv")
    os.symlink("../input/home-data-for-ml-course/test.csv", "../input/test.csv")
from learntools.core import binder
binder.bind(globals())
```

In [2]:
```
train_data = pd.read_csv('../input/train.csv', index_col='Id')
train_data.head()
```

Out[2]:

| Id | MSSubClass | MSZoning | LotFrontage | LotArea | Street | Alley | LotShape | LandContour | Utilities | Lc |
|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 60 | RL | 65.0 | 8450 | Pave | NaN | Reg | Lvl | AllPub | |
| **2** | 20 | RL | 80.0 | 9600 | Pave | NaN | Reg | Lvl | AllPub | |
| **3** | 60 | RL | 68.0 | 11250 | Pave | NaN | IR1 | Lvl | AllPub | |
| **4** | 70 | RL | 60.0 | 9550 | Pave | NaN | IR1 | Lvl | AllPub | |
| **5** | 60 | RL | 84.0 | 14260 | Pave | NaN | IR1 | Lvl | AllPub | |

5 rows × 80 columns

In [3]:
```
test_data = pd.read_csv('../input/test.csv', index_col='Id')
test_data.head()
```

Out[3]:

| Id | MSSubClass | MSZoning | LotFrontage | LotArea | Street | Alley | LotShape | LandContour | Utilities |
|---|---|---|---|---|---|---|---|---|---|
| **1461** | 20 | RH | 80.0 | 11622 | Pave | NaN | Reg | Lvl | AllPub |
| **1462** | 20 | RL | 81.0 | 14267 | Pave | NaN | IR1 | Lvl | AllPub |
| **1463** | 60 | RL | 74.0 | 13830 | Pave | NaN | IR1 | Lvl | AllPub |
| **1464** | 60 | RL | 78.0 | 9978 | Pave | NaN | IR1 | Lvl | AllPub |
| **1465** | 120 | RL | 43.0 | 5005 | Pave | NaN | IR1 | HLS | AllPub |

5 rows × 79 columns

In [4]:
```
test_data.columns
```

Out[4]:
```
Index(['MSSubClass', 'MSZoning', 'LotFrontage', 'LotArea', 'Street', 'Alley',
       'LotShape', 'LandContour', 'Utilities', 'LotConfig', 'LandSlope',
       'Neighborhood', 'Condition1', 'Condition2', 'BldgType', 'HouseStyle',
       'OverallQual', 'OverallCond', 'YearBuilt', 'YearRemodAdd', 'RoofStyle',
       'RoofMatl', 'Exterior1st', 'Exterior2nd', 'MasVnrType', 'MasVnrArea',
       'ExterQual', 'ExterCond', 'Foundation', 'BsmtQual', 'BsmtCond',
       'BsmtExposure', 'BsmtFinType1', 'BsmtFinSF1', 'BsmtFinType2',
       'BsmtFinSF2', 'BsmtUnfSF', 'TotalBsmtSF', 'Heating', 'HeatingQC',
       'CentralAir', 'Electrical', '1stFlrSF', '2ndFlrSF', 'LowQualFinSF',
       'GrLivArea', 'BsmtFullBath', 'BsmtHalfBath', 'FullBath', 'HalfBath',
       'BedroomAbvGr', 'KitchenAbvGr', 'KitchenQual', 'TotRmsAbvGrd',
       'Functional', 'Fireplaces', 'FireplaceQu', 'GarageType', 'GarageYrBlt',
       'GarageFinish', 'GarageCars', 'GarageArea', 'GarageQual', 'GarageCond',
       'PavedDrive', 'WoodDeckSF', 'OpenPorchSF', 'EnclosedPorch', '3SsnPorch',
       'ScreenPorch', 'PoolArea', 'PoolQC', 'Fence', 'MiscFeature', 'MiscVal',
       'MoSold', 'YrSold', 'SaleType', 'SaleCondition'],
      dtype='object')
```

Here I can see that both datasets contain the 79 explanitory variables such as, *MSSubClass*, *MSZoning*, *LotFrontage*, and *LotArea* just to name the first few. The **train_data** also contains the

*SalePrice* variable, which will obviously be used as the target variable.

I will now take a closer look at the variables using the descibe method.

In [5]:
```
train_data.describe()
```

Out[5]:

| | MSSubClass | LotFrontage | LotArea | OverallQual | OverallCond | YearBuilt | YearRemodAd |
|---|---|---|---|---|---|---|---|
| **count** | 1460.000000 | 1201.000000 | 1460.000000 | 1460.000000 | 1460.000000 | 1460.000000 | 1460.0000 |
| **mean** | 56.897260 | 70.049958 | 10516.828082 | 6.099315 | 5.575342 | 1971.267808 | 1984.8657 |
| **std** | 42.300571 | 24.284752 | 9981.264932 | 1.382997 | 1.112799 | 30.202904 | 20.6454 |
| **min** | 20.000000 | 21.000000 | 1300.000000 | 1.000000 | 1.000000 | 1872.000000 | 1950.0000 |
| **25%** | 20.000000 | 59.000000 | 7553.500000 | 5.000000 | 5.000000 | 1954.000000 | 1967.0000 |
| **50%** | 50.000000 | 69.000000 | 9478.500000 | 6.000000 | 5.000000 | 1973.000000 | 1994.0000 |
| **75%** | 70.000000 | 80.000000 | 11601.500000 | 7.000000 | 6.000000 | 2000.000000 | 2004.0000 |
| **max** | 190.000000 | 313.000000 | 215245.000000 | 10.000000 | 9.000000 | 2010.000000 | 2010.0000 |

8 rows × 37 columns

In [6]:
```
test_data.describe()
```

Out[6]:

| | MSSubClass | LotFrontage | LotArea | OverallQual | OverallCond | YearBuilt | YearRemodAd |
|---|---|---|---|---|---|---|---|
| **count** | 1459.000000 | 1232.000000 | 1459.000000 | 1459.000000 | 1459.000000 | 1459.000000 | 1459.00000 |
| **mean** | 57.378341 | 68.580357 | 9819.161069 | 6.078821 | 5.553804 | 1971.357779 | 1983.66278 |
| **std** | 42.746880 | 22.376841 | 4955.517327 | 1.436812 | 1.113740 | 30.390071 | 21.13046 |
| **min** | 20.000000 | 21.000000 | 1470.000000 | 1.000000 | 1.000000 | 1879.000000 | 1950.00000 |
| **25%** | 20.000000 | 58.000000 | 7391.000000 | 5.000000 | 5.000000 | 1953.000000 | 1963.00000 |
| **50%** | 50.000000 | 67.000000 | 9399.000000 | 6.000000 | 5.000000 | 1973.000000 | 1992.00000 |
| **75%** | 70.000000 | 80.000000 | 11517.500000 | 7.000000 | 6.000000 | 2001.000000 | 2004.00000 |
| **max** | 190.000000 | 200.000000 | 56600.000000 | 10.000000 | 9.000000 | 2010.000000 | 2010.00000 |

8 rows × 36 columns

There are a couple of things that stand out:

- The count for a few variables is different from the number of houses in the datasets. This indicates that some houses are missing values in their data.
- There are only 36 explanatory variables appearing in the result of the describe method meaning there are 43 variables that are categorical.

- In general all the variables tend to have similar numbers across both datasets hopefully indicating that the data was properly/randomly split when separating the **train** and **test** datasets.

# Split the Data

Next I will split the train_data into 4 groups using the train_test_split method. (Using 80% train, 20% Validation)

- X_train: 80% of the data with all variables except the target variable. Will be used to build the model(s).
- X_valid: 20% of the data with all variables except the target variable. Will be used to test the accuracy of model(s), and check results of adjustments against the model.
- y_train: The same 80% of the data as the X_train set, but only comprised of the target *SalePrice* variable.
- y_test: The same 20% of the data as the X_test set, but only comprised of the target *SalePrice* variable.

I need to split the data since like I mentioned before I will not know the *SalePrice* variable of the test_data. Therefore, I cannot use that data to assess the accuracy of my models and determine what proper adjustments to make against it.

In [7]:
```
# Create a copy of the train_data that I will make adjustments against.
X = train_data

# Separate the target variable from the explanatory variables.
X.dropna(axis=0, subset=['SalePrice'], inplace=True)
y = X.SalePrice
X.drop(['SalePrice'], axis=1, inplace=True)

# Split the taining data set and validation set from training data
X_train, X_valid, y_train, y_valid = train_test_split(X, y, train_size=0.8, test_size=
                                                      random_state=0)
```

# Assess and Clean Variables in the Data

## Missing Data

Next I will handle the missing data points. There are a couple of ways deal with missing values in data. In this case I will be using simple imputation to fill in the missing values. For this project I will be calculating the missing values using the "constant" strategy of the SimpleImputer method.

In [8]:
```
# Select numerical columns to impute against
numerical_cols = [cname for cname in X_train.columns if
                  X_train[cname].dtype in ['int64', 'float64']]
```

```
# Preprocessing for numerical data
numerical_transformer = SimpleImputer(strategy = "constant")
```

### Categorical Data

As mentioned earlier, 43 of the explanatory variables are categorical. There are also a few approaches to handle categorical variables. In this project, I will be using One-Hot Encoding which creates new columns in the data indicating the presence (or absence) of each possible value of the categorical data. Since this typically does not perform well with categorical variables with a large number of differing values, I will find the cardinality of the categorical variables, and then use One-Hot Encoding on those with less than 10 cardinality, and drop the variables with 10 or more cardinality from the data.

In [9]:
```
# "Cardinality" means the number of unique values in a column
# Select categorical columns with relatively low cardinality.
categorical_cols = [cname for cname in X_train.columns if
                    X_train[cname].nunique() < 10 and
                    X_train[cname].dtype == "object"]

# Using Pipeline to help preprocess the categorical data both Imputing and One-Hot Enc
categorical_transformer = Pipeline(steps=[
    ("imputer",SimpleImputer(strategy = "constant")),
    ("one hot",OneHotEncoder(handle_unknown = "ignore"))])
```

# Selecting Variables For Model

Now based on how I approached the data in the previous section, I will select which columns to use for the model to predict. I will be keeping only the numerical and low cardinality categorical variables for each of the 3 X datasets. (**X_train**, **X_valid**, **X_test**)

In [10]:
```
# Keep selected columns only
my_cols = categorical_cols + numerical_cols

X_train_mycol = X_train[my_cols].copy()
X_valid_mycol = X_valid[my_cols].copy()
X_test_mycol = test_data[my_cols].copy()
```

I will run the head method and columns method on the valid dataset to show the number and names of the columns I am keeping.

In [11]:
```
X_valid_mycol.head()
```

Out[11]:

| Id | MSZoning | Street | Alley | LotShape | LandContour | Utilities | LotConfig | LandSlope | Condition1 | C |
|---|---|---|---|---|---|---|---|---|---|---|
| **530** | RL | Pave | NaN | IR1 | Lvl | AllPub | CulDSac | Gtl | Norm | |
| **492** | RL | Pave | NaN | Reg | Lvl | AllPub | Inside | Gtl | Artery | |
| **460** | RL | Pave | NaN | IR1 | Bnk | AllPub | Corner | Gtl | Norm | |
| **280** | RL | Pave | NaN | Reg | Lvl | AllPub | Inside | Gtl | Norm | |
| **656** | RM | Pave | NaN | Reg | Lvl | AllPub | Inside | Gtl | Norm | |

5 rows × 76 columns

In [12]:
```
X_valid_mycol.columns
```

Out[12]:
```
Index(['MSZoning', 'Street', 'Alley', 'LotShape', 'LandContour', 'Utilities',
       'LotConfig', 'LandSlope', 'Condition1', 'Condition2', 'BldgType',
       'HouseStyle', 'RoofStyle', 'RoofMatl', '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', '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'],
      dtype='object')
```

In [13]:
```
# Bundle preprocessing for numerical and categorical data
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, numerical_cols),
        ('cat', categorical_transformer, categorical_cols)
    ])
```

# Creating The Models

For this project I will be using both Random Forest Regression and XGBoost Regression (gradient boosting) to create models.

The Random Forest Regression works by constructing a collection of decision trees. Data is inputted into each decision tree in the forest, and each tree independently makes a prediction. The final prediction is then determined by taking the average of all the predictions from the decision trees.

Gradient boosting works by going through cycles to iteratively add models into an ensemble. It begins by initializing the ensemble with a single model, whose predictions can be pretty naive. It then iteratively trains new models to correct the errors made by previous models in the ensemble. The errors from the previous models are used as targets for the next model to improve upon. This process continues iteratively, with each new model trying to minimize the residual errors of the ensemble.

I will create 5 models for using each technique with differing parameters.

In [14]:
```
# Define the Random Forsest models for different n_estimators.
model_RF1 = RandomForestRegressor(n_estimators = 100, random_state = 0)
model_RF2 = RandomForestRegressor(n_estimators = 300, random_state = 0)
model_RF3 = RandomForestRegressor(n_estimators = 500, random_state = 0)
model_RF4 = RandomForestRegressor(n_estimators = 700,  random_state = 0)
model_RF5 = RandomForestRegressor(n_estimators = 900, random_state = 0)

# Define the XGBoost models for different n_estimators.
model_XGB1 = XGBRegressor(n_estimators = 100, learning_rate = 0.05,
                                                    random_state = 0)
model_XGB2 = XGBRegressor(n_estimators = 400, learning_rate = 0.05,
                                                      random_state = 0)
model_XGB3 = XGBRegressor(n_estimators = 700, learning_rate = 0.05,
                                                      random_state = 0)
model_XGB4 = XGBRegressor(n_estimators = 1000, learning_rate = 0.05,
                                                       random_state = 0)
model_XGB5 = XGBRegressor(n_estimators = 1300, learning_rate = 0.05,
                                                       random_state = 0)
```

# Model Testing

Now I will create a score_model function that uses mean absolute error between the predictions based on the X-valid data (calculated using the model trained on the X_train_mycol and y_train data) and the y_valid data. The model that gives the lowest mean absolute error should be considered the best model.

In [15]:
```
def score_model(model, X_t=X_train_mycol, X_v=X_valid_mycol,
                                            y_t=y_train, y_v=y_valid):
    my_pipeline = Pipeline(steps=[('preprocessor', preprocessor),
                                  ('model', model)
                                 ])
    my_pipeline.fit(X_t, y_t)
    preds = my_pipeline.predict(X_v)
    return mean_absolute_error(y_v, preds)

#create set of all the combined models and then iterate
#through them calculating their mean absolute error.
models = [model_RF1, model_RF2, model_RF3, model_RF4, model_RF5,
              model_XGB1, model_XGB2, model_XGB3, model_XGB4, model_XGB5]
for i in range(0, len(models)):
    mae = score_model(models[i])
    print("Model " + str(i+1) + " MAE: " + str(mae))
```

```
Model 1 MAE: 17621.3197260274
Model 2 MAE: 17305.304303652967
Model 3 MAE: 17287.301842465753
Model 4 MAE: 17214.610132093934
Model 5 MAE: 17232.302907153728
Model 6 MAE: 17511.70408818493
Model 7 MAE: 17219.78846050942
Model 8 MAE: 17210.830037992295
Model 9 MAE: 17207.32646618151
Model 10 MAE: 17208.56204516267
```

It appears **model_XGB4** should be considered the best and final model as it is about tied for having the lowest mean absolute error with a more complex models having a larger error, most likely due to overfitting.

## Creating and Fitting the Final Model

I will now create the final model, once again calculate its mean absolute value under its new final model name, and then use the **X_test** data to predict the Sale Price of all houses in the **X_test** dataset. An **output** dataset consisting of the House Ids and predicted SalePrice values will be created and submitted to the Kaggle compition.

In [16]:
```
#Creating the Final Model.
model_final = XGBRegressor(n_estimators = 1000, learning_rate = 0.05,
                                                    random_state = 0)
```

In [17]:
```
# Bundle preprocessing and modeling code in a pipeline
my_pipeline = Pipeline(steps=[('preprocessor', preprocessor),
                              ('model', model_final)])

# Preprocessing of training data, fit model
my_pipeline.fit(X_train_mycol, y_train)

# Preprocessing of validation data, get predictions
preds = my_pipeline.predict(X_valid)

# Evaluate the model
score = mean_absolute_error(y_valid, preds)
print('MAE:', score)
```

```
MAE: 17207.32646618151
```

In [18]:
```
# Preprocessing of test data, fit model using the X_test_mycol dataset
preds_test = my_pipeline.predict(X_test_mycol)
```

In [19]:
```
#Create the output dataset and take a quick look at it.
output = pd.DataFrame({'Id': test_data.index,
                       'SalePrice': preds_test})
```

In [20]:
```
output.to_csv('submission.csv', index=False)
print("Submission was successfully saved!")
```

```
Submission was successfully saved!
```

# Result

After submitting the output, the result is that my submission received a final score of 14893. In this case the lower the score the more accurate the model, as I believe the score indicates the average error in the predicted house price compared to the real sale price from the houses in the **test** dataset.