Narges Babaahmadi 610398102

Dataset

- **This dataset contains house sale prices**
- **The dataset has 6 columns(features)**
- **It include 1460 rows ### Feature Columns**
- **id: Unique ID for each home sold**
- **MSSubClass: The building class**
- **MSZoning: The general zoning classification**
- **LotArea: Lot size in square feet**
- **SaleCondition: Condition of sale**
- **Sale Price: Dependent Variable**

imports

In [1]:

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import seaborn as sns
from sklearn import metrics
from sklearn.tree import DecisionTreeClassifier,  DecisionTreeRegressor
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LinearRegression
from sklearn import metrics
from scipy import stats
from sklearn.preprocessing import OneHotEncoder, LabelEncoder, PolynomialFeatures, MinMaxScaler
from sklearn import preprocessing
from tensorflow.keras import Sequential
from keras.layers import Dense
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV, cross_val_score
from sklearn.ensemble import RandomForestRegressor
from sklearn.datasets import make_regression
from sklearn.metrics import mean_absolute_error,  mean_squared_error
```

now, I read our dataset using pandas packages. I start by acquiring the datasets into Pandas DataFrames.

In [2]:

```
train = pd.read_csv("train.csv")
train.head()
```

Out[2]:

	Id	MSSubClass	MSZoning	LotArea	SaleCondition	SalePrice
0	1	60	RL	8450	Normal	208500
1	2	20	RL	9600	Normal	181500
2	3	60	RL	11250	Normal	223500
3	4	70	RL	9550	Abnorml	140000
4	5	60	RL	14260	Normal	250000

in order to find the minimum, maximum, standard deviation, first quartile, second quartile, third quartile and mean, we run the code below:

In [3]:

```
train.describe().T
```

Out[3]:

	count	mean	std	min	25%	50%	75%	max
Id	1460.0	730.500000	421.610009	1.0	365.75	730.5	1095.25	1460.0
MSSubClass	1460.0	56.897260	42.300571	20.0	20.00	50.0	70.00	190.0
LotArea	1460.0	10516.828082	9981.264932	1300.0	7553.50	9478.5	11601.50	215245.0
SalePrice	1460.0	180921.195890	79442.502883	34900.0	129975.00	163000.0	214000.00	755000.0

here I find the categorical features(in order to encode them) and the features that have NULL values.

In [4]:

```
train.info()
train.isnull().sum()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1460 entries, 0 to 1459
Data columns (total 6 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   LotArea        1460 non-null   int64 
 4   SaleCondition  1460 non-null   object
```

```
 5   SalePrice      1460 non-null   int64
dtypes: int64(4), object(2)
memory usage: 68.6+ KB
```

Out[4]:

```
Id               0
MSSubClass       0
MSZoning         0
LotArea          0
SaleCondition    0
SalePrice        0
dtype: int64
```

so MSZoning and SaleCondition are categorical features. This problem is a Regression problem in which categorical data shall be converted in numeric format.

I can encode the categorical data with different methods of encoding. Here, i tried label encoding and one hot encoding.

- **label encoding: assigning(labling) each categorical value with a number.**
- **one hot endoing: i will explain this using an example. assume that we have a feature named 'color' and it has 3 categories. so we add 3 columns and eliminate the color column. for example if the color of the first row is red, the color of the second row is green and the color of the third row is blue, we have the following columns: [red, green, blue]=[[1,0,0],[0,1,0],[0,0,1]] one hot encoding was more accurate than label encodig here so i used one hot encoding.**

In [5]:

```
#i save the unchanged train, in case i need access to cateegorical data before encoding
train_unchanged = train
one_hot = pd.get_dummies(train['MSZoning'])
train = train.drop('MSZoning',axis = 1)
train = train.join(one_hot)
one_hot2 = pd.get_dummies(train['SaleCondition'])
train = train.drop('SaleCondition',axis = 1)
train = train.join(one_hot2)
train.head()
```

Out[5]:

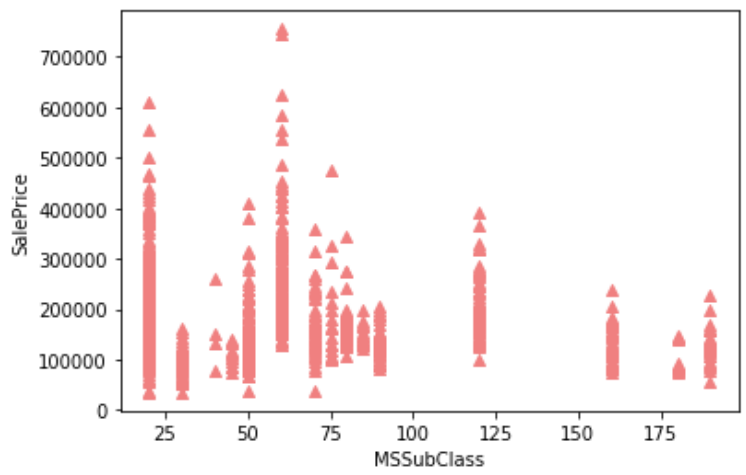
	Id	MSSubClass	LotArea	SalePrice	C (all)	FV	RH	RL	RM	Abnorml	AdjLand	Alloca	Family	Normal	Partial
0	1	60	8450	208500	0	0	0	1	0	0	0	0	0	1	0
1	2	20	9600	181500	0	0	0	1	0	0	0	0	0	1	0
2	3	60	11250	223500	0	0	0	1	0	0	0	0	0	1	0
3	4	70	9550	140000	0	0	0	1	0	1	0	0	0	0	0
4	5	60	14260	250000	0	0	0	1	0	0	0	0	0	1	0

EDA

Let's first plot a scatter plot visualizing MSSubClass and SalePrice:

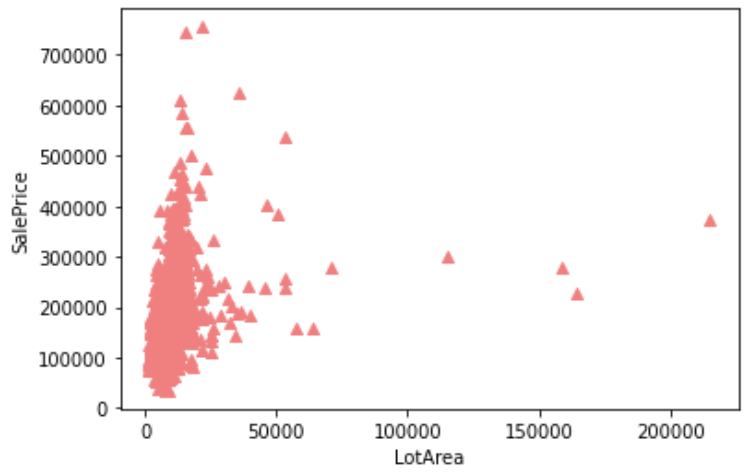
In [6]:

```
plt.scatter(train.MSSubClass, train.SalePrice, c = "lightcoral", marker ="^")
plt.xlabel("MSSubClass")
plt.ylabel("SalePrice")
plt.show()
```



In [7]:

```
plt.scatter(train.LotArea, train.SalePrice, c = "lightcoral", marker ="^")
plt.xlabel("LotArea")
plt.ylabel("SalePrice")
plt.show()
```



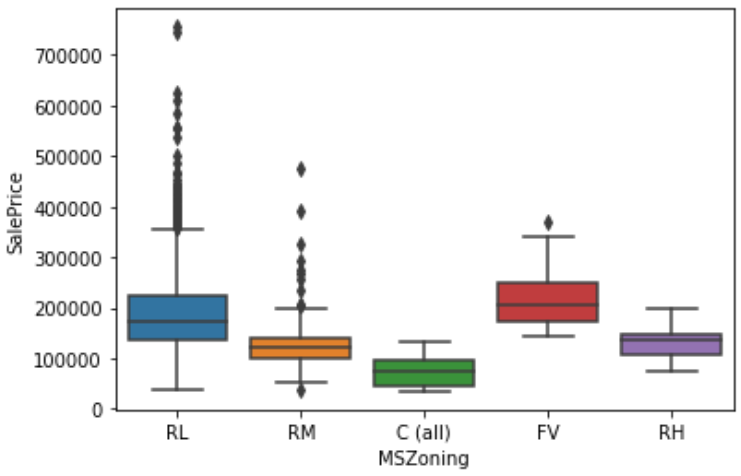
I will consider LotArea greater than 200000 as outliers

In [8]:

```
train = train[train.LotArea < 200000]
```

In [9]:

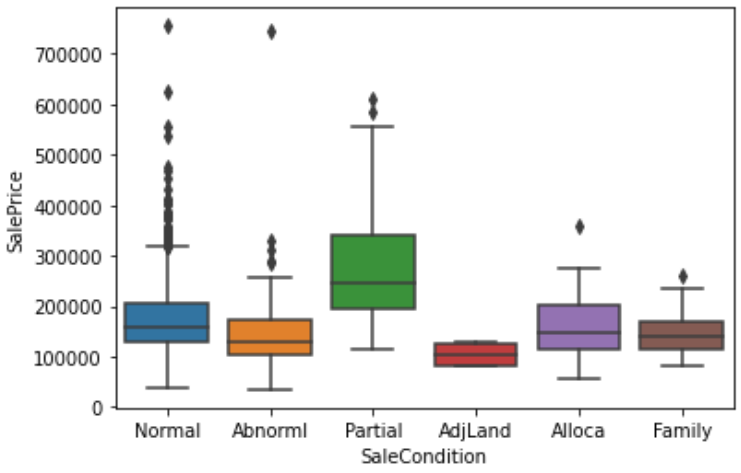
```
ax = sns.boxplot(x='MSZoning', y='SalePrice', data=train_unchanged)
```



From this boxplot, i get that houses the overal price of houses with MSZoning = FV is more expensive than others and houses wih MSZoning = C(all) are the least expensive ones.

In [10]:

```
ax = sns.boxplot(x='SaleCondition', y='SalePrice', data=train_unchanged)
```



From this plot, i get that people ,usually, pay more for the houses with SaleCondition = Partial.

I will consider SalePrice greater than 700000 as outliers.

In [11]:

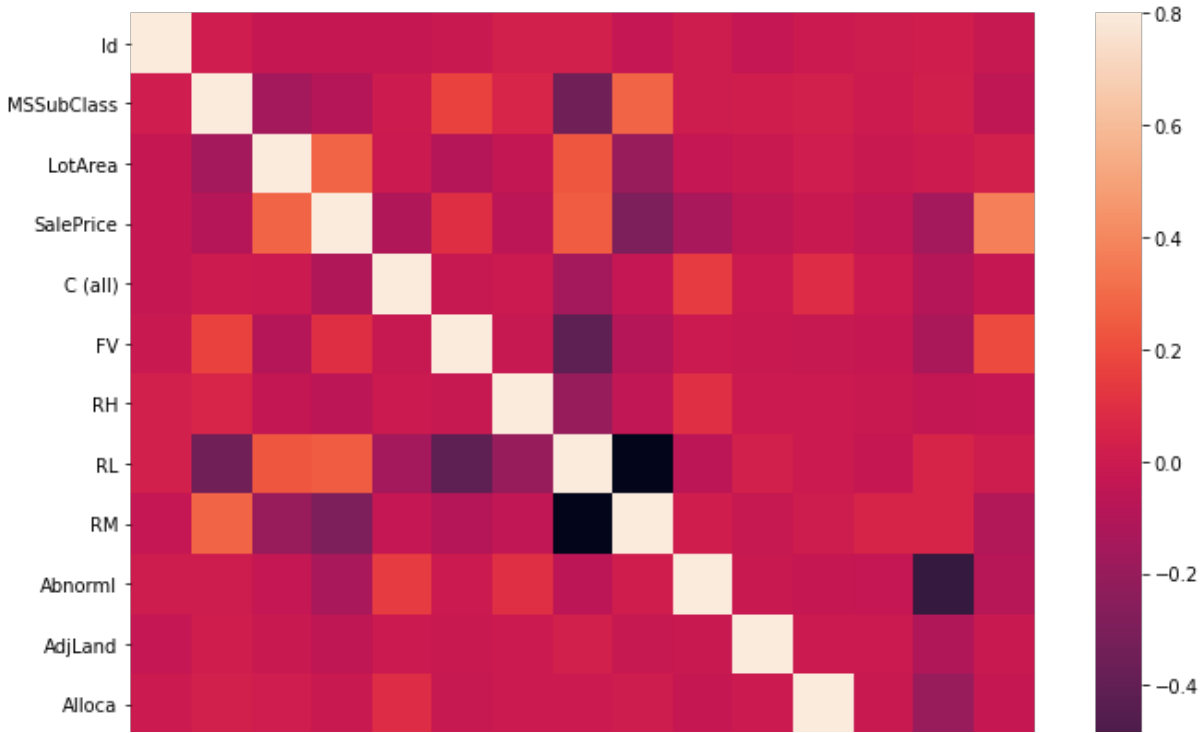
```
train = train[train.SalePrice < 700000]
```

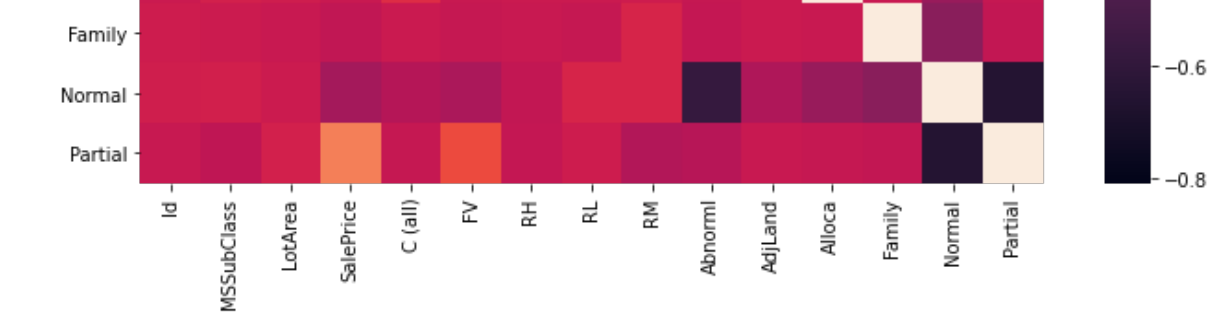
Assumtions based on data analysis:

- **Correlating: We want to know how well does each feature correlate with Price. We want to do this early in our project and match these quick correlations with modelled correlations later in the project.**
- **Completing: Since there are no missing values we do not need to complete any values. #### correlation matrix i use correlation matrix to examine the strength and direction of the linear relationship between two continuous variables. The correlation coefficient can range in value from −1 to +1. The larger the absolute value of the coefficient, the stronger the relationship between the variables. For correlation, an absolute value of 1 indicates a perfect linear relationship. A correlation close to 0 indicates no linear relationship between the variables. constructing pearson matrix:**

In [12]:

```
corrmat = train.corr()
f, ax = plt.subplots(figsize=(12, 9))
sns.heatmap(corrmat, vmax=.8, square=True);
```





In [13]:

```
cols = corrmat.nlargest(15,'SalePrice')['SalePrice'].index
most_corr = pd.DataFrame(cols)
most_corr.columns = ['Most Correlated Features']
most_corr
```

Out[13]:

	Most Correlated Features
0	SalePrice
1	Partial
2	LotArea
3	RL
4	FV
5	Alloca
6	Id
7	Family
8	AdjLand
9	RH
10	MSSubClass
11	C (all)
12	Abnorml
13	Normal
14	RM

Using the matrix and the table above, partial and RL and LotArea seems to be the most correlated features. But overall the correlations are ,mostly, close to zero, so that Linear regression wouldn't be a good model.

Modeling

Now it's time to create our independent and dependent matrix of feature. we can normalize our dataset using the scikit-learn object MinMaxScaler. MinMaxScaler normalizes the value as follows:

- **y = (x – min) / (max – min)**

In [14]:

```
scaler = MinMaxScaler()
X_train = train.drop('SalePrice', 1)
Y_train = train['SalePrice']
X_train = scaler.fit_transform(X_train)
```

Now we should read test dataset

In [15]:

```
test = pd.read_csv("test1.csv")
```

after reading the dataset, we encode the categorical data using one hot encoding method.

In [16]:

```
one_hot = pd.get_dummies(test['MSZoning'])
test = test.drop('MSZoning',axis = 1)
test = test.join(one_hot)
one_hot2 = pd.get_dummies(test['SaleCondition'])
test = test.drop('SaleCondition',axis = 1)
test = test.join(one_hot2)
test.head()
```

Out[16]:

	Id	MSSubClass	LotArea	SalePrice	C (all)	FV	RH	RL	RM	Abnorml	AdjLand	Alloca	Family	Normal	Partial
0	16	45	6120	132000	0	0	0	0	1	0	0	0	0	1	0
1	23	20	9742	230000	0	0	0	1	0	0	0	0	0	1	0
2	25	20	8246	154000	0	0	0	1	0	0	0	0	0	1	0
3	30	30	6324	68500	0	0	0	0	1	0	0	0	0	1	0
4	35	120	7313	277500	0	0	0	1	0	0	0	0	0	1	0

normalizing and splitting the data set:

In [17]:

```
X_test = test.drop('SalePrice', 1)
Y_test = test['SalePrice']
X_test = scaler.transform(X_test)
```

Linear regression

Linear regression attempts to model the relationship between two variables by fitting a linear equation to observed data. One variable is considered to be an explanatory variable, and the other is considered to be a dependent variable.

In [18]:

```
model = LinearRegression()
model.fit(X_train, Y_train)
Y_pred = model.predict(X_test)
mse = mean_squared_error(Y_test,Y_pred)
print('accuracy on test: ',np.sqrt(mse))
Y_pred = model.predict(X_train)
mse = mean_squared_error(Y_train,Y_pred)
print('accuracy on train: ',np.sqrt(mse))
```

```
accuracy on test:  67212.67322055003
accuracy on train:  65012.934685335
```

The relation between dependant variable and independant variables is not linear(as we previously saw in correlation matrix), so linear regression is not a good model for it.

Polynomial Linear Regression

Polynomial regression is a special case of linear regression where we fit a polynomial equation on the data with a curvilinear relationship between the target variable and the independent variables. To convert the original features into their higher order terms we will use the PolynomialFeatures class provided by scikit-learn. Next, we train the model using Linear Regression. This is still considered to be linear model as the coefficients/weights associated with the features are still linear.

In [18]:

```
features = PolynomialFeatures(degree=7)
X_train = features.fit_transform(X_train)
X_test = features.fit_transform(X_test)
model = LinearRegression()
model.fit(X_train, Y_train)
Y_pred = model.predict(X_test)
mse = mean_squared_error(Y_test,Y_pred)
print('accuracy on test: ',np.sqrt(mse))
Y_pred = model.predict(X_train)
mse = mean_squared_error(Y_train,Y_pred)
print('accuracy on train: ',np.sqrt(mse))
```

```
accuracy on test:  46770.39593751407
accuracy on train:  42349.681866222
```

dgree	mae on test	mae on train
2	60313	60145
3	58933	57123
4	56634	53930
5	55099	50756
6	49529	45792
(best params)7	46770	42349
8	kernel died	kernel died

Neural Network

Neural network is a machine learning model that tries to mimic the way of working of the biological brain. A neural network consists of multiple layers. Each layer consists of a number of nodes. The nodes of each layer are connected to the nodes of adjacent layers. We estimate the number of neurons (units) from our features. Ex: X_train.shape (1457, 14). The optimizer is asking how you want to perform this gradient descent. In this case we are using the Adam optimizer and the mean square error loss function.

In [73]:

```
model = Sequential()
#input layer
model.add(Dense(19,activation='relu'))

#hidden layers
model.add(Dense(19,activation='relu'))
model.add(Dense(19,activation='relu'))
model.add(Dense(19,activation='relu'))

#output layer
model.add(Dense(1))

model.compile(optimizer='adam', loss='mse')
```

Now that the model is ready, we can fit the model into the data. Since the dataset is large, we are going to use batch_size. It is typical to use batches of the power of 2 (32, 64, 128, 256...). In this case we are using 32. The smaller the batch size, the longer is going to take

In [ ]:

```
model.fit(x=X_train, y=Y_train.values,
```

```
        validation_data=(X_test, Y_test.values),
        batch_size=64, epochs=100)
```

In [77]:

```
Y_pred = model.predict(X_test)
mse = mean_squared_error(Y_test,Y_pred)
print('accuracy on test: ',np.sqrt(mse))
Y_pred = model.predict(X_train)
mse = mean_squared_error(Y_train,Y_pred)
print('accuracy on train: ',np.sqrt(mse))
```

```
accuracy on test:  67796.75778530823
accuracy on train:  65611.72912569232
```

batch_size	epochs	mae on test	mae on train
32	600	67186	65016
64	600	67178	64995
128	600	67144	64968
256	600	67099	64951
256	400	67082	64940
256	800	67027	64920
256	100	67011	64913
64	100	67796	65611
64	50	71936	70062

Decision Trees

Decision tree is the most powerful and popular tool for classification and prediction. A Decision tree is a flowchart like tree structure, where each internal node denotes a test on an attribute, each branch represents an outcome of the test, and each leaf node (terminal node) holds a class label. Here in this technique the goal is to create a model that predicts the value of a target variables.

In [19]:

```
model = DecisionTreeRegressor(max_depth=30, min_samples_split=3, max_features=14, random_state=0)
cross_val_score(model, X_train, Y_train, cv=10)
model.fit(X_train, Y_train)
Y_pred_test = model.predict(X_test)
mse = mean_squared_error(Y_test,Y_pred_test)
print('accuracy on test: ',np.sqrt(mse))
Y_pred = model.predict(X_train)
mse = mean_squared_error(Y_train,Y_pred)
print('accuracy on train: ',np.sqrt(mse))
```

```
accuracy on test:  12866.774363285725
accuracy on train:  11702.436252360565
```

max_depth	min_samples_split	max_features	random_state	mae on test	mae on train
10	5	14	0	43035	39336
20	5	14	0	23246	25390
30	5	14	0	21129	23236
35	5	14	0	21129	23236
30	4	14	0	16068	18121
(best params)30	3	14	0	12866	11702
30	6	14	0	15724	15568
30	3	10	0	12972	14674
30	3	9	0	15923	13744
30	3	10	1	15840	13287
30	3	5	0	14788	16611

In [20]:

```
prediction = pd.DataFrame(Y_pred_test, columns=['SalePrice']).to_csv('prediction.csv')
```

Random Forest

Random forest is a supervised learning algorithm. The "forest" it builds, is an ensemble of decision trees, usually trained with the “bagging” method. The general idea of the bagging method is that a combination of learning models increases the overall result. Random forest adds additional randomness to the model, while growing the trees. Instead of searching for the most important feature while splitting a node, it searches for the best feature among a random subset of features. This results in a wide diversity that generally results in a better model.

In [35]:

```
rf_reg = RandomForestRegressor(n_estimators=170 ,max_depth=30, random_state=0).fit(X_train, Y_train)
Y_pred = rf_reg.predict(X_test)
mse = mean_squared_error(Y_test,Y_pred)
mse = mean_squared_error(Y_test,Y_pred)
print('accuracy on test: ',np.sqrt(mse))
Y_pred = model.predict(X_train)
mse = mean_squared_error(Y_train,Y_pred)
print('accuracy on train: ',np.sqrt(mse))
```

```
accuracy on test:  22641.52735009306
accuracy on train:  65012.934685335
```

n_estimator	random_state	max_depth	mae on test	mae on train
1000	0	20	22796	65012
100	0	20	22956	65012
10	0	20	27027	65012
10	1	20	25497	65012
170	0	30	22641	65012

the accuracy on test is better than the accuracy on train so the model is underfitting.

Conclusion

After examinig the accuracy of different models with different parameters, Desicion Tree with max_depth=30, min_samples_split=3, max_features=14 and random_state=0 had the best accuracy which was 12866 on test and 11702 on train.

Refferences

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html
https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html
https://scikit-learn.org/stable/modules/neural_networks_supervised.html
https://scikit-learn.org/stable/modules/tree.html
https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html