IT for Business Analytics - Project

Name: Hector Alarcon

alt text

Project Description

The project is the House Prices: Advanced Regression Techniques competition on Kaggle. With 79 explanatory variables describing (almost) every aspect of residential homes in Ames, Iowa, this competition challenges you to predict the final price of each home. Your goal is to get the highest ranking on Kaggle within the class. However, while doing this, you need to clearly explain what you are doing between your code chunks. Basically, you should tell a story of your Kaggle score.

Evaluation metric is RMSE - lowest score will get the highest ranking.

Do not leave this project to last day. You can rush and finish it quickly but you will not get a good grade. Getting a high score on Kaggle requires a good amount of work. Keep in mind that **you can submit only 5 entries per day.**

Project Setup

You can make a copy of this notebook and save it to your Binghamton Google Drive. All your answers will be on your copy of this notebook. Name it appropriately.

Helpful DataCamp courses

The project is on building regression based machine learning models. You completed various chapters on DataCamp covering most of the algorithms you are likely to use. Below is a list of the courses/chapters you may want to go back to refresh your knowledge. You can always re-watch the videos and look at the slides for quick tips. To download chapter slides on DataCamp, start the chapter, and click on the pdf icon on the top right.

- Supervised Learning with scikit-learn: chapter 2
- Extreme Gradient Boosting with XGBoost: chapter 2, 3, 4 (*This method is not covered in the Titanic example*)
- Machine Learning with Tree-Based Models: all chapters

Don't forget to read the submission and grading guidelines at the end of the notebook

---

Hands-on Part

Important notes

Code sharing
Sharing is defined as copying or looking at another code. You are not allowed to

- share your code/solutions with other students,
- seek for code/solutions from other students.

Please report any code sharing violation to the instructor. Consider the fact that the assignments will have many correct solutions. Any similarity in the order, syntax, variable names, mistakes, typos, and other small details are proofs of code sharing. While you are allowed to discuss homeworks, assignments, and the project with other students, you are expected to write your own code.

Use of Kaggle Kernels
You can look at the available kernels on Kaggle, read the discussions, or search for examples on the Internet. This will be helpful and may speed up your project. However, you are not allowed to copy and use code available online (Kaggle or elsewhere).

Tips

Explanatory Data Analysis
Familiarize yourself with the dataset. Run some simple descriptives and graphs to find out more about the variables. Keep some of the good ones for your submission. Delete the ones that doesn't give any information.

Missing Data
Look at the missing variables and make a plan to address them if necessary. Make sure you address the missing data in the `test` dataset as well.

Feature Engineering
You can build your model with existing variables, but you should also create new variables from the existing variables (e.g., you can create new categorical variable by grouping ages of houses). If you create new variables in your `train` dataset (which is a strongly suggested), make sure to create

them in your `test` dataset as well. You can check Titanic competition Age2 variable in class notes as an example of a new variable both in train and test datasets.

> Picking the right variables, cleaning them, and engineering good ones will take the most time/effort but they will increase your score more than anything. Consider a model without Gender variable for the Titanic competition. No matter how you tune your hyperparameters or engineer new variables, a model without Gender will highly likely do worse than a simple model with Gender.

Hands-on Tasks

When you are ready with your variables, the next step is to create your model. You are asked to use the following methods to train your model and do predictions. You are also asked to fine-tune the hyperparameters. Follow the Titanic example for decision tree and random forests, and use either DataCamp notes, Kernels, or online sources to do the rest of the methods. Use gridsearch for tuning similar to Titanic example.

Model Training and Tuning

- Decision Tree (no need to create the actual decision tree with graphviz)
- Random Forests
- Regularized linear regression - ridge
- Regularized linear regression - lasso
- XGboost

```
# load the libraries
from pandas import Series,DataFrame
from sklearn.preprocessing import LabelEncoder, KBinsDiscretizer
from sklearn.metrics import mean_squared_error

#Classifiers
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.linear_model import Ridge, Lasso

# import GridSearchCV for fine-tuning
from sklearn.model_selection import GridSearchCV
# to download files from Colab to our computer
from google.colab import files

import re
import numpy as np
import pandas as pd
```

```
# load the data to dataframes
train_df = pd.read_csv("https://s3.amazonaws.com/it4ba/Kaggle/train.csv")
test_df    = pd.read_csv("https://s3.amazonaws.com/it4ba/Kaggle/test.csv")


print("Train shape: ", train_df.shape)
print("Test shape: ", test_df.shape)
```

```
Train shape:  (1460, 81)
Test shape:  (1459, 80)
```

Get to know the data

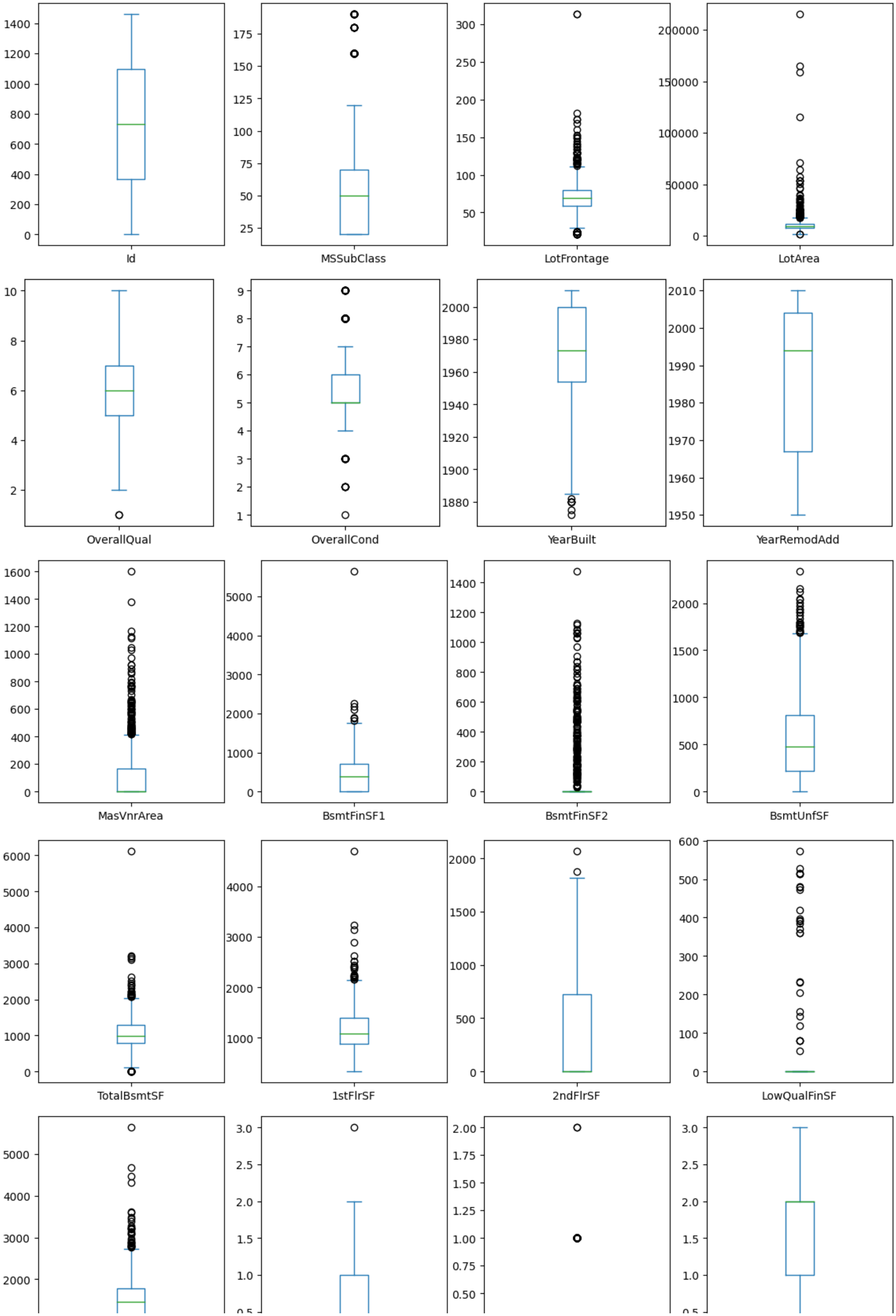
```
train_df
```

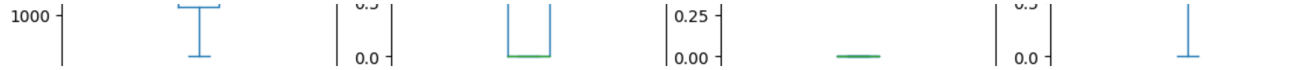
	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandCor
0	1	60	RL	65.0	8450	Pave	NaN	Reg	
1	2	20	RL	80.0	9600	Pave	NaN	Reg	
2	3	60	RL	68.0	11250	Pave	NaN	IR1	
3	4	70	RL	60.0	9550	Pave	NaN	IR1	
4	5	60	RL	84.0	14260	Pave	NaN	IR1	
...	...	...	...	...	...	...	...	...	
1455	1456	60	RL	62.0	7917	Pave	NaN	Reg	
1456	1457	20	RL	85.0	13175	Pave	NaN	Reg	
1457	1458	70	RL	66.0	9042	Pave	NaN	Reg	
1458	1459	20	RL	68.0	9717	Pave	NaN	Reg	
1459	1460	20	RL	75.0	9937	Pave	NaN	Reg	

1460 rows × 81 columns

```
cols_graphed_box = []

#Let's visualize the distribution of the numeric data
for col in train_df.columns:
  if(train_df[col].dtype == int or train_df[col].dtype == float):
    cols_graphed_box.append(col)
    if(len(cols_graphed_box) == 4):
      train_df.plot.box(column=cols_graphed_box, sharey=False, subplots=True, figsize=(14,
      cols_graphed_box = []
```


Id
MSSubClass
LotFrontage
LotArea
OverallQual
OverallCond
YearBuilt
YearRemodAdd
MasVnrArea
BsmtFinSF1
BsmtFinSF2
BsmtUnfSF
TotalBsmtSF
1stFlrSF
2ndFlrSF
LowQualFinSF



What are the features that are highly correlated with the target?

```
corr_matrix = train_df.corr(min_periods = 100, numeric_only=True)
corr_series = corr_matrix.abs().unstack()
index_pairs = corr_series.SalePrice.sort_values(ascending=False)[1:]
print("Corr to Sale Price")
print(index_pairs)
print()

#Keep the ones above 0.3 coeff
#Tested this but didn't improve the score, I reimplemented after the best submission to see
print("\nTop Abs Corr to Price")
index_pairs = index_pairs[index_pairs > 0.3]
print(index_pairs)
print()
'''
#This is from implementation, leaving
train_id = train_df.copy()[["Id","SalePrice"]]
test_id = test_df.copy()["Id"]

train_df = train_df[index_pairs.index.values]
test_df = test_df[index_pairs.index.values]

train_df.insert(0, "Id", train_id["Id"])
train_df["SalePrice"] = train_id["SalePrice"]
test_df.insert(0, "Id", test_id)
'''
```

```
Corr to Sale Price
OverallQual      0.790982
GrLivArea        0.708624
GarageCars       0.640409
GarageArea       0.623431
TotalBsmtSF      0.613581
1stFlrSF         0.605852
FullBath         0.560664
TotRmsAbvGrd     0.533723
YearBuilt        0.522897
YearRemodAdd     0.507101
GarageYrBlt      0.486362
MasVnrArea       0.477493
Fireplaces       0.466929
BsmtFinSF1       0.386420
LotFrontage      0.351799
WoodDeckSF       0.324413
2ndFlrSF         0.319334
OpenPorchSF      0.315856
HalfBath         0.284108
LotArea          0.263843
BsmtFullBath     0.227122
BsmtUnfSF        0.214479
BedroomAbvGr     0.168213
KitchenAbvGr     0.135907
EnclosedPorch    0.128578
ScreenPorch      0.111447
PoolArea         0.092404
MSSubClass       0.084284
OverallCond      0.077856
MoSold           0.046432
3SsnPorch        0.044584
YrSold           0.028923
LowQualFinSF     0.025606
Id               0.021917
MiscVal          0.021190
BsmtHalfBath     0.016844
BsmtFinSF2       0.011378
dtype: float64


Top Abs Corr to Price
OverallQual     0.790982
GrLivArea       0.708624
GarageCars      0.640409
GarageArea      0.623431
TotalBsmtSF     0.613581
1stFlrSF        0.605852
FullBath        0.560664
TotRmsAbvGrd    0.533723
YearBuilt       0.522897
YearRemodAdd    0.507101
GarageYrBlt     0.486362
MasVnrArea      0.477493
Fireplaces      0.466929
BsmtFinSF1      0.386420
```

```
    LotFrontage     0.351799
    WoodDeckSF      0.324413
'''
nan_cols_test = [i for i in test_df.columns if test_df[i].isnull().any()]
nan_cols_train = [i for i in train_df.columns if train_df[i].isnull().any()]

print(nan_cols_test)
print(nan_cols_train)'''
```

```
'\nnan_cols_test = [i for i in test_df.columns if test_df[i].isnull().any()]\nnan_cols
_train = [i for i in train_df.columns if train_df[i].isnull().any()]\n\nprint(nan_cols
_test)\nprint(nan_cols_train)'
```

Feature engineering

Tested on the main training set and later reimplemented to improve best score but didn't hit the mark.

```
'''
#Feature engineering - Adding the total area

train_df["TotalSF"] = 0
test_df["TotalSF"] = 0

#Making a new feature, total squared footage
for col in train_df.columns:
  if("SF" in col and col != "TotalSF"):
    #Treating nans before sum
    train_df[col]= train_df[col].replace(np.nan, 0)
    test_df[col]= test_df[col].replace(np.nan, 0)

    train_df["TotalSF"] = train_df["TotalSF"] + train_df[col]
    test_df["TotalSF"] = test_df["TotalSF"] + test_df[col]

    #Dropping after adding
    train_df.drop(col, inplace=True, axis=1)
    test_df.drop(col, inplace=True, axis=1)

#Adding total areas to it
for col in train_df.columns:
  if("Area" in col and col != "TotalArea"):
    #Treating nans before sum
    train_df[col]= train_df[col].replace(np.nan, 0)
    test_df[col]= test_df[col].replace(np.nan, 0)
    train_df["TotalSF"] = train_df["TotalSF"] + train_df[col]
    test_df["TotalSF"] = test_df["TotalSF"] + test_df[col]

    #Dropping after adding
    train_df.drop(col, inplace=True, axis=1)
    test_df.drop(col, inplace=True, axis=1)
    print(col)

'''
```

```
'\n#Feature engineering - Adding the total area\n\ntrain_df["TotalSF"] = 0\ntest_df["T
otalSF"] = 0\n\n#Making a new feature, total squared footage\nfor col in train_df.colu
mns:\n  if("SF" in col and col != "TotalSF"):\n    #Treating nans before sum\n    trai
n_df[col]= train_df[col].replace(np.nan, 0)\n    test_df[col]= test_df[col].replace(n
p.nan, 0)\n\n    train_df["TotalSF"] = train_df["TotalSF"] + train_df[col]\n    test_d
f["TotalSF"] = test_df["TotalSF"] + test_df[col]\n\n    #Dropping after adding\n    tr
ain_df.drop(col, inplace=True, axis=1)\n    test_df.drop(col, inplace=True, axis=1)\n
\n#Adding total areas to it\nfor col in train_df.columns:\n  if("Area" in col and col
!= "TotalArea"):\n    #Treating nans before sum\n    train_df[col]= train_df[col].repl
ace(np.nan, 0)\n    test_df[col]= test_df[col].replace(np.nan, 0)\n    train_df["Total
SF"] = train_df["TotalSF"] + train_df[col]\n    test_df["TotalSF"] = test_df["TotalS
```

Data preprocessing

Taking care of nans, encoding and normalization

```
    cols_to_drop = []
    cols_to_encode = []
    cols_to_normalize = []

    for column in train_df.columns[1:-1]:
      column_dtype = train_df[column].dtype
      type_values = train_df[column].apply(type).value_counts()
      #Drop overwhelmingly nan filled columns
      if(train_df[column].isna().sum() > type_values[0]/2):
        cols_to_drop.append(column)
      #Encoding candidate, probably a string
      elif(column_dtype == "object" or column_dtype == "O"):
        #print("adding column ",column, " to encode list")
        cols_to_encode.append(column)


      if(column_dtype == "float"):
        train_df[column] = train_df[column].fillna(train_df[column].mode())
        test_df[column] = test_df[column].fillna(test_df[column].mode())

        try:
          train_df[column] = train_df[column].astype("np.integer")
          test_df[column] = test_df[column].astype("np.integer")
        except:
          print("This column: ", column)
      if(len(type_values) >=2):
        values = train_df[column].value_counts(dropna=False)
        #Drop overly dominant features
        if(train_df[column].value_counts(normalize=True)[0] > 0.5):
          cols_to_drop.append(column)

      #Explore distribution of values
      if(len(train_df[column].value_counts()) > 2 and column not in cols_to_drop):
        if(train_df[column].value_counts(normalize=True).iat[0] > 0.75):
          if(isinstance(train_df.iloc[0][column], np.integer) or isinstance(train_df.iloc[0][co
            cols_to_normalize.append(column)

      #No need to encode if they are going to be dropped
      if(column in cols_to_drop and column in cols_to_encode):
        cols_to_encode.remove(column)
      #If its an unbalanced variable, needs to be normalized then encoded
      if(column in cols_to_encode and column in cols_to_normalize):
        cols_to_encode.remove(column)

    print(cols_to_drop)
    print(cols_to_encode)
    print(cols_to_normalize)

    train_df = train_df.drop(cols_to_drop, axis=1)
    test_df = test_df.drop(cols_to_drop, axis=1)
```

```
This column:  LotFrontage
This column:  MasVnrArea
This column:  GarageYrBlt
['Alley', 'Alley', 'MasVnrType', 'BsmtCond', 'BsmtExposure', 'BsmtFinType2', 'Electrica
['MSZoning', 'Street', 'LotShape', 'LandContour', 'Utilities', 'LotConfig', 'LandSlope'
['BsmtFinSF2', 'LowQualFinSF', 'BsmtHalfBath', 'KitchenAbvGr', 'EnclosedPorch', '3SsnPo
```

```
enc = LabelEncoder()

nan_cols_train = [i for i in cols_to_encode if train_df[i].isnull().any()]
nan_cols_test = [i for i in cols_to_encode if test_df[i].isnull().any()]

for col in nan_cols_train:
  mode = train_df[col].value_counts(normalize=True).index[0]
  train_df[col]= train_df[col].replace(np.nan, mode)

for col in nan_cols_test:
  mode = test_df[col].value_counts(normalize=True).index[0]
  test_df[col]= test_df[col].replace(np.nan, mode)


for col in cols_to_encode:
  enc.fit(train_df[col])
  train_df[col]= enc.transform(train_df[col])
  test_df[col]= enc.transform(test_df[col])



nan_cols_train = [i for i in cols_to_normalize if train_df[i].isnull().any()]
nan_cols_test = [i for i in cols_to_normalize if test_df[i].isnull().any()]

for col in nan_cols_train:
  mode = train_df[col].value_counts(normalize=True).index[0]
  train_df[col]= train_df[col].replace(np.nan, mode)

for col in nan_cols_test:
  mode = test_df[col].value_counts(normalize=True).index[0]
  test_df[col]= test_df[col].replace(np.nan, mode)

#These are heavily unbalanced features
for col in cols_to_normalize:
  print(train_df[col].value_counts(bins=3))

#Let's remove them
for col in cols_to_normalize:
  train_df.drop(col, axis=1, inplace=True)
  test_df.drop(col, axis=1, inplace=True)
```

```
(-1.4749999999999999, 491.333]    1405
(491.333, 982.667]                  45
```

```
(982.667, 1474.0]                  10
Name: BsmtFinSF2, dtype: int64
(-0.573, 190.667]     1441
(381.333, 572.0]        13
(190.667, 381.333]       6
Name: LowQualFinSF, dtype: int64
(-0.003, 0.667]    1378
(0.667, 1.333]       80
(1.333, 2.0]          2
Name: BsmtHalfBath, dtype: int64
(-0.004, 1.0]    1393
(1.0, 2.0]         65
(2.0, 3.0]          2
Name: KitchenAbvGr, dtype: int64
(-0.553, 184.0]    1391
(184.0, 368.0]       67
(368.0, 552.0]        2
Name: EnclosedPorch, dtype: int64
(-0.509, 169.333]     1447
(169.333, 338.667]      11
(338.667, 508.0]         2
Name: 3SsnPorch, dtype: int64
(-0.481, 160.0]    1388
(160.0, 320.0]       65
(320.0, 480.0]        7
Name: ScreenPorch, dtype: int64
(-0.739, 246.0]    1453
(492.0, 738.0]        6
(246.0, 492.0]        1
Name: PoolArea, dtype: int64
(-15.501, 5166.667]      1458
(5166.667, 10333.333]       1
(10333.333, 15500.0]        1
Name: MiscVal, dtype: int64
```

```
print("Train shape: ", train_df.shape)
print("Test shape: ", test_df.shape)
```

```
Train shape:  (1460, 59)
Test shape:  (1459, 58)
```

```
#Dealing with remaining NaNs
train_df.fillna(train_df.mode(), inplace=True)
test_df.fillna(test_df.mode(), inplace=True)

nan_cols_train = [i for i in train_df.columns if train_df[i].isnull().any()]
nan_cols_test = [i for i in test_df.columns if test_df[i].isnull().any()]

for col in nan_cols_test:
  test_df[col]= test_df[col].replace(np.nan, 0)

for col in nan_cols_train:
  train_df[col]= train_df[col].replace(np.nan, 0)


print(train_df["LotFrontage"].value_counts(normalize=True).index[0])
print(train_df["LotFrontage"].value_counts(normalize=True, bins=5))
print(train_df["LotFrontage"].mean())
print(train_df["LotFrontage"].median())
print(train_df["LotFrontage"][train_df["LotFrontage"] < 5])
```

```
0.0
(-0.314, 62.6]    0.495205
(62.6, 125.2]     0.488356
(125.2, 187.8]    0.015068
(250.4, 313.0]    0.001370
(187.8, 250.4]    0.000000
Name: LotFrontage, dtype: float64
57.62328767123287
63.0
7       0.0
12      0.0
14      0.0
16      0.0
24      0.0
       ...
1429    0.0
1431    0.0
1441    0.0
1443    0.0
1446    0.0
Name: LotFrontage, Length: 259, dtype: float64
```

```
'''
#Dealing with outliers
def removeOutliers(col):
  #Most represented value in data
  most_rep = train_df["LotFrontage"].value_counts(normalize=True).index[0]

  q3 = col.quantile(0.75)
  q1 = col.quantile(0.25)
  iqr = q3 - q1
  mean = col.mean()
  median = col.median()
  threshold = 1.5*iqr
  upper_fence = q3+threshold
  lower_fence = q3-threshold

  if(abs(mean - most_rep) < abs(median - most_rep)):
    col[col > upper_fence] = mean
    col[col < upper_fence] = mean
  else:
    col[col > upper_fence] = median
    col[col < upper_fence] = median

  return col

train_df.iloc[:, 1:-1] = train_df.iloc[:, 1:-1].apply(removeOutliers, axis=1)
test_df.iloc[:, 1:-1] = test_df.iloc[:, 1:].apply(removeOutliers, axis=1)
'''
```

```
'\n#Dealing with outliers\ndef removeOutliers(col):\n  #Most represented value in data
\n  most_rep = train_df["LotFrontage"].value_counts(normalize=True).index[0]\n\n  q3 =
col.quantile(0.75)\n  q1 = col.quantile(0.25)\n  iqr = q3 - q1\n  mean = col.mean()\n
median = col.median()\n  threshold = 1.5*iqr\n  upper_fence = q3+threshold\n  lower_fe
nce = q3-threshold\n\n  if(abs(mean - most_rep) < abs(median - most_rep)):\n    col[co
l > upper_fence] = mean\n    col[col < upper_fence] = mean\n  else:\n    col[col > upp
er_fence] = median\n    col[col < upper_fence] = median\n\n  return col\n\ntrain_df.il
oc[:, 1:-1] = train_df.iloc[:, 1:-1].apply(removeOutliers, axis=1)\ntest_df.iloc[:, 1:
```

```
train_df
```

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	LotShape	LandContour	l
0	1	60	3	65.0	8450	1	3	3	
1	2	20	3	80.0	9600	1	3	3	
2	3	60	3	68.0	11250	1	0	3	
3	4	70	3	60.0	9550	1	0	3	
4	5	60	3	84.0	14260	1	0	3	
...	...	...	...	...	...	...	...	...	
1455	1456	60	3	62.0	7917	1	3	3	
1456	1457	20	3	85.0	13175	1	3	3	
1457	1458	70	3	66.0	9042	1	3	3	
1458	1459	20	3	68.0	9717	1	3	3	
1459	1460	20	3	75.0	9937	1	3	3	

1460 rows × 59 columns

```
X_train = train_df.drop(["Id", "SalePrice"], axis=1)
Y_train = train_df["SalePrice"]
X_test  = test_df.drop("Id", axis=1)
```

```
print(X_train.shape)
print(X_test.shape)
```

```
(1460, 57)
(1459, 57)
```

```
#Double checking nulls
print(X_train.isnull().sum().sum())
print(X_test.isnull().sum().sum())
```

```
0
0
```

Classifiers

Decision Tree

Decision trees are supervised learning algorithms that are used both in classification and regression problems. They can be prone to bias and overfitting depending on our features and the shape of our tree in case of an unbalanced tree (prunning can also help with that). It works by using our features to answer questions and split into branches that lead to nodes where it essentially reaches a classification or regression result.

Hyperparameters description

max_depth: how tall the resulting tree is going to be
min_samples_split: min number of samples needed to branch out

```
# train and predict
dt = DecisionTreeRegressor (criterion="friedman_mse", random_state = 54)


params_grid = {"max_depth":[10, 30, 50, 70], "min_samples_split":[4, 6, 8 , 10, 20, 25, 40,
clf = GridSearchCV(dt, params_grid, scoring="neg_root_mean_squared_error")


clf.fit(X_train, Y_train)
```

```
▸ GridSearchCV
▸ estimator: DecisionTreeRegressor
    ▸ DecisionTreeRegressor
```

```
Y_pred = clf.predict(X_test)


# prepare a submission file
submission = pd.DataFrame({
        "Id": test_df["Id"],
        "SalePrice": Y_pred
    })
submission.to_csv('dt1.csv', index=False)

files.download('dt1.csv')
```

Kaggle Score: 0.18829

Random Forests

Random forest is an ensemble machine learning lagorithm that combines the output of multiple decision trees to reach a single result. Random forests unlike decision trees alone also considers subsets of splits across our features as opposed to all of them. In addition it uses bagging, so all of our weak individual trees are trained in parallel to reach a single result.

Hyperparameters description

n_estimators: numbers of weak learners to use
max_depth: how tall the resulting tree is going to be
min_samples_split: min number of samples needed to branch out

```
params_grid = {"n_estimators":[80, 100, 120, 140, 160, 180, 200],"max_depth":[5, 10, 30, 50
rfr = RandomForestRegressor(criterion="friedman_mse", random_state = 54)
clf_rfr = GridSearchCV(rfr , params_grid, scoring="neg_root_mean_squared_error")
```

```
clf_rfr.fit(X_train, Y_train)
```

```
GridSearchCV
estimator: RandomForestRegressor
RandomForestRegressor
```

```
Y_pred = clf_rfr.predict(X_test)
```

```
print(clf_rfr.best_params_)
```

```
{'max_depth': 30, 'min_samples_split': 6, 'n_estimators': 100}
```

```
# prepare a submission file
submission = pd.DataFrame({
        "Id": test_df["Id"],
        "SalePrice": Y_pred
    })
submission.to_csv('rfr.csv', index=False)

files.download('rfr.csv')
```

{'max_depth': 30, 'min_samples_split': 6, 'n_estimators': 100} : **Kaggle Score: 0.14716**

```
params_grid = {"n_estimators":[100, 140, 180, 200],"max_depth":[30], "min_samples_split":[6
rfr = RandomForestRegressor(criterion="friedman_mse", random_state = 54)
clf_rfr = GridSearchCV(rfr , params_grid, scoring="neg_root_mean_squared_error", cv=5)
```

```
clf_rfr.fit(X_train, Y_train)
```

```
▸ GridSearchCV
▸ estimator: RandomForestRegressor
  ▸ RandomForestRegressor
```

```
Y_pred = clf_rfr.predict(X_test)
```

```
print(clf_rfr.best_params_)
print(clf_rfr.best_score_)
```

```
{'ccp_alpha': 0.01, 'max_depth': 30, 'min_impurity_decrease': 0, 'min_samples_split': 6
-29106.029586321634
```

Previous best

{'ccp_alpha': 0.01, 'max_depth': 30, 'min_samples_split': 6, 'n_estimators': 100}

-29106.029586321634 -- 0.14 kaggle

```
# prepare a submission file
submission = pd.DataFrame({
        "Id": test_df["Id"],
        "SalePrice": Y_pred
    })
submission.to_csv('rfr.csv', index=False)

files.download('rfr.csv')
```

Ridge Regression

Ridge is a type of regularized regression that aims to penalize parameters that bring complexity into the model by shrinking them however the coefficients are squared as opposed to the magnitude like in lasso.

Hyperparameters description

alpha: regularization parameter, how much it penalizes irrelevant features

```
params_grid = {"alpha":[0.8, 1, 1.2, 1.4, 1.6, 1.8, 2]}
ridge = Ridge(random_state=54)
clf_ridge = GridSearchCV(ridge , params_grid, scoring="neg_root_mean_squared_error")
```

```
clf_ridge.fit(X_train, Y_train)
```

```
▸ GridSearchCV
▸ estimator: Ridge
    ▸ Ridge
```

```
Y_pred = clf_ridge.predict(X_test)
```

```
print(clf_ridge.best_params_)
```

```
{'alpha': 2}
```

```
# prepare a submission file
submission = pd.DataFrame({
        "Id": test_df["Id"],
        "SalePrice": Y_pred
    })
submission.to_csv('ridge.csv', index=False)

files.download('ridge.csv')
```

Lasso Regression

Lasso is a type of regularized regression that aims to penalize parameters that introduce too much complexity to the model by multiplying a parameter alpha; progressively shrinking them. However, one of the weaknesses is that is can entirely shrink certain weights out of our model depending on the regularization strength.

Hyperparameters description

alpha: regularization parameter, how much it penalizes irrelevant features

```
params_grid = {"alpha":[0.2, 0.4, 0.5, 1, 1.2, 1.4, 1.6, 1.8, 2, 3, 5, 7]}
#1000 iterations were not enough for convergence
lasso = Lasso(random_state=54, max_iter=1000000, selection="random")
clf_lasso = GridSearchCV(lasso , params_grid, scoring="neg_root_mean_squared_error")
```

```
clf_lasso.fit(X_train, Y_train)
```

```
▸ GridSearchCV
▸ estimator: Lasso
  ▸ Lasso
```

```
Y_pred = clf_lasso.predict(X_test)
```

```
print(clf_lasso.best_params_)
```

```
{'alpha': 7}
```

```
# prepare a submission file
submission = pd.DataFrame({
        "Id": test_df["Id"],
        "SalePrice": Y_pred
    })
submission.to_csv('lasso.csv', index=False)

files.download('lasso.csv')
```

XGradientBoost

Gradient boosting is a type of ensemble model machine learning algorithm. It uses "boosting" which is a update sequential models denoted by the number of boosting rounds to perform. Every boosting round learns from the previous one making even weak learners perform better than individual decision trees. It also differentiates from Bagging because it doesn't averge a set number of models. This particular scikit model also has the ability to specify prunning parameters so that we can adjust models that are being affected by the data's inherent variability.

Hyperparameters description

loss: the way the distance between predictions are calculated (expected - output)
learning_rate: parameter that controls the rate of change throughout the learning process
n_estimators: numbers of weak learners to use
max_depth: how tall the resulting tree is going to be
min_samples_split: min number of samples needed to branch out
max_features: applying an operation for the top number of features to consider
subsample: fraction of samples to train the base learners
ccp_alpha: prunning parameter, (reduces variance)

```
params_grid = {"loss":["squared_error", "quantile"], "learning_rate": [0.05, 0.1, 0.115, 0.
xgbr = GradientBoostingRegressor(random_state=54)
clf_xgbr = GridSearchCV(xgbr , params_grid, scoring="neg_root_mean_squared_error")
```

```
clf_xgbr.fit(X_train, Y_train)
```

```
GridSearchCV
estimator: GradientBoostingRegressor
GradientBoostingRegressor
```

```
Y_pred = clf_xgbr.predict(X_test)
```

```
print(clf_xgbr.best_params_)
```

```
{'learning_rate': 0.1, 'loss': 'squared_error', 'max_depth': 5, 'max_features': None, '
```

```
# prepare a submission file
submission = pd.DataFrame({
        "Id": test_df["Id"],
        "SalePrice": Y_pred
    })
submission.to_csv('xgbr.csv', index=False)

files.download('xgbr.csv')
```

Using the base line from before for fine tunning

```
params_grid = {"subsample":[0.4, 0.6, 0.8, 1], "learning_rate": [0.12], "n_estimators":[100
xgbr = GradientBoostingRegressor(random_state=54,  n_iter_no_change=100, loss="absolute_err
clf_xgbr = GridSearchCV(xgbr , params_grid, scoring="neg_root_mean_squared_error", cv=10)
```

```
clf_xgbr.fit(X_train, Y_train)
```

```
GridSearchCV
estimator: GradientBoostingRegressor
GradientBoostingRegressor
```

```
Y_pred = clf_xgbr.predict(X_test)
```

```
print(clf_xgbr.best_params_)
print(clf_xgbr.best_score_)
```

```
{'ccp_alpha': 0.1, 'learning_rate': 0.12, 'max_depth': 5, 'min_samples_leaf': 2, 'min_s
-45228.794299142275
```

```
# prepare a submission file
submission = pd.DataFrame({
        "Id": test_df["Id"],
        "SalePrice": Y_pred
    })
submission.to_csv('xgbr.csv', index=False)

files.download('xgbr.csv')
```

More fine tunning using the previous best score

Trying some prunning using impurities and different alpha values on previous best estimators

```
params_grid = {"subsample":[0.8], "learning_rate": [0.115], "n_estimators":[110] , "max_dep
xgbr_2 = GradientBoostingRegressor(random_state=54, min_samples_split= 30, min_samples_leaf
clf_xgbr_2 = GridSearchCV(xgbr_2 , params_grid, scoring="neg_root_mean_squared_error", cv=5
```

```
clf_xgbr_2.fit(X_train, Y_train)
```

```
▸ GridSearchCV
▸ estimator: GradientBoostingRegressor
    ▸ GradientBoostingRegressor
```

```
Y_pred = clf_xgbr_2.predict(X_test)
```

```
print(clf_xgbr_2.best_params_)
print(clf_xgbr_2.best_score_)
```

```
{'alpha': 0.04, 'ccp_alpha': 0.01, 'learning_rate': 0.115, 'max_depth': 5, 'min_impurit
-124051.20754867699
```

Best Score

model: {'learning_rate': 0.115, 'loss': 'absolute_error', 'max_depth': 5, 'min_samples_leaf': 2, 'min_samples_split': 30, 'n_estimators': 110, 'subsample': 0.8}

RMSE: -28064.08723214544

```
params_grid = {"subsample":[0.8], "learning_rate": [0.115], "n_estimators":[110] , "max_dep
xgbr_2 = GradientBoostingRegressor(random_state=54, min_samples_split= 30, min_samples_leaf
clf_xgbr_2 = GridSearchCV(xgbr_2 , params_grid, scoring="neg_root_mean_squared_error", cv=1
```

```
clf_xgbr_2.fit(X_train, Y_train)
Y_pred = clf_xgbr_2.predict(X_test)
```

```
print(clf_xgbr_2.best_params_)
print(clf_xgbr_2.best_score_)
```

```
{'learning_rate': 0.115, 'max_depth': 5, 'min_impurity_decrease': 0, 'n_estimators': 11
-28064.08723214544
```

```
# prepare a submission file
submission = pd.DataFrame({
        "Id": test_df["Id"],
        "SalePrice": Y_pred
    })
submission.to_csv('xgbr.csv', index=False)

files.download('xgbr.csv')
```

Doing some more transformations to improve XGradient Boosting

```
for col in X_train.columns:
  #Let's take a look again at the distribution of values
  print(X_train[col].value_counts(normalize=True))
```

```
20     0.367123
60     0.204795
50     0.098630
120    0.059589
30     0.047260
160    0.043151
70     0.041096
80     0.039726
90     0.035616
190    0.020548
85     0.013699
75     0.010959
45     0.008219
180    0.006849
40     0.002740
Name: MSSubClass, dtype: float64
3    0.788356
4    0.149315
1    0.044521
2    0.010959
0    0.006849
Name: MSZoning, dtype: float64
0.0      0.177397
60.0     0.097945
70.0     0.047945
80.0     0.047260
50.0     0.039041
           ...
137.0    0.000685
38.0     0.000685
33.0     0.000685
150.0    0.000685
46.0     0.000685
Name: LotFrontage, Length: 111, dtype: float64
7200     0.017123
9600     0.016438
6000     0.011644
9000     0.009589
8400     0.009589
           ...
14601    0.000685
13682    0.000685
4058     0.000685
17104    0.000685
9717     0.000685
```

```
Name: LotArea, Length: 1073, dtype: float64
1    0.99589
0    0.00411
Name: Street, dtype: float64
3    0.633562
0    0.331507
1    0.028082
2    0.006849
Name: LotShape, dtype: float64
3    0.897945
0    0.043151
1    0.034247
2    0.024658
```

```
train_df.insert(1, "TotalSF", value=0)
test_df.insert(1, "TotalSF", value=0)

#Making a new feature, total squared footage
for col in train_df.columns:
  if("SF" in col and col != "TotalSF"):
    #Treating nans before sum
    train_df[col]= train_df[col].replace(np.nan, 0)
    test_df[col]= test_df[col].replace(np.nan, 0)

    train_df["TotalSF"] = train_df["TotalSF"] + train_df[col]
    test_df["TotalSF"] = test_df["TotalSF"] + test_df[col]

    #Dropping after adding
    train_df.drop(col, inplace=True, axis=1)
    test_df.drop(col, inplace=True, axis=1)
  elif("Porch" in col):
    train_df["TotalSF"] = train_df["TotalSF"] + train_df[col]
    test_df["TotalSF"] = test_df["TotalSF"] + test_df[col]

    #Dropping after adding
    train_df.drop(col, inplace=True, axis=1)
    test_df.drop(col, inplace=True, axis=1)
  elif("Area" in col and col != "TotalArea"): #Making a new feature, total area
    #Treating nans before sum
    train_df[col]= train_df[col].replace(np.nan, 0)
    test_df[col]= test_df[col].replace(np.nan, 0)
    train_df["TotalSF"] = train_df["TotalSF"] + train_df[col]
    test_df["TotalSF"] = test_df["TotalSF"] + test_df[col]

    #Dropping after adding
    train_df.drop(col, inplace=True, axis=1)
    test_df.drop(col, inplace=True, axis=1)
```

```
train_df
```

	Id	TotalSF	MSSubClass	MSZoning	LotFrontage	Street	LotShape	LandContour
0	1	14387.0	60	3	65.0	1	3	3
1	2	15406.0	20	3	80.0	1	3	3
2	3	17474.0	60	3	68.0	1	0	3
3	4	15173.0	70	3	60.0	1	0	3
4	5	22408.0	60	3	84.0	1	0	3
...	...	...	...	...	...	...	...	...
1455	1456	13617.0	60	3	62.0	1	3	3
1456	1457	21210.0	20	3	85.0	1	3	3
1457	1458	16338.0	70	3	66.0	1	3	3
1458	1459	13606.0	20	3	68.0	1	3	3
1459	1460	15751.0	20	3	75.0	1	3	3

1460 rows × 49 columns

```
print("Range of values per remainining features")
for col in train_df.columns[1:-1]:
  print("Column: ", col, "\n range: ", len(train_df[col].value_counts(normalize=True)))
```

```
Range of values per remainining features
Column:  TotalSF
 range:  1389
Column:  MSSubClass
 range:  15
Column:  MSZoning
 range:  5
Column:  LotFrontage
 range:  111
Column:  Street
 range:  2
Column:  LotShape
 range:  4
Column:  LandContour
 range:  4
Column:  Utilities
 range:  2
Column:  LotConfig
 range:  5
Column:  LandSlope
 range:  3
Column:  Neighborhood
 range:  25
Column:  Condition1
 range:  9
```

```
Column:  Condition2
 range:  8
Column:  BldgType
 range:  5
Column:  HouseStyle
 range:  8
Column:  OverallQual
 range:  10
Column:  OverallCond
 range:  9
Column:  YearBuilt
 range:  112
Column:  YearRemodAdd
 range:  61
Column:  RoofStyle
 range:  6
Column:  RoofMatl
 range:  8
Column:  Exterior1st
 range:  15
Column:  Exterior2nd
 range:  16
Column:  ExterQual
 range:  4
Column:  ExterCond
 range:  5
Column:  Foundation
 range:  6
Column:  BsmtQual
 range:  4
Column:  BsmtFinType1
 range:  6
Column:  Heating
```

```
train_df.drop("YearRemodAdd", axis=1, inplace=True)
test_df.drop("YearRemodAdd", axis=1, inplace=True)
```

```
binerizer_100 = []
binerizer_10 = []

for col in train_df.columns[1:-2]:
  count = len(train_df[col].value_counts(normalize=True))
  if(count > 100):
    binerizer_100.append(col)
  elif(count > 10):
    binerizer_10.append(col)

print("binerizer_100 : ",binerizer_100)
print("binerizer_10 : ",binerizer_10)

#Let's use kmeans because of the spread of the values
est = KBinsDiscretizer(encode="ordinal", strategy="kmeans")
train_df[binerizer_100] = est.fit_transform(train_df[binerizer_100])
test_df[binerizer_100] = est.transform(test_df[binerizer_100])

est = KBinsDiscretizer(encode="ordinal", strategy="kmeans")
train_df[binerizer_10] = est.fit_transform(train_df[binerizer_10])
test_df[binerizer_10] = est.transform(test_df[binerizer_10])
```

```
binerizer_100 :  ['TotalSF', 'LotFrontage', 'YearBuilt']
binerizer_10 :  ['MSSubClass', 'Neighborhood', 'Exterior1st', 'Exterior2nd', 'TotRmsAbv
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_discretization.py:268: C
  centers = km.fit(column[:, None]).cluster_centers_[:, 0]
```

```
quality_cols = []

for col in train_df.columns:
  if("QC" in col or "Qual" in col):
    quality_cols.append(col)

#The other qualities don't seem to be directly related to overall quality yet, overall qual
print(train_df[quality_cols])
quality_cols.remove("OverallQual")

train_df.drop(quality_cols, inplace=True, axis=1)
test_df.drop(quality_cols, inplace=True, axis=1)
```

```
      OverallQual  ExterQual  BsmtQual  HeatingQC  KitchenQual
0               7          2         2          0            2
1               6          3         2          0            3
2               7          2         2          0            2
3               7          3         3          2            2
4               8          2         2          0            2
...           ...        ...       ...        ...          ...
1455            6          3         2          0            3
1456            6          3         2          4            3
1457            7          0         3          0            2
1458            5          3         3          2            2
```

```
1459            5           2          3           2             3

[1460 rows x 5 columns]
```

```
train_df.columns
```

```
Index(['Id', 'TotalSF', 'MSSubClass', 'MSZoning', 'LotFrontage', 'Street',
       'LotShape', 'LandContour', 'Utilities', 'LotConfig', 'LandSlope',
       'Neighborhood', 'Condition1', 'Condition2', 'BldgType', 'HouseStyle',
       'OverallQual', 'OverallCond', 'YearBuilt', 'RoofStyle', 'RoofMatl',
       'Exterior1st', 'Exterior2nd', 'ExterCond', 'Foundation', 'BsmtFinType1',
       'Heating', 'CentralAir', 'BsmtFullBath', 'FullBath', 'HalfBath',
       'BedroomAbvGr', 'TotRmsAbvGrd', 'Functional', 'Fireplaces',
       'GarageYrBlt', 'GarageFinish', 'GarageCars', 'PavedDrive', 'MoSold',
       'YrSold', 'SaleType', 'SaleCondition', 'SalePrice'],
      dtype='object')
```

```
bath_cols = []
```