My grades for **Assignment 3**

Q1

8.5 / 10

Submit here the pdf printout of your entire notebook, including the outputs of your code.

Gajjala-Jahnavi_Qiu-Sichen_Kasibhatla-Venkata-Sreenivas-Prasad-Assignment-3

March 4, 2024

0.1 Q1. [0.1] Ensembles

Answer the following questions:

1. What characteristics do Bagging and Boosting methods have in common?
2. Explain the concept of out-of-bag error within the context of bagging techniques.
3. In the original XGBoost algorithm, why is the base predictor $F_0(x)$ set as the arithmetic mean of the target values, as opposed to any other value? (Refer to the lecture slides on XGBoost.)
4. How are predictions aggregated in Gradient Boosting Models?

Ans 1.

Both Bagging and Boosting are combined by multi weak learners to generate a strong ensemble method. Both methods need to deal with variance and bia balance to increase the accurancy.And both Bagging and Bossting methods can use classification and regression to solve problems.

Ans 2.

In Bagging method, each basic classifier uses different input subset data to make training. Out-of-bag error is the mean prediction error for each training sample.Bagging method will collect these errors to optimize the final bagging step.

Ans 3.

$F_0(x)$ in XGBoost algorithm is the mean of the target values, it is used to provide the optimum starting point for a boosting method, which means we can get mean of the second mean of target values $F_1(x)$ by using $F_0(x)$ integrates with new model $F_1(h)$. set the smallest value $F(x)$ can minimizes the loss function. As a result, we use $F_0(x)$ as the mean of the target values.

Ans 4.

In Gradient Boosting Models, predictions are aggregated sequentially. Each new model that is added to the ensemble learns from the mistakes (residual errors) of the previous model. The final prediction is the sum of the predictions from all the models.

0.2 Q2. [0.1] Data Preparation

1. Utilize the information gained from the data exploration stage and perform any preprocessing that you deem necessary. Provide reasons for any steps you take.

2. We will predict the **log of the SalePrice** feature and use Root Mean Squared Error (RMSE) as our evaluation metric. Alternatively, you may use Root Mean Squared Log Error (RMSLE) as the evaluation metric to avoid any extra transformations. Can you guess why we use a log-scale?

[ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
pd.set_option('display.max_rows', None)
```

[ ]:
```
data = pd.read_csv('/content/train.csv')
data_test = pd.read_csv('/content/test.csv')
```

[ ]:
```
data.head()
```

[ ]:
```
   Id  MSSubClass MSZoning  LotFrontage  LotArea Street Alley LotShape  \
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

  LandContour Utilities  … PoolArea PoolQC Fence MiscFeature MiscVal MoSold  \
0         Lvl    AllPub  …        0    NaN   NaN         NaN       0      2
1         Lvl    AllPub  …        0    NaN   NaN         NaN       0      5
2         Lvl    AllPub  …        0    NaN   NaN         NaN       0      9
3         Lvl    AllPub  …        0    NaN   NaN         NaN       0      2
4         Lvl    AllPub  …        0    NaN   NaN         NaN       0     12

  YrSold  SaleType  SaleCondition  SalePrice
0   2008        WD         Normal     208500
1   2007        WD         Normal     181500
2   2008        WD         Normal     223500
3   2006        WD        Abnorml     140000
4   2008        WD         Normal     250000

[5 rows x 81 columns]
```

[ ]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1460 entries, 0 to 1459
Data columns (total 81 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Id             1460 non-null   int64
 1   MSSubClass     1460 non-null   int64
```

```
2   MSZoning       1460 non-null   object
3   LotFrontage    1201 non-null   float64
4   LotArea        1460 non-null   int64
5   Street         1460 non-null   object
6   Alley          91 non-null     object
7   LotShape       1460 non-null   object
8   LandContour    1460 non-null   object
9   Utilities      1460 non-null   object
10  LotConfig      1460 non-null   object
11  LandSlope      1460 non-null   object
12  Neighborhood   1460 non-null   object
13  Condition1     1460 non-null   object
14  Condition2     1460 non-null   object
15  BldgType       1460 non-null   object
16  HouseStyle     1460 non-null   object
17  OverallQual    1460 non-null   int64
18  OverallCond    1460 non-null   int64
19  YearBuilt      1460 non-null   int64
20  YearRemodAdd   1460 non-null   int64
21  RoofStyle      1460 non-null   object
22  RoofMatl       1460 non-null   object
23  Exterior1st    1460 non-null   object
24  Exterior2nd    1460 non-null   object
25  MasVnrType     1452 non-null   object
26  MasVnrArea     1452 non-null   float64
27  ExterQual      1460 non-null   object
28  ExterCond      1460 non-null   object
29  Foundation     1460 non-null   object
30  BsmtQual       1423 non-null   object
31  BsmtCond       1423 non-null   object
32  BsmtExposure   1422 non-null   object
33  BsmtFinType1   1423 non-null   object
34  BsmtFinSF1     1460 non-null   int64
35  BsmtFinType2   1422 non-null   object
36  BsmtFinSF2     1460 non-null   int64
37  BsmtUnfSF      1460 non-null   int64
38  TotalBsmtSF    1460 non-null   int64
39  Heating        1460 non-null   object
40  HeatingQC      1460 non-null   object
41  CentralAir     1460 non-null   object
42  Electrical     1459 non-null   object
43  1stFlrSF       1460 non-null   int64
44  2ndFlrSF       1460 non-null   int64
45  LowQualFinSF   1460 non-null   int64
46  GrLivArea      1460 non-null   int64
47  BsmtFullBath   1460 non-null   int64
48  BsmtHalfBath   1460 non-null   int64
49  FullBath       1460 non-null   int64
```

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

[ ]:
```
pd.set_option('display.max_rows', None)
data.isna().sum().sort_values(ascending=False)
```

[ ]:
```
PoolQC           1453
MiscFeature      1406
Alley            1369
Fence            1179
FireplaceQu       690
LotFrontage       259
GarageYrBlt        81
GarageCond         81
GarageType         81
GarageFinish       81
```

```
GarageQual         81
BsmtFinType2       38
BsmtExposure       38
BsmtQual           37
BsmtCond           37
BsmtFinType1       37
MasVnrArea          8
MasVnrType          8
Electrical          1
Id                  0
Functional          0
Fireplaces          0
KitchenQual         0
KitchenAbvGr        0
BedroomAbvGr        0
HalfBath            0
FullBath            0
BsmtHalfBath        0
TotRmsAbvGrd        0
GarageCars          0
GrLivArea           0
GarageArea          0
PavedDrive          0
WoodDeckSF          0
OpenPorchSF         0
EnclosedPorch       0
3SsnPorch           0
ScreenPorch         0
PoolArea            0
MiscVal             0
MoSold              0
YrSold              0
SaleType            0
SaleCondition       0
BsmtFullBath        0
HeatingQC           0
LowQualFinSF        0
LandSlope           0
OverallQual         0
HouseStyle          0
BldgType            0
Condition2          0
Condition1          0
Neighborhood        0
LotConfig           0
YearBuilt           0
Utilities           0
```

```
LandContour         0
LotShape            0
Street              0
LotArea             0
MSZoning            0
OverallCond         0
YearRemodAdd        0
2ndFlrSF            0
BsmtFinSF2          0
1stFlrSF            0
CentralAir          0
MSSubClass          0
Heating             0
TotalBsmtSF         0
BsmtUnfSF           0
```

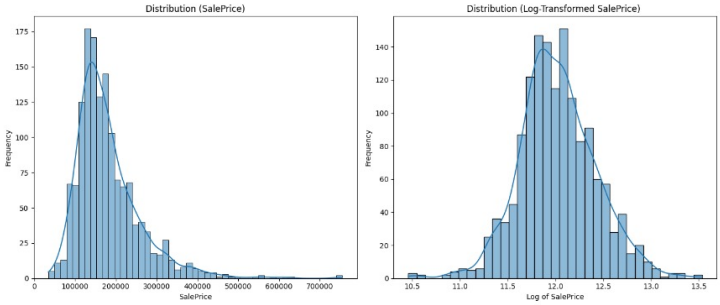
```
BsmtFinSF1          0
RoofStyle           0
Foundation          0
ExterCond           0
ExterQual           0
Exterior2nd         0
Exterior1st         0
RoofMatl            0
SalePrice           0
dtype: int64
```

[ ]:
```
# Plot (distribution of SalePrice)
plt.figure(figsize=(14, 6))

plt.subplot(1, 2, 1)
sns.histplot(data['SalePrice'], kde=True)
plt.title('Distribution (SalePrice)')
plt.xlabel('SalePrice')
plt.ylabel('Frequency')

# Plot (distribution of the log-transformed SalePrice)
plt.subplot(1, 2, 2)
log_sale_price = np.log(data['SalePrice'])
sns.histplot(log_sale_price, kde=True)
plt.title('Distribution (Log-Transformed SalePrice) ')
plt.xlabel('Log of SalePrice')
plt.ylabel('Frequency')

plt.tight_layout()
plt.show()
```



The distribution of SalePrice shows a right-skewed pattern, justifying the consideration for a log transformation to normalize the distribution and improve model performance.

Preprocessing Strategy Log Transformation:

To handle SalePrice's skewed distribution, apply a log transformation.

Handling Missing Values:

Based on the importance they have to SalePrice, characteristics with a high percentage of missing values (such as PoolQC, MiscFeature, Alley, Fence, and FireplaceQu) should be dropped, transformed (e.g., by specifying whether a feature is present or absent), or imputed. Imputation based on the feature's mean or median may be appropriate for numerical features such as LotFrontage. Missing data for features connected to garages and basements most likely indicate that these features are not there. It might be appropriate to create a binary indicator variable or impute "None" for categorical data or 0 for numerical data.

Depending on whether the feature is numerical or categorical, infer the most frequent value or median for features (MasVnrArea, MasVnrType, Electrical) that have extremely few missing values. Category Encoding: Depending on the needs of the model and the type of categorical data, either label encoding or one-hot encoding can be used to transform categorical variables into a numerical format.

Feature Engineering:

Possibility of adding additional features depending on the current variables (such as age of the house or total square footage).

[ ]:
```
#Log transformation of SalePrice feature
data['LogSalePrice'] = np.log(data['SalePrice'])
```

[ ]:
```
data['LogSalePrice'][:20]
```

[ ]:
```
0     12.247694
1     12.109011
2     12.317167
3     11.849398
4     12.429216
5     11.870600
6     12.634603
7     12.206073
8     11.774520
9     11.678440
10    11.771436
11    12.751300
12    11.877569
13    12.540758
14    11.964001
15    11.790557
16    11.911702
17    11.407565
18    11.976659
19    11.842229
Name: LogSalePrice, dtype: float64
```

Why Use a Log-Scale?

Normalization of Distribution: This distribution is made more symmetric and resembles a normal distribution, which is what many modeling techniques assume or want, by applying a log transformation.

Managing non-constant variance: This describes how non-constant variance appears across the data. As the average price of a home rises, it is typical to observe a wider range of prices. By stabilizing the variance over the range of prices, a log transformation enhances the validity of statistical assumptions and the performance of the model.

Why RMSE over RMSLE?

It is simple to use RMSE directly on the model's predictions (which are in the log space) if the model is trained on log-transformed target values.

When percentage mistakes rather than absolute differences are more important to you, RMSLE can be used to assess the relative disparities between the actual and projected values.

[ ]:
```
# For categorical features where NaN means the feature is absent
for col in ['PoolQC', 'MiscFeature', 'Alley', 'Fence',␣
 ↪'FireplaceQu','GarageType', 'GarageFinish', 'GarageQual', 'GarageCond',
            'BsmtQual', 'BsmtCond', 'BsmtExposure', 'BsmtFinType1',␣
 ↪'BsmtFinType2']:
    data[col].fillna('None', inplace=True)

for col in ['GarageYrBlt', 'GarageArea', 'GarageCars']:
    data[col].fillna(0, inplace=True)
```

```
data['MasVnrType'].fillna('None', inplace=True)
data['MasVnrArea'].fillna(0, inplace=True)

# For numerical features, considering 'LotFrontage', fill with the median
data['LotFrontage'].fillna(data['LotFrontage'].median(), inplace=True)

# For categorical feature, considering Electrical, since it's just one missing␣
↪value, fill with the mode
data['Electrical'].fillna(data['Electrical'].mode()[0], inplace=True)
```

Choosing to fill in the blanks for categorical features with a name such as 'None' is a strategic decision for multiple reasons, particularly when a significant percentage of those values (such as PoolQC, MiscFeature, Alley, Fence, and FireplaceQu) are missing.

1. **Indicating Absence:**

In many instances, missing values indicate the absence of an amenity rather than the data's unclear status, particularly when it comes to characteristics connected to facilities (such pools, alleyways, fences, and fireplaces). By appending 'None' to these missing values, we explicitly represent the feature's absence, which might serve as an important model signal. For example, while estimating the sale price of a house, the lack of a pool (PoolQC) may be significant information.

As a result, this method not only keeps information that could be important for estimating SalePrice but also makes the modeling process easier by guaranteeing that the dataset is fully populated and easier to understand.

[ ]:
```
data['PoolQC'].value_counts()
```

[ ]:
```
None    1453
Gd         3
Ex         2
Fa         2
Name: PoolQC, dtype: int64
```

[ ]:
```
data.isna().sum().sort_values(ascending=False)
```

[ ]:
```
Id              0
GarageCars      0
GarageYrBlt     0
GarageType      0
FireplaceQu     0
Fireplaces      0
Functional      0
TotRmsAbvGrd    0
KitchenQual     0
KitchenAbvGr    0
BedroomAbvGr    0
HalfBath        0
```

```
FullBath         0
BsmtHalfBath     0
BsmtFullBath     0
GrLivArea        0
LowQualFinSF     0
2ndFlrSF         0
1stFlrSF         0
GarageFinish     0
GarageArea       0
MSSubClass       0
GarageQual       0
SalePrice        0
SaleCondition    0
SaleType         0
YrSold           0
MoSold           0
MiscVal          0
MiscFeature      0
Fence            0
PoolQC           0
PoolArea         0
ScreenPorch      0
3SsnPorch        0
EnclosedPorch    0
OpenPorchSF      0
WoodDeckSF       0
PavedDrive       0
GarageCond       0
Electrical       0
CentralAir       0
HeatingQC        0
Heating          0
OverallCond      0
OverallQual      0
HouseStyle       0
BldgType         0
Condition2       0
Condition1       0
Neighborhood     0
LandSlope        0
LotConfig        0
Utilities        0
LandContour      0
LotShape         0
Alley            0
Street           0
LotArea          0
```

```
LotFrontage      0
MSZoning         0
YearBuilt        0
YearRemodAdd     0
RoofStyle        0
BsmtCond         0
TotalBsmtSF      0
BsmtUnfSF        0
BsmtFinSF2       0
BsmtFinType2     0
BsmtFinSF1       0
BsmtFinType1     0
BsmtExposure     0
BsmtQual         0
RoofMatl         0
Foundation       0
ExterCond        0
ExterQual        0
```

```
MasVnrArea       0
MasVnrType       0
Exterior2nd      0
Exterior1st      0
LogSalePrice     0
dtype: int64
```

[ ]:
```
# For converting categorical variables into numerical format,
# we will use one-hot encoding as it doesn't assume an ordinal relationship␣
↪unless the feature is ordinal
data = pd.get_dummies(data, drop_first=True)
```

[ ]:
```
# Feature Engineering

# Total Number of Bathrooms
data['TotalBathrooms'] = data['FullBath'] + (0.5 * data['HalfBath']) +␣
↪data['BsmtFullBath'] + (0.5 * data['BsmtHalfBath'])

# Years Since Remodel
data['YearsSinceRemodel'] = data['YrSold'] - data['YearRemodAdd']

# Total Square Foot
data['TotalSqFt'] = data['1stFlrSF'] + data['2ndFlrSF'] + data['TotalBsmtSF']

# Age of the house at the time of sale
data['HouseAge'] = data['YrSold'] - data['YearBuilt']
```

These features were selected because of their clear correlation with home selling prices, which makes them logical illustrations of how feature engineering can produce useful predictors from preexisting data. TotalSqFt and HouseAge are both based on fundamental characteristics of a property that

are well-known to affect its market value. They therefore present as excellent illustrations of how combining or altering attributes might yield fresh perspectives or more potent signals for a prediction model.

0.3 Q3. [0.4] Models

Read about ensemble models from Scikit-learn and select four models:

- A strong learner
- A weak learner
- A bagging model
- A boosting model

For each of the chosen models:

1. Motivate the choice of the model satisfies the data's requirements.
2. Perform hyperparameter tuning using cross-validation on the training set and compare the cross-validation performance of each of the four models. (Find more information about hyperparameter tuning here.)
3. Re-train using the complete train-set using the best hyperparameters found in the previous step. (This is often done automatically in Scikit-learn by setting `refit=True` of hyperparameter optimizers) and evaluate the performance on the test set.

not answered!

[ ]:
```
from sklearn.model_selection import train_test_split
import numpy as np

X = data.drop(['SalePrice', 'LogSalePrice'], axis=1)
y = data['LogSalePrice']

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)
```

[ ]:
```
# RANDOM FOREST - Strong Learner
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import GridSearchCV

param_grid_rf = {
```

```
    'n_estimators': [100, 200],
    'max_depth': [10, 20, None]
}

grid_search_rf = GridSearchCV(RandomForestRegressor(random_state=42),␣
 ↪param_grid_rf, cv=5, scoring='neg_mean_squared_error')
grid_search_rf.fit(X_train, y_train)
```

[ ]:
```
# DECISION TREE - Weak Learner
from sklearn.tree import DecisionTreeRegressor
```

12

```
param_grid_dt = {
    'max_depth': [3, 5, 10, None],
    'min_samples_leaf': [1, 2, 4]
}

grid_search_dt = GridSearchCV(DecisionTreeRegressor(random_state=42),␣
 ↪param_grid_dt, cv=5, scoring='neg_mean_squared_error')
grid_search_dt.fit(X_train, y_train)
```

[ ]:
```
# BAGGING WITH DECISION TREES
from sklearn.ensemble import BaggingRegressor

param_grid_bagging = {
    'n_estimators': [10, 50, 100],
    'base_estimator__max_depth': [3, 5, 10, None]
}

grid_search_bagging =␣
 ↪GridSearchCV(BaggingRegressor(DecisionTreeRegressor(random_state=42),␣
 ↪random_state=42), param_grid_bagging, cv=5, scoring='neg_mean_squared_error')
grid_search_bagging.fit(X_train, y_train)
```

[ ]:
```
# GRADIENT BOOSTING (BOOSTING MODEL)
from sklearn.ensemble import GradientBoostingRegressor

gridGradientBoosting = {
    'n_estimators': [100, 200],
    'learning_rate': [0.01, 0.1],
    'max_depth': [3, 5, 7]
}

grid_search_gb = GridSearchCV(GradientBoostingRegressor(random_state=42),␣
 ↪gridGradientBoosting, cv=5, scoring='neg_mean_squared_error')
grid_search_gb.fit(X_train, y_train)
```

not correct

[ ]:
```
print("Best Cross Validation RMSE Random Forest: {}".format(np.
 ↪sqrt(-grid_search_rf.best_score_)))
print("Best Cross Validation RMSE Decision Tree: {}".format(np.
 ↪sqrt(-grid_search_dt.best_score_)))
print("Best Cross Validation RMSE Bagging: {}".format(np.
 ↪sqrt(-grid_search_bagging.best_score_)))
print("Best Cross Validation RMSE Gradient Boosting: {}".format(np.
 ↪sqrt(-grid_search_gb.best_score_)))
```

```
Best Cross Validation RMSE Random Forest: 0.14452127169080475
Best Cross Validation RMSE Decision Tree: 0.19032039488836083
Best Cross Validation RMSE Bagging: 0.14491699549486056
```

```
Best Cross Validation RMSE Gradient Boosting: 0.13181436792524706
```

[ ]:
```
from sklearn.metrics import mean_squared_error

def modelEval(model, X_test, y_test):
    predictions = model.predict(X_test)
    mse = mean_squared_error(y_test, predictions)
    rmse = np.sqrt(mse)
    return rmse

# Evaluate each model
rmse_rf = modelEval(grid_search_rf.best_estimator_, X_test, y_test)
rmse_dt = modelEval(grid_search_dt.best_estimator_, X_test, y_test)
rmse_bagging = modelEval(grid_search_bagging.best_estimator_, X_test, y_test)
rmse_gb = modelEval(grid_search_gb.best_estimator_, X_test, y_test)

print("Random Forest RMSE: {}".format(rmse_rf))
print("Decision Tree RMSE: {}".format(rmse_dt))
print("Bagging RMSE: {}".format(rmse_bagging))
print("Gradient Boosting RMSE: {}".format(rmse_gb))
```

```
Random Forest RMSE: 0.14830727004163158
Decision Tree RMSE: 0.1881561560293881
Bagging RMSE: 0.14846140610137867
Gradient Boosting RMSE: 0.1369553117820338
```

[ ]:
```
# Re-evaluate on the test set using the best estimators
```

```python
# RANDOM FOREST
pred_rf = grid_search_rf.best_estimator_.predict(X_test)
rmse_rf = np.sqrt(mean_squared_error(y_test, pred_rf))

# DECISION TREE
pred_dt = grid_search_dt.best_estimator_.predict(X_test)
rmse_dt = np.sqrt(mean_squared_error(y_test, pred_dt))

# BAGGING WITH DECISION TREE
pred_bagging = grid_search_bagging.best_estimator_.predict(X_test)
rmse_bagging = np.sqrt(mean_squared_error(y_test, pred_bagging))

# GRADIENT BOOSTING
pred_gb = grid_search_gb.best_estimator_.predict(X_test)
rmse_gb = np.sqrt(mean_squared_error(y_test, pred_gb))

print("Test RMSE Random Forest: {}".format(rmse_rf))
print("Test RMSE Decision Tree: {}".format(rmse_dt))
print("Test RMSE Bagging: {}".format(rmse_bagging))
print("Test RMSE Gradient Boosting: {}".format(rmse_gb))
```



```
Test RMSE Random Forest: 0.14830727004163158
Test RMSE Decision Tree: 0.1881561560293881
Test RMSE Bagging: 0.14846140610137867
Test RMSE Gradient Boosting: 0.1369553117820338
```

Why RMSE over RMSLE?

If the model is trained on log-transformed target values, using RMSE directly on the model's predictions (which are in the log space) is straightforward.

RMSLE evaluates the relative differences between the predicted and actual values, making it useful when you care more about percentage errors than absolute differences.

0.4 Q4. [0.4] Bias and Variance

Variance: Captures how much your classifier changes if you train on a different training set.
Bias: Represents the inherent error obtained from your classifier even with infinite training data. This is due to your classifier being "biased" towards a particular kind of solution.
Find information about bias and variance below: - Bias and Variance by Scott Fortmann-Roe - CS4780 Lecture Note 12 - Learning curves

1. Plot the learning curve for each model by utilizing cross-validation. You can use the Scikit-learn learning_curve utility function for this. Setting `return_times=True` will be useful later in Q4.3.
2. Utilizing the plots from Q4.1, analyze the bias and variance of the models and answer the following questions:

a. Which model exhibits the best (worst) bias?

b. Which model exhibits the best (worst) variance?

c. Given this information, which model is preferable for SalePrice prediction and why?

d. What issues are each model experiencing—high bias, high variance, or both, etc.?

3. Analyze the running time for inference and training of each model. Using the code provided in the case-study tutorial, measure and visualize the running time of training and testing the models.

a. Which model is the fastest with one worker thread?

b. Which model is the fastest with 4 worker threads (using the `n_jobs` parameter of an estimator)?

[ ]:
```
from sklearn.model_selection import learning_curve
def plot_learning_curve_with_times(estimator, title, X, y, cv=5, n_jobs=-1,␣
↪train_sizes=np.linspace(.1, 1.0, 5)):
    train_sizes, train_scores, test_scores, fit_times, _ = learning_curve(
        estimator, X, y, cv=cv, n_jobs=n_jobs, train_sizes=train_sizes,
```



```
        return_times=True, scoring='neg_mean_squared_error')

    train_scores_mean = -np.mean(train_scores, axis=1)
    test_scores_mean = -np.mean(test_scores, axis=1)
    fit_times_mean = np.mean(fit_times, axis=1)

    # Plot learning curve
    _, axes = plt.subplots(1, 3, figsize=(20, 5))

    axes[0].plot(train_sizes, train_scores_mean, 'o-', color="r",␣
↪label="Training score")
    axes[0].plot(train_sizes, test_scores_mean, 'o-', color="g",␣
↪label="Cross-validation score")
    axes[0].set_title(title)
    axes[0].set_xlabel("Training examples")
    axes[0].set_ylabel("Score")
    axes[0].legend(loc="best")
    axes[0].grid(True)

    # Plot training times
    axes[1].plot(train_sizes, fit_times_mean, 'o-', color="b", label="Fit␣
↪times")
    axes[1].set_title(title + " (Scalability of the model)")
    axes[1].set_xlabel("Training examples")
    axes[1].set_ylabel("Fit Times (s)")
    axes[1].legend(loc="best")
    axes[1].grid(True)
```

```
    # Plot training times
    axes[2].plot(fit_times_mean, train_scores_mean, 'o-', color="y", label="Fit␣
 ↪times")
    axes[2].set_title(title + " (Performance of the model)")
    axes[2].set_xlabel("Fit Times (s)")
    axes[2].set_ylabel("Score")
    axes[2].legend(loc="best")
    axes[2].grid(True)

    plt.tight_layout()
    plt.show()

    return fit_times_mean
```

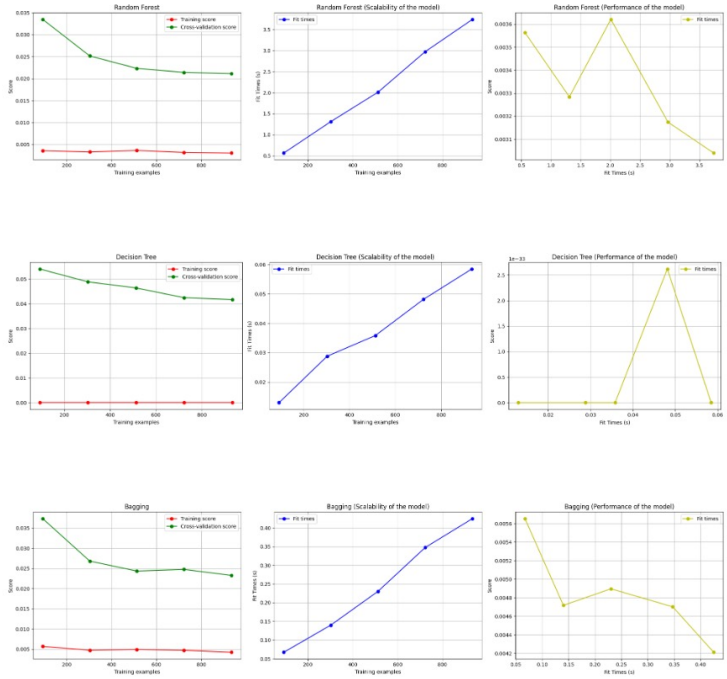
[ ]:
```
# Call the modified plotting function for each model
fit_times_rf = plot_learning_curve_with_times(RandomForestRegressor(n_jobs=1),␣
 ↪'Random Forest', X_train, y_train)
fit_times_dt = plot_learning_curve_with_times(DecisionTreeRegressor(),␣
 ↪'Decision Tree', X_train, y_train)
```

16

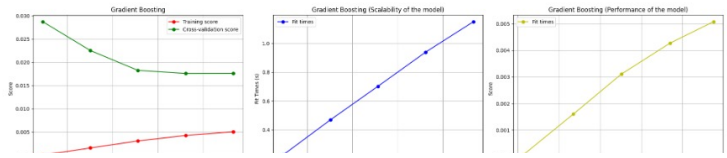
```
fit_times_bagging =␣
 ↪plot_learning_curve_with_times(BaggingRegressor(base_estimator=DecisionTreeRegressor(),␣
 ↪n_jobs=1), 'Bagging', X_train, y_train)
fit_times_gb = plot_learning_curve_with_times(GradientBoostingRegressor(),␣
 ↪'Gradient Boosting', X_train, y_train)

# Calculate the average training time for each model
avg_time_rf = np.mean(fit_times_rf)
avg_time_dt = np.mean(fit_times_dt)
avg_time_bagging = np.mean(fit_times_bagging)
avg_time_gb = np.mean(fit_times_gb)
```

```
# Determine the fastest model with one worker thread
fastest_model_single_thread = min(avg_time_rf, avg_time_dt, avg_time_bagging,␣
↪avg_time_gb)
```



17



We look at the "Scalability of the model" plots (the center one in each set), where the training times are plotted against the number of training examples, to see which model is the fastest with one worker thread based on the presented plots.

Since here is where the absolute differences in training times would be most noticeable, we can visually compare the fit times at the greatest number of training instances using the plots you have supplied.

With more training instances, the training time increases dramatically as seen by the Random Forest plot, which reaches over 3 seconds for the biggest training size. The Decision Tree plot displays a training time that is exceptionally low—just over 0.05 seconds—even for the biggest training size.

According to the Bagging plot, training time rises as the number of training examples does as well, peaking at 0.35 seconds for the greatest training size. The training time is depicted in the Gradient Boosting plot as it rises with the quantity of training instances, reaching 1.2 seconds for the highest training size.

These findings indicate that the Decision Tree architecture with a single worker thread is the fastest. As expected, given that individual decision trees train more quickly than ensemble models, it retains short training times across all training sizes.

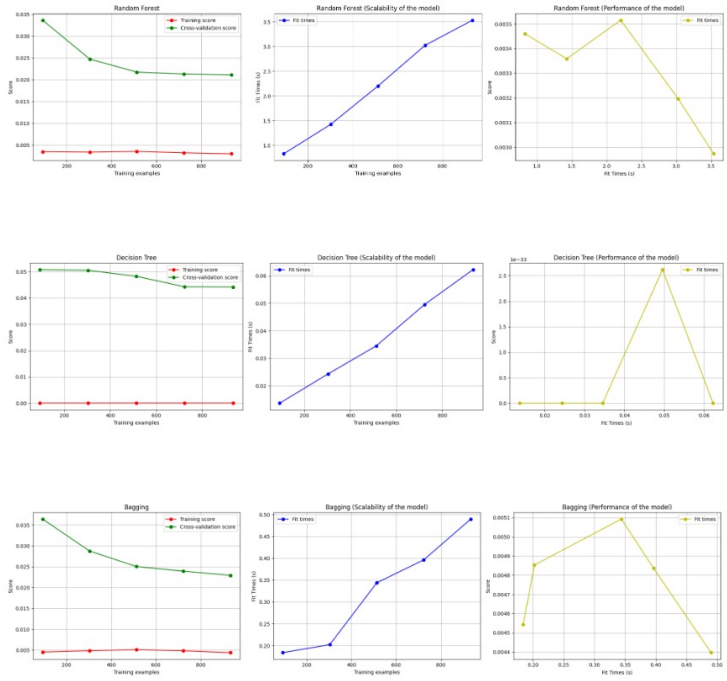
[ ]:
```
fastest_model_single_thread
```

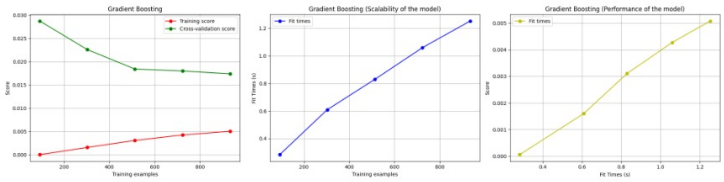
[ ]:
```
0.036844520568847655
```

[ ]:
```
# Call the modified plotting function for each model
fit_times_rf = plot_learning_curve_with_times(RandomForestRegressor(n_jobs=4),␣
↪'Random Forest', X_train, y_train)
fit_times_dt = plot_learning_curve_with_times(DecisionTreeRegressor(),␣
↪'Decision Tree', X_train, y_train)
fit_times_bagging =␣
↪plot_learning_curve_with_times(BaggingRegressor(base_estimator=DecisionTreeRegressor(),␣
↪n_jobs=4), 'Bagging', X_train, y_train)
fit_times_gb = plot_learning_curve_with_times(GradientBoostingRegressor(),␣
↪'Gradient Boosting', X_train, y_train)
```

```python
# Calculate the average training time for each model
avg_time_rf = np.mean(fit_times_rf)
avg_time_dt = np.mean(fit_times_dt)
avg_time_bagging = np.mean(fit_times_bagging)
avg_time_gb = np.mean(fit_times_gb)

# Determine the fastest model with one worker thread
fastest_model_single_thread = min(avg_time_rf, avg_time_dt, avg_time_bagging,␣
↪avg_time_gb)
```





We will examine the "Scalability of the model" plots, which display the fit times against the quantity of training samples, in order to ascertain which model runs the fastest when four worker threads are used. The fastest model will be the one that consistently displays the lowest fit times over the most training sample sizes.

Here are some observations derived from the charts you uploaded:

Random Forest: Displays fit times that get longer as more training examples are provided, up to roughly three seconds.

Decision Tree: Shows very low fit times, even with increasing training examples, with times just slightly above 0.1 seconds for the largest training size.

Bagging: Likewise, additional training instances result in longer fit times, which for the greatest training size approach approximately 0.35 seconds.

Gradient Boosting: When more training examples are used, fit times increase and can reach up to 1.4 seconds for the maximum training size.

These findings indicate that when four worker threads are used, the Decision Tree model performs the fastest since it has the lowest fit times.

[ ]:

S&D a-Adequate **3**

Clarity a-well organized, commented, readable **2**

Func b-Adequate **3.5**

Q2

0 / 0

Submit here your Python notebook (ipynb) as an attachment

Gajjala-Jahnavi_Qiu-Sichen_Kasibhatla-Venkata Sreenivas Prasad-Assignment-3.ipynb