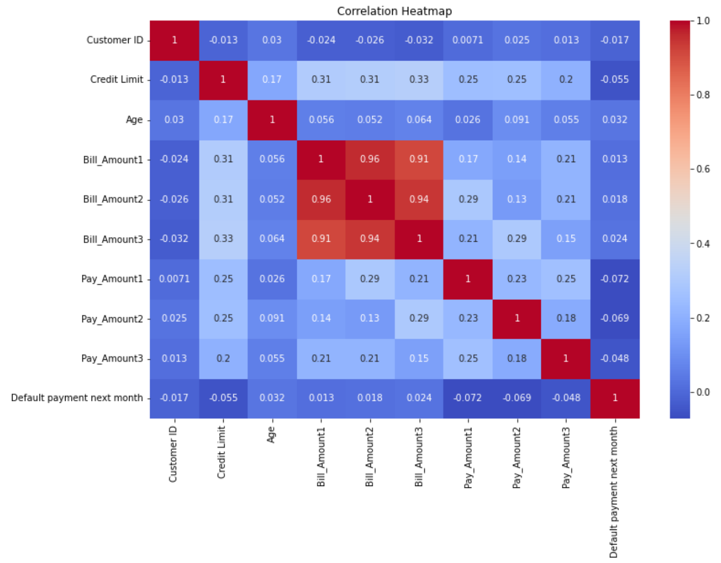
AIML CA1: Part A Classification

Name: Glenn Wu

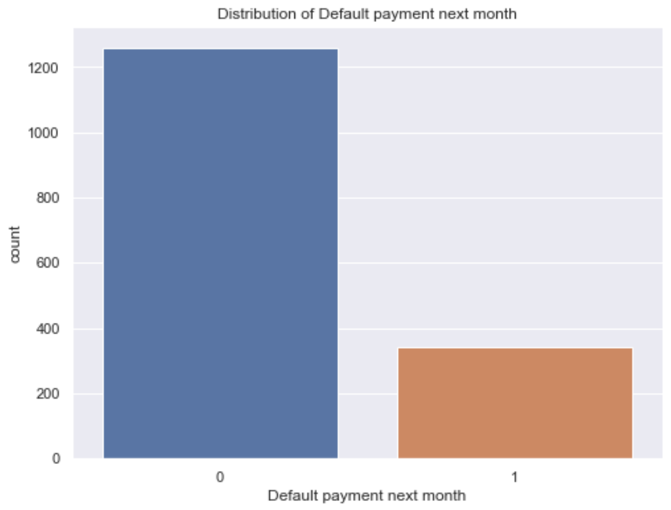
Admin No.: 2214395

Class: DAAA/FT/2A/01

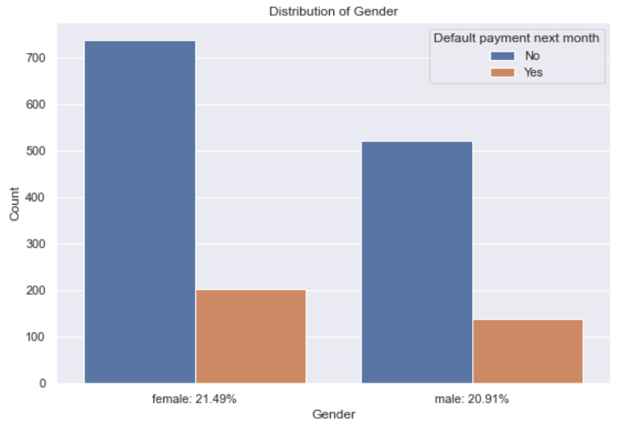
Data Exploration

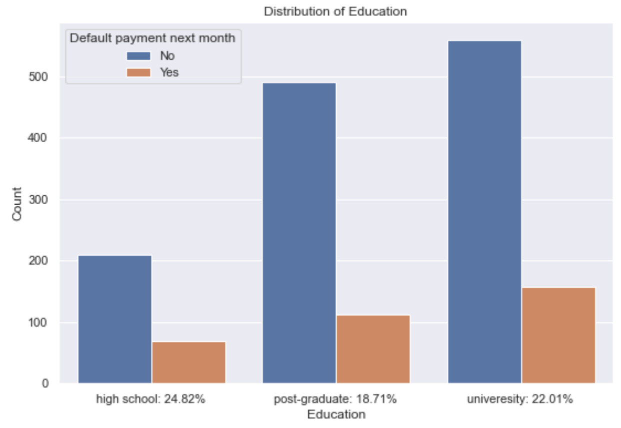


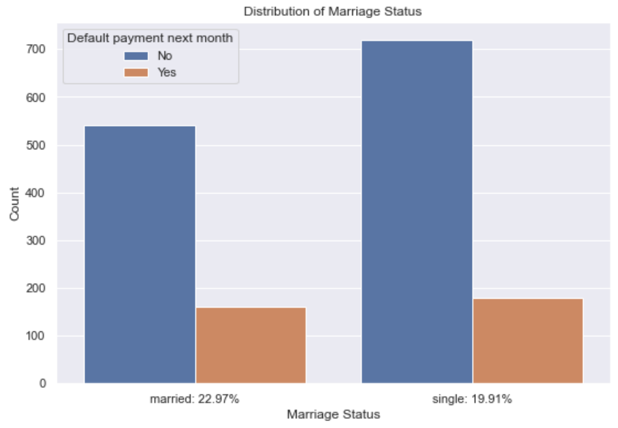
- The customer ID column is unique to each customer and independent from the rest of the columns, thus I will remove it.



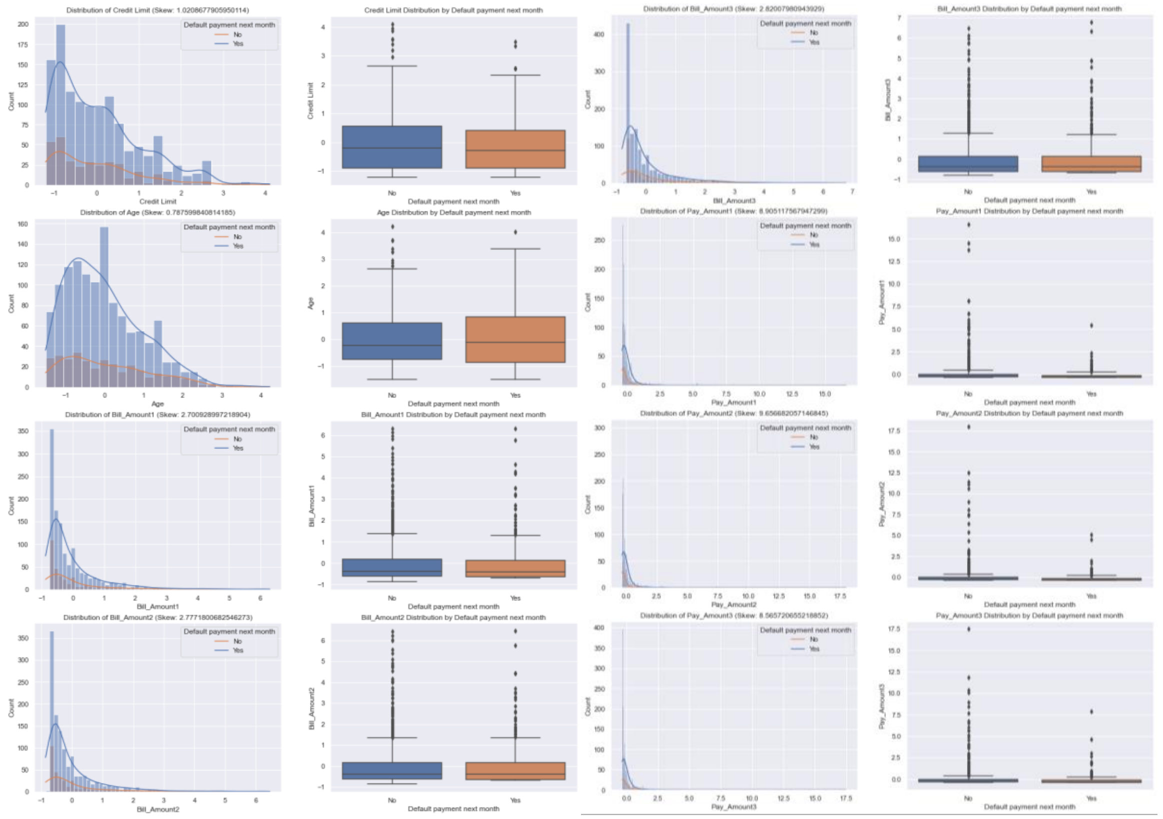
- We must <u>predict whether a customer will default given certain information</u>. There is an imbalance in the number of Positive (1) and Negative (0) classes.



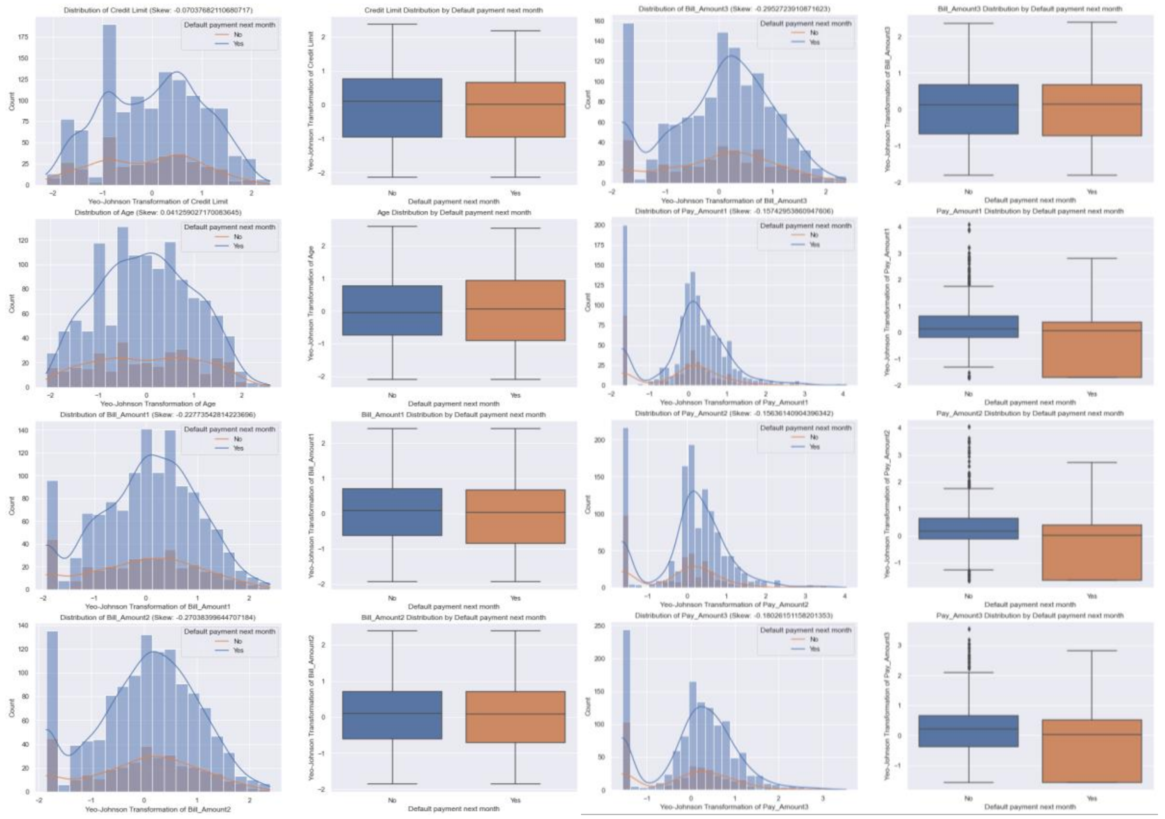




Data Exploration/Feature Engineering: Yeo-Johnson Transformation

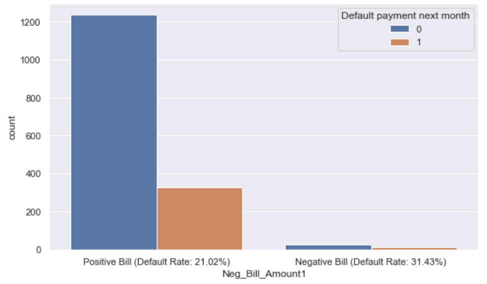


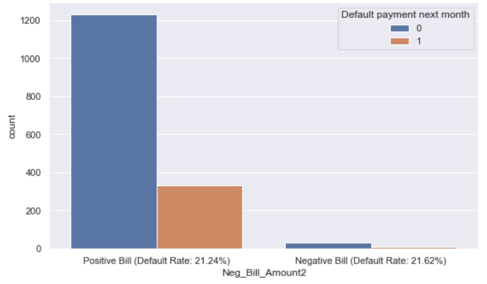
All the continuous numerical columns are highly skewed

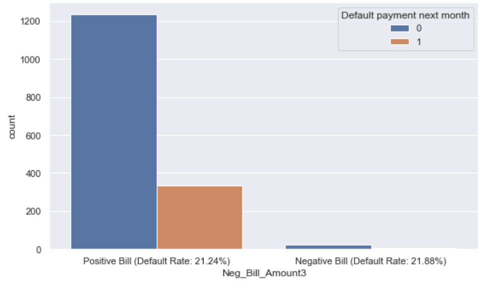


After applying the Yeo-Johnson Transformation they are un-skewed (-0.5 < skew < 0.5)

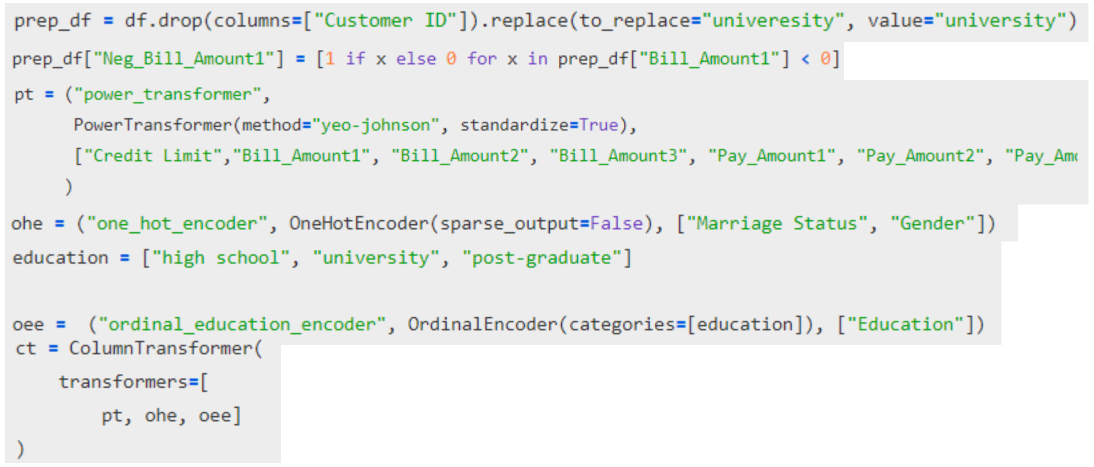
Data Exploration/Feature Engineering







- Some rows in the Bill_Amounts are less than zero.
- For Bill_Amount1, when the value is less than zero, there is a higher chance that the customer will default, 31.43% compared to 21.02% if they have a bill equal to or more than zero.
- Thus, I also One-Hot encode this column



```
prep_df = df.drop(columns=["Customer ID"]).replace(to_replace="univeresity", value="university")
prep_df["Neg_Bill_Amount1"] = [1 if x else 0 for x in prep_df["Bill_Amount1"] < 0]
pt = ("power_transformer",
      PowerTransformer(method="yeo-johnson", standardize=True),
      ["Credit Limit","Bill_Amount1", "Bill_Amount2", "Bill_Amount3", "Pay_Amount1", "Pay_Amount2", "Pay_Am
     )
ohe = ("one_hot_encoder", OneHotEncoder(sparse_output=False), ["Marriage Status", "Gender"])
education = ["high school", "university", "post-graduate"]

oee =  ("ordinal_education_encoder", OrdinalEncoder(categories=[education]), ["Education"])
ct = ColumnTransformer(
    transformers=[
        pt, ohe, oee]
)
```

Usage example with pipeline:

```
make_pipeline(ct, resampler, estimator)
```

Model Selection

- Scoring Metrics:
  - F1 Score: Harmonic Mean of Precision and Recall of the Positive Class

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

$$precision = \frac{TruePositive}{TruePositive + FalsePositive}$$

$$recall = \frac{TruePositive}{TruePositive + FalseNegative}$$

$$F_1 = \frac{2 \cdot TruePositive}{2 \cdot TruePositive + FalsePositive + FalseNegative}$$

  - Average Precision: The area under the precision recall curve.
  - Accuracy:

$$accuracy = \frac{TruePositive + TrueNegative}{TruePositive + TrueNegative + FalsePositive + FalseNegative}$$

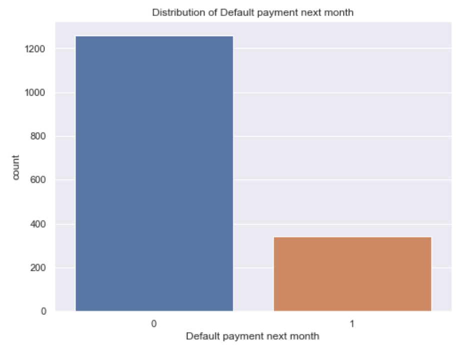
- Optimized Probability Threshold F1 Score

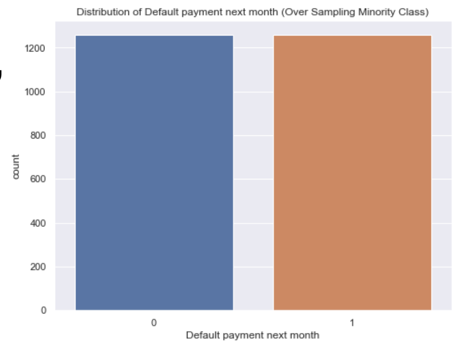
```python
def optimized_proba_f1_score(y_valid, proba, return_threshold=False):
    precision, recall, thresholds = precision_recall_curve(y_valid, proba)
    f1_scores = 2 * (precision * recall) / (precision + recall)
    f1_scores = np.nan_to_num(f1_scores)
    if return_threshold:
        return thresholds[f1_scores.argmax()]
    else:
        return f1_scores.max()
```

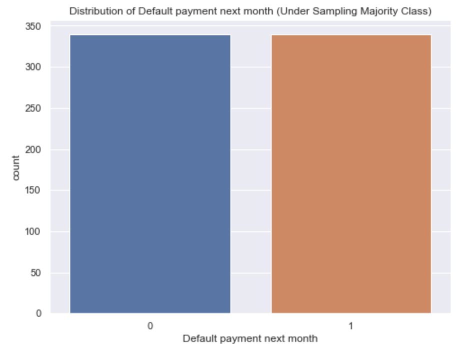
Predicted / Actual	**Negative (did not default)**	**Positive (defaulted)**
Negative (did not default)	True Negative	False Positive
Positive (defaulted)	False Negative	True Positive

Model Selection: Resampling

- Resampling Methods to compensate for Class Imbalance:
    - RandomOverSampler
    - RandomUnderSampler
    - SMOTE
    - SMOTENC
- RandomOverSampler, SMOTE, and SMOTENC are all over sampling methods that increase the number of values in the minority class to match the majority class
- RandomUnderSampler decreases the number of values in the majority class to match the minority class.

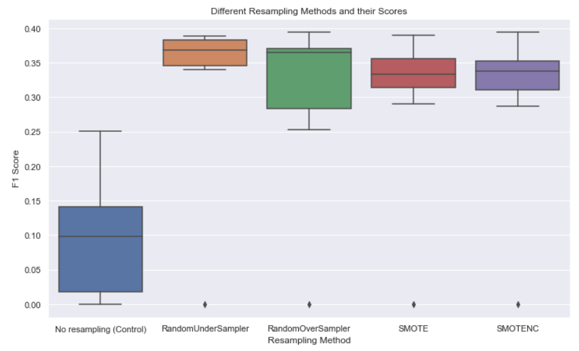


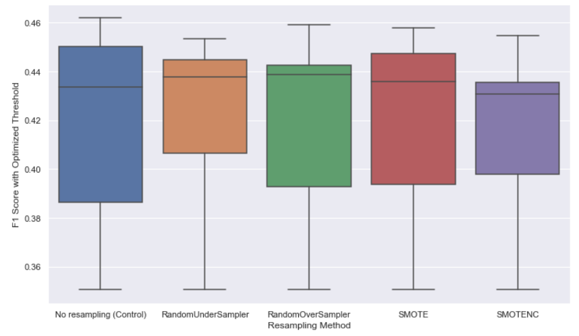


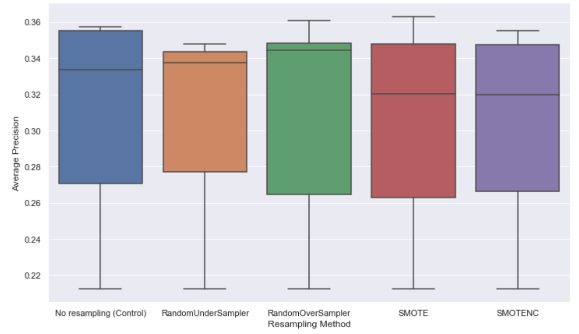


After comparing all 4 of these methods with the original data (no resampling) I conclude that **RandomOverSampler** is the best method since it has the highest median score for both threshold optimized F1 Score and Average Precision Score.

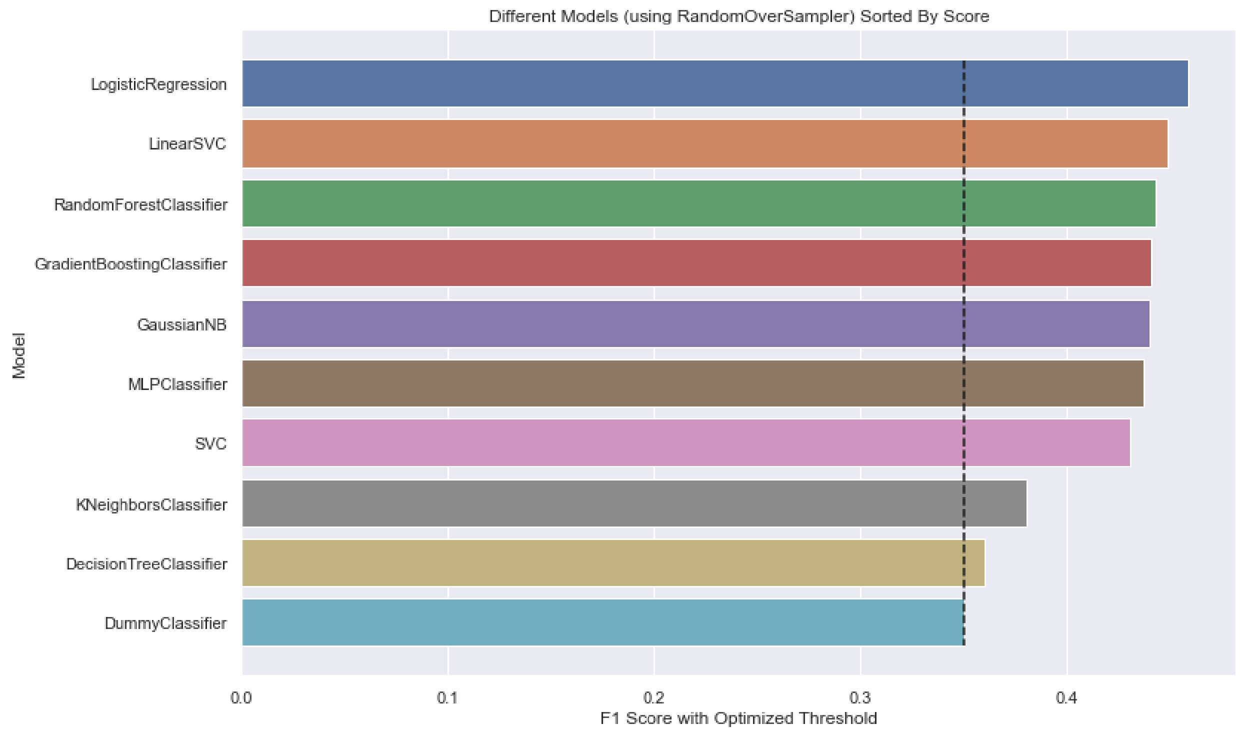
For all the evaluation I used **Stratified KFold Cross Validation**. This prevents overfitting of the model and ensures that the test set has the same distribution of target classes as the entire dataset



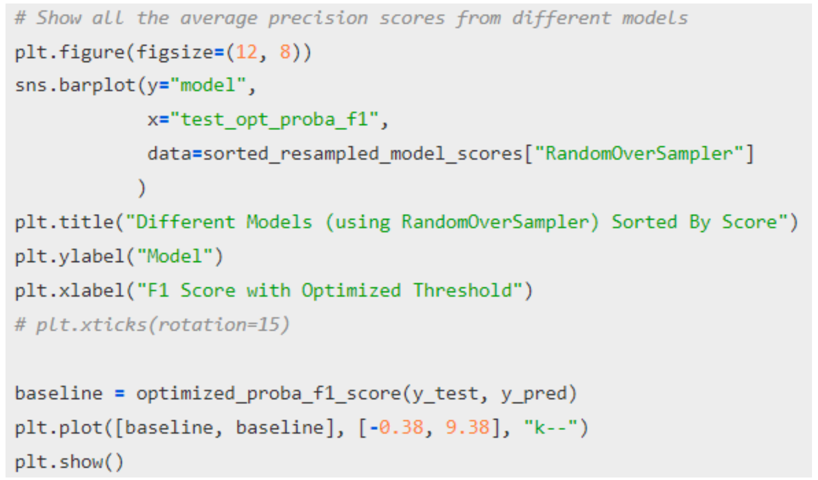




Model Selection



- All models perform better than DummyClassifier
- Best Performing Untuned Models: LogisticRegression, LinearSVC, RandomForestClassifier



```
# Show all the average precision scores from different models
plt.figure(figsize=(12, 8))
sns.barplot(y="model",
            x="test_opt_proba_f1",
            data=sorted_resampled_model_scores["RandomOverSampler"]
           )
plt.title("Different Models (using RandomOverSampler) Sorted By Score")
plt.ylabel("Model")
plt.xlabel("F1 Score with Optimized Threshold")
# plt.xticks(rotation=15)

baseline = optimized_proba_f1_score(y_test, y_pred)
plt.plot([baseline, baseline], [-0.38, 9.38], "k--")
plt.show()
```

```
From DummyClassifier:
0.3505154639175258

If all predications are zeros:
[[252   0]
 [ 68   0]]
0.0

If all predications are ones:
[[  0 252]
 [  0  68]]
0.3505154639175258
```

Model Improvement: Hyperparameter Tuning

GridSearchCV

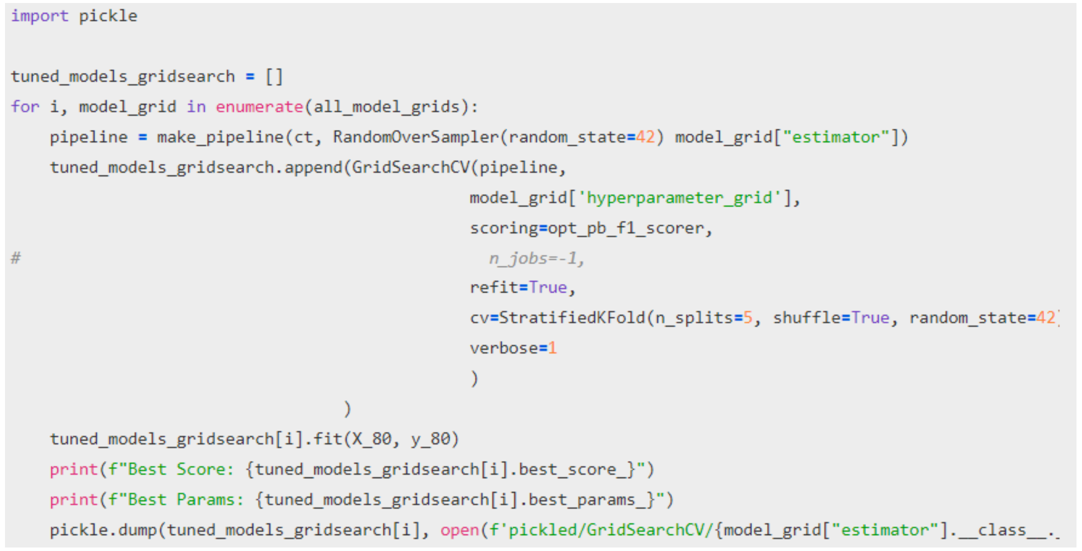
- GridSearchCV exhaustively searches all the specified hyperparameter combinations within a predefined grid.
- It evaluates and compares the performance of each combination and returns the best set of hyperparameters that maximize a chosen performance metric.
- It is a brute-force approach that considers all possible hyperparameter combinations within the defined grid, which can be computationally expensive for large search spaces.

Artificial Bee Colony Algorithm

- ABC is a nature-inspired metaheuristic optimization algorithm that mimics the foraging behavior of honeybees.
- In ABC hyperparameter tuning, the algorithm creates a population of candidate solutions (representing different sets of hyperparameters) and iteratively improves them.
- The algorithm uses the concept of employed bees, onlooker bees, and scout bees to explore and exploit the search space of hyperparameters.
- Does not rely on exhaustive grid search but rather employs a population-based search and exploration strategy to find good hyperparameter configurations.

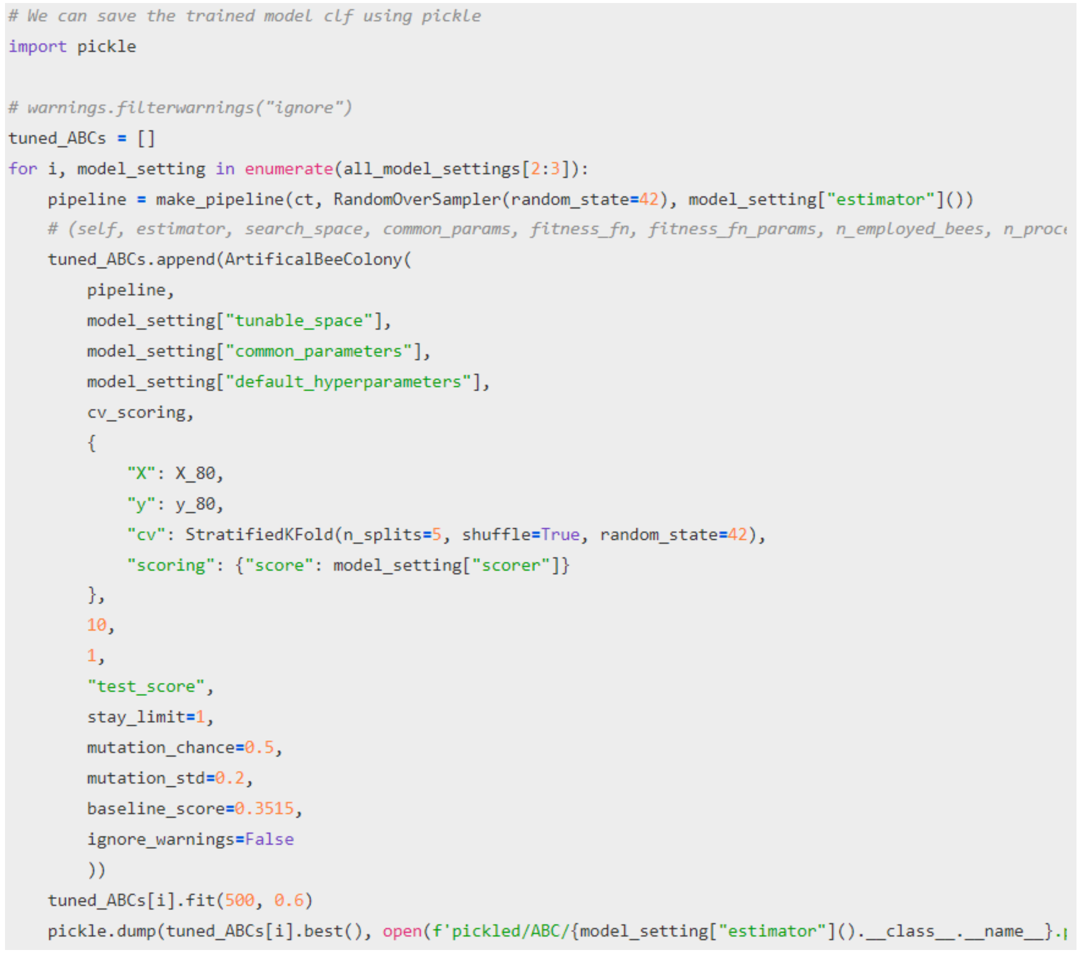
Model Improvement: Hyperparameter Tuning

GridSearchCV



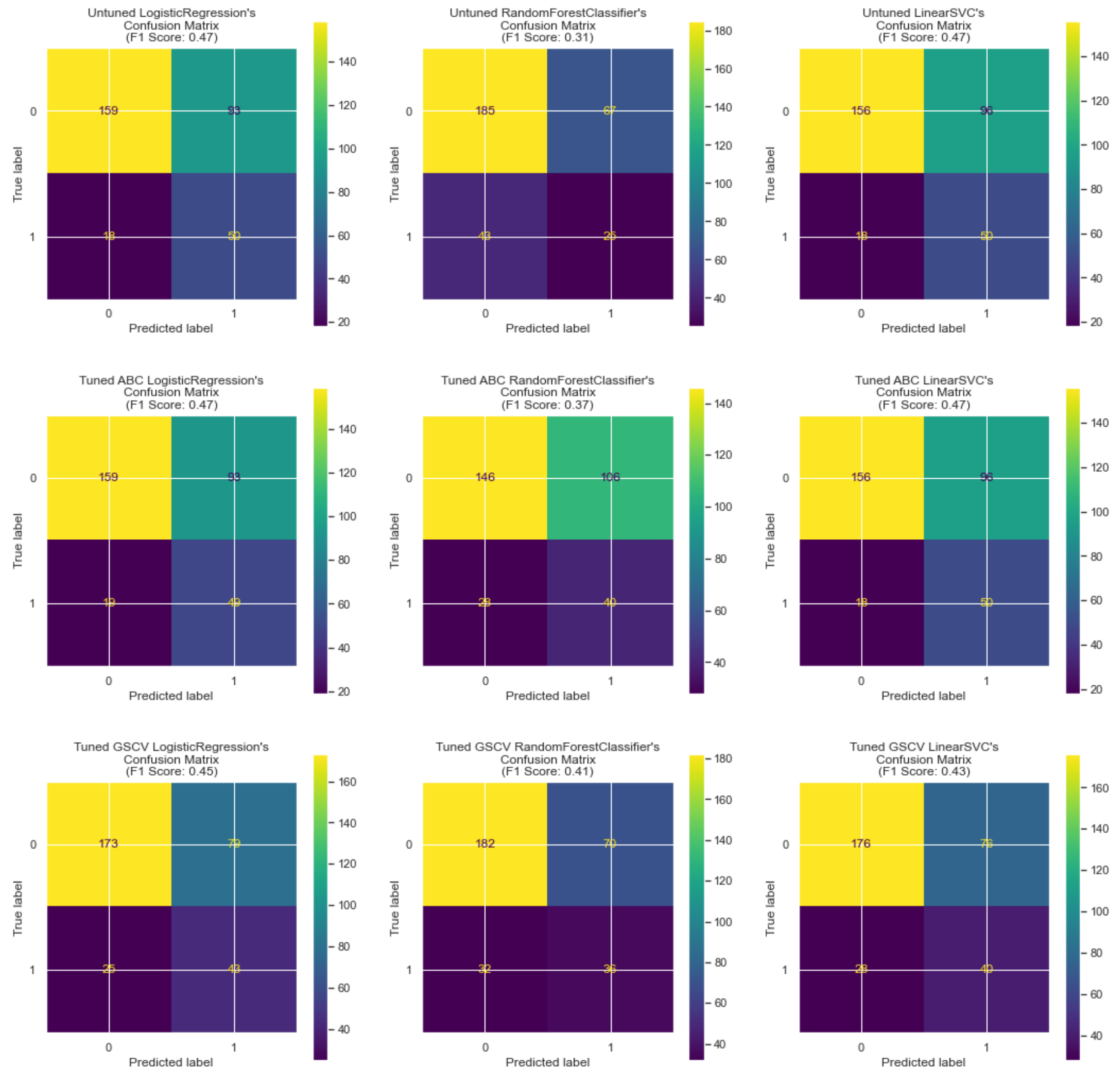
```
import pickle

tuned_models_gridsearch = []
for i, model_grid in enumerate(all_model_grids):
    pipeline = make_pipeline(ct, RandomOverSampler(random_state=42) model_grid["estimator"])
    tuned_models_gridsearch.append(GridSearchCV(pipeline,
                                                model_grid['hyperparameter_grid'],
                                                scoring=opt_pb_f1_scorer,
#                                                 n_jobs=-1,
                                                refit=True,
                                                cv=StratifiedKFold(n_splits=5, shuffle=True, random_state=42
                                                verbose=1
                                                )
                                   )
    tuned_models_gridsearch[i].fit(X_80, y_80)
    print(f"Best Score: {tuned_models_gridsearch[i].best_score_}")
    print(f"Best Params: {tuned_models_gridsearch[i].best_params_}")
    pickle.dump(tuned_models_gridsearch[i], open(f'pickled/GridSearchCV/{model_grid["estimator"].__class__._
```

Artificial Bee Colony Algorithm

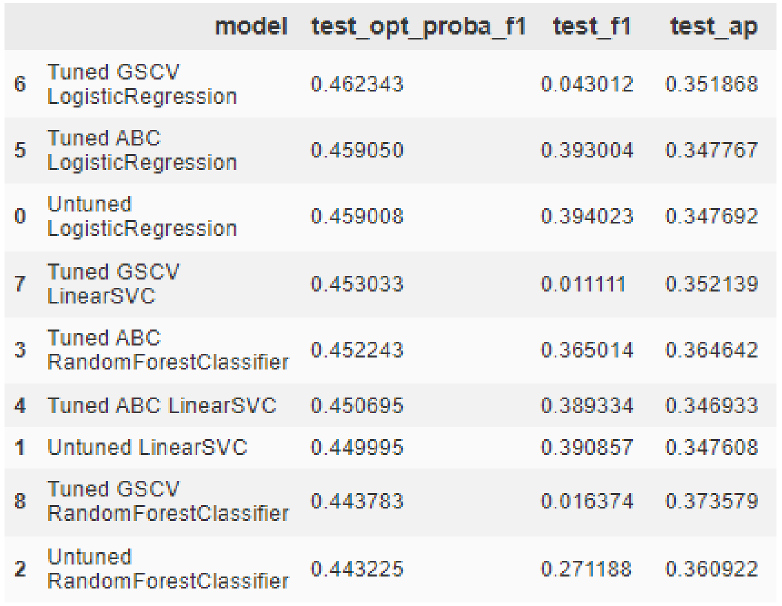


```
# We can save the trained model clf using pickle
import pickle

# warnings.filterwarnings("ignore")
tuned_ABCs = []
for i, model_setting in enumerate(all_model_settings[2:3]):
    pipeline = make_pipeline(ct, RandomOverSampler(random_state=42), model_setting["estimator"]())
    # (self, estimator, search_space, common_params, fitness_fn, fitness_fn_params, n_employed_bees, n_proc
    tuned_ABCs.append(ArtificalBeeColony(
        pipeline,
        model_setting["tunable_space"],
        model_setting["common_parameters"],
        model_setting["default_hyperparameters"],
        cv_scoring,
        {
            "X": X_80,
            "y": y_80,
            "cv": StratifiedKFold(n_splits=5, shuffle=True, random_state=42),
            "scoring": {"score": model_setting["scorer"]}
        },
        10,
        1,
        "test_score",
        stay_limit=1,
        mutation_chance=0.5,
        mutation_std=0.2,
        baseline_score=0.3515,
        ignore_warnings=False
        ))
    tuned_ABCs[i].fit(500, 0.6)
    pickle.dump(tuned_ABCs[i].best(), open(f'pickled/ABC/{model_setting["estimator"]().__class__.__name__}.
```

Model Improvement



Cross Validation Results:



	model	test_opt_proba_f1	test_f1	test_ap
6	Tuned GSCV LogisticRegression	0.462343	0.043012	0.351868
5	Tuned ABC LogisticRegression	0.459050	0.393004	0.347767
0	Untuned LogisticRegression	0.459008	0.394023	0.347692
7	Tuned GSCV LinearSVC	0.453033	0.011111	0.352139
3	Tuned ABC RandomForestClassifier	0.452243	0.365014	0.364642
4	Tuned ABC LinearSVC	0.450695	0.389334	0.346933
1	Untuned LinearSVC	0.449995	0.390857	0.347608
8	Tuned GSCV RandomForestClassifier	0.443783	0.016374	0.373579
2	Untuned RandomForestClassifier	0.443225	0.271188	0.360922

- Logistic Regression is the best performing overall model based on optimized F1 Score
- Hyperparameter Tuning only improved the score of RandomForestClassifier
- The optimized F1 Scores for the models tunes with GridSearchCV are generally higher than those from ArtificialBeeColony
- Best model according to cross-validation: LogisticRegression Tuned using GridSearchCV

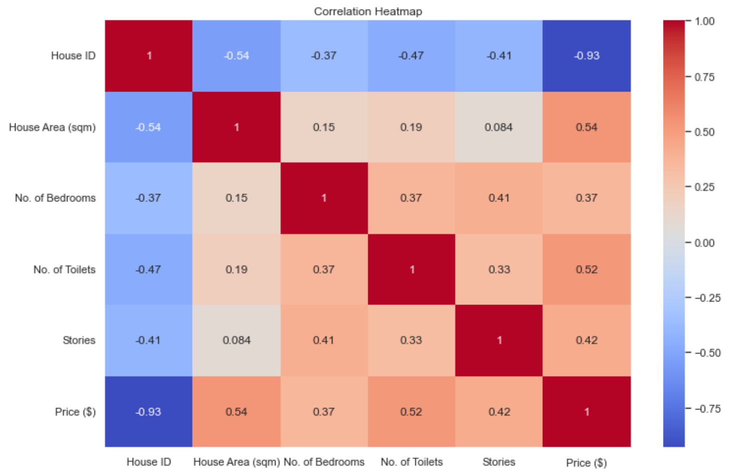
AIML CA1: Part B Classification

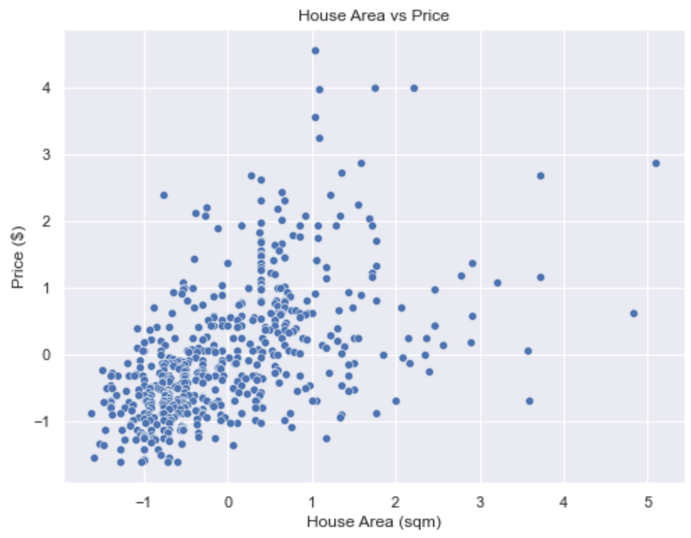
Name: Glenn Wu

Admin No.: 2214395

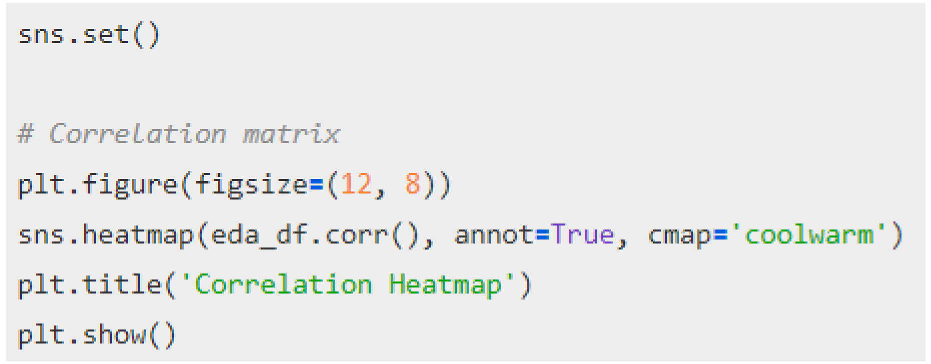
Class: DAAA/FT/2A/01

Data Exploration

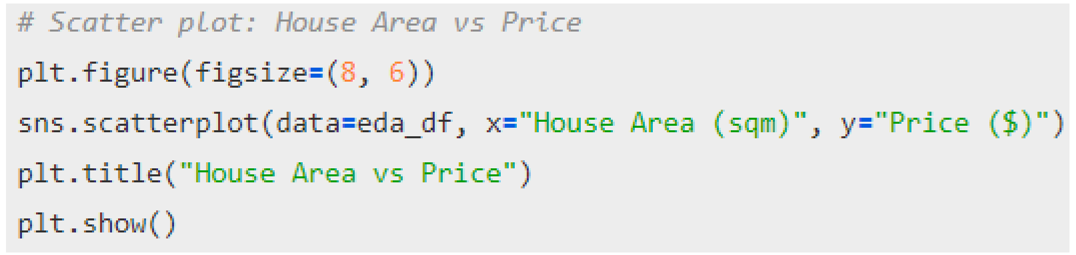




- The customer ID column is unique to each customer and independent from the rest of the columns, thus I will remove it.
- House Area and Price have a positive correlation as seen in the Correlation Matrix and in the Scatterplot



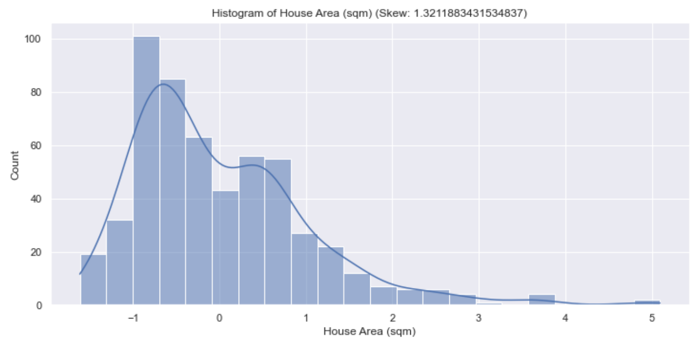
```
sns.set()


# Correlation matrix
plt.figure(figsize=(12, 8))
sns.heatmap(eda_df.corr(), annot=True, cmap='coolwarm')
plt.title('Correlation Heatmap')
plt.show()
```

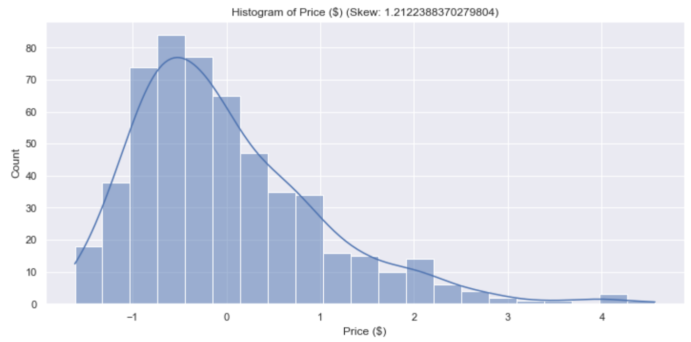


```
# Scatter plot: House Area vs Price
plt.figure(figsize=(8, 6))
sns.scatterplot(data=eda_df, x="House Area (sqm)", y="Price ($)")
plt.title("House Area vs Price")
plt.show()
```

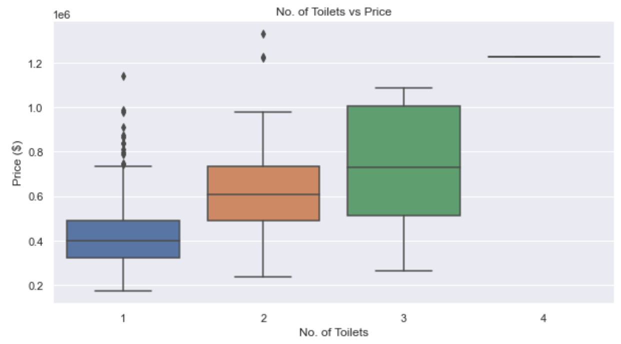
There is some correlation between all the features and the target “Price ($)” which we are trying to predict, as a regression problem.

Data Exploration

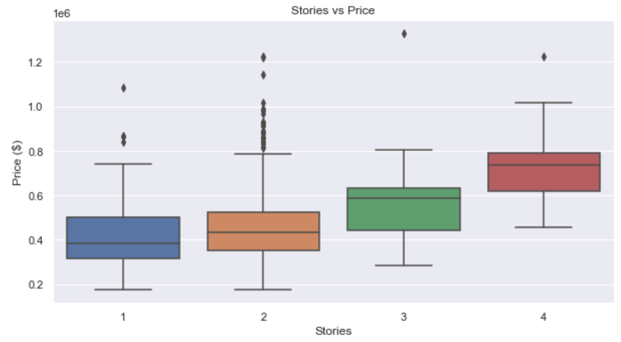


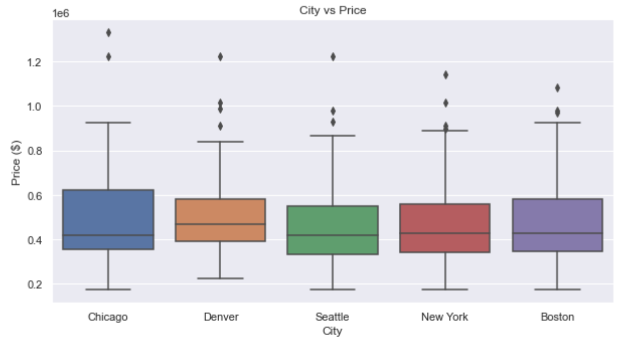


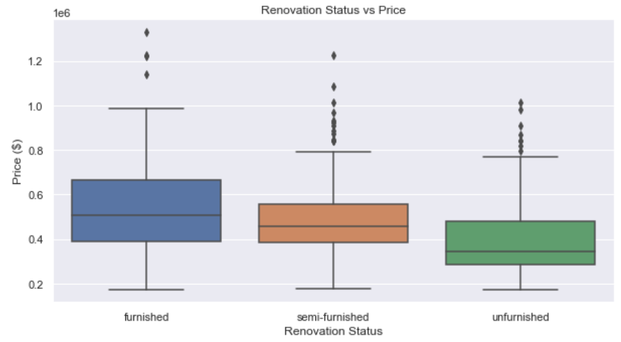
- Both House Area and Price are skewed to the right, with high skews. (>0.5)

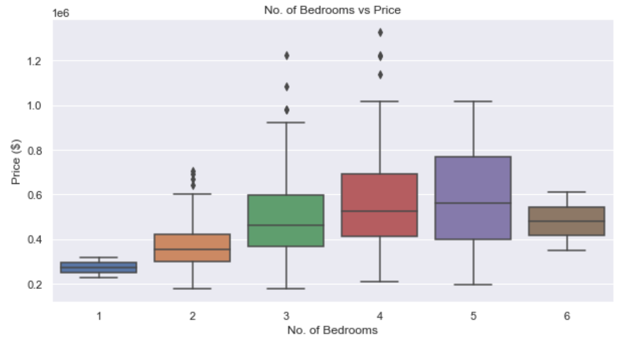


- Notable: Houses with 6 bedrooms are cheaper than houses with 4 bedrooms and more expense than houses with 3 bedrooms

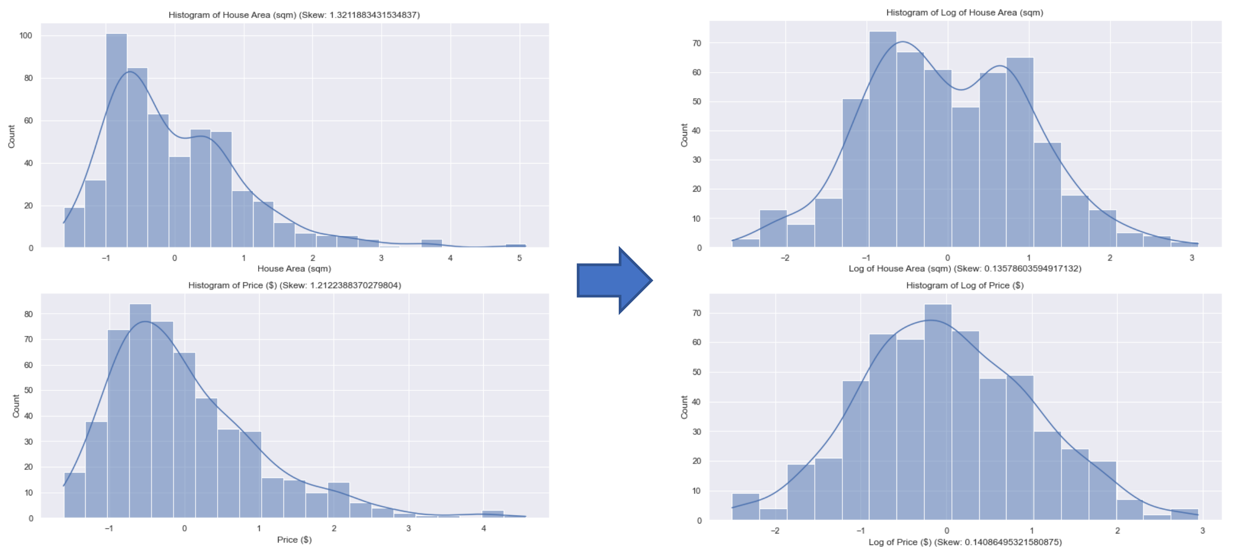




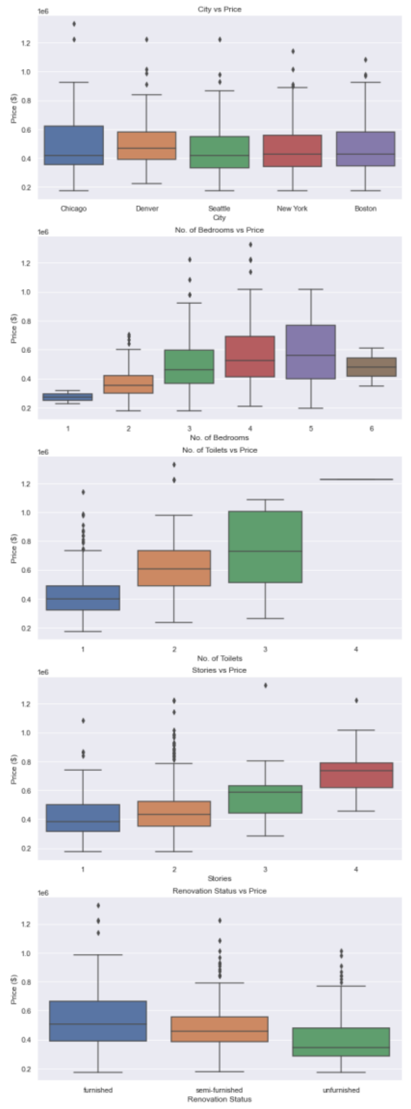




Feature Engineering



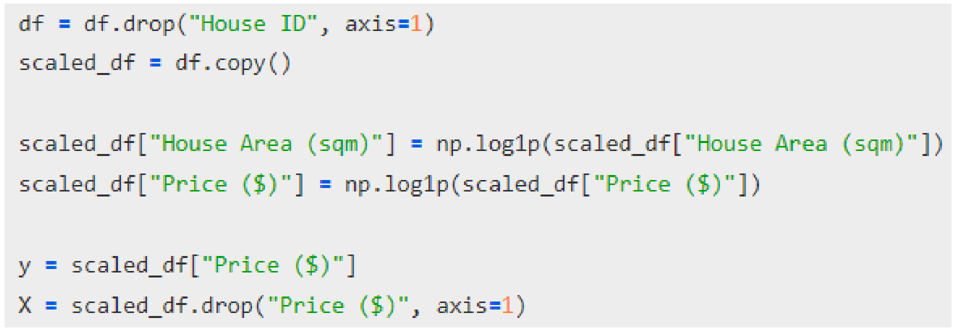
- Both House Area and Price are skewed to the right, with high skews. (>0.5)
- I use a Log transformation to unskew the data. (<0.5)
- I also use a standard scaler within the pipeline after the Log transformation.



- One Hot Encode “Cities”
- Ordinal encode “Renovation Status” since it seems that houses with more furnishing are generally priced higher. (unfurnished > semi-furnished > furnished)
- No. of Bedrooms, No. of Toilets, and Stories are already in an ordinal form, but I will encode them such that they start from 0 instead of 1.
- I change the order of No. of Bedrooms such that 1 > 2 > 3 > 6 > 4 > 5

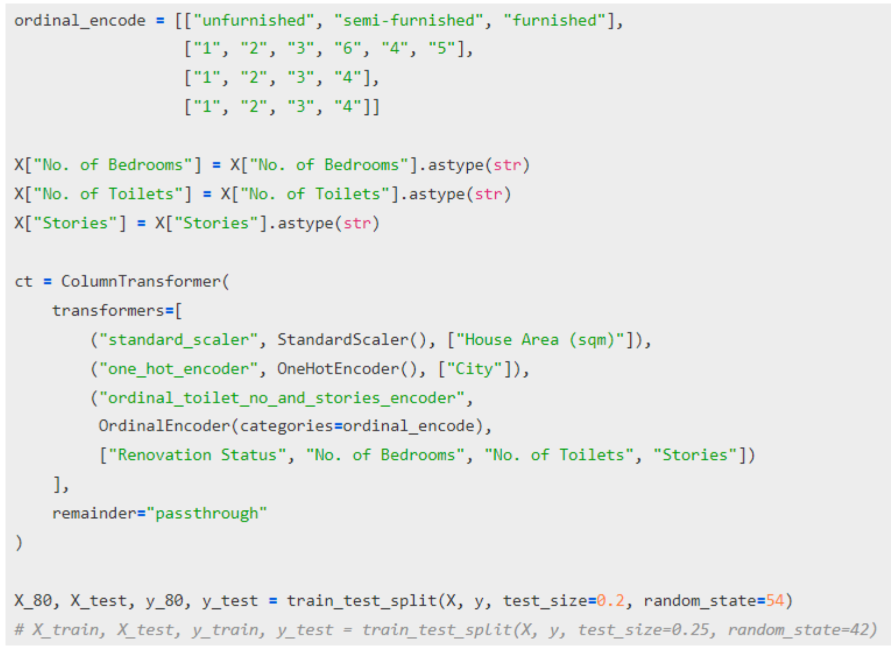
Feature Engineering

Log Transformation



```python
df = df.drop("House ID", axis=1)
scaled_df = df.copy()

scaled_df["House Area (sqm)"] = np.log1p(scaled_df["House Area (sqm)"])
scaled_df["Price ($)"] = np.log1p(scaled_df["Price ($)"])

y = scaled_df["Price ($)"]
X = scaled_df.drop("Price ($)", axis=1)
```

Preprocessing Steps



```python
ordinal_encode = [["unfurnished", "semi-furnished", "furnished"],
                  ["1", "2", "3", "6", "4", "5"],
                  ["1", "2", "3", "4"],
                  ["1", "2", "3", "4"]]

X["No. of Bedrooms"] = X["No. of Bedrooms"].astype(str)
X["No. of Toilets"] = X["No. of Toilets"].astype(str)
X["Stories"] = X["Stories"].astype(str)

ct = ColumnTransformer(
    transformers=[
        ("standard_scaler", StandardScaler(), ["House Area (sqm)"]),
        ("one_hot_encoder", OneHotEncoder(), ["City"]),
        ("ordinal_toilet_no_and_stories_encoder",
         OrdinalEncoder(categories=ordinal_encode),
         ["Renovation Status", "No. of Bedrooms", "No. of Toilets", "Stories"])
    ],
    remainder="passthrough"
)

X_80, X_test, y_80, y_test = train_test_split(X, y, test_size=0.2, random_state=54)
# X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42)
```

Model Selection

- Scoring Metrics:
  - R2: Coefficient of determination

$$R^2 = 1 - \frac{RSS}{TSS}$$

    - RSS: Sum of square of residuals

$$TSS = \sum_{i=1}^{n} (y_i - \bar{y})^2$$

    - TSS: Total sum of squares

$$RSS = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

  - Root Mean Square Error (RMSE):

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n} \|y_i - \hat{y}_i\|^2}{n}}$$

  - Mean Absolute Error (RMSE):

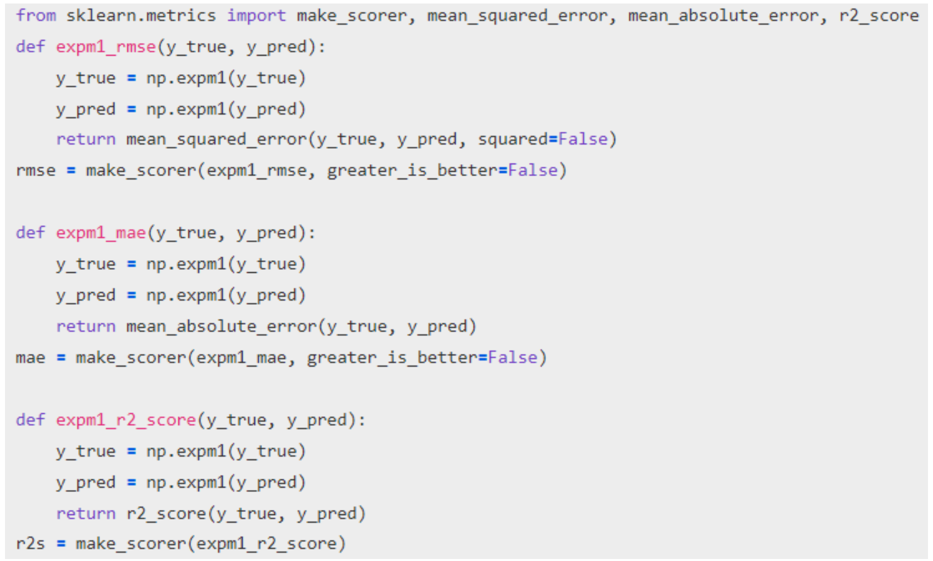
$$RMSE = \frac{\sum_{i=1}^{n} |y_i - \hat{y}_i|}{n}$$

$$n = \text{number of data points}$$

$$y_i = \text{i-th true value of y}$$

$$\hat{y}_i = \text{i-th predicted value of y}$$

RMSE is more sensitive to outliers than MAE which is important in our context to not neglect predictions that are very wrong.

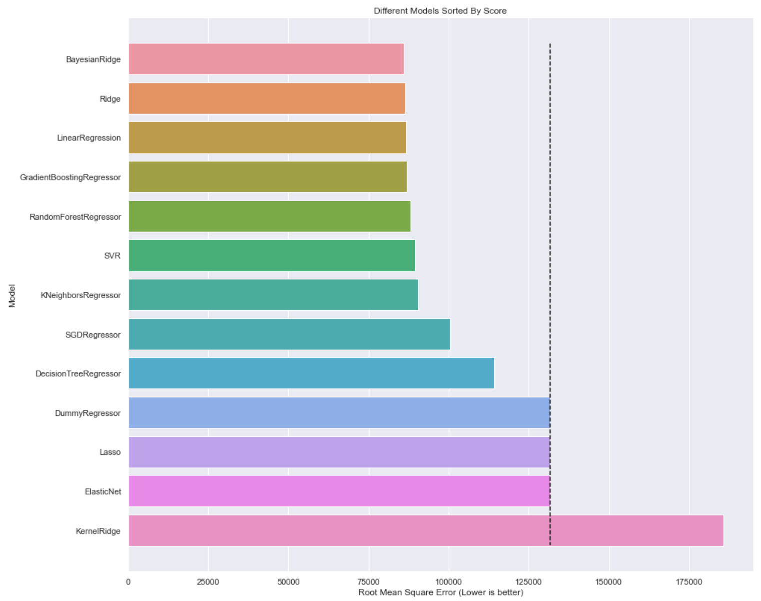


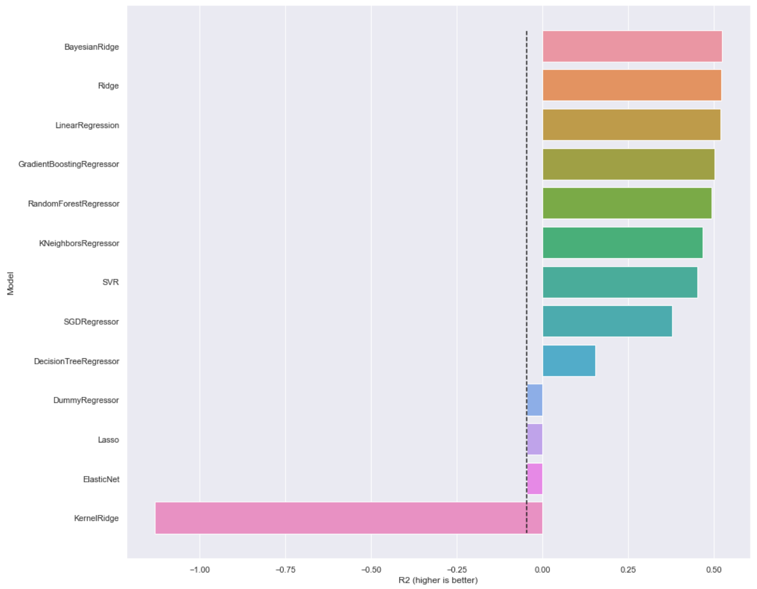
```
from sklearn.metrics import make_scorer, mean_squared_error, mean_absolute_error, r2_score
def expm1_rmse(y_true, y_pred):
    y_true = np.expm1(y_true)
    y_pred = np.expm1(y_pred)
    return mean_squared_error(y_true, y_pred, squared=False)
rmse = make_scorer(expm1_rmse, greater_is_better=False)

def expm1_mae(y_true, y_pred):
    y_true = np.expm1(y_true)
    y_pred = np.expm1(y_pred)
    return mean_absolute_error(y_true, y_pred)
mae = make_scorer(expm1_mae, greater_is_better=False)

def expm1_r2_score(y_true, y_pred):
    y_true = np.expm1(y_true)
    y_pred = np.expm1(y_pred)
    return r2_score(y_true, y_pred)
r2s = make_scorer(expm1_r2_score)
```

I have to create custom loss/score functions since “Price ($)” was Log transformed. So, to calculate the score I convert it back to the original form using exponents.

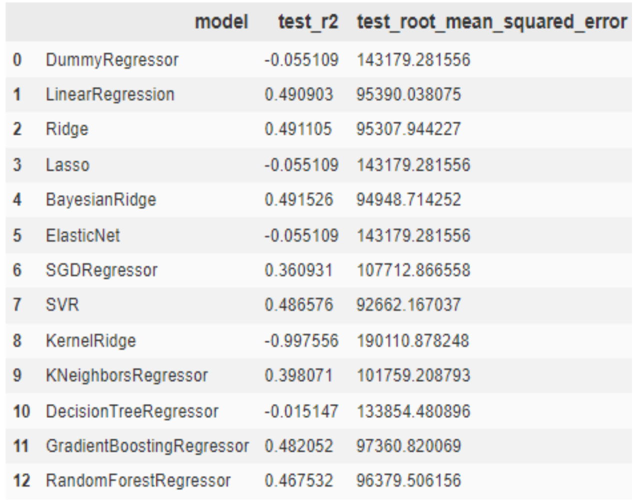
Model Selection

- Untuned Models Ranked by RMSE and R2 respectively:

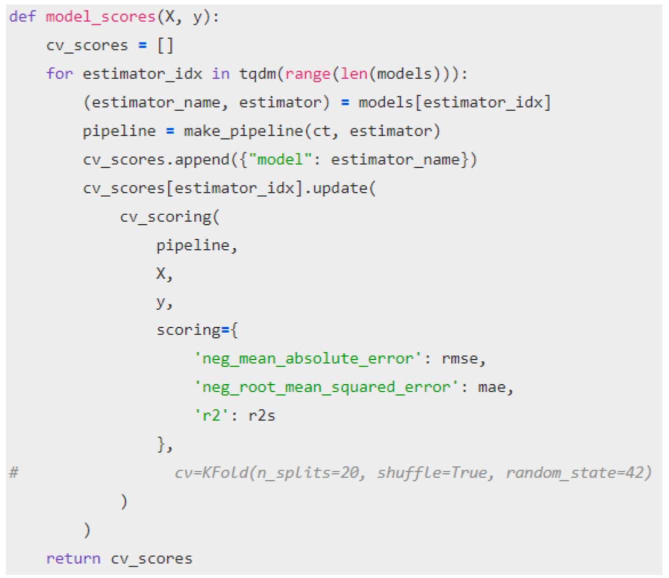




- Baseline R2 from DummyRegressor: -0.04650839316728548
- Baseline Root Mean Squared Error from DummyRegressor: 131734.69118270132
- Best Untuned Models: BayesianRidge, Ridge, LinearRegression

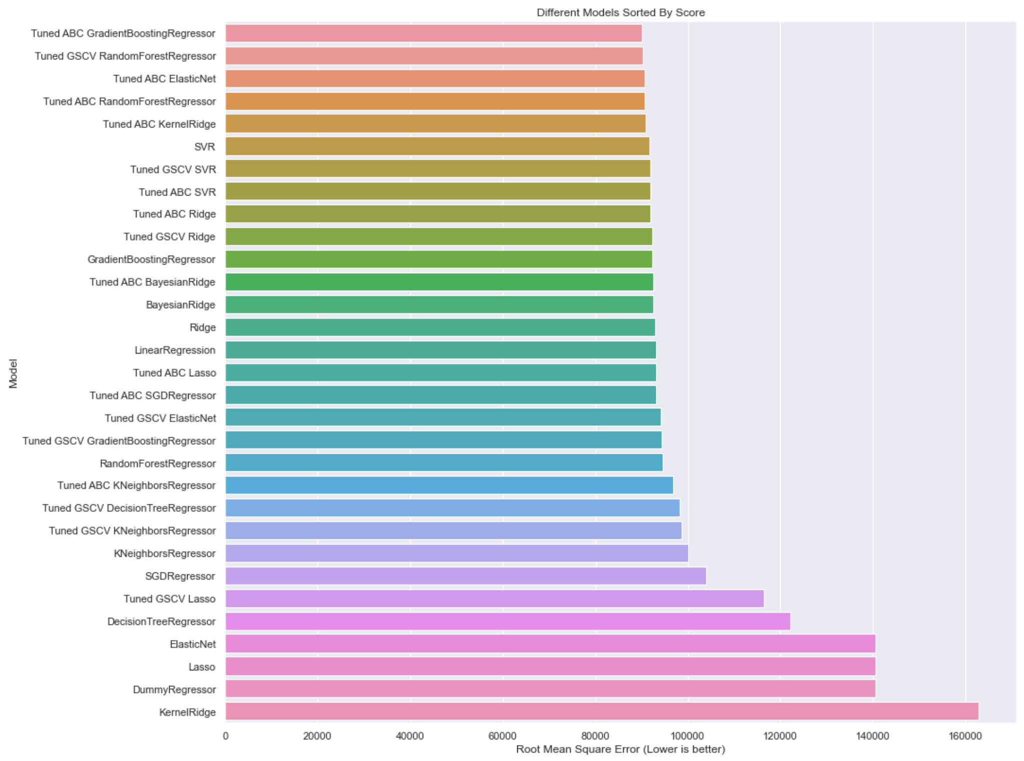


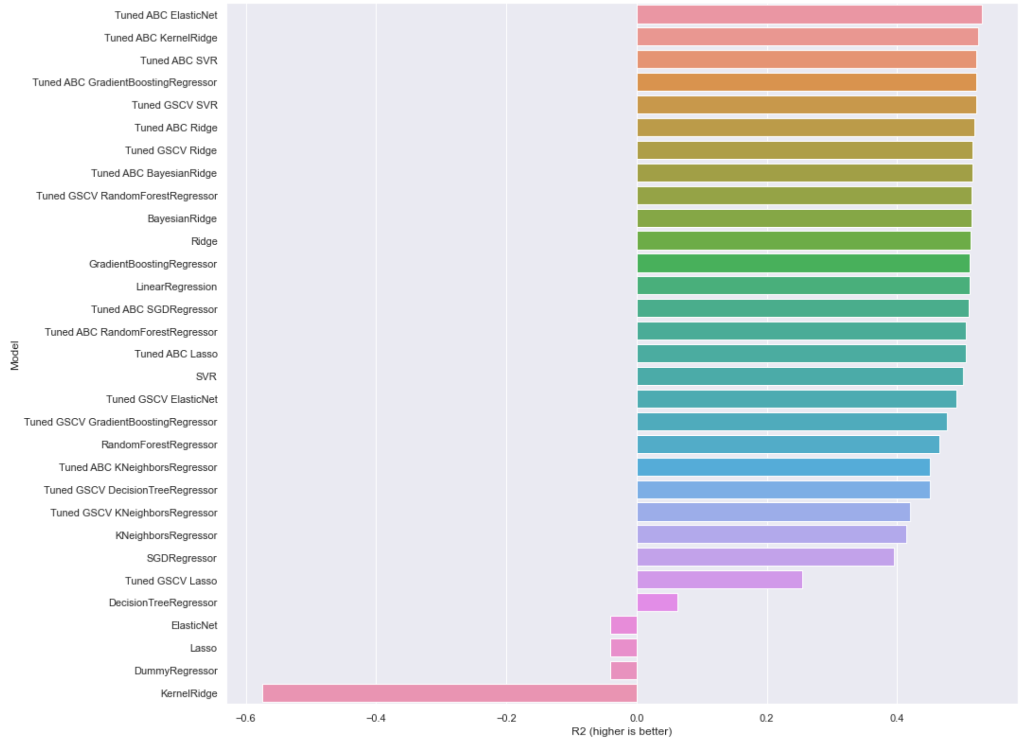
	model	test_r2	test_root_mean_squared_error
0	DummyRegressor	-0.055109	143179.281556
1	LinearRegression	0.490903	95390.038075
2	Ridge	0.491105	95307.944227
3	Lasso	-0.055109	143179.281556
4	BayesianRidge	0.491526	94948.714252
5	ElasticNet	-0.055109	143179.281556
6	SGDRegressor	0.360931	107712.866558
7	SVR	0.486576	92662.167037
8	KernelRidge	-0.997556	190110.878248
9	KNeighborsRegressor	0.398071	101759.208793
10	DecisionTreeRegressor	-0.015147	133854.480896
11	GradientBoostingRegressor	0.482052	97360.820069
12	RandomForestRegressor	0.467532	96379.506156



```python
def model_scores(X, y):
    cv_scores = []
    for estimator_idx in tqdm(range(len(models))):
        (estimator_name, estimator) = models[estimator_idx]
        pipeline = make_pipeline(ct, estimator)
        cv_scores.append({"model": estimator_name})
        cv_scores[estimator_idx].update(
            cv_scoring(
                pipeline,
                X,
                y,
                scoring={
                    'neg_mean_absolute_error': rmse,
                    'neg_root_mean_squared_error': mae,
                    'r2': r2s
                },
#                cv=KFold(n_splits=20, shuffle=True, random_state=42)
            )
        )
    return cv_scores
```

Model Improvement

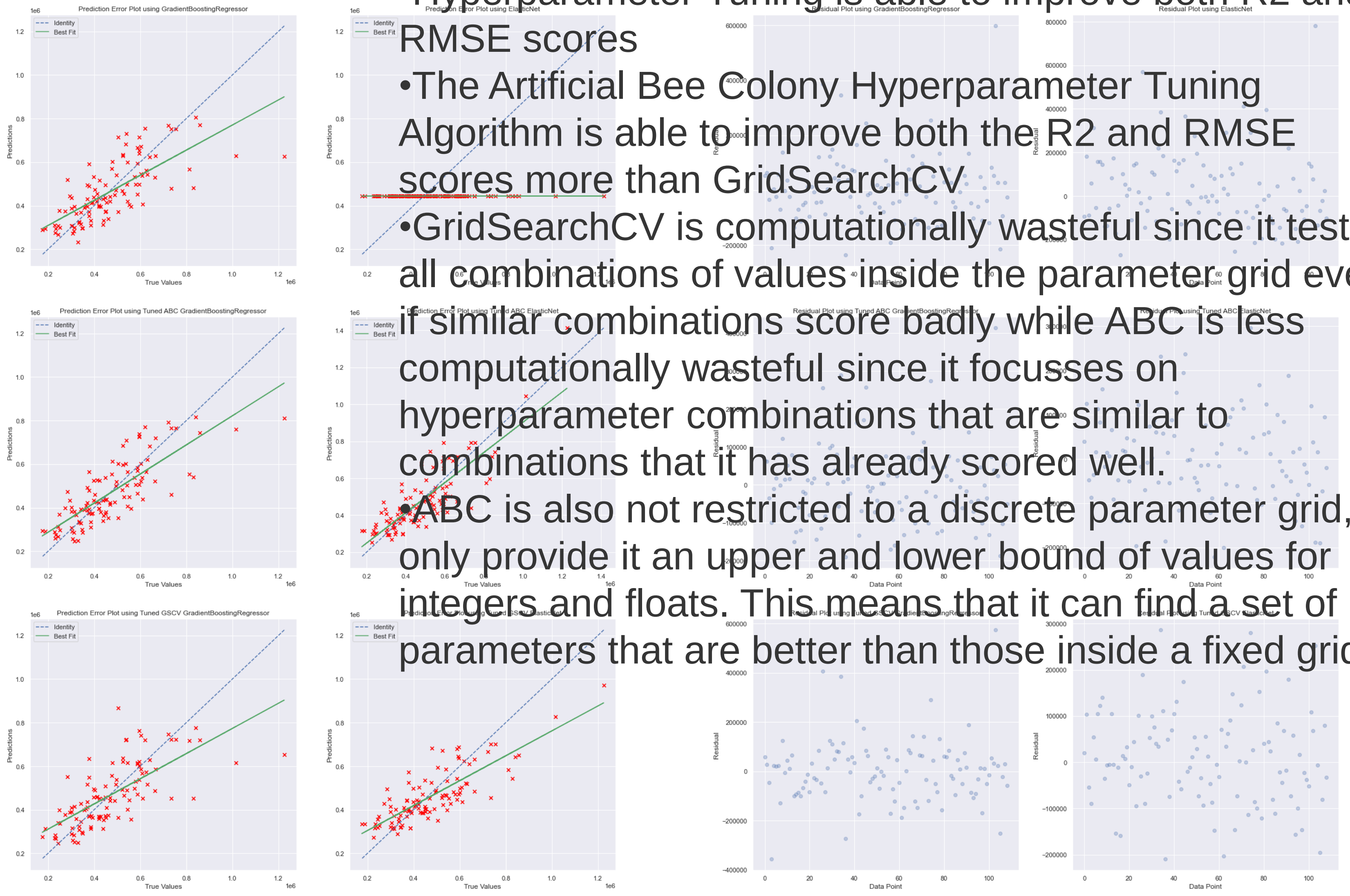




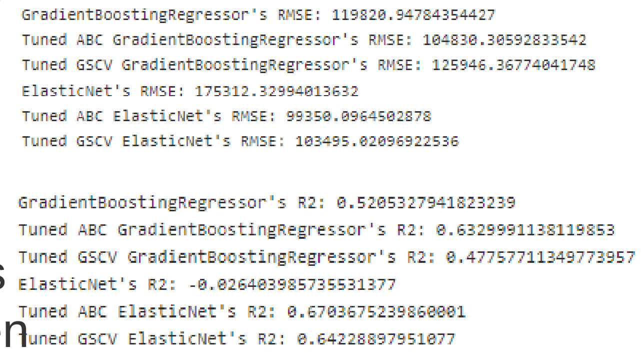
Models tuned with ABC generally score higher than those tuned with GridSearchCV, except for RandomForestRegressor

Best model according to R2 score: GradientBoostingRegressor Tuned using ABC
Best model according to RMSE: ElasticNet Tuned using ABC

Model Evaluation



- Hyperparameter Tuning is able to improve both R2 and RMSE scores
- The Artificial Bee Colony Hyperparameter Tuning Algorithm is able to improve both the R2 and RMSE scores more than GridSearchCV
- GridSearchCV is computationally wasteful since it tests all combinations of values inside the parameter grid even if similar combinations score badly while ABC is less computationally wasteful since it focusses on hyperparameter combinations that are similar to combinations that it has already scored well.
- ABC is also not restricted to a discrete parameter grid, I only provide it an upper and lower bound of values for integers and floats. This means that it can find a set of parameters that are better than those inside a fixed grid.



```
GradientBoostingRegressor's RMSE: 119820.94784354427
Tuned ABC GradientBoostingRegressor's RMSE: 104830.30592833542
Tuned GSCV GradientBoostingRegressor's RMSE: 125946.36774041748
ElasticNet's RMSE: 175312.32994013632
Tuned ABC ElasticNet's RMSE: 99350.0964502878
Tuned GSCV ElasticNet's RMSE: 103495.02096922536

GradientBoostingRegressor's R2: 0.5205327941823239
Tuned ABC GradientBoostingRegressor's R2: 0.6329991138119853
Tuned GSCV GradientBoostingRegressor's R2: 0.47757711349773957
ElasticNet's R2: -0.026403985735531377
Tuned ABC ElasticNet's R2: 0.6703675239860001
Tuned GSCV ElasticNet's R2: 0.64228897951077
```

Tuned ABC ElasticNet is the best performing based on both R2 score and RMSE. The Prediction Error Plot is very close to the ideal.

Conclusions

- Hyperparameter Tuning is able to improve both R2 and RMSE scores
- The Artificial Bee Colony Hyperparameter Tuning Algorithm is able to improve both the R2 and RMSE scores more than GridSearchCV
- GridSearchCV is computationally wasteful since it tests all combinations of values inside the parameter grid even if similar combinations score badly while ABC is less computationally wasteful since it focusses on hyperparameter combinations that are similar to combinations that it has already scored well.
- ABC is also not restricted to a discrete parameter grid, I only provide it an upper and lower bound of values for integers and floats. This means that it can find a set of parameters that are better than those inside a fixed grid like GridSearchCV requires.