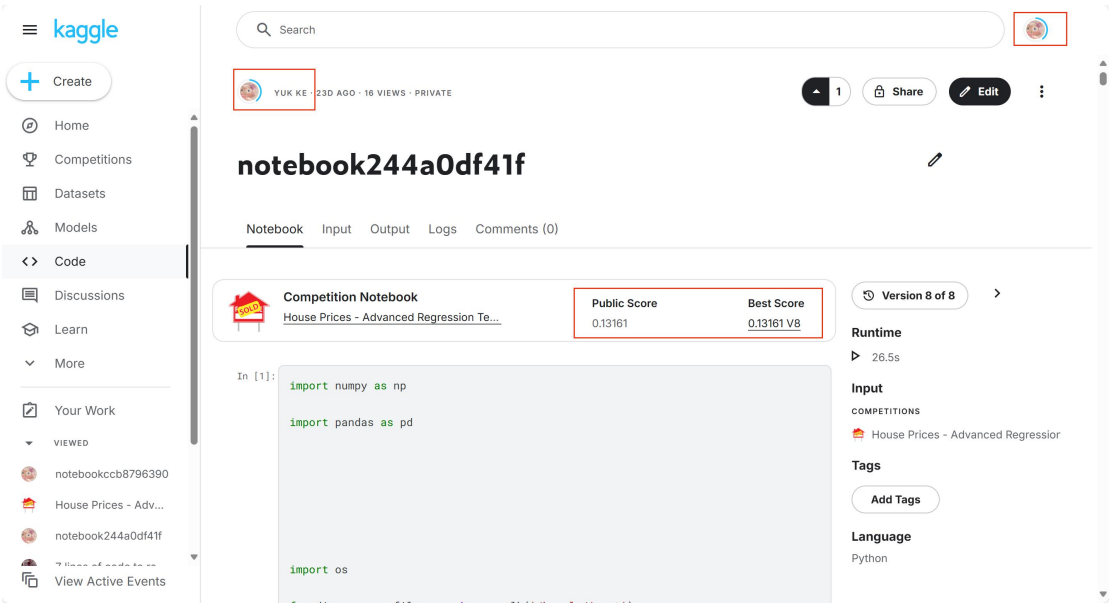
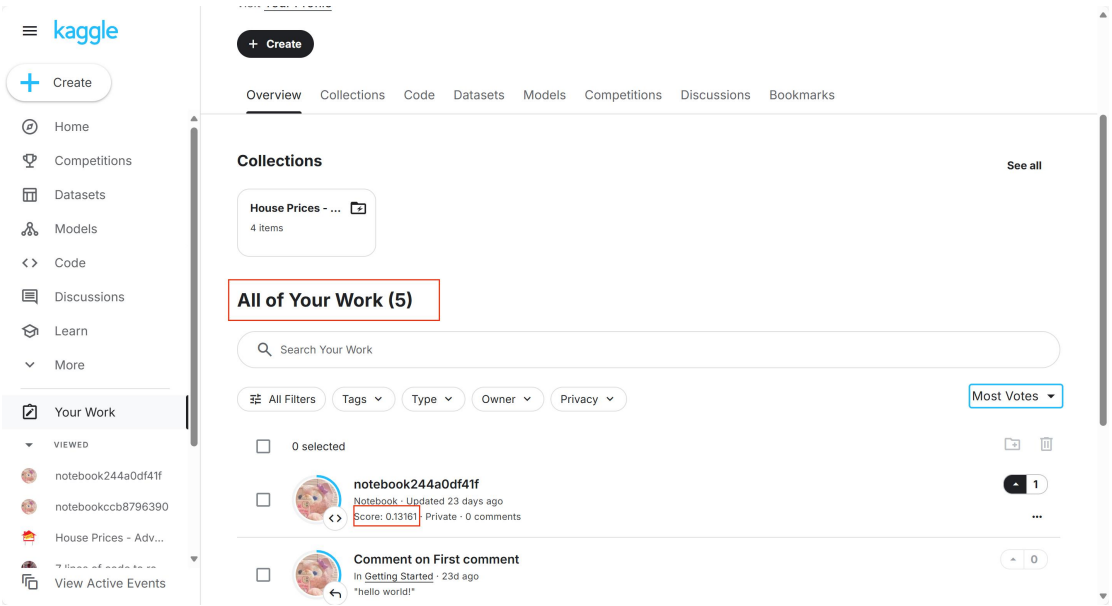
1.实验内容

完成 kaggle 上的比赛题目——House Prices - Advanced Regression Techniques。即基于不同的描述变量预测爱荷华州艾姆斯的住宅最终价格。实验的关键技术包括特征工程和回归预测，目的是通过特征工程提取和回归模型训练，利用给定的属性数据预测测试集中每个住宅的成交价格。

在 kaggle 上最终成绩为 **0.13161**，截图如下：





2.实验方法

1. 数据加载和初步处理

（1）导入必要的库

In [2]:
```
import os

import warnings

import numpy as np

import pandas as pd

from sklearn.linear_model import Lasso

from sklearn.linear_model import LassoCV

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler, LabelEncoder
```

数据处理：使用 numpy 和 pandas 进行数值计算和数据操作；
特征工程：使用 LabelEncoder 进行标签编码，StandardScaler 进行数据标准化；
模型训练：使用多种回归模型，包括 XGBoost、LightGBM、随机森林和 CatBoost，以获得更好的预测性能；
模型评估：使用 train_test_split 划分训练集和验证集，以评估模型性能。

（2）加载数据集

In [4]:
```
train = pd.read_csv('/kaggle/input/house-prices-advanced-regression-techniques/train.csv')
test = pd.read_csv('/kaggle/input/house-prices-advanced-regression-techniques/test.csv')
sample = pd.read_csv('/kaggle/input/house-prices-advanced-regression-techniques/sample_submission.csv')
```

（3）提取目标变量并合并数据

In [6]:
```
#Seprate taget value "Id" "Saleprice(more importantly)" from the features

y     = train[['Id','SalePrice']]

#After storing the target value "SalePrice" in y we can now drop it from the feature set

train = train.drop('SalePrice',axis=1)
```

In [7]:
```
#creates a list with both Training and Testing Data sets -> all_dfs

all_dfs = [train,test]

# "d.concat(all_dfs)" merges the train and test datasets into a single DataFrame called all_df. It stacks

# ".reset_index(drop=True)" resets the index of the new concatenated DataFrame and drops the old index.

# Without this, the concatenated DataFrame would retain the indices from both train and test

all_df = pd.concat(all_dfs).reset_index(drop=True);
```

目标变量提取：将 SalePrice 提取出来，方便后续的特征工程，不受目标变量的影响。
数据合并：将训练集和测试集合并，统一进行数据预处理和特征工程，确保在训练和测试时使用相同的数据处理方式。

2. 数据预处理

（1）缺失值分析

```
In [8]: display_all((all_df.isnull().sum()/all_df.shape[0])*100)
```

```
Id                0.000000
MSSubClass        0.000000
MSZoning          0.137033
LotFrontage      16.649538
LotArea           0.000000
                    ...
MiscVal           0.000000
MoSold            0.000000
YrSold            0.000000
SaleType          0.034258
SaleCondition     0.000000
Length: 80, dtype: float64
```

（2）删除高缺失值特征

```
In [9]: # Drop all the features with very high number in null rows

all_df.drop(['Alley','PoolQC','MiscFeature','Fence','FireplaceQu','Utilities'],axis=1,inplace=True)
```

对于缺失值比例过高的特征（Alley：小巷类型、PoolQC：游泳池质量、MiscFeature：杂项特征、Fence：围栏质量、FireplaceQu：壁炉质量、Utilities：公共设施类型），直接删除，避免填充带来的偏差和噪声。

（3）填充缺失值

```
In [10]: # with all feature with few null views, we manually filled the missing data

all_df['LotFrontage'].fillna(value=all_df['LotFrontage'].median(),inplace=True)

all_df['MasVnrType'].fillna(value='None',inplace=True)

all_df['MasVnrArea'].fillna(0,inplace=True)

all_df['BsmtCond'].fillna(value='TA',inplace=True)

all_df['BsmtExposure'].fillna(value='No',inplace=True)

all_df['Electrical'].fillna(value='SBrkr',inplace=True)

all_df['BsmtFinType2'].fillna(value='Unf',inplace=True)

all_df['GarageType'].fillna(value='Attchd',inplace=True)

all_df['GarageYrBlt'].fillna(value=all_df['GarageYrBlt'].median(),inplace=True)

all_df['GarageFinish'].fillna(value='Unf',inplace=True)
```

中位数填充：对于偏态分布的数值特征，使用中位数。

均值填充：对于正态分布的数值特征，使用均值。

众数填充：对于类别型特征，使用出现频率最高的值。

特殊值填充：根据业务逻辑，填充特定的值。

3. 特征工程

（1）编码分类特征

```
labelencoder=LabelEncoder()



all_df['MSZoning']      = labelencoder.fit_transform(all_df['MSZoning'].astype(str))

all_df['Exterior1st']   = labelencoder.fit_transform(all_df['Exterior1st'].astype(str))

all_df['Exterior2nd']   = labelencoder.fit_transform(all_df['Exterior2nd'].astype(str))

all_df['KitchenQual']   = labelencoder.fit_transform(all_df['KitchenQual'].astype(str))

all_df['Functional']    = labelencoder.fit_transform(all_df['Functional'].astype(str))

all_df['SaleType']      = labelencoder.fit_transform(all_df['SaleType'].astype(str))

all_df['Street']        = labelencoder.fit_transform(all_df['Street'])

all_df['LotShape']      = labelencoder.fit_transform(all_df['LotShape'])
```

将类别型特征转换为数值型，以便模型可以处理。使用 LabelEncoder 对每个类别特征进行编码，将不同的类别映射为整数值。

（2）特征缩放

```
In [12]: #scales features by removing the mean

         Scaler = StandardScaler()

         # ".fit_transform()" calculates the mean and standard deviation for scaling and then applies the tran

         all_scaled   = pd.DataFrame(Scaler.fit_transform(all_df))
```

将数值特征进行标准化处理，使其均值为 0，方差为 1，消除不同特征之间的量纲差异。使用 StandardScaler 对所有特征进行标准化。

（3）拆分数据集

```
train_scaled = pd.DataFrame(all_scaled[:1460])

test_scaled  = pd.DataFrame(all_scaled[1460:2920])
```

将标准化后的数据重新拆分为训练集和测试集，保证与原始数据对应。根据原始训练集的行数（1460 行），将标准化后的数据进行切片。

4. 模型训练

（1）准备训练和验证集

```
In [13]: X = train_scaled

         X_train, X_test, y_train, y_test = train_test_split(X, y['SalePrice'], test_size=0.1, random_state=42)
```

使用train_test_split，按照9:1的比例将训练集进一步划分为训练集和验证集，以评估模型的泛化能力。

（2）定义和训练模型

通过使用不同的超参数配置，训练多个模型（XGBoost、随机森林），增加

模型的多样性，有助于提高集成效果。使用验证集评估模型性能，调整模型复杂度参数（如 max_depth、min_child_weight 等）以避免过拟合；设置 n_jobs=-1，利用所有可用的 CPU 资源进行并行计算，加速模型训练。

（a）XGBoost

XGBoost 参数：①max_depth：树的最大深度，控制模型的复杂度；②learning_rate：学习率，控制每棵树的贡献程度；③n_estimators：树的数量，即提升迭代次数；④reg_alpha：L1 正则化系数，控制模型的稀疏性；⑤reg_lambda：L2 正则化系数，防止过拟合；⑥min_child_weight：最小叶子节点样本权重和，防止过拟合。

XGBoost 模型 1：

```
XGB1 = XGBRegressor(
    max_depth=3,        # maximum depth
    learning_rate=0.17851317118854057,   # 学习率
    n_estimators=319,   # boosting rounds
    reg_alpha=0.1070232678681972,   # alpha
    reg_lambda=9.21620230766857e-05,   # lambda
    min_child_weight=3,   # minimum child weight
    n_jobs=-1,   # 使用所有可用的 CPU
)                  #``(A higher value makes the model more conservative by preventing the tree from growing too complex )
XGB1.fit(X_train,y_train)
```

Out[14]:
```
XGBRegressor(base_score=None, booster=None, callbacks=None,
             colsample_bylevel=None, colsample_bynode=None,
             colsample_bytree=None, device=None, early_stopping_rounds=None,
             enable_categorical=False, eval_metric=None, feature_types=None,
             gamma=None, grow_policy=None, importance_type=None,
             interaction_constraints=None, learning_rate=0.17851317118854057,
             max_bin=None, max_cat_threshold=None, max_cat_to_onehot=None,
             max_delta_step=None, max_depth=3, max_leaves=None,
             min_child_weight=3, missing=nan, monotone_constraints=None,
             multi_strategy=None, n_estimators=319, n_jobs=-1,
             num_parallel_tree=None, random_state=None, ...)
```

XGBoost 模型 2：

In [15]:
```
XGB2 = XGBRegressor(
    max_depth=3,   # maximum depth
    learning_rate=0.0364507719136609,   # 学习率
    n_estimators=965,   # boosting rounds
    reg_alpha=0.00018131710013969177,   # alpha
    reg_lambda=2.3152099456723933e-07,   # lambda
    min_child_weight=3,   # minimum child weight
    n_jobs=-1   # 使用所有可用的 CPU
)                  #``(A higher value makes the model more conservative by preventing the tree from growing too complex )
XGB2.fit(X_train,y_train)
```

Out[15]:
```
XGBRegressor(base_score=None, booster=None, callbacks=None,
             colsample_bylevel=None, colsample_bynode=None,
             colsample_bytree=None, device=None, early_stopping_rounds=None,
             enable_categorical=False, eval_metric=None, feature_types=None,
             gamma=None, grow_policy=None, importance_type=None,
             interaction_constraints=None, learning_rate=0.0364507719136609,
             max_bin=None, max_cat_threshold=None, max_cat_to_onehot=None,
             max_delta_step=None, max_depth=3, max_leaves=None,
             min_child_weight=3, missing=nan, monotone_constraints=None,
             multi_strategy=None, n_estimators=965, n_jobs=-1,
             num_parallel_tree=None, random_state=None, ...)
```

（b）随机森林

随机森林参数：①n_estimators：决策树的数量；②max_depth：树的最大深

度；③min_samples_split：最小样本分割数；④min_samples_leaf：叶子节点最小样本数；⑤max_features：最大特征比例；⑥bootstrap：是否有放回地抽样。

随机森林模型 1：

In [16]:
```
RF1 = RandomForestRegressor(
    n_estimators=198,  # 决策树数量
    max_depth=18,  # 最大树深度
    min_samples_split=2,  # 最小样本分割数
    min_samples_leaf=2,  # 叶节点最小样本数
    max_features=0.7084875029768692,  # 最大特征比例
    bootstrap=False,  # 是否启用bootstrap
    random_state=42  # 保持结果可重复
)
RF1.fit(X_train, y_train)
```

Out[16]:
```
RandomForestRegressor(bootstrap=False, max_depth=18,
                      max_features=0.7084875029768692, min_samples_leaf=2,
                      n_estimators=198, random_state=42)
```

随机森林模型 2：

In [17]:
```
RF2 = RandomForestRegressor(
    n_estimators=293,  # 决策树数量
    max_depth=20,  # 最大树深度
    min_samples_split=2,  # 最小样本分割数
    min_samples_leaf=1,  # 叶节点最小样本数
    max_features=0.496408431158935,  # 最大特征比例
    bootstrap=False,  # 是否启用bootstrap
    random_state=42  # 保持结果可重复
)

RF2.fit(X_train, y_train)
```

Out[17]:
```
RandomForestRegressor(bootstrap=False, max_depth=20,
                      max_features=0.496408431158935, n_estimators=293,
                      random_state=42)
```

5. 超参数优化

使用 Optuna 这一超参数优化框架，对 XGBoost 和随机森林两个模型进行超参数优化（分别通过设置 xgb_tuning = False 和 rf_tuning = False 来控制是否执行优化过程）。

（a）XGBoost 超参数优化（xgb_tuning 为 True 时启用）

（1）优化目标函数

```
def objective(trial):
    params = {
        'max_depth': trial.suggest_int('max_depth', 2, 10),
        'learning_rate': trial.suggest_loguniform('learning_rate', 1e-3, 0.3),
        'n_estimators': trial.suggest_int('n_estimators', 100, 1000),
        'reg_alpha': trial.suggest_loguniform('reg_alpha', 1e-8, 1.0),
        'reg_lambda': trial.suggest_loguniform('reg_lambda', 1e-8, 1.0),
        'min_child_weight': trial.suggest_int('min_child_weight', 1, 10),
        'subsample': trial.suggest_uniform('subsample', 0.6, 1.0),
        'colsample_bytree': trial.suggest_uniform('colsample_bytree', 0.6, 1.0)
    }

    model = XGBRegressor(**params, n_jobs=-1)
    model.fit(X_train, y_train,
              eval_set=[(X_val, y_val)],
              eval_metric='rmse',  # 使用RMSE作为评估指标
              early_stopping_rounds=50,
              verbose=False)

    # 对目标值和预测值取对数
    y_pred = model.predict(X_val)
    y_pred_log = np.log(y_pred + 1)  # 避免对数0的问题
    y_val_log = np.log(y_val + 1)

    # 计算对数 RMSE
    log_rmse = np.sqrt(mean_squared_error(y_val_log, y_pred_log))
    return log_rmse
```

①使用当前试验的超参数创建 XGBRegressor 模型；②训练模型时使用 early_stopping_rounds=50，如果在 50 轮内验证集的性能没有提升，提前停止训练，防止过拟合；③对预测值和真实值取对数，避免数值过大导致的误差放大，同时处理目标值的偏态分布，并以计算对数均方根误差（Log RMSE）作为优化目标。

（2）执行超参数优化

```
if xgb_tunning==True:
    # 创建文件以保存试验结果
    output_file = "optuna_trials.txt"

    # 定义输出函数，将每个 trial 的信息保存到文件
    def save_trial_result(study, trial):
        with open(output_file, 'a') as f:
            f.write(f"Trial {trial.number} finished with value: {trial.value} and parameters: {trial.params}. "
                    f"Best is trial {study.best_trial.number} with value: {study.best_value}.\n")

    # 使用Optuna进行超参数调优
    study = optuna.create_study(direction='minimize')
    study.optimize(objective, n_trials=100, callbacks=[save_trial_result])

    print("Best parameters:", study.best_params)
    print("Best Log RMSE:", study.best_value)
```

①创建文件保存试验结果：将每次试验的结果保存到 optuna_trials.txt，便于后续分析；②定义回调函数 save_trial_result：在每次试验完成后，记录试验编号、结果、参数和当前最佳结果；③创建 Optuna 的 Study 对象：指定优化方向为最小化（即最小化 Log RMSE）；④执行优化：调用 study.optimize，传入目标函数 objective，指定试验次数为 100，并添加回调函数；⑤输出最佳结果：打印最佳超参数组合和对应的最小 Log RMSE 值。

（3）使用最佳参数训练最终模型

```
    # 使用最佳参数训练最终模型
    best_params = study.best_params
    final_model = XGBRegressor(**best_params, n_jobs=-1)
    final_model.fit(X_train, y_train)

    # 在验证集上评估最终模型
    y_pred = final_model.predict(X_val)
    y_pred_log = np.log(y_pred + 1)
    y_val_log = np.log(y_val + 1)
    final_log_rmse = np.sqrt(mean_squared_error(y_val_log, y_pred_log))
    print("Final model Log RMSE:", final_log_rmse)
```

从 study.best_params 中获取最优的超参数组合，并在整个训练集上（X_train 和 y_train）训练模型，以充分利用数据。最终在验证集上计算 Log RMSE，验证模型的泛化能力。

（b）随机森林超参数优化（rf_tuning 为 True 时启用）

（1）优化目标函数

```
def objective(trial):
    params = {
        'n_estimators': trial.suggest_int('n_estimators', 100, 1000),  # 森林中的树木数目
        'max_depth': trial.suggest_int('max_depth', 2, 20),  # 树的最大深度
        'min_samples_split': trial.suggest_int('min_samples_split', 2, 10),  # 内部节点再分裂所需的最小样本数
        'min_samples_leaf': trial.suggest_int('min_samples_leaf', 1, 10),  # 叶子节点所需的最小样本数
        'max_features': trial.suggest_uniform('max_features', 0.1, 1.0),  # 每次分裂时的最大特征数
        'bootstrap': trial.suggest_categorical('bootstrap', [True, False])  # 是否有放回地采样
    }

    # 创建随机森林模型
    rf = RandomForestRegressor(**params, random_state=42, n_jobs=-1)

    # 训练模型
    rf.fit(X_train, y_train)

    # 在验证集上进行预测
    y_pred = rf.predict(X_val)

    # 对目标值和预测值取对数，避免对数为0的问题
    y_pred_log = np.log(y_pred + 1)
    y_val_log = np.log(y_val + 1)

    # 计算对数 RMSE
    log_rmse = np.sqrt(mean_squared_error(y_val_log, y_pred_log))
    return log_rmse
```

（2）执行超参数优化

```
if rf_tuning == True:
    # 创建文件以保存试验结果
    output_file = "rf_optuna_trials.txt"

    # 定义输出函数，将每个 trial 的信息保存到文件
    def save_trial_result(study, trial):
        with open(output_file, 'a') as f:
            f.write(f"Trial {trial.number} finished with value: {trial.value} and parameters: {trial.params}. "
                    f"Best is trial {study.best_trial.number} with value: {study.best_value}.\n")

    # 使用 Optuna 进行超参数调优
    study = optuna.create_study(direction='minimize')
    study.optimize(objective, n_trials=100, callbacks=[save_trial_result])

    # 输出最佳参数和结果
    print("Best parameters:", study.best_params)
    print("Best Log RMSE:", study.best_value)
```

（3）使用最佳参数训练最终模型

```
# 使用最佳参数训练最终模型

best_params = study.best_params

final_rf = RandomForestRegressor(**best_params, random_state=42, n_jobs=-1)

final_rf.fit(X_train, y_train)


# 在验证集上评估最终模型

y_pred_final = final_rf.predict(X_val)

y_pred_log = np.log(y_pred_final )

y_val_log = np.log(y_val )

final_log_rmse = np.sqrt(mean_squared_error(y_val_log, y_pred_log))

print("Final model Log RMSE:", final_log_rmse)
```

6. 模型预测与集成

（1）生成预测结果

```
In [22]: #generate the predictions using ".predict" for the "test_scaled" for both models and store it to merge later

         y_pred_xgb1  = pd.DataFrame( XGB1.predict(test_scaled))

         y_pred_xgb2  = pd.DataFrame( XGB2.predict(test_scaled))

         y_pred_rf1 = pd.DataFrame(RF1.predict(test_scaled))

         y_pred_rf2 = pd.DataFrame(RF2.predict(test_scaled))
```

（2）模型集成

```
y_pred=pd.DataFrame()




y_pred['SalePrice'] = 0.42 * y_pred_xgb1[0]+0.42 * y_pred_xgb2[0] +0.08*y_pred_rf1[0]+0.08*y_pred_rf2[0]

#store the Id column in the dataset

y_pred['Id'] = test['Id']
```

对多个模型的预测结果进行加权平均，权重之和为 1。其中，XGBoost 模型的权重较高，各为 0.42；随机森林模型的权重较低，各为 0.08。

（3）结果保存

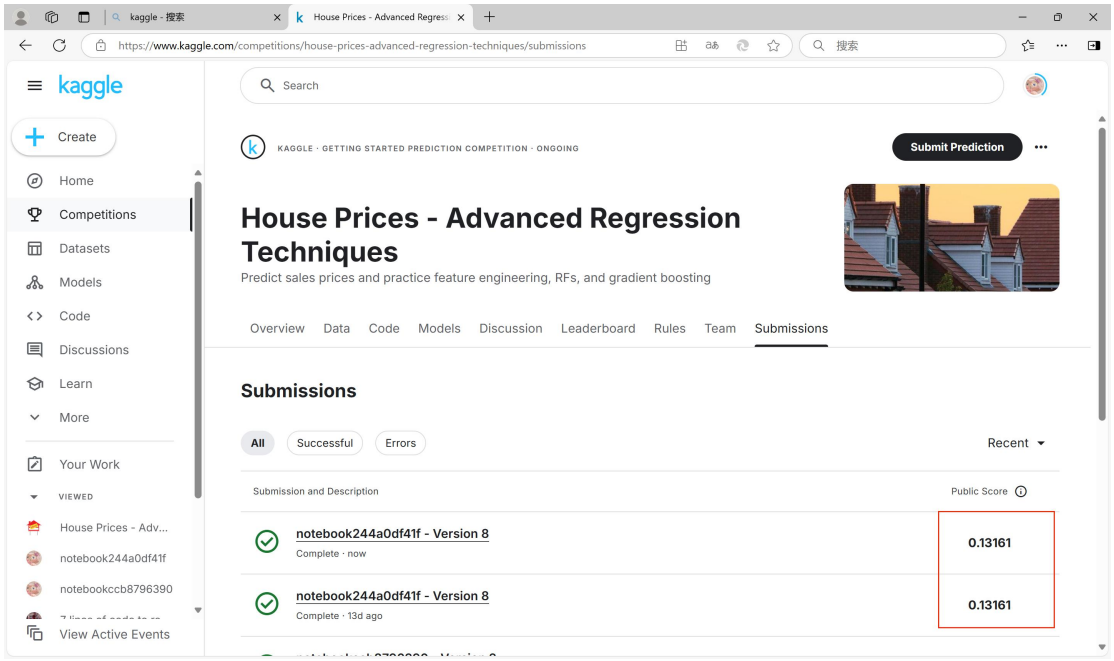
```
In [23]: #save dataset into a csv file for submission

         y_pred.to_csv('submission.csv',index=False)
```

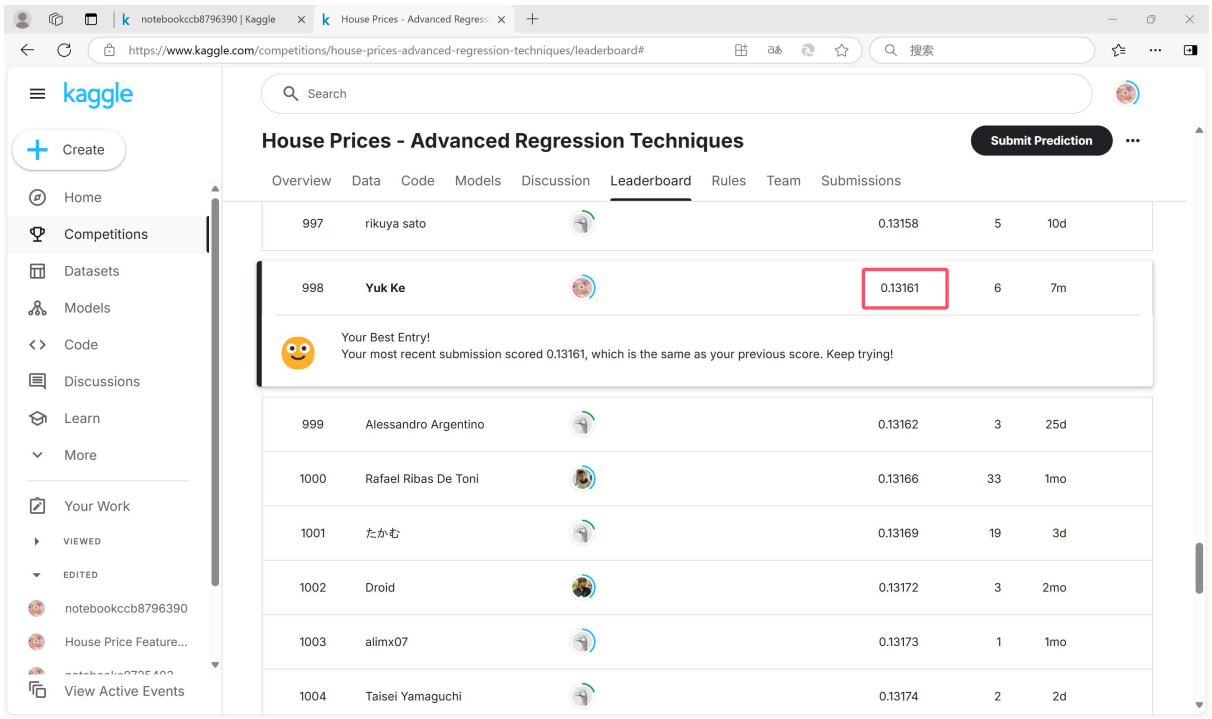
将最终结果保存在 submission.csv 文件中。

3.实验结果和分析

在 kaggle 上的分数为 **0.13161**，排名 998/5267（2024.10.27 日在排行榜中排名）
kaggle 成绩截图：



排行榜截图：



4.总结

在这次大作业中，我体验了使用机器学习方法进行回归预测的完整流程：从数据的获取、清洗到模型的建立和优化，每一个环节都让我对数据分析和处理有了更深入的理解；在数据预处理阶段，我学会了如何应对缺失值、如何正确编码类别型特征以及进行特征缩放等操作。这些实践经验让我更加理解了数据质量对模型性能的重要性。

在模型训练过程中，我尝试了不同的算法，包括 XGBoost 和随机森林等，并对超参数进行了细致的调优。这让我对各类机器学习方法的优势和适用场景有了更清晰的认识，也巩固了理论课中老师所讲授的知识。在超参数优化的过程中，我体会到了参数选择对模型效果的显著影响，学会了如何使用工具来自动化地寻找最佳参数组合。

《大数据导论》这门课真的教会了我很多，我所收获的不仅有理论知识，更有非常宝贵的、在一次次“令人头秃”的 debug 中收获的实操经验。比如最开始做作业一时，被豆瓣的反爬机制折磨了很久，反反复复改代码、修改应对反爬的方案，最终才有了那五万多条评论。本以为这就封顶了，结果到了实验一又出现了各种各样的状况：实训搭建平台时由于对命令的完全不熟悉导致进度非常缓慢、终于进入实验一核心部分后才发现原来作业一的评论还要再进行分词处理遂连夜学习等等等等。虽然每个实验总会遇到代码无法运行、结果不符合预期等问题、大作业也耗时良久，但通过自己的努力，也在老师助教的帮助下，最终这些问题都被解决了。

“道阻且长，行则将至；行而不辍，未来可期”。虽然改代码的过程是痛苦而曲折的，但每次克服困难后，所收获的总会赋予我勇敢面对下一次的 bug 的勇气，更激励我继续深入探索。我相信这些收获将为我今后的学习和发展打下坚实的基础。