第八章 回归分析 房价预测模型

回归分析用于对连续变量的预测，拟合训练集的曲线，预测未知的值。常用的有线性回归，多项式回归等，这些是连续可导的函数回归，前面的时间序列分析则是基于概率论的回归。对于更复杂的变量分布，比如本章的房价预测，没有明显的函数规律，上述回归方法处理的效果可能不够好，这就需要树回归与深度学习等其它一些的回归方法去拟合，它们共同的特点就是回归曲线不是连续可导的。深度学习下一章会详细介绍，本章先与树回归等放在一起，看看它用于房价回归预测的效果，比较一下。本章的结论是，梯度提升决策树GBDT模型是解决这类复杂回归问题最有效的模型。

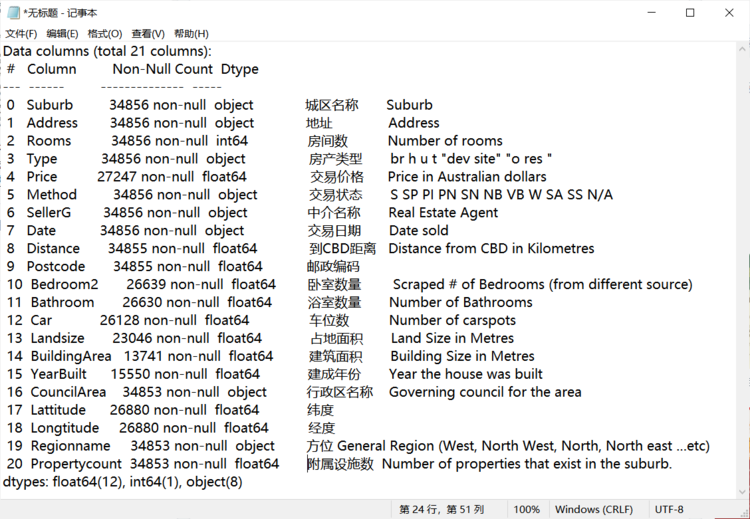
第一节 集成回归与深度学习

房价预测是大家都比较关注的一个主题，税收工作中的应用就是二手房交易价格的监管。[Kaggle上有一项参与队伍很多的竞赛](https://www.kaggle.com/c/house-prices-advanced-regression-techniques/overview)，可以提供很多有用的参考资料。为方便讨论，本章使用[墨尔本市的房产交易数据集](https://www.kaggle.com/anthonypino/melbourne-housing-market)，因为它提供了经纬度坐标，房产价格的决定性因素是地段，数字经纬度坐标比较便于机器学习处理。在GIS应用一章中我们已经看到，现在可以通过地理编码服务，把文本的门牌号码转换为相对准确的经纬度坐标，我相信在全球数据科学家的积极参与下，这个课题一定会产生比较好的模型。

因为kaggle上这些代码大部分是Python，本章也是用Python，不过同样的逻辑用R语言实现也是可以的。所以不必纠缠于具体的语言、平台与框架，重要的是处理逻辑与模型，它们是通用的，总要从其中一个开始。

一、墨尔本市房产交易数据集

这个数据集有21列34856条数据，具体可参阅kaggle上的文档。



二、数据探索

本节主要参考[参考资料1](https://www.kaggle.com/poojpillai/pre-processing-housing-data)，[参考资料2](https://www.kaggle.com/pmarcelino/comprehensive-data-exploration-with-python)。

1、载入数据并初步了解数据集

# -\*- coding: utf-8 -\*-

"""

Created on Wed Sep 1 16:23:15 2021

@author: Jean

"""

# Data Pre-processing

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.linear\_model import LinearRegression

from sklearn.model\_selection import train\_test\_split

from sklearn.impute import SimpleImputer

from sklearn.preprocessing import LabelEncoder

from sklearn.metrics import r2\_score

# Reading a CSV File

# 34857

df = pd.read\_csv("D:/temp/data/Melbourne\_housing/Melbourne\_housing\_FULL.csv", encoding="utf-8")

# 34856

df = df.drop\_duplicates()

# 8887, too less data, so we must use impution to deal with null values

df2 = df.dropna()

# Displaying first five records of datset

df.head()

# Displays dimension of the dataset i.e no. of rows and columns

df.shape

df.info()

# Describe the dataset

df.describe()

df.describe().T

# Slicing for first 20 rows for the column named 'Method'.

df[0:20]['Method']

# Displaying first 10 records of attributes 'Distane' and 'Price'

df.loc[0:10,['Distance','Price']]

# Count no. of unique values in the column 'Method'

df['Method'].value\_counts()

#separate the numeric columns from the categorical columns

# select numerical columns

data\_numeric = df.select\_dtypes(include=[np.number])

numeric\_cols = data\_numeric.columns.values

# select non-numeric columns

data\_non\_numeric = df.select\_dtypes(exclude=[np.number])

non\_numeric\_cols = data\_non\_numeric.columns.values

numeric\_cols

non\_numeric\_cols

# Printing contents of attribute 'Price'

df['Price']

2、Bedroom2与Rooms列差异很小，删除Bedroom2列。

# After carefully evaluating data, it can be noticed that variables "Rooms" and "Bedroom2"

# are pretty much similar and one of the columns should be removed to avoid duplication of data

df["b 2 r"] = df["Bedroom2"] - df["Rooms"]

df[["b 2 r", "Bedroom2", "Rooms"]].head()

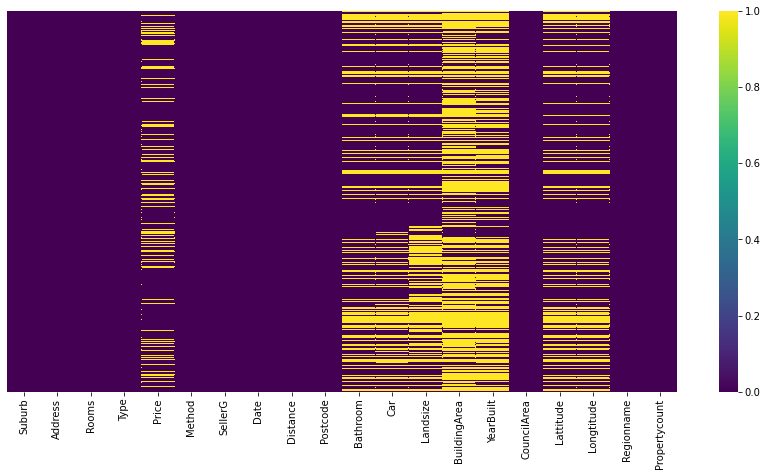
# we can see that the difference is very minimal here that will be wise to

# remove one of the 2 columns

df = df.drop(["b 2 r", "Bedroom2"],1)

3、查看各字段缺失值的分布与百分比

各字段缺失值分布图



# Check for null values

df.isnull().sum()

# visualizing missing values

fig, ax = plt.subplots(figsize=(15,7))

sns.heatmap(df.isnull(), yticklabels=False,cmap="viridis")

# Percentage of missing values

df.isnull().sum()/len(df)\*100

# From the information above, we can notice that few feature varaibles still have large percentage of

# missing values. At this point we are ignoring it, but at later state if we will take those as our feature

# variables for our model, we will explore ways to fill in those information or to remove those from our data.

#df = df.drop(["Landsize","BuildingArea","YearBuilt"],axis=1)

# Also since our target variable is price, it makes sense to drop rows for

# price columns where price values are missing

# Longtitude and Lattitude are critical that should not be null too.

# 34856-->20993

df.dropna(subset=["Price","Longtitude","Lattitude"], inplace=True)

# 20993-->9020

df.dropna(subset=["Landsize","BuildingArea","YearBuilt"], inplace=True)

# 8887

df2 = df.dropna()

各字段缺失值百分比，可以看到，建筑面积、建成年份、占地面积、车位数、浴室数量、纬度、经度、交易价格，这些字段的缺失数据比较多。根据生活经验，对于价格预测建模来说，交易价格与经纬度是比较重要的数据，建筑面积、建成年份、占地面积也是比较重要的数据，为提高模型的准确度，这些数据不使用插值去补缺，而是选择非缺失的样本去训练、验证与测试，其它列有缺失的，则通过插值去补缺。这样剩下的数据有9020条，这样规模的样本集训练的模型，应该有比较好的泛化性能。当然，样本数据是越多越好。

In [**19**]: df.isnull().sum()/len(df)\*100

Out[**19**]:

Suburb 0.000000

Address 0.000000

Rooms 0.000000

Type 0.000000

Price 21.829814

Method 0.000000

SellerG 0.000000

Date 0.000000

Distance 0.002869

Postcode 0.002869

Bathroom 23.599954

Car 25.040165

Landsize 33.882258

BuildingArea 60.577806

YearBuilt 55.387882

CouncilArea 0.008607

Lattitude 22.882717

Longtitude 22.882717

Regionname 0.008607

Propertycount 0.008607

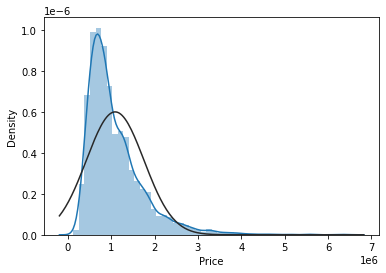
dtype: float64

4、查看目标变量交易价格的分布

可以看到交易价格（蓝色拟合线）是带尖峰的偏态分布。

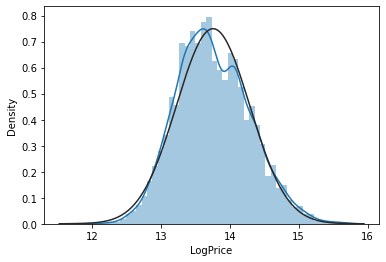
Skewness: 2.131305，Kurtosis: 7.580344。

黑色拟合线是正态分布。

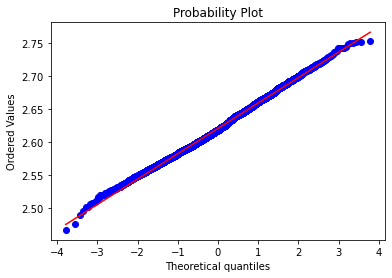


经过对数转换后（蓝色拟合线），接近正态分布。

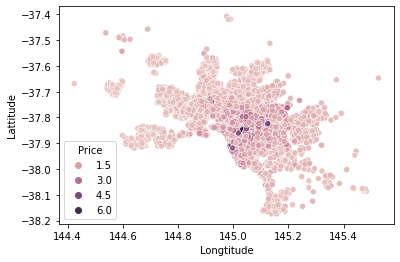
mu = 13.76 ，sigma = 0.53。



对数转换后，QQ图说明接近正态分布。



5、最后，可以看看房价在地理上的分布，有一些集中的高房价地区。



# 画房价的分布图观察

sns.scatterplot(x='Longtitude',y='Lattitude',data=df,hue='Price')

#histogram

from scipy.stats import norm

sns.distplot(df['Price'], fit=norm);

# It'll be better to transform target variable Price with a log transofrm.

df["LogPrice"] = np.log(df['Price'])

sns.distplot(df["LogPrice"], fit=norm);

#skewness and kurtosis

print("Skewness: %f" % df['Price'].skew())

print("Kurtosis: %f" % df['Price'].kurt())

# Get the fitted parameters used by the function

(mu, sigma) = norm.fit(df["LogPrice"])

print( '\n mu = {:.2f} and sigma = {:.2f}\n'.format(mu, sigma))

#Get also the QQ-plot

from scipy import stats

fig = plt.figure()

res = stats.probplot(np.log(df['LogPrice']), plot=plt)

plt.show()

三、数据预处理

具体参阅以下[参考资料1](https://www.kaggle.com/poojpillai/pre-processing-housing-data)，[参考资料2](https://www.kaggle.com/pmarcelino/comprehensive-data-exploration-with-python)。

1、缺失值插值，对不同的字段，使用均值、中位数、填零、相邻的后一个值、相邻的前一个值等不同的方式插值补缺。

# Pre-processing attributes having null values， Cleaning / Filling Missing Data

# =============================================================================

# There are 5 ways to find the null values if present in the dataset

# isnull() — provides the boolean value for the complete dataset to know if any null value is present or not

# isna() — same as the isnull() function, provides the same output

# isna().any() — gives a boolean value if any null value is present or not, but it gives results column-wise, not in tabular format

# isna().sum() — gives the sum of the null values preset in the dataset column-wise

# isna().any().sum() — gives output in a single value if any null is present or not

# Imputing techniquess

#

# fillna — filling in null values based on given value (mean, median, mode, or specified value)

# bfill / ffill — stands for backward fill and forward fill (filling in missing values based on the value after or before the column.)

# Simple Imputer — Sklearn’s built-in function that imputes missing values (commonly used alongside a pipeline when building ML models)

#

#

# =============================================================================

# Mean and Median of values in column 'Price'

print(f"Median : {df['Price'].median()}")

print(f"Mean : {df['Price'].mean()}")

# All occurrences of missing\_values are imputed by median Price

price\_imputer = SimpleImputer(missing\_values = np.nan, strategy='median')

df[['Price']] = price\_imputer.fit\_transform(df[['Price']])

df['Price'].head(10)

# Checking null values of attribute 'Distance'

df['Distance'].isnull().sum()

df['Distance'].mean()

# Filling null values of attribute 'Distane' using mean

distance\_mean = round(df['Distance'].mean(),1)

df['Distance'].fillna(distance\_mean, inplace = True)

# Checking null values of attribute 'Postcode'

df['Postcode'].isnull().sum()

# Counting unique postcodes in column 'Postcode'

df['Postcode'].value\_counts()

df['Postcode'].median()

# fillna() replaces null values of attribute 'Postcode' by median

postcode\_median = round(df['Postcode'].median())

df['Postcode'].fillna(postcode\_median, inplace = True)

'''

# This column is droped already

# value\_counts() finds the unique no. of bedroom counts

df['Bedroom2'].value\_counts()

# Checking null values in column 'Bedroom2'

df['Bedroom2'].isnull().sum()

# Displaying first 20 values of attribute 'Bedroom2'

print("Bedrooms with NULL values")

df['Bedroom2'].head(20)

# Null values of Bedroom replaced by 0

df['Bedroom2'].fillna(0, inplace= True)

print("Bedrooms after replacing NULL values")

df['Bedroom2'].head(20)

'''

# Checking null values in column 'Bathroom'

df['Bathroom'].isnull().sum()

# Displaying first 20 values of attribute 'Bathroom'

print("Bathroom with NULL values")

df['Bathroom'].head(20)

# Null values of Bathroom replaced by 1

df['Bathroom'].fillna(1, inplace= True)

print("Bathroom after replacing NULL values")

df['Bathroom'].head(20)

# Checking null values in column 'Car'

df['Car'].isnull().sum()

# Displaying first 20 values of attribute 'Car'

print("Car with NULL values")

df['Car'].head(20)

# Null values of Car replaced by 0

df['Car'].fillna(0, inplace= True)

print("Car after replacing NULL values")

df['Car'].head(20)

# Checking null values in column 'Landsize'

df['Landsize'].isna().sum()

# Mean and Median values of column 'Landsize'

print(f"Median : {df['Landsize'].median()}")

print(f"Mean : {round(df['Landsize'].mean(),0)}")

# Displaying first 20 values of attribute 'Landsize'

print("Landsize with NULL values")

df['Landsize'].head(20)

# All occurrences of missing\_values are imputed by median Landsize

from sklearn.impute import SimpleImputer

land\_imputer = SimpleImputer(missing\_values = np.nan, strategy='median')

df[['Landsize']] = land\_imputer.fit\_transform(df[['Landsize']])

df['Landsize'].head(20)

# Checking NULL values in column BuildingArea

df['BuildingArea'].isna().sum()

# Mean and Median of values in column 'BuildingArea'

print(f"Median : {df['BuildingArea'].median()}")

print(f"Mean : {round(df['BuildingArea'].mean())}")

# Dosplaying first 20 values of attribute 'BuildingArea'

print("BuildingArea with NULL values")

df['BuildingArea'].head(20)

# All occurrences of missing\_values are imputed by mean building area

from sklearn.impute import SimpleImputer

area\_imputer = SimpleImputer(missing\_values = np.nan, strategy='mean')

df[['BuildingArea']] = land\_imputer.fit\_transform(df[['BuildingArea']])

df['BuildingArea'].head(20)

# Mean and Median of values in column 'YearBuilt'

print(f"Median : {df['YearBuilt'].median()}")

print(f"Mean : {round(df['YearBuilt'].mean())}")

# All occurrences of missing\_values are imputed by mean YearBuilt

from sklearn.impute import SimpleImputer

year\_imputer = SimpleImputer(missing\_values = np.nan, strategy='mean')

df[['YearBuilt']] = land\_imputer.fit\_transform(df[['YearBuilt']])

df['YearBuilt'].head(20)

# Displaying first 20 values of 'CouncilArea'

df['CouncilArea'].head(20)

# =============================================================================

# - bfill() :

#

# It is used to backward fill the missing values in the dataset.

# The missing values are replaced by values in next row of the same column

# =============================================================================

# Appying bfill for NULL values in attribute 'CouncilArea'

df['CouncilArea'].bfill(inplace=True)

df['CouncilArea']

# Checking NULL values in column 'CouncilArea' after bfill

df['CouncilArea'].isnull().sum()

# Checking NULL values in column 'Latitude'

df['Lattitude'].isnull().sum()

# ffill: Forward fill (NULL values are replaced by corresponding value in the previous row)

df['Lattitude'].fillna(method = 'ffill' , inplace=True)

# Checking NULL values in column 'Lattitude' after ffill

df['Lattitude'].isnull().sum()

# Checking NULL values in column 'Longtitude'

df['Longtitude'].isnull().sum()

# ffill: Forward fill (NULL values are replaced by corresponding value in the previous row)

df['Longtitude'].fillna(method = 'ffill' , inplace=True)

# Checking NULL values in column 'Lattitude' after ffill

df['Longtitude'].isnull().sum()

# Checking NULL values in column 'Regionname'

df['Regionname'].isnull().sum()

# bfill: Backward fill (NULL values are replaced by corresponding value in the next row)

df['Regionname'].bfill(inplace=True)

df['Regionname']

# Checking NULL values in column 'Regionname' after bfill

df['Regionname'].isnull().sum()

# Checking NULL values in column 'Propertycount'

df['Propertycount'].isnull().sum()

# value\_counts() finds number of unique properties in each suburb

df['Propertycount'].value\_counts()

# NULL values of attribute 'Propertycount' are replaced by ffill

df['Propertycount'].ffill(inplace=True)

# Checking NULL values in column 'Propertycount' after ffill

df['Propertycount'].isnull().sum()

# Now all NULL values of all attributes are replaced

df.isnull().sum()

df.info()

# Changing Data type

objdtype\_cols = df.select\_dtypes(["object"]).columns

df[objdtype\_cols] = df[objdtype\_cols].astype("category")

# looking at data information above, we can notice that "Date" is also converted

# to category.

# in this step we will cast date to datetime

df["Date"] = pd.to\_datetime(df["Date"])

df.info()

2、分类（字符）型变量转换为数值型变量，机器学习都要使用数值型变量。用sklearn包的LabelEncoder类的fit\_transform()函数转换大部分的分类变量，结果相当于R语言的factor类型，即几个整数取值的变量。用Pandas的get\_dummies()函数转换关键的房产类型为onehot类型的独热编码变量，属于某类的取值为1，否则取值为0，以增加模型拟合的精度。

# change category type into numeric

le=LabelEncoder()

df['Suburb']=le.fit\_transform(df.Suburb)

df.Method = le.fit\_transform(df.Method)

df.SellerG=le.fit\_transform(df.SellerG)

df.Regionname=le.fit\_transform(df.Regionname)

df.CouncilArea = le.fit\_transform(df.CouncilArea)

# df.Type = le.fit\_transform(df.Type)

#convert categorical variable into dummy

df = pd.get\_dummies(df, prefix="Type",columns=["Type"])

df.info()

df.columns.values

3、从交易日期中提取交易的年与月，并看看几类房价近年的变化趋势。房价同时具有时间序列的特征，后面会看到，交易年份的影响颇大。

# Price trend against year per house

## extract year and month from date

df["Year"] = df["Date"].apply(lambda x:x.year)

df['Month']=pd.DatetimeIndex(df['Date']).month

df.head(5)

# have a look at the last dataset we get

# sns.distplot(df, kde=False, bins=20).set(xlabel="Price");

numerics = ["int16", "int32", "int64", "float16", "float32", "float64"]

# df.select\_dtypes(include = numerics)

df.select\_dtypes(include = numerics).hist(bins=15, figsize=(15, 6),layout=(4,4))

# data subset by type

# house price

df\_h = df[df["Type"]=="h"]

# condo price

df\_u = df[df["Type"]=="u"]

# townhouse price

df\_t = df[df["Type"]=="t"]

#house, condo and townhouse price groupby "year" and "mean"

df\_h\_y = df\_h.groupby("Year").mean()

df\_u\_y = df\_u.groupby("Year").mean()

df\_t\_y = df\_t.groupby("Year").mean()

df\_h\_y.head()

# sns.implot(x="Year", y="Price", hue="Type", data=df,

# x\_estimator=np.mean);

df\_h\_y["Price"].plot(kind="line", color="r", label="House")

df\_u\_y["Price"].plot(kind="line", color="g", label="Condo")

df\_t\_y["Price"].plot(kind="line", color="b", label="TownHouse")

year\_xticks=[2016,2017,2018]

plt.ylabel("Price")

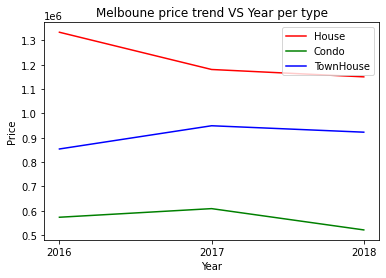
plt.xticks( year\_xticks)

plt.title("Melboune price trend VS Year per type")

plt.legend()

df.shape

df.columns



4、交易价格作对数转换，算法将对价格的对数作回归，因为统计学中的很多假设都基于正态分布，这个转换将模型的精度提高了4%~5%。在评估模型最终效果时，作指数变换回原量纲后再评估。

# It'll be better to transform target variable Price with a log transofrm.

df["LogPrice"] = np.log(df['Price'])

5、排除异常数据。这里排除房价超高与房间数超多的少数样本。这些城堡一样的超级豪宅，它们引入的方差或绝对误差很大，但对预测模型的意义不大。

# Finding Outliers

df.describe().T

# From the statstical summary above we can see that max price in our data is nearly $11.2 million.

# That looks like a clear outlier. But before removing it, lets first ensure that we have very few values

# in that range.

## to findout outliers lets divide data into different price ranges to identify number of occurences of data in different price ranges

df['PriceRange'] = np.where(df['Price'] <= 100000, '0-100,000',

np.where ((df['Price'] > 100000) & (df['Price'] <= 1000000), '100,001 - 1M',

np.where((df['Price'] > 1000000) & (df['Price'] <= 3000000), '1M - 3M',

np.where((df['Price']>3000000) & (df['Price']<=5000000), '3M - 5M',

np.where((df['Price']>5000000) & (df['Price']<=6000000), '5M - 6M',

np.where((df['Price']>6000000) & (df['Price']<=7000000), '6M - 7M',

np.where((df['Price']>7000000) & (df['Price']<=8000000), '7M-8M',

np.where((df['Price']>8000000) & (df['Price']<=9000000), '8M-9M',

np.where((df['Price']>9000000) & (df['Price']<=10000000), '9M-10M',

np.where((df['Price']>10000000) & (df['Price']<=11000000), '10M-11M',

np.where((df['Price']>11000000) & (df['Price']<=12000000), '11M-12M', '')

))))))))))

df.groupby(["PriceRange"]).agg({"PriceRange": ["count"]})

# Lets drop those outliers of price, 27242

df.drop(df[(df["PriceRange"] == "0-100,000") |

(df["PriceRange"] == "7M-8M") |

(df["PriceRange"] == "8M-9M") |

(df["PriceRange"] == "11M-12M")].

index, inplace=True)

df.describe().T

df.groupby(["Rooms"])["Rooms"].count()

# drop the outliers in rooms,27233

# df.drop(df[(df["Rooms"] == 12) |(df["Rooms"] == 16)].index,inplace=True)

df.drop(df[df["Rooms"] >8].index,inplace=True)

df.describe().T

df['Distance'] = round(df['Distance'])

df = df.drop(["PriceRange"],axis=1)

df.shape

6、对样本数据集重新随机排序，这样模型的准确度提高了1%~2%，泛化能力也提高了。

# 随机重新排序

df = df.sample(frac=1).reset\_index(drop=True)

df.isnull().sum()

最后，得到了（9015，26）的样本数据集，根据生活经验，后面将使用下列的部分字段进行训练与预测。

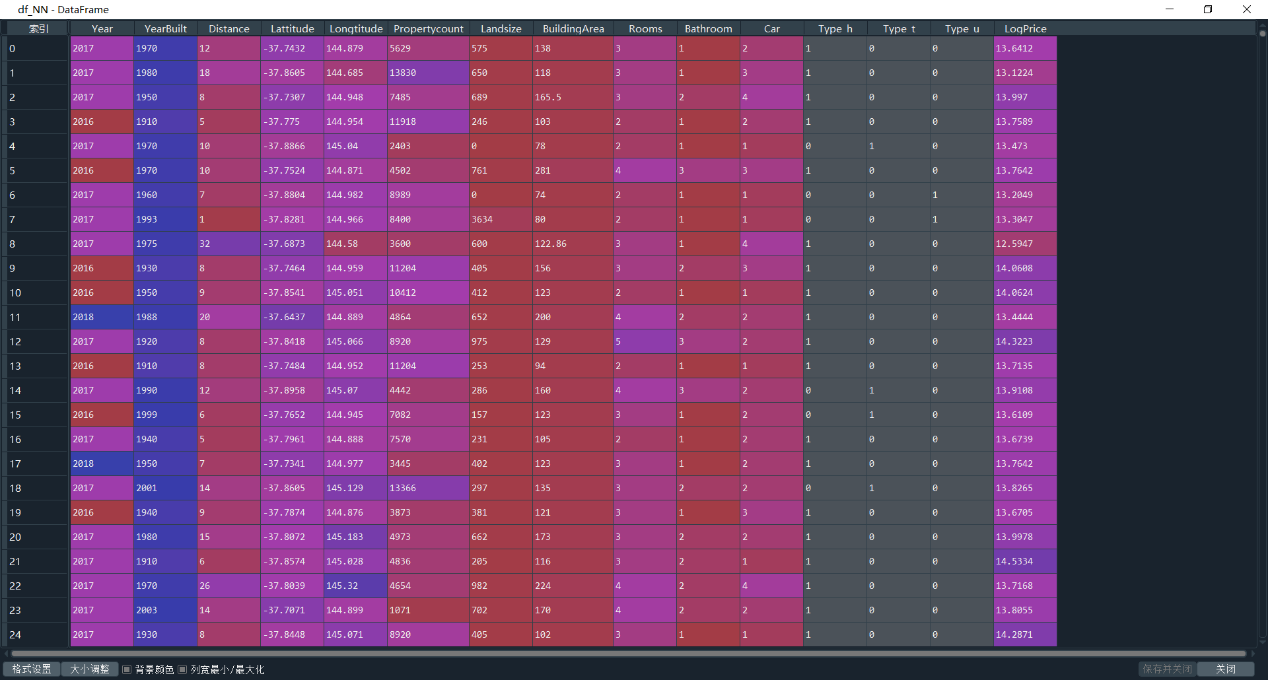
# Final dataset

df\_NN=df[['Year','YearBuilt','Distance','Lattitude','Longtitude','Propertycount',

'Landsize','BuildingArea', 'Rooms','Bathroom', 'Car','Type\_h','Type\_t','Type\_u','LogPrice']]

df\_NN.isnull().sum()

df\_NN.shape



四、划分训练集与测试集

参与回归的变量包括交易年份、建成年份、到CBD距离、纬度、经度、附属设施数、占地面积、建筑面积、房间数、浴室数量、车位数、房产类型（独热变量，每类一列），经测试交易月份的影响不大，目标变量是房价对数，80%训练集，20%测试集。

# Sample and target, train and test

X=df\_NN[['Year','YearBuilt','Distance','Lattitude','Longtitude','Propertycount',

'Landsize','BuildingArea', 'Rooms','Bathroom', 'Car','Type\_h','Type\_t','Type\_u']]

y=df\_NN['LogPrice']

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,y, test\_size = .20, random\_state=42)

对数据作标准化处理，消除各变量量纲影响。这个处理对基于决策树的随机森林等算法影响不大，但对线性回归评估各变量作用的影响较大。该处理对总体准确率的影响也不大。

# Data standardization

mean = X\_train.mean(axis=0)

X\_train -= mean

std = X\_train.std(axis=0)

X\_train /= std

X\_test -= mean

X\_test /= std

五、线性回归基线模型

这是最常见的连续变量回归模型，先看看效果如何。注意因样本随机排序，每重跑一次各模型输出的准确率都会略有不同，这里主要是了解相关的方法与模型。

可以看到，模型对测试集对数房价回归的打分为 68.7%，其实这是个R2打分，即反映了68.7%的方差。为了与深度学习的模型比较，把目标变量与预测值作指数变换回原量纲后，计算其平均绝对误差MAE，然后计算 准确率=1-MAE/样本均值，这样得到训练集与测试集的准确率分别为78.6%，68.7%，还行，因为这显然不是一个线性回归能解决的问题。

个人觉得一个模型要部署到生产环境应用，准确率需要90%以上，85%以上是勉强可用，所以其它模型要优于线性回归基线模型才有说服力。

# 线性回归模型是用于比较的基线模型

regressor = LinearRegression()

# Fit model to training data

regressor.fit(X\_train,y\_train)

# Predict

# Predicting test set results

y\_pred = regressor.predict(X\_test)

# 0.742346869078488, the original on kaggle is 0.570662925365457, great improvment

# 0.742346869078488

regressor.score(X\_test,y\_test)

# 0.742346869078488

r2\_score(y\_test,y\_pred)

plt.scatter(y\_test, y\_pred)

# Histogram of the distribution of residuals

sns.distplot((y\_test - y\_pred), fit=norm)

cdf = pd.DataFrame(data = regressor.coef\_, index = X.columns,columns = ['Coefficients'])

cdf

y\_pred\_t\_o = np.round(np.exp(regressor.predict(X\_train)),2)

y\_pred\_o = np.round(np.exp(y\_pred),2)

y\_train\_o = np.round(np.exp(y\_train),2)

y\_test\_o = np.round(np.exp(y\_test),2)

err\_t = abs(y\_train\_o - y\_pred\_t\_o)

mae\_lt = sum(err\_t)/len(err\_t)

err\_t2 = abs(y\_test\_o - y\_pred\_o)

mae\_lt2 = sum(err\_t2)/len(err\_t2)

# LinearRegression MAE train: 238152.264223516 Accuracy: 0.7810946104912015 MAE test: 252793.69442595684 Test Accuracy: 0.7728025889428027

print("LinearRegression MAE train: ",mae\_lt," Accuracy: ",1-mae\_lt/y\_train\_o.mean(),

"MAE test: ",mae\_lt2," Test Accuracy: ",1-mae\_lt2/y\_test\_o.mean())

# 画特征重要性

# https://deephub.blog.csdn.net/article/details/105487021?utm\_medium=distribute.pc\_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7Edefault-1.no\_search\_link&depth\_1-utm\_source=distribute.pc\_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7Edefault-1.no\_search\_link

for i,v in enumerate(regressor.coef\_):

print('Feature: %0d, Score: %.5f' % (i,v))

df\_importance = pd.DataFrame({"Feature":X\_train.columns,"importance":abs(regressor.coef\_)})

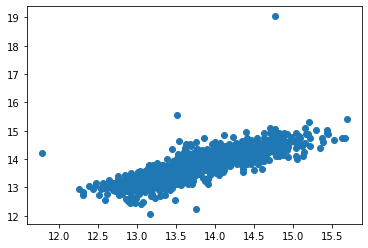
df\_importance.sort\_values(["importance"],ascending = False, inplace=True)

plt.bar(df\_importance["Feature"], df\_importance["importance"])

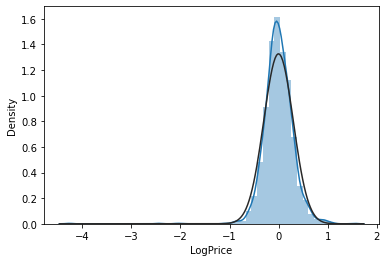
plt.xticks(rotation=45)

plt.show()

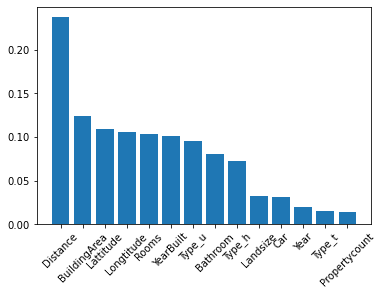
测试集拟合的效果，可以看到有明显的异常值，偏差较大。



输入变量经标准化后，目标变量残差接近正态分布，线性回归模型已充分提取输入的信息，达到了该模型的最大性能。



画特征重要性图，变量标准化后，各变量的重要性得到充分展现，该模型中到CBD距离、建筑面积、纬度、经度、房间数、建成年份、房产类型（3个独热变量累加）是最有影响的变量。



六、集成学习随机森林回归

随机森林算法可用于分类，也可用于回归，区别是其中的树是分类树还是回归树，[参阅资料1](https://www.jianshu.com/p/af6b9f15200f)，[参阅资料2](https://www.cnblogs.com/Lin-Yi/p/8972051.html)。[Kaggle上的这篇文章](https://www.kaggle.com/bidiptabikashgogoi/price-prediction-with-88-accuracy-5-model-tested)测试了5种回归算法，达到了88%的准确率，可以参考。

随机森林回归模型测试集打分为87.6%，指数变换回原值后，训练集与测试集的准确率分别为94.5%与85.3%，非常不错。可以看到，训练集的精度已经很高了，测试集精度降低说明了有过拟合，影响了泛化能力。随机森林回归的预测值是森林中各棵回归树预测值的均值，所以它不是多项式等函数式的回归，正因如此，普适能力很强，性能稳定。

# 随机森林回归模型是更好的机器学习回归模型， https://www.jianshu.com/p/af6b9f15200f

# https://www.kaggle.com/bidiptabikashgogoi/price-prediction-with-88-accuracy-5-model-tested

# Python集成回归模型 https://www.cnblogs.com/Lin-Yi/p/8972051.html

from sklearn.model\_selection import GridSearchCV

from sklearn.ensemble import RandomForestRegressor

rf = RandomForestRegressor()

params = {"max\_depth":[25,30,35,40], "n\_estimators":[42,45,48,51]}

rf\_reg = GridSearchCV(rf, params, cv = 10, n\_jobs =10)

rf\_reg.fit(X\_train, y\_train)

print(rf\_reg.best\_estimator\_)

regressor\_RF = rf\_reg.best\_estimator\_

regressor\_RF.fit(X\_train,y\_train)

y\_pred\_RF = regressor\_RF.predict(X\_test)

# 0.8468594088054577， Price对数转换后0.8855182522807963, Type作独热编码后0.8926109722380867

regressor\_RF.score(X\_test,y\_test)

# 0.8069878879987716，r2\_score稍有不同，稍为低一点。Price对数转换后0.8855182522807963，Type作独热编码后 0.8926109722380867

r2\_score(y\_test,y\_pred\_RF)

plt.scatter(y\_test, y\_pred,c = "blue",label = "Linear")

plt.scatter(y\_test, y\_pred\_RF,c = "green",label = "Random Forest")

plt.legend(loc = "upper left")

plt.show()

sns.distplot((y\_test - y\_pred\_RF), fit = norm)

y\_pred\_t\_RF\_o = np.round(np.exp(regressor\_RF.predict(X\_train)),2)

y\_pred\_RF\_o = np.round(np.exp(y\_pred\_RF),2)

y\_train\_o = np.round(np.exp(y\_train),2)

y\_test\_o = np.round(np.exp(y\_test),2)

err\_t\_RF = abs(y\_train\_o - y\_pred\_t\_RF\_o)

mae\_lt\_RF = sum(err\_t\_RF)/len(err\_t\_RF)

err\_t2\_RF = abs(y\_test\_o - y\_pred\_RF\_o)

mae\_lt2\_RF = sum(err\_t2\_RF)/len(err\_t2\_RF)

# RandomForestRegressor MAE train: 59602.10845284314 Accuracy: 0.9452147859733556 MAE test: 164940.00928056418 Test Accuracy: 0.8517607681101779

# RandomForestRegressor MAE train: 58412.18957154735 Accuracy: 0.9463085385649727 MAE test: 160344.32230726557 Test Accuracy: 0.8558911250193315

print("RandomForestRegressor MAE train: ",mae\_lt\_RF," Accuracy: ",1-mae\_lt\_RF/y\_train\_o.mean(),

"MAE test: ",mae\_lt2\_RF," Test Accuracy: ",1-mae\_lt2\_RF/y\_test\_o.mean())

# 画特征重要性

df\_importance = pd.DataFrame({"Feature":X\_train.columns,"importance":regressor\_RF.feature\_importances\_})

df\_importance.sort\_values(["importance"],ascending = False, inplace=True)

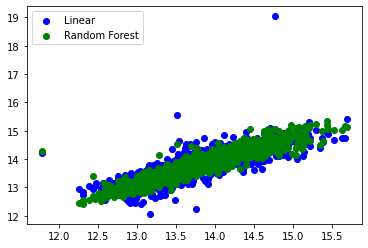
plt.bar(df\_importance["Feature"], df\_importance["importance"])

plt.xticks(rotation=45)

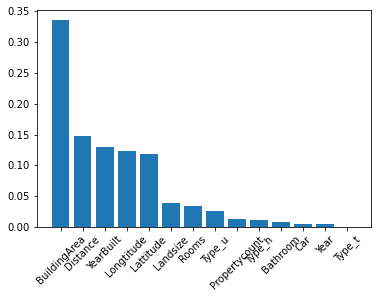
plt.show()

[Kaggle上的这个竞赛项目](https://www.kaggle.com/c/house-prices-advanced-regression-techniques/overview)提供了对各种回归算法深入的探讨，比如Lasso、Ridge（这两个是线性回归的正则化模型）、XGB、Stacking等，我还没有深入去了解，随机森林回归的性能已经很不错，相信经过正则化、堆叠等处理克服过拟合后，这条路线应该可以最终达到90%以上的投产精度要求。

拟合结果，异常点有所减少，更接近对角线，即更准确了。



建筑面积、与CBD距离、建成年份、经度、纬度是影响房价的主要因素，个人觉得这个解释可能更合理。建筑面积其实是与房间数和占地面积等相关的。



七、集成学习Stacking回归

具体可参阅[参考资料1](https://zhouchen.blog.csdn.net/article/details/89253879)，本节主要参考[参考资料2](https://www.kaggle.com/lavanyashukla01/how-i-made-top-0-3-on-a-kaggle-competition)。

通过堆叠回归模型进行二次回归，以进一步提高精度。参与测试的回归模型有GBR、LGBM、XGB、RF、SVR与Lasso，我还没有深入了解这些算法，先跑起来看看，先看看它们各自在训练集上的均方差(均方差标准差)：

lightgbm: 0.1702 (0.0101)

xgboost: 0.1796 (0.0107)

SVR: 0.2648 (0.0124)

lasso: 0.2781 (0.0157)

rf: 0.1868 (0.0099)

gbr: 0.1635 (0.0101)

在训练集中gbr模型性能最好（**下一节再介绍性能更平衡的CatBoost模型，本节写在测试CatBoost之前**），lgbm, xgb,rf其次，SVR与Lasso最后。先定义各模型，它们的超参数先通过网格搜索初步调优，以提高Stacking的精度。

from sklearn.model\_selection import KFold, cross\_val\_score, GridSearchCV

from sklearn.metrics import mean\_squared\_error

from lightgbm import LGBMRegressor

from xgboost import XGBRegressor

from sklearn.pipeline import make\_pipeline

from sklearn.preprocessing import RobustScaler

from sklearn.linear\_model import Lasso

from sklearn.svm import SVR

from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor

from mlxtend.regressor import StackingCVRegressor

# Define error metrics, mean\_squared\_error

def rmsle(y, y\_pred):

return np.sqrt(mean\_squared\_error(y, y\_pred))

# Setup cross validation folds to 12

kf = KFold(n\_splits=12, random\_state=42, shuffle=True)

# mean\_squared\_error for k folds cross validation

def cv\_rmse(model, X=X\_train, y=y\_train):

rmse = np.sqrt(-cross\_val\_score(model, X, y, scoring="neg\_mean\_squared\_error", cv=kf))

return (rmse)

# Light Gradient Boosting Regressor

# GridSearch for better hyper parameters, must list all candidates

import time

t1 = time.time()

lgbm = LGBMRegressor(n\_estimators=6000) # No n\_estimators=6000, 1.1110258102416992

# num\_threads =2 and n\_jobs=4 is best for this GPU

# lgbm = LGBMRegressor(device='gpu',gpu\_use\_dp=False, max\_bin=15, num\_threads =2, n\_estimators=6000) # No n\_estimators=6000, 29.460313081741333

params = {'learning\_rate': [0.01,0.05,0.1, 0.15], 'max\_depth': [4,5,6,7] }

lgbm\_reg = GridSearchCV(lgbm, params, cv = 5, n\_jobs=4)

lgbm\_reg.fit(X\_train,y\_train)

t2 = time.time()

print("Time: ",t2-t1)

print("Best params:{}".format(lgbm\_reg.best\_params\_))

lightgbm = LGBMRegressor(n\_estimators=6000,learning\_rate=0.01, max\_depth=5, random\_state=42)

lgb\_model\_full\_data = lightgbm.fit(X\_train, y\_train)

print(lgb\_model\_full\_data.score(X\_train,y\_train))

print(lgb\_model\_full\_data.score(X\_test,y\_test))

acc = cross\_val\_score(lightgbm, X\_train, y\_train).mean()

print(acc)

# XGBoost Regressor

# GridSearch for better hyper parameters, must list all candidates

import time

t1 = time.time()

xgbr = XGBRegressor(n\_estimators=6000)

# xgbr = XGBRegressor(tree\_method='gpu\_hist')

params = {'learning\_rate': [0.01,0.05,0.1, 0.15], 'max\_depth': [4,5,6,7] }

xgbr\_reg = GridSearchCV(xgbr, params, cv = 5)

xgbr\_reg.fit(X\_train,y\_train)

t2 = time.time()

print("Time: ",t2-t1)

print("Best params:{}".format(xgbr\_reg.best\_params\_))

xgboost = XGBRegressor(n\_estimators=6000, learning\_rate=0.01, max\_depth=6, random\_state=42)

# Gradient Boosting Regressor

# GridSearch for better hyper parameters, must list all candidates

params = {'learning\_rate': [0.01,0.05,0.1, 0.15], 'max\_depth': [4,5,6,7] }

gbr = GradientBoostingRegressor(n\_estimators=6000)

gbr\_reg = GridSearchCV(gbr, params, cv = 5, n\_jobs =12)

gbr\_reg.fit(X\_train,y\_train)

print("Best params:{}".format(gbr\_reg.best\_params\_))

gbr = GradientBoostingRegressor(n\_estimators=6000, learning\_rate=0.01, max\_depth= 6, random\_state=42)

# Random Forest Regressor

# GridSearch for better hyper parameters, must list all candidates

rf = RandomForestRegressor(n\_estimators=1200)

params = {"max\_depth":[10,15,20,25]}

rf\_reg = GridSearchCV(rf, params, cv = 5, n\_jobs =12)

rf\_reg.fit(X\_train, y\_train)

print(rf\_reg.best\_estimator\_)

rf = RandomForestRegressor(n\_estimators=1200, max\_depth=25)

# Support Vector Regressor

# GridSearch for better hyper parameters, must list all candidates

svr = SVR()

params = {'C': [10,20,30,40], 'gamma': [0.00003,0.0003,0.003,0.03] }

svr\_reg = GridSearchCV(svr, params, cv = 5, n\_jobs =10)

svr\_reg.fit(X\_train,y\_train)

print("Best params:{}".format(svr\_reg.best\_params\_))

svr = make\_pipeline(RobustScaler(), SVR(C= 10, gamma=0.03))

# Lasso Regressor

# GridSearch for better hyper parameters, must list all candidates

lasso =Lasso()

parameters= {'alpha':[x for x in [0.0005,0.001,0.01,0.1,1]]}

Lasso\_reg=GridSearchCV(lasso, param\_grid=parameters, cv = 5, n\_jobs =16)

Lasso\_reg.fit(X\_train,y\_train)

print("The best value of Alpha is: ",Lasso\_reg.best\_params\_,Lasso\_reg.best\_score\_)

lasso =Lasso(alpha=0.0005)

scores = {}

score = cv\_rmse(lightgbm)

print("lightgbm: {:.4f} ({:.4f})".format(score.mean(), score.std()))

scores['lgb'] = (score.mean(), score.std())

score = cv\_rmse(xgboost)

print("xgboost: {:.4f} ({:.4f})".format(score.mean(), score.std()))

scores['xgb'] = (score.mean(), score.std())

score = cv\_rmse(svr)

print("SVR: {:.4f} ({:.4f})".format(score.mean(), score.std()))

scores['svr'] = (score.mean(), score.std())

score = cv\_rmse(lasso)

print("lasso: {:.4f} ({:.4f})".format(score.mean(), score.std()))

scores['lasso'] = (score.mean(), score.std())

score = cv\_rmse(rf)

print("rf: {:.4f} ({:.4f})".format(score.mean(), score.std()))

scores['rf'] = (score.mean(), score.std())

score = cv\_rmse(gbr)

print("gbr: {:.4f} ({:.4f})".format(score.mean(), score.std()))

scores['gbr'] = (score.mean(), score.std())

因为SVR与Lasso模型的精度不高，只堆叠了GBR、LGBM、XGB、RF 4个模型。使用精度较高的GBR为元数据二次回归的算法。

然后各模型独立拟合与预测，并打印它们的得分，可以看到GBR的精度较高，它与LGBM、XGB在测试集上的得分都接近90%，比RF与SVR都高，Stacking后，模型在训练集上的准确度是95.3%，测试集上是91.2%，精度再提高了1%左右，可以看到比随机森林回归有较大的改善，但还是有轻微的过拟合(注意因样本随机排序，每次重跑的结果会略有出入)。

# Stack up all the models above, optimized using xgboost

# stack\_gen = StackingCVRegressor(regressors=(xgboost, lightgbm, svr, lasso, gbr, rf),

stack\_gen = StackingCVRegressor(regressors=(xgboost, lightgbm, gbr,rf),

meta\_regressor=gbr, # 用本数据集中性能最好的GBR模型作元数据回归

use\_features\_in\_secondary=True)

print('stack\_gen')

stack\_gen\_model = stack\_gen.fit(X\_train, y\_train)

print(stack\_gen\_model.score(X\_train,y\_train))

print(stack\_gen\_model.score(X\_test,y\_test))

print('lightgbm')

lgb\_model\_full\_data = lightgbm.fit(X\_train, y\_train)

print(lgb\_model\_full\_data.score(X\_train,y\_train))

print(lgb\_model\_full\_data.score(X\_test,y\_test))

print('xgboost')

xgb\_model\_full\_data = xgboost.fit(X\_train, y\_train)

print(xgb\_model\_full\_data.score(X\_train,y\_train))

print(xgb\_model\_full\_data.score(X\_test,y\_test))

print('RandomForest')

rf\_model\_full\_data = rf.fit(X\_train, y\_train)

print(rf\_model\_full\_data.score(X\_train,y\_train))

print(rf\_model\_full\_data.score(X\_test,y\_test))

print('GradientBoosting')

gbr\_model\_full\_data = gbr.fit(X\_train, y\_train)

print(gbr\_model\_full\_data.score(X\_train,y\_train))

print(gbr\_model\_full\_data.score(X\_test,y\_test))

print('Svr')

svr\_model\_full\_data = svr.fit(X\_train, y\_train)

print(svr\_model\_full\_data.score(X\_train,y\_train))

print(svr\_model\_full\_data.score(X\_test,y\_test))

print('Lasso')

lasso\_model\_full\_data = lasso.fit(X\_train, y\_train)

print(lasso\_model\_full\_data.score(X\_train,y\_train))

print(lasso\_model\_full\_data.score(X\_test,y\_test))

输出：

stack\_gen

0.9847646566694989

0.8958598532574242

lightgbm

0.9664793384090974

0.8965985785483989

xgboost

0.9792174097323387

0.899801219120223

RandomForest

0.9833117332959057

0.8764745616232377

GradientBoosting

0.9901047516413873

0.8940361382726137

Svr

0.8914315230077958

0.8539980041617118

Lasso

0.739710591933525

0.6933811352865575

[参考资料2](https://www.kaggle.com/lavanyashukla01/how-i-made-top-0-3-on-a-kaggle-competition)最后用上面的模型组成一个混合的模型，测试集得分可以稳定在91%以上。混合模型的目标不是得分最高，而是克服过拟合的健壮性，还是可以参考的。

# Blend models in order to make the final predictions more robust to overfitting

def blended\_predictions(X):

return ((0.01 \* lasso\_model\_full\_data.predict(X)) + \

(0.03 \* svr\_model\_full\_data.predict(X)) + \

(0.08 \* gbr\_model\_full\_data.predict(X)) + \

(0.07 \* xgb\_model\_full\_data.predict(X)) + \

(0.06 \* lgb\_model\_full\_data.predict(X)) + \

(0.05 \* rf\_model\_full\_data.predict(X)) + \

(0.70 \* stack\_gen\_model.predict(np.array(X))))

# Get final precitions from the blended model

blended\_score = rmsle(y\_train, blended\_predictions(X\_train))

scores['blended'] = (blended\_score, 0)

print('RMSLE score on train data:')

print(blended\_score)

y\_train\_pred\_b = blended\_predictions(X\_train)

y\_val\_pred\_b = blended\_predictions(X\_test)

y\_pred\_t\_blend\_o = np.round(np.exp(y\_train\_pred\_b),2)

y\_pred\_blend\_o = np.round(np.exp(y\_val\_pred\_b),2)

y\_train\_o = np.round(np.exp(y\_train),2)

y\_test\_o = np.round(np.exp(y\_test),2)

err\_t\_blend = abs(y\_train\_o - y\_pred\_t\_blend\_o)

mae\_lt\_blend = sum(err\_t\_blend)/len(err\_t\_blend)

err\_t2\_blend = abs(y\_test\_o - y\_pred\_blend\_o)

mae\_lt2\_blend = sum(err\_t2\_blend)/len(err\_t2\_blend)

print("blend Regressor MAE train: ",mae\_lt\_blend," Accuracy: ",1-mae\_lt\_blend/y\_train\_o.mean(),

"MAE test: ",mae\_lt2\_blend," Test Accuracy: ",1-mae\_lt2\_blend/y\_test\_o.mean())

r2\_score(y\_test,y\_val\_pred\_b)

In [**63**]: r2\_score(y\_test,y\_val\_pred\_b)

Out[**63**]: 0.8993054596596561

测试集上的拟合结果比较见下图，可见Stacking的精度又提高了一点。

y\_train\_pred\_s = stack\_gen\_model.predict(X\_train)

y\_val\_pred\_s = stack\_gen\_model.predict(X\_test)

plt.scatter(y\_test, y\_pred,c = "blue",label = "Linear")

plt.scatter(y\_test, y\_pred\_RF,c = "green",label = "Random Forest")

plt.scatter(y\_test, y\_val\_pred\_s, c = "red",label = "Stacking")

plt.title("Regressions on LogPrice")

plt.xlabel("Test values")

plt.ylabel("Predicted values")

plt.legend(loc = "upper left")

plt.show()

sns.distplot((y\_test - y\_val\_pred\_s), fit = norm)

stack\_gen\_model.score(X\_test,y\_test)

r2\_score(y\_test,y\_val\_pred\_s)

r2\_score(y\_train,y\_train\_pred\_s)

y\_pred\_t\_stack\_o = np.round(np.exp(y\_train\_pred\_s),2)

y\_pred\_stack\_o = np.round(np.exp(y\_val\_pred\_s),2)

y\_train\_o = np.round(np.exp(y\_train),2)

y\_test\_o = np.round(np.exp(y\_test),2)

err\_t\_stack = abs(y\_train\_o - y\_pred\_t\_stack\_o)

mae\_lt\_stack = sum(err\_t\_stack)/len(err\_t\_stack)

err\_t2\_stack = abs(y\_test\_o - y\_pred\_stack\_o)

mae\_lt2\_stack = sum(err\_t2\_stack)/len(err\_t2\_stack)

print("stack Regressor MAE train: ",mae\_lt\_stack," Accuracy: ",1-mae\_lt\_stack/y\_train\_o.mean(),

"MAE test: ",mae\_lt2\_stack," Test Accuracy: ",1-mae\_lt2\_stack/y\_test\_o.mean())

In [**72**]: r2\_score(y\_train,y\_train\_pred\_s)

Out[**72**]: 0.9847646566694989

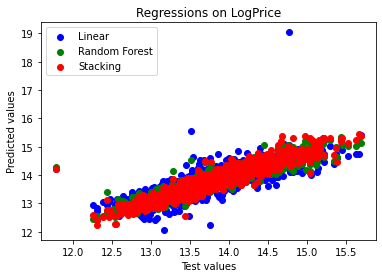
In [**73**]: r2\_score(y\_test,y\_val\_pred\_s)

Out[**73**]: 0.8958598532574242

In [**71**]: print("stack Regressor MAE train: ",mae\_lt\_stack," Accuracy: ",1-mae\_lt\_stack/y\_train\_o.mean(),

    ...: "MAE test: ",mae\_lt2\_stack," Test Accuracy: ",1-mae\_lt2\_stack/y\_test\_o.mean())

stack Regressor MAE train: 56683.99558097635 Accuracy: 0.947989799278768 MAE test: 151803.7307820299 Test Accuracy: 0.8626088532010626



调参是个技术活，要掌握算法才能做好，我只是随便试试。[参考资料2](https://www.kaggle.com/lavanyashukla01/how-i-made-top-0-3-on-a-kaggle-competition)中的参数应该是作者经过大量尝试得到的，主要是2个特点：低学习率与超多的估计器（过饱和的结果是更稳定的拟合，代价就是计算量很大，耗时大增）。这里用网格搜索尝试从上面的参数出发调整，可以看到除Lasso外各算法的性能都提高了，在训练集上都达到了非常高的精度，**过拟合现象都比较明显**，GBR、XGB、LGB在测试集上都超过了91%，Stacking后训练集有提高，测试集略有下降,性能还算稳定。所以还是要寻求其它办法克服过拟合的问题，混合模型也许是个有效的方案。

网格搜索的缺点是要列举所有可能的超参数组合逐个测试，这其实是个超耗时的体力活，[参考资料3](https://www.kaggle.com/pavansanagapati/automated-hyperparameter-tuning)与[参考资料4](https://blog.csdn.net/FontThrone/article/details/85100328)给出了用hyperopt包自动搜索最佳超参数的方法，给定超参数的取值范围，通过贝叶斯优化有效减少搜索的次数，一觉醒来，程序已找到了更好的超参数组合。注意这样得到的超参数是与训练集高度相关的，当训练数据增加或改变了，需要重新寻找。**下一节将详细探讨一下**。

前面的调参过程中，同时也用交叉验证的准确率来衡量模型的性能，可以看到模型交叉验证（默认5折）的准确率>89.9%，非常接近90%，但小于测试集91%以上的准确率，说明模型在测试集新数据上的性能比验证集（从训练集分割而来）上要好。如果以新数据上90%的准确率为标准，那么Stacking模型和混合模型是可以实际应用的，回归预测这条路线是可以走通的。但**Stacking模型和混合模型的计算量非常大，速度比较慢，在实际的应用中建议谨慎考虑**。后面将通过贝叶斯优化让CatBoost、LGBM等单个模型在新数据集上达到90%的标准，克服过拟合的问题，从而平衡精度与速度的要求。

Scikit-Learn不支持GPU，其自带的GBR、RF、SVM、Lasso等算法都不能使用GPU。调参与大数据集上的速度都比较慢。[H2O.ai](https://www.h2o.ai/)提供了开源的Scikit-Learn接口的软件包[h2o4gpu](https://pypi.org/project/h2o4gpu/)，支持Python与R语言，提供了GPU版的Scikit-Learn替代算法，目前只有Linux版的（大多数GPU云端环境都是Linux），并且只支持Python3.6与3.7，通过接口包封装Pyhton的实现去支持R语言。LGBM与XGB都支持用GPU，参考[这篇文章](https://blog.csdn.net/weixin_44405644/article/details/108813007)，通过whl文件安装的xgboost包已经支持GPU，安装测试也通过了，不过在本例中性能反而不如CPU版，上面自动搜索参数，每次迭代GPU版要86.38s，CPU版32.97s。 参考[这篇文章](https://zhuanlan.zhihu.com/p/55259112)，因为LightGBM GPU版要重新编译安装，后面再研究一下。

八、深度学习基线模型DNN

《Python深度学习》与《R语言深度学习》上的DNN基线模型，用的是波士顿房价数据集，没有经纬度坐标。这里换成墨尔本数据集跑一下。为了跑Keras与Tensorflow，数据需要转换一下，前面已经做了标准化，这里直接用即可。

# copy to tensor dataset

train\_data = X\_train.copy()

test\_data = X\_test.copy()

train\_targets = np.reshape(np.array(y\_train.copy()),(-1,1))

test\_targets = np.reshape(np.array(y\_test.copy()),(-1,1))

train\_data.shape

test\_data.shape

train\_targets

DNN基线模型使用了2个64单元的稠密层及一个1单元的输出层，以MAE为评价指标，跑了几次后发觉性能很不稳定，这是跑得比较好的一次：模型在训练集的R2为86.5%，测试集R2为82.8%，指数变换回原量纲后，训练集的准确率为84.3%，测试集准确率为74%，可以看到精度损失不大泛化性能不错。

from tensorflow import keras

from tensorflow.keras import layers

# 只有稠密层的神经网络DNN， 神经网络的基础模型，后面会尝试卷积神经网络

# Building your model, only dense layer

def build\_model():

model = keras.Sequential([

layers.Dense(64, activation="relu"),

# layers.Dropout(0.5),

layers.Dense(64, activation="relu"),

# layers.Dropout(0.5),

layers.Dense(1)

])

model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])

return model

# Validating your approach using K-fold validation

k = 4

num\_val\_samples = len(train\_data) // k

# Saving the validation logs at each fold

num\_epochs = 200

all\_mae\_histories = []

for i in range(k):

print(f"Processing fold #{i}")

val\_data = train\_data[i \* num\_val\_samples: (i + 1) \* num\_val\_samples]

val\_targets = train\_targets[i \* num\_val\_samples: (i + 1) \* num\_val\_samples]

partial\_train\_data = np.concatenate(

[train\_data[:i \* num\_val\_samples],

train\_data[(i + 1) \* num\_val\_samples:]],

axis=0)

partial\_train\_targets = np.concatenate(

[train\_targets[:i \* num\_val\_samples],

train\_targets[(i + 1) \* num\_val\_samples:]],

axis=0)

model = build\_model()

history = model.fit(partial\_train\_data, partial\_train\_targets,

validation\_data=(val\_data, val\_targets),

epochs=num\_epochs, batch\_size=32, verbose=1)

mae\_history = history.history["val\_mae"]

all\_mae\_histories.append(mae\_history)

# Building the history of successive mean K-fold validation scores

average\_mae\_history = [

np.mean([x[i] for x in all\_mae\_histories]) for i in range(num\_epochs)]

# Plotting validation scores, 迭代超过150次后，MAE的改进很小，收敛很慢。

plt.plot(range(1, len(average\_mae\_history) + 1), average\_mae\_history)

plt.xlabel("Epochs")

plt.ylabel("Validation MAE")

plt.show()

# Plotting smoothed validation scores, excluding the first 10 data points

# 将每个数据点的值替换为前面数据点的指数移动平均值

def smooth\_curve(points, factor=0.9):

smoothed\_points = []

for point in points:

if smoothed\_points:

previous = smoothed\_points[-1]

smoothed\_points.append(previous \* factor + point \* (1 - factor))

else:

smoothed\_points.append(point)

return smoothed\_points

# drop the 10 points in the front

smooth\_mae\_history = smooth\_curve(average\_mae\_history[10:])

plt.plot(range(1, len(smooth\_mae\_history) + 1), smooth\_mae\_history)

plt.xlabel("Epochs")

plt.ylabel("Validation MAE")

plt.show()

# Training the final model

model = build\_model()

model.fit(train\_data, train\_targets,

epochs=200, batch\_size=32, verbose=1)

test\_mse\_score, test\_mae\_score = model.evaluate(test\_data, test\_targets)

# Generating predictions on new data

predictions = model.predict(test\_data)

predictions[0]

test\_targets[0]

# 0.5385328194827406

# 0.8134494671523879

r2\_score(y\_test,predictions)

# plot the predictions, better than LinearRegression

plt.scatter(y\_test, y\_pred,c = "blue",label = "Linear")

plt.scatter(test\_targets, predictions, c = "orange", label = "DNN")

plt.title("Regressions on LogPrice")

plt.xlabel("Test values")

plt.ylabel("Predicted values")

plt.legend(loc = "upper left")

plt.show()

train\_targets\_o = np.round(np.exp(train\_targets),2)

test\_targets\_o = np.round(np.exp(test\_targets),2)

predictions\_t\_o = np.round(np.exp(model.predict(train\_data)),2)

predictions\_o = np.round(np.exp(predictions),2)

# 0.6916433392638972

r2\_score(train\_targets\_o,predictions\_t\_o)

# 0.6408760675521747

r2\_score(test\_targets\_o,predictions\_o)

err\_t\_DNN = abs(train\_targets\_o - predictions\_t\_o)

mae\_lt\_DNN = sum(err\_t\_DNN)/len(err\_t\_DNN)

err\_t2\_DNN = abs(test\_targets\_o - predictions\_o)

mae\_lt2\_DNN = sum(err\_t2\_DNN)/len(err\_t2\_DNN)

# Keras DNN MAE train: [170840.17254337] Accuracy: [0.84402656] MAE test: [186165.20406094] Test Accuracy: [0.82811673]

print("Keras DNN MAE train: ",mae\_lt\_DNN," Accuracy: ",1-mae\_lt\_DNN/train\_targets\_o.mean(),

"MAE test: ",mae\_lt2\_DNN," Test Accuracy: ",1-mae\_lt2\_DNN/test\_targets\_o.mean())

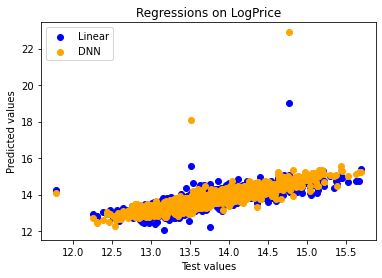
In [**87**]: test\_mse\_score, test\_mae\_score = model.evaluate(test\_data, test\_targets)

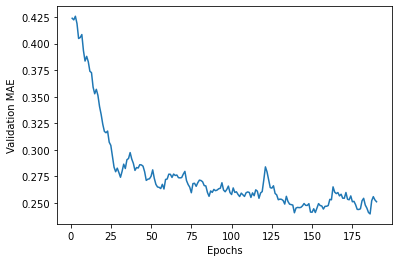
57/57 [==============================] - 0s 1ms/step - loss: 0.0956 - mae: 0.1707

In [**92**]: r2\_score(y\_test,predictions)

Out[**92**]: 0.6753726227389214

拟合效果可以看到离群点比较多，这应该是该DNN结构性能不稳定的原因。深度学习基线DNN模型比线性回归基线模型性能要稍好一点。





对于深度学习，我完全是菜鸟，关键应该是找到合适的神经网络结构，因为卷积神经网络CNN善于发现空间中的局部结构，也许可以尝试一下一维与二维的卷积神经网络。

Kaggle上的各路大神们大概可以提供一些参考的方案，全球化时代有全球的资源可资利用，实在是一件好事，让我们擦一擦神灯吧。

第二节 超参数调整 贝叶斯优化

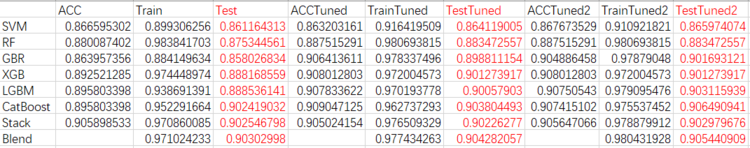
超参数是机器学习模型在训练中自己无法学习到的参数，需要在训练前指定。它们对模型的性能有重要的影响，所以超参数优化是模型落地应用之前必经的环节。有人说一个模型输出垃圾还是金子，往往取决于它的参数，我的水平有限，从下表可以看到优化只是改善了0.5%~1%的一点点，但这已经足以让一些模型跨过90%准确率的门槛，达到优秀的水准。从前一节可以看到，提高准确率的关键措施，还是找到黄金特征（经纬度坐标）和有效的预处理变换（缺失值补缺、排除异常、正态化、标准化）等。而超参数优化，则可以让模型发挥出最大的性能。

一、先看看调整前后的效果对比

下表是各模型调整前后的两组性能数据，ACC是训练集5折交叉验证的准确率，即训练集划分为5份，取4份训练1份验证，依次循环后验证集上5次准确率的平均值，它反映了模型抗过拟合的能力，后面将作为模型的损失函数指标用于超参数优化。选这个指标是因为它决定了模型的泛化能力，即在新数据集上应用的效果。Train是模型在训练集上的准确率，它反映了模型拟合的程度，Test是模型在测试集上的准确率，它反映了模型在新数据集上的泛化能力。这3个指标都等价于R2统计量，即模型反映了样本方差的百分比。第一组用模型默认的参数，带Tuned后缀的是调参后相应的指标，这两组是稳定的；Tuned2后缀的是直接用测试集准确率负值作损失函数的极限情况，泛化能力是不稳定的。

从下表可以看到，调参前一些模型如RF随机森林回归Train与ACC之差达到了10%，Test略小于ACC，这说明过拟合现象比较明显。CatBoost回归Train与ACC之差约为5%，Test略大于ACC，抗过拟合与泛化能力都不错。调参后在新数据上的准确率都有所提高，CatBoost、LGBM、XGB三种Kaggle上主流的Gradient Boosting模型都跨过了90%的门槛，Bagging模型RF、GBR还差一点点。调参后的堆叠模型Blend泛化性能最好，准确率达到了90.4%。

堆叠模型Stack是CatBoost、LGBM、XGB、GBR、RF五种基于决策树模型的堆叠，Blend是直接把上面5种模型（不含GBR，加入后准确率下降，加RF后抗过拟合能力更好）按权重线性堆叠，没有计算它的ACC，见下图。选择基于决策树的两类模型是因为单个运行的时候准确率与泛化能力都比其它模型好，SVM只是作为非树模型对比参考，没有加入Stack和Blend中。



# Stack up all the models above, optimized using catboost

stack\_gen\_best = StackingCVRegressor(regressors=(gbr\_best,xgb\_best,lgbm\_best,cat\_best,rf\_best),

meta\_regressor=cat\_best, # 用本数据集中性能最好的cat\_best模型作元数据回归

use\_features\_in\_secondary=True)

print("stack\_gen")

t1 = time.time()

stack\_gen\_model\_best = stack\_gen\_best.fit(train\_X, train\_y)

acc = cross\_val\_score(stack\_gen\_model\_best, train\_X, train\_y).mean()

t2= time.time()

# 578.914528131485

print(t2-t1)

print('acc: ',acc)

print('Train R2: ',stack\_gen\_model\_best.score(train\_X,train\_y))

print('Test R2: ', stack\_gen\_model\_best.score(valid\_X,valid\_y))

# Blend models in order to make the final predictions more robust to overfitting

def blended\_predictions2(X):

return (

(0.09 \* cat\_model\_best.predict(X)) + \

(0.08 \* lgb\_model\_best.predict(X)) + \

(0.07 \* xgb\_model\_best.predict(X)) + \

#(0.06 \* gbr\_model\_best.predict(X)) +

(0.06 \* rf\_model\_best.predict(X)) +

(0.70 \* stack\_gen\_model\_best.predict(np.array(X))))

print("Blend")

print('Train R2: ',r2\_score(train\_y, blended\_predictions2(train\_X)))

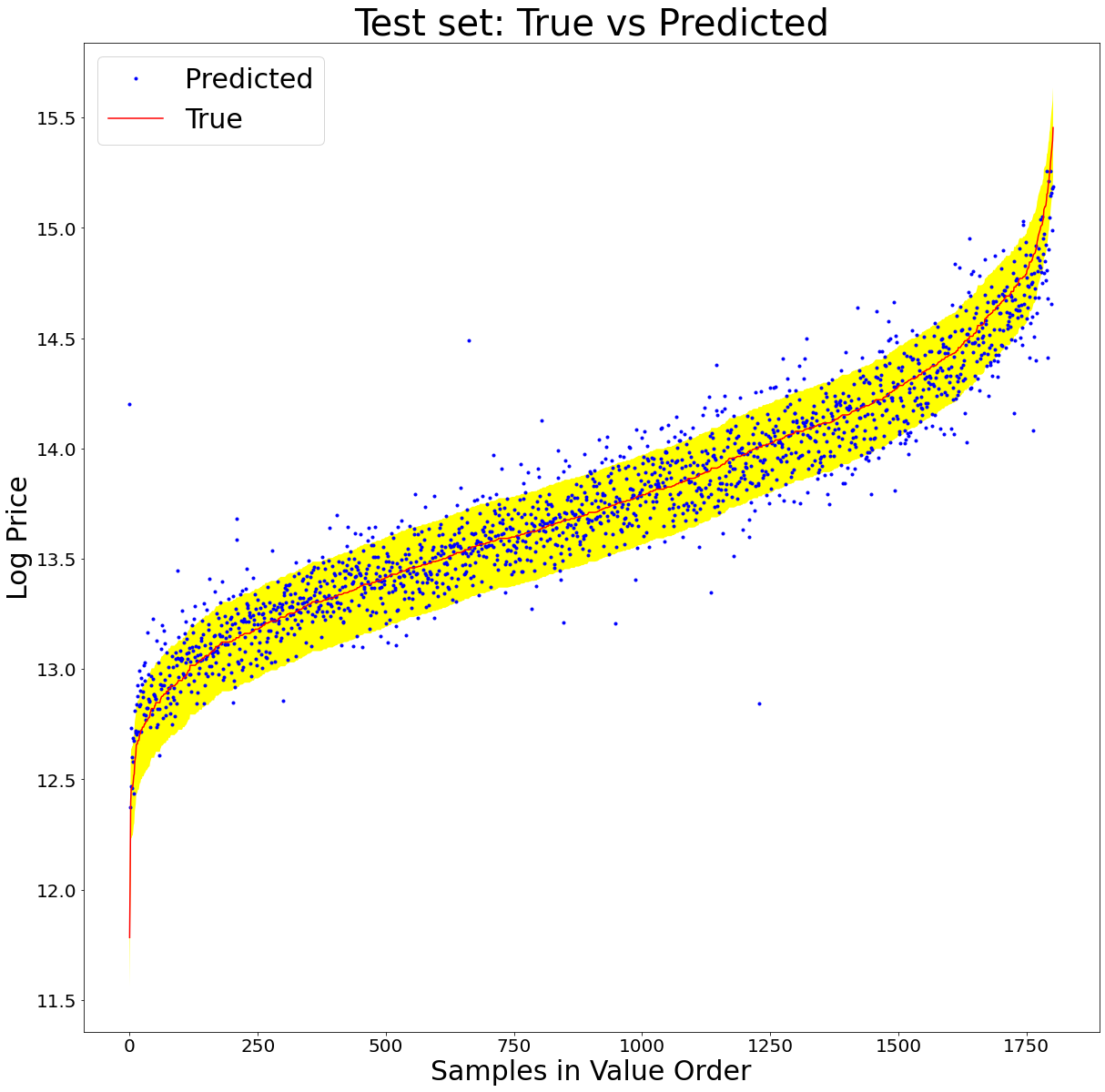
print('Test R2: ',r2\_score(valid\_y, blended\_predictions2(valid\_X)))

模型的性能与具体的数据集及参数密切相关。本人的水平有限，所以上表不一定反映了各算法模型真实的性能，这不是一个测评，只是个人试用一下。都是很优秀的开源产品，都足以落地应用解决现实中的问题。我的高数、概率论与数理统计、线性代数等都早已还给老师了，所以只是大致了解算法的原理，目标是学会使用。调参也是用hyperopt包的贝叶斯优化这样的黑匣子傻瓜式方法，只了解原理，学会使用，然后交给工具去完成。贝叶斯优化是概率模型，所以得到的也不一定是最优的参数。也就是说，试试而已，大家参考一下即可，欢迎讨论，但千万别跟我较真。

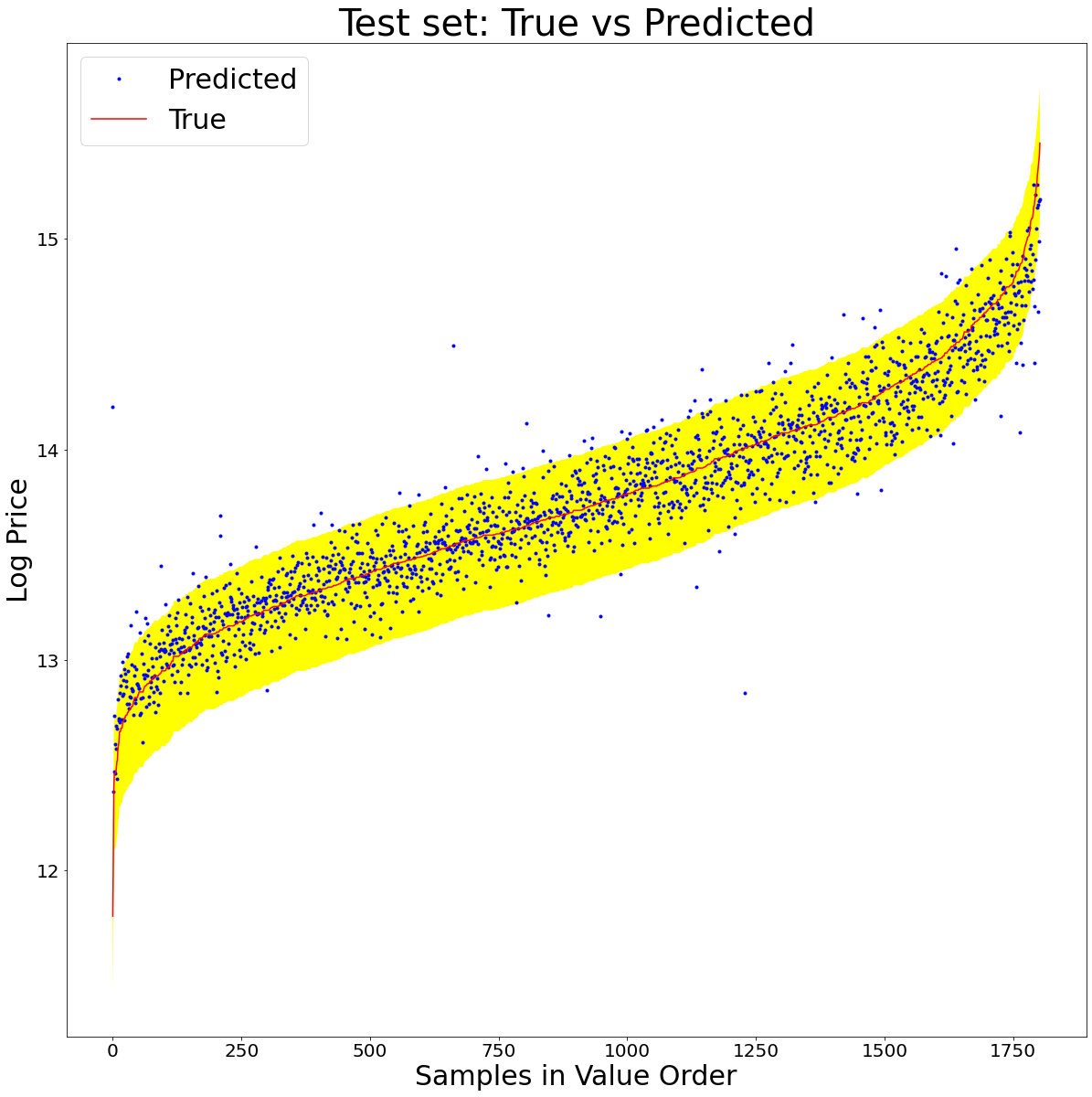
二、算法的选择

具体在什么场景中选择用什么算法，要看对速度、准确率、泛化能力等的具体要求，平衡各方面去考虑。具体到房价预测，如果是用于二手房交易价格风控的话，88%与90%的准确率差别可能并不大，泛化能力可能是更重要的。比如可以划定一个适当的预测准确率基线，70%之类，准确率在70%以下的，挑出来了解一下，可能是模型的问题，也可能是真的有问题。如果真的有问题的样本集中于某岗某人，那就是一个需要处理的风险点了。这是个高风险领域，常出问题，需要艺术的处理。这条线上难免要做点人情，法不责众，全国恐怕都一样。在现有数据上训练模型等于承认现状，而划线处理则可以识别出真正的高风险点预防和处理。

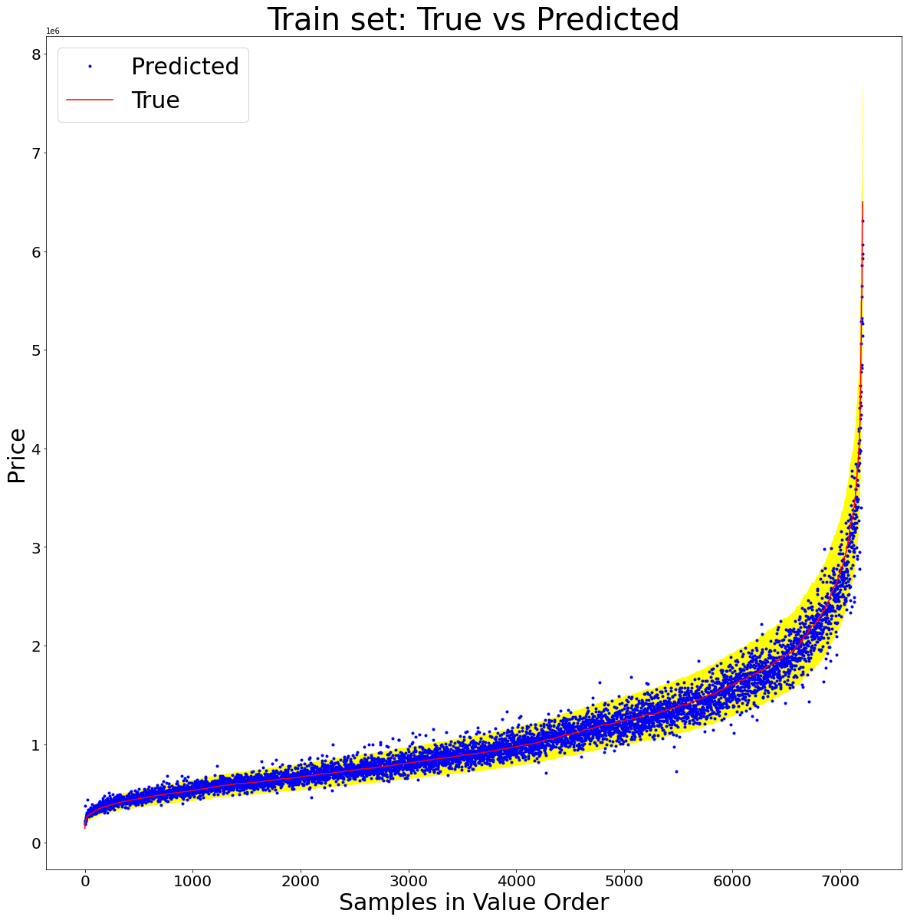
比如下图在测试集上的预测，黄色区域是正负20%的误差范围，可以见到一些明显偏离真实值红线的点，那是模型的问题，是要加以甄别的。如果是事前审核，则可以根据坐标调半径几百米范围内的其它交易记录对比详细了解一下。

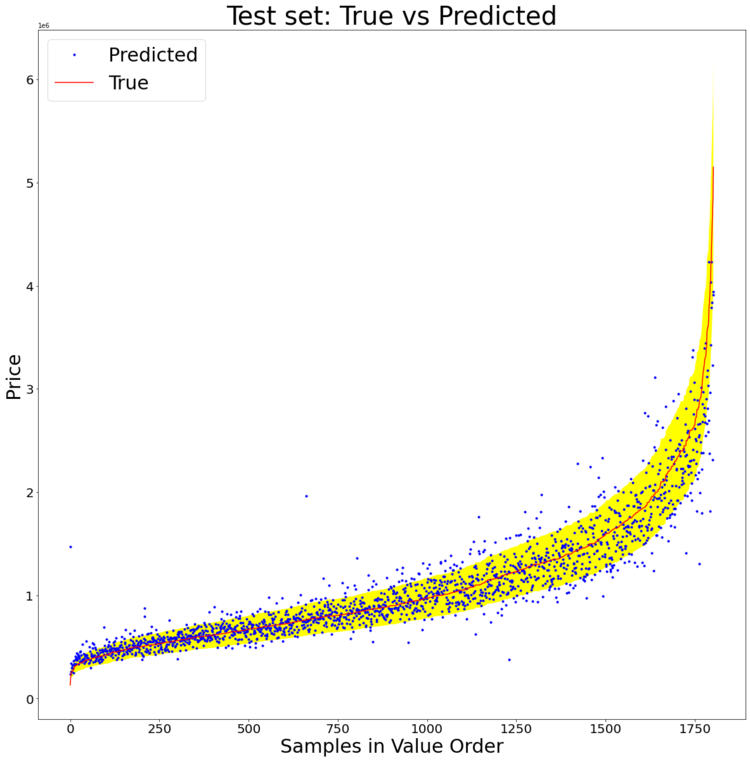


误差范围调整到正负30%后，黄色区域外的预测异常点明显减少。



下图是训练集与测试集真实值拟合图，通过指数运算恢复为真实值后，训练集上维持了97%的准确率，测试集上为88%，可以看到指数运算放大了高价段的误差，准确率拉低了2%左右，还不错，落地应用是没有问题的。





从上表可见，几个梯度提升决策树GBDT模型CatBoost、LGBM、XGB、GBR的准确率与泛化能力都比RF要好，从下图的原理图可知，GBDT模型是串行训练的，Bagging模型是并行训练的，速度比GBDT模型要快。所以在满足精度和泛化能力要求的情况下，RF等Bagging模型也是有它们适用的场景的。上面几个GBDT模型的另一个优势是支持GPU，在比较大的数据集上会有比较好的性能（Scikit-Learn不支持GPU，所以自带的GBR算法也不支持）。CatBoost、LGBM、XGB都同时支持Python、R等多种语言及多个操作系统平台，这也是一个优势。根据后面的参考资料，本文的堆叠模型有可以改善的地方，但一般来说堆叠模型计算量太大，速度较慢，竞赛则可以，落地应用就不建议了。

这里补充说明一下，LGBM是微软的开源产品，XGB是华盛顿大学的开源产品，CatBoost是俄罗斯最大的搜索引擎Yandex的开源产品。当篮子里有多个选择时，我们就可以在不同的应用场景中有效应对各种技术以外的风险，系统集成时可以设计成算法热插拔的架构以方便切换。



[参考资料1](https://www.cnblogs.com/lvdongjie/p/11393140.html)和[参考资料2](https://blog.csdn.net/aaa_aaa1sdf/article/details/81391735)简要介绍了Bagging/RandomForest和Boosting/GBDT的算法要点和主要区别。

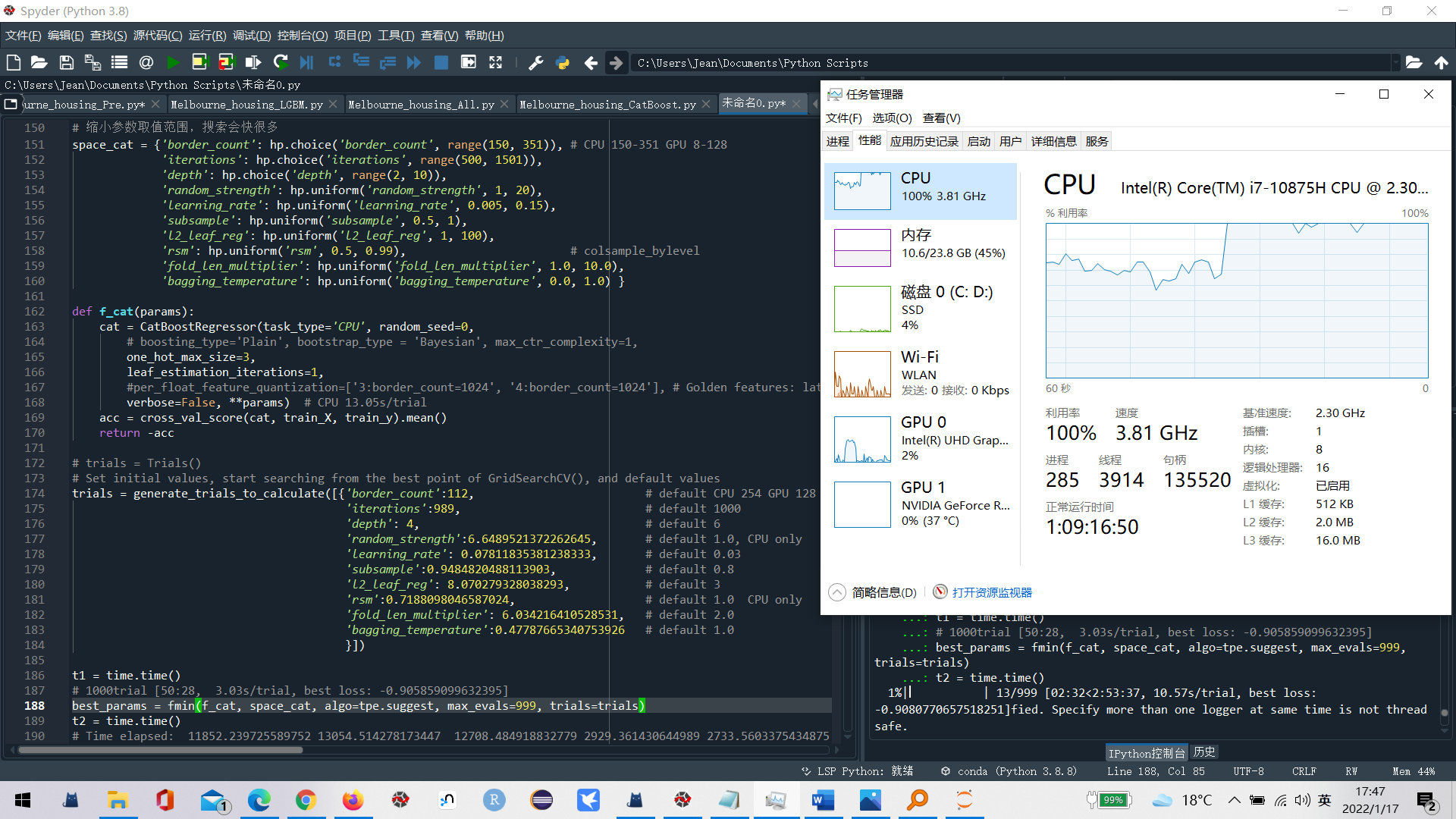
[参考资料3](https://segmentfault.com/a/1190000021373632?utm_source=tag-newest)简要介绍了CatBoost、LGBM、XGB三种主流Gradient Boosting模型的主要特点。

[参考资料4](https://zhuanlan.zhihu.com/p/25836678)简要介绍了Stacking的原理，[参考资料5](https://blog.csdn.net/zhouchen1998/article/details/89253879)介绍了Python的实例。

三、超参数优化

给定一个损失函数，比如交叉验证准确率ACC，Scikit-Learn自带的网格搜索可以在较小的参数空间内找到最优的超参数组合（损失函数极值），随机搜索可以在更大一点的空间内以较少的步数找到一些更好参数组合，但不是确定的，要看运气。贝叶斯优化则通过高斯过程建立的动态概率模型，参考过往的搜索结果以较少的步数在比较大的参数空间内（20维以内效果较好），通过确定的过程找到更优的参数组合。机器学习训练都是比较耗时耗资源的，所以有效减少优化过程中训练的消耗是很有意义的。对于机器学习算法来说，大多数情况下，我们不知道超参数与ACC等损失函数指标之间的确切函数关系，或者象神经网络那样，根本就没有确切的函数关系，即没有办法通过求导（梯度下降）来求得损失函数的极值，这样就要用贝叶斯优化了。具体的数学推导已经看不懂了，知道原理，学会使用即可。具体可参阅[参考资料1](https://www.cnblogs.com/marsggbo/p/9866764.html)，[参考资料2](https://zhuanlan.zhihu.com/p/76269142)等。这里有一个[完整的应用例子](https://www.cnblogs.com/wmx24/p/10025600.html)。

从前面的性能表格可以看到，CatBoost、LGBM、XGB、RF四种模型在训练集上都有相当高的准确率，3个95%以上，而交叉验证及测试集上准确率都有较大的损失，存在不同程度的过拟合现象。所以调参主要是看能在多大程度上克服过拟合。先看看CatBoost，从读入上一节预处理好的数据开始。



1、找到影响过拟合的参数列表，确定这些参数的取值范围，定义好参数空间。

2、然后定义好损失函数，以算法交叉验证（默认5折）的准确率取负为损失函数返回值，这样准确率越高，损失函数越小。

3、然后给定一个开始的地方，我从参数的默认值开始，也可以从Kaggle上一些比较好的经验参数开始，或者从网格搜索或随机搜索得到的比较好的参数开始。

4、迭代1000次，这样找到了更优的参数，然后从上次的最优参数开始，再迭代1000次......，直到找不到更优的参数为止。

调参的结果，训练集的准确率也许会有所下降（RF、XGB），交叉验证准确率和测试集准确率有所上升，说明一定程度上克服了过拟合的问题。不要有太大的期望，大概是0.5%~1%的提升，作用是让几个GBDT算法跨过了90%优秀的门槛。对抗过拟合起决定性作用的还是算法实现的本身。

# Imports

# Basic Imports

import numpy as np

import pandas as pd

import time

# Plotting

import matplotlib.pyplot as plt

# Preprocessing

from sklearn.model\_selection import train\_test\_split, cross\_val\_score

# Metrics

from sklearn.metrics import mean\_squared\_error, mean\_absolute\_error, r2\_score

# ML Models

from catboost import CatBoostRegressor

# Model Tuning

from hyperopt import fmin, tpe, hp

from hyperopt.fmin import generate\_trials\_to\_calculate

# Feature Importance

import shap

# Ignore Warnings

import warnings

warnings.filterwarnings('ignore')

# Reading a CSV File

# 9015

df\_NN = pd.read\_csv("D:/temp/data/Melbourne\_housing/Melbourne\_housing\_pre.csv", encoding="utf-8")

X=df\_NN[['Year','YearBuilt','Distance','Lattitude','Longtitude','Propertycount',

'Landsize','BuildingArea', 'Rooms','Bathroom', 'Car','Type\_h','Type\_t','Type\_u']]

y=df\_NN['LogPrice']

train\_X, valid\_X, train\_y, valid\_y = train\_test\_split(X,y, test\_size = .20, random\_state=42)

train\_X2 = train\_X.copy()

valid\_X2 = valid\_X.copy()

# Data standardization

mean = train\_X.mean(axis=0)

train\_X -= mean

std = train\_X.std(axis=0)

train\_X /= std

valid\_X -= mean

valid\_X /= std

##% evaluateRegressor

# from sklearn.metrics import mean\_squared\_error, mean\_absolute\_error

def evaluateRegressor(true,predicted,message = "Test set"):

MSE = mean\_squared\_error(true,predicted,squared = True)

MAE = mean\_absolute\_error(true,predicted)

RMSE = mean\_squared\_error(true,predicted,squared = False)

LogRMSE = mean\_squared\_error(np.log(true),np.log(predicted),squared = False)

R2 = r2\_score(true,predicted)

print(message)

print("MSE:", MSE)

print("MAE:", MAE)

print("RMSE:", RMSE)

print("LogRMSE:", LogRMSE)

print("R2 :", R2)

##% Plot True vs predicted values. Useful for continuous y

def PlotPrediction(true,predicted, title = "Dataset: "):

df = pd.DataFrame({"Real":true,"Predicted":predicted})

df.sort\_values(by=["Real"], inplace=True)

df = df.reset\_index(drop=True)

fig = plt.figure(figsize=(20,20))

plt.tick\_params(labelsize=20)

ax1 = fig.add\_subplot(111)

ax1.set\_title(title + 'True vs Predicted', fontsize=40)

ax1.plot(list(range(0,len(df["Predicted"]))),df["Predicted"],'b.',label='Predicted')

ax1.plot(list(range(0,len(df["Real"]))),df["Real"],'r-',label='True')

ax1.fill\_between(list(range(0,len(df["Real"]))),df["Real"]\*0.8,

df["Real"]\*1.2,facecolor='yellow',interpolate=True)

plt.xlabel("Samples in Value Order", fontsize=30)

plt.ylabel("Price", fontsize=30)

plt.legend(loc='upper left',fontsize=30 );

plt.show()

def PlotPrediction2(true,predicted, title = "Dataset: "):

df = pd.DataFrame({"Real":true,"Predicted":predicted})

df.sort\_values(by=["Real"], inplace=True)

df = df.reset\_index(drop=True)

fig = plt.figure(figsize=(20,20))

plt.tick\_params(labelsize=20)

ax1 = fig.add\_subplot(111)

ax1.set\_title(title + 'True vs Predicted', fontsize=40)

ax1.plot(list(range(0,len(df["Predicted"]))),df["Predicted"],'b.',label='Predicted')

ax1.plot(list(range(0,len(df["Real"]))),df["Real"],'r-',label='True')

ax1.fill\_between(list(range(0,len(df["Real"]))),df["Real"]+np.log(0.8),

df["Real"]+np.log(1.2),facecolor='yellow',interpolate=True)

plt.xlabel("Samples in Value Order", fontsize=30)

plt.ylabel("Log Price", fontsize=30)

plt.legend(loc='upper left',fontsize=30 );

plt.show()

# -------------------------------------------------------------------------------

# Auto search for better hyper parameters with hyperopt, only need to give a range

# Reference: https://github.com/talperetz/hyperspace/tree/master/GBDTs

# https://catboost.ai/docs/concepts/python-reference\_parameters-list.html#python-reference\_parameters-list

# https://catboost.ai/docs/concepts/parameter-tuning.html

# https://affine.ai/catboost-a-new-game-of-machine-learning/

'''

https://catboost.ai/en/docs/concepts/speed-up-training

Speeding up the training

1. iterations, worked

2. learning\_rate, worked

2. boosting\_type, Ordered, Plain, not worked

3. bootstrap\_type, Bayesian, Bernoulli, MVS, Poisson, not worked

4. subsample, not worked

This parameter can be used if one of the following bootstrap types is selected:

Poisson Bernoulli MVS

5. one\_hot\_max\_size, One-hot encoding

6. rsm, colsample\_bylevel, Random subspace method

7.leaf\_estimation\_iterations, worked, set to 1.

Try setting the value to "1" or "5" to speed up the training on datasets with a small number of features.

8. max\_ctr\_complexity, worked, 0 or 2 to speed up trainning.

This parameter can affect the training time only if the dataset contains categorical features.

9. border\_count, worked, set to less.

10.Reusing quantized datasets in Python, not applyable to cross\_val\_score()

11.Golden features. If the dataset has a feature, which is a strong predictor of the result, the

pre-quantisation of this feature may decrease the information that the model can get from it.

It is recommended to use an increased number of borders (1024) for this feature.

per\_float\_feature\_quantization=['0:border\_count=1024', '1:border\_count=1024']

'''

# default values

# trials = generate\_trials\_to\_calculate([{

# 'border\_count':254-150, # default CPU 254 GPU 128

# 'iterations':1000-500, # default 1000

# 'depth': 6-2, # default 6

# 'random\_strength':1.0, # default 1.0, CPU only

# 'learning\_rate': 0.03, # default 0.03

# 'subsample':0.8, # default 0.8

# 'l2\_leaf\_reg': 3.0, # default 3

# 'rsm':0.8, # default 1.0 CPU only

# 'fold\_len\_multiplier':2.0, # default 2.0

# 'bagging\_temperature':1.0 # default 1.0

# }])

# 缩小参数取值范围，搜索会快很多

space\_cat = {'border\_count': hp.choice('border\_count', range(150, 351)), # CPU 150-351 GPU 8-128

'iterations': hp.choice('iterations', range(500, 1501)),

'depth': hp.choice('depth', range(2, 10)),

'random\_strength': hp.uniform('random\_strength', 1, 20),

'learning\_rate': hp.uniform('learning\_rate', 0.005, 0.15),

'subsample': hp.uniform('subsample', 0.5, 1),

'l2\_leaf\_reg': hp.uniform('l2\_leaf\_reg', 1, 100),

'rsm': hp.uniform('rsm', 0.5, 0.99), # colsample\_bylevel

'fold\_len\_multiplier': hp.uniform('fold\_len\_multiplier', 1.0, 10.0),

'bagging\_temperature': hp.uniform('bagging\_temperature', 0.0, 1.0) }

def f\_cat(params):

cat = CatBoostRegressor(task\_type='CPU', random\_seed=0,

# boosting\_type='Plain', bootstrap\_type = 'Bayesian', max\_ctr\_complexity=1,

one\_hot\_max\_size=3,

leaf\_estimation\_iterations=1,

#per\_float\_feature\_quantization=['3:border\_count=1024', '4:border\_count=1024'], # Golden features: lat, long

verbose=False, \*\*params) # CPU 13.05s/trial

#cat\_model = cat.fit(train\_X, train\_y)

#acc = cat\_model.score(valid\_X,valid\_y)

acc = cross\_val\_score(cat, train\_X, train\_y).mean()

return -acc

# trials = Trials()

# Set initial values, start searching from the best point of GridSearchCV(), and default values

trials = generate\_trials\_to\_calculate([{

'border\_count':112, # default CPU 254 GPU 12

'iterations':989, # default 1000

'depth': 4, # default 6

'random\_strength':6.6489521372262645, # default 1.0, CPU only

'learning\_rate': 0.07811835381238333, # default 0.03

'subsample':0.9484820488113903, # default 0.8

'l2\_leaf\_reg': 8.070279328038293, # default 3

'rsm':0.7188098046587024, # default 1.0 CPU only

'fold\_len\_multiplier': 6.034216410528531, # default 2.0

'bagging\_temperature':0.47787665340753926 # default 1.0

}])

t1 = time.time()

# 1000trial [3:17:32, 11.85s/trial, best loss: -0.908607011971843]

# 1000trial [3:37:34, 13.05s/trial, best loss: -0.9090471252105156]

# 1000trial [3:31:48, 12.71s/trial, best loss: -0.9090471252105156]

# 1000trial [48:49, 2.93s/trial, best loss: -0.9064909405009918]

# 1000trial [45:33, 2.73s/trial, best loss: -0.905595803200792]

# 1000trial [4:15:39, 15.34s/trial, best loss: -0.9247947954437636]

# 1000trial [50:28, 3.03s/trial, best loss: -0.905859099632395]

best\_params = fmin(f\_cat, space\_cat, algo=tpe.suggest, max\_evals=999, trials=trials)

t2 = time.time()

# Time elapsed: 11852.239725589752 13054.514278173447 12708.484918832779 2929.361430644989 2733.5603375434875 15340.029090881348 3028.758241176605

print("Time elapsed: ", t2-t1)

print('best:')

# {'bagging\_temperature': 0.9257320870418309, 'border\_count': 81, 'depth': 6, 'fold\_len\_multiplier': 3.9464296202589044, 'iterations': 819, 'l2\_leaf\_reg': 36.301263538300276, 'learning\_rate': 0.11522312552993323, 'random\_strength': 18.96544522884934, 'rsm': 0.69678623336759, 'subsample': 0.7098942047631396}

# {'bagging\_temperature': 0.4504229015029012, 'border\_count': 119, 'depth': 4, 'fold\_len\_multiplier': 2.0919942770421853, 'iterations': 737, 'l2\_leaf\_reg': 13.057928619717039, 'learning\_rate': 0.10701852869378331, 'random\_strength': 3.559681852246134, 'rsm': 0.8496879993042271, 'subsample': 0.9886883465409299}

# {'bagging\_temperature': 0.47787665340753926, 'border\_count': 112, 'depth': 4, 'fold\_len\_multiplier': 6.034216410528531, 'iterations': 989, 'l2\_leaf\_reg': 8.070279328038293, 'learning\_rate': 0.07811835381238333, 'random\_strength': 6.6489521372262645, 'rsm': 0.7188098046587024, 'subsample': 0.9484820488113903}

# {'bagging\_temperature': 0.5402870554069704, 'border\_count': 183, 'depth': 5, 'fold\_len\_multiplier': 4.43906516804156, 'iterations': 899, 'l2\_leaf\_reg': 8.334167765336101, 'learning\_rate': 0.0997818676941431, 'random\_strength': 6.564979609549752, 'rsm': 0.8975065545697877, 'subsample': 0.857395221266925}

# {'bagging\_temperature': 0.963604410523803, 'border\_count': 20, 'depth': 4, 'fold\_len\_multiplier': 3.724055148568295, 'iterations': 969, 'l2\_leaf\_reg': 3.1380215904899176, 'learning\_rate': 0.07896753655162898, 'random\_strength': 1.2901139148639478, 'rsm': 0.9698099104015329, 'subsample': 0.9664695329669599}

# {'bagging\_temperature': 0.6730567766972233, 'border\_count': 114, 'depth': 4, 'fold\_len\_multiplier': 6.746920236421965, 'iterations': 871, 'l2\_leaf\_reg': 8.071698998237064, 'learning\_rate': 0.10378896820797036, 'random\_strength': 7.84645871379397, 'rsm': 0.8514600128978964, 'subsample': 0.9603810112084823}

# {'bagging\_temperature': 0.44539253072336643, 'border\_count': 142, 'depth': 5, 'fold\_len\_multiplier': 3.9112871147780752, 'iterations': 541, 'l2\_leaf\_reg': 5.814810215893133, 'learning\_rate': 0.11348487512579726, 'random\_strength': 7.757238485588866, 'rsm': 0.8579059894823821, 'subsample': 0.6970992858202274}

print(best\_params)

# verify

#params = best\_params.copy()

# restore best hyper parameters

params = {'bagging\_temperature': 0.5402870554069704, 'border\_count': 183, 'depth': 5, 'fold\_len\_multiplier': 4.43906516804156, 'iterations': 899, 'l2\_leaf\_reg': 8.334167765336101, 'learning\_rate': 0.0997818676941431, 'random\_strength': 6.564979609549752, 'rsm': 0.8975065545697877, 'subsample': 0.857395221266925}

# params = {'bagging\_temperature': 0.6730567766972233, 'border\_count': 114, 'depth': 4, 'fold\_len\_multiplier': 6.746920236421965, 'iterations': 871, 'l2\_leaf\_reg': 8.071698998237064, 'learning\_rate': 0.10378896820797036, 'random\_strength': 7.84645871379397, 'rsm': 0.8514600128978964, 'subsample': 0.9603810112084823}

params['border\_count'] = params['border\_count']+150

params['depth'] = params['depth']+2

params['iterations'] = params['iterations']+500

print(params)

# Original best parameters of GridSearchCV()

cat\_best = CatBoostRegressor(task\_type='CPU',

random\_seed=0,

one\_hot\_max\_size=3,

leaf\_estimation\_iterations=1,

#max\_ctr\_complexity=0,

verbose=False, \*\*params) # CPU 44.64s/trial

acc = cross\_val\_score(cat\_best, train\_X, train\_y).mean()

# 0.9070147487202945 0.9075121102244383 0.9090471252105156 0.9074151020398504 0.9247947954437636

print(acc)

# predict

cat\_model\_train = cat\_best.fit(train\_X, train\_y)

# 0.9642262907727034 0.9510781506876833 0.9627372929633919 0.9755374523207733 0.974539467468884

print(cat\_model\_train.score(train\_X,train\_y))

# 0.9035659553161598 0.9035094883595367 0.903804492576797 0.9064909405009918 0.9032257307932621

print(cat\_model\_train.score(valid\_X,valid\_y))

# evalute model using the entire dataset from Train.csv

evaluateRegressor(train\_y,cat\_model\_train.predict(train\_X),"Train set ")

evaluateRegressor(valid\_y,cat\_model\_train.predict(valid\_X),"Valid set ")

PlotPrediction2(train\_y,cat\_model\_train.predict(train\_X),"Train set: ")

PlotPrediction2(valid\_y,cat\_model\_train.predict(valid\_X),"Test set: ")

train\_targets\_o = np.round(np.exp(train\_y),2)

predictions\_t\_o = np.round(np.exp(cat\_model\_train.predict(train\_X)),2)

evaluateRegressor(train\_targets\_o, predictions\_t\_o,"Train Set")

PlotPrediction(list(train\_targets\_o),list(predictions\_t\_o),"Train set: ")

test\_targets\_o = np.round(np.exp(valid\_y),2)

predictions\_o = np.round(np.exp(cat\_model\_train.predict(valid\_X)),2)

evaluateRegressor(test\_targets\_o, predictions\_o,"Test Set")

PlotPrediction(list(test\_targets\_o),list(predictions\_o),"Test set: ")

四、[优化过程可视化](https://www.statestitle.com/resource/visualizing-hyperparameter-optimization-with-hyperopt-and-plotly/)

优化过程可视化可以帮助理解优化的过程，选择更合适的参数区间。trials变量记录了搜索过程的参数组合与损失函数返回值，是一个字典列表，通过下面的程序先转换为pandas DF。

# print one trail to see the result structure

for t in trials:

print(t)

break

# hyperparameter is not relevant to a particular trial.

def unpack(x):

if x:

return x[0]

return np.nan

# We'll first turn each trial into a series and then stack those series together as a dataframe.

trials\_df = pd.DataFrame([pd.Series(t["misc"]["vals"]).apply(unpack) for t in trials])

# Then we'll add other relevant bits of information to the correct rows and perform a couple of

# mappings for convenience

trials\_df["loss"] = [t["result"]["loss"] for t in trials]

trials\_df["trial\_number"] = trials\_df.index

trials\_df.to\_csv("D:/temp/data/CatBoost\_1000\_trials2.csv")

trials\_df["depth"].value\_counts()

# draw contour plot with plotly in jupyter notebook then.

def PlotTrial(trials,loss):

fig = plt.figure(figsize=(20,20))

plt.tick\_params(labelsize=20)

ax1 = fig.add\_subplot(111)

ax1.set\_title('Loss per Trial', fontsize=40)

ax1.plot(trials,loss,'b.',label='loss')

# ax1.plot(list(range(0,len(df["Real"]))),df["Real"],'r-',label='True')

plt.xlabel("Trial", fontsize=30)

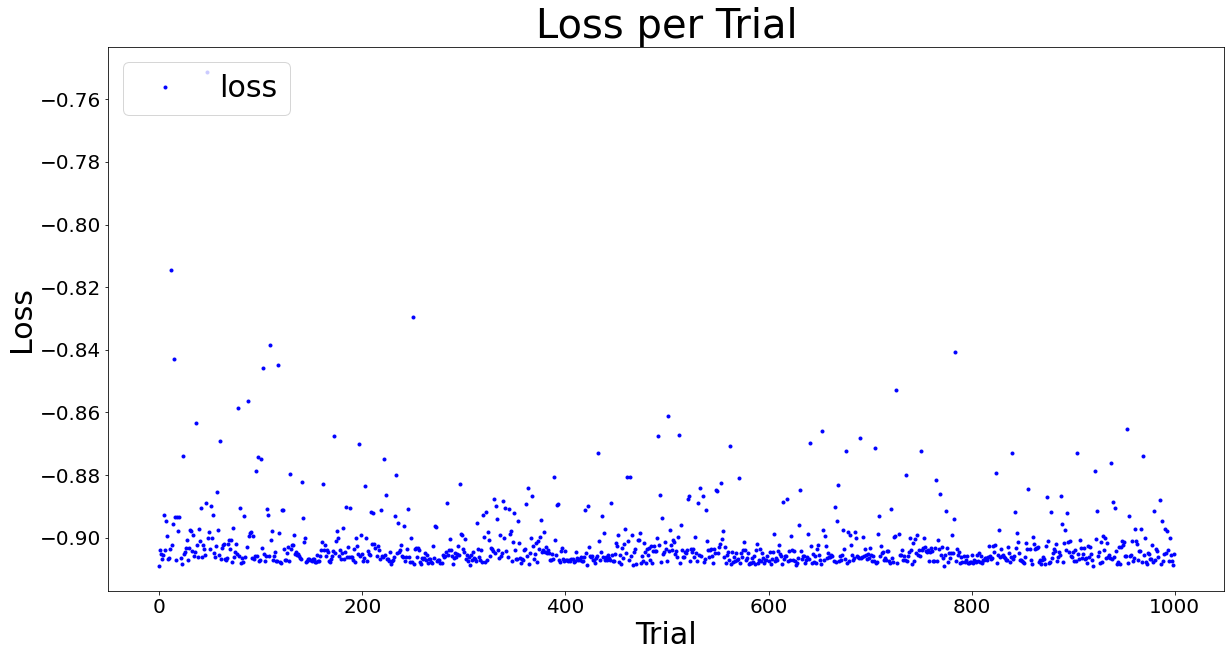
plt.ylabel("Loss", fontsize=30)

plt.legend(loc='upper left',fontsize=30 );

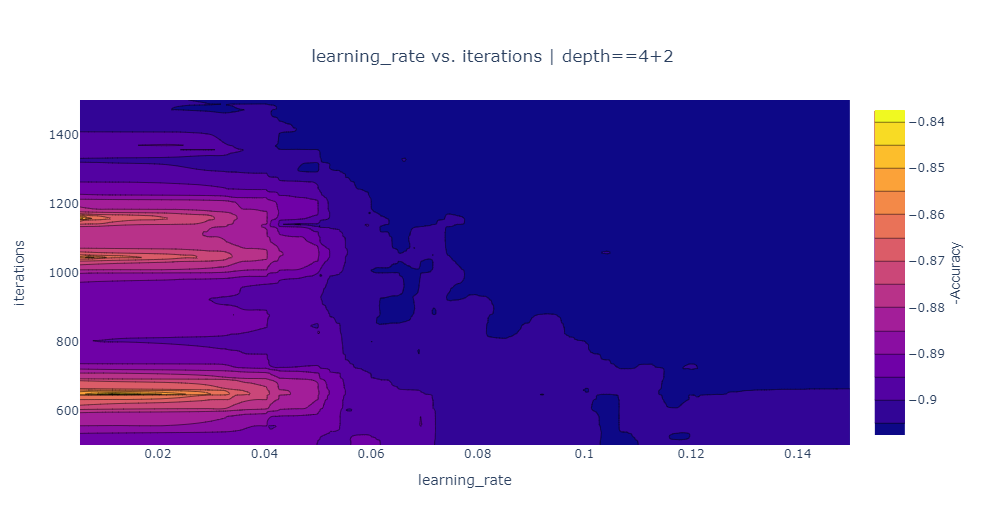
plt.show()

PlotTrial(trials\_df['trial\_number'], trials\_df['loss'])

画损失函数返回值在各次迭代中的变化，可以看到贝叶斯优化在选择下次迭代的参数组合时，导向了损失函数更小的参数空间，在图上表现为集中于loss更小的一端，它倾向于选择减少损失函数概率更大的方向。从上面的参数空间可知，超参数的组合是无穷的（因为包含了实数），贝叶斯优化的确可以在有限的步数内找到更好的参数，非常有效。



对于GBDT算法来说，迭代训练次数与学习率是两个比较重要的超参数，可以画它们的等高线图来看看它们对损失函数的影响，先看结果。低学习率不容易过拟合，这是Kaggle上的经验，CatBoost默认的学习率是0.03，我在[0.005,0.15]之间搜索，可以看到本数据集在0.08至0.15之间的表现比较稳定。GBDT算法是串行训练的，迭代训练的次数决定了性能的两个方面：准确率和速度，在保证准确率的前提下要尽量减少迭代次数以提高速度。CatBoost默认的次数是1000，我在[500,1500]之间搜索。从图上可见，当学习率>0.08时，迭代次数800到1500之间的表现是稳定的。贝叶斯优化最后找到的最优组合中，iterations=1489，learning\_rate=0.07811835381238333。还可以画其它感兴趣参数组合的等高线图以观察它们的交互作用及对损失函数的共同影响。这里的一个主要经验是，尽量缩小参数的取值区间可以有效加快搜索和迭代训练的速度。因为模型在落地应用后要随着样本数据的积累不断的重新训练与优化，所以训练与优化的效率也是很重要的。



Matplotlib画等高线图需要一个（X,Y,Z）网格矩阵，分别对应上面的iterations，learning\_rate，loss(-ACC)，但trials DF提供的只是部分的点，网格中绝大部分的数据是缺失的，(X,Y)网格可以通过numpy.meshgrid(x, y)很容易生成，但Z轴是训练的结果，这就需要算法插值。plotly包的等高线图就提供了这样的插值功能，如下图所示，当x,y,z都是一维数组时，它会通过算法插值自动生成所需的(X,Y,Z)网格矩阵。不过当数据行数较大时，如本例的1000行，可能会导致插值函数挂起输出白图，所以本例选择了另一个超参数depth=4+2切片画图。hp.choice类型的参数是个列表，内容可以是字符串等，在hyperopt中返回的值是列表的下标，所以真实值要加上列表的开始值，如depth要+2，iterations要+500，如果定义为hp.quniform，均匀的离散类型，就不用加开始值。另外还有hp.uniform实数均匀分布，hp.loguniform对数下均匀分布等。

plotly是运行在浏览器上的js库，后端要用jupyter notebook，所以要把trials转换为pandas DF，从Spyder输出成CSV，然后在jupyter notebook中读入处理。

# code block1, 读入训练记录

import numpy as np

import pandas as pd

trials\_df = pd.read\_csv("D:/temp/data/CatBoost\_1000\_trials.csv")

print(trials\_df.shape)

print(trials\_df.columns)

# code block2,估算个切片的数据点数

print(trials\_df["depth"].value\_counts())

trial\_filter = (trials\_df["depth"] ==4)

z=trials\_df.loc[trial\_filter, "loss"]

x=trials\_df.loc[trial\_filter, "learning\_rate"]

y=trials\_df.loc[trial\_filter, "iterations"]

print(len(x))

print(len(y))

print(len(z))

# code block3,自动插值画等高线图

from plotly import express as px

from plotly import graph\_objects as go

from plotly import offline as pyo

# creates a boolean array that we will use to filter down to relevant rows in the `trials\_df`dataframe.

trial\_filter = (trials\_df["depth"] ==4)

# plotly express does not support contour plots so we will use `graph\_objects` instead.

# go.Contour automatically interpolates "z" values for our loss.

fig = go.Figure(

data=go.Contour(

z=trials\_df.loc[trial\_filter, "loss"],

x=trials\_df.loc[trial\_filter, "learning\_rate"],

y=trials\_df.loc[trial\_filter, "iterations"]+500,

colorbar=dict(

title='-Accuracy', # title here

titleside='right',

titlefont=dict(size=14,family='Arial, sans-serif')

)

)

)

fig.update\_layout(

xaxis\_title="learning\_rate",

yaxis\_title="iterations",

title={

"text": "learning\_rate vs. iterations | depth==4+2",

"xanchor": "center",

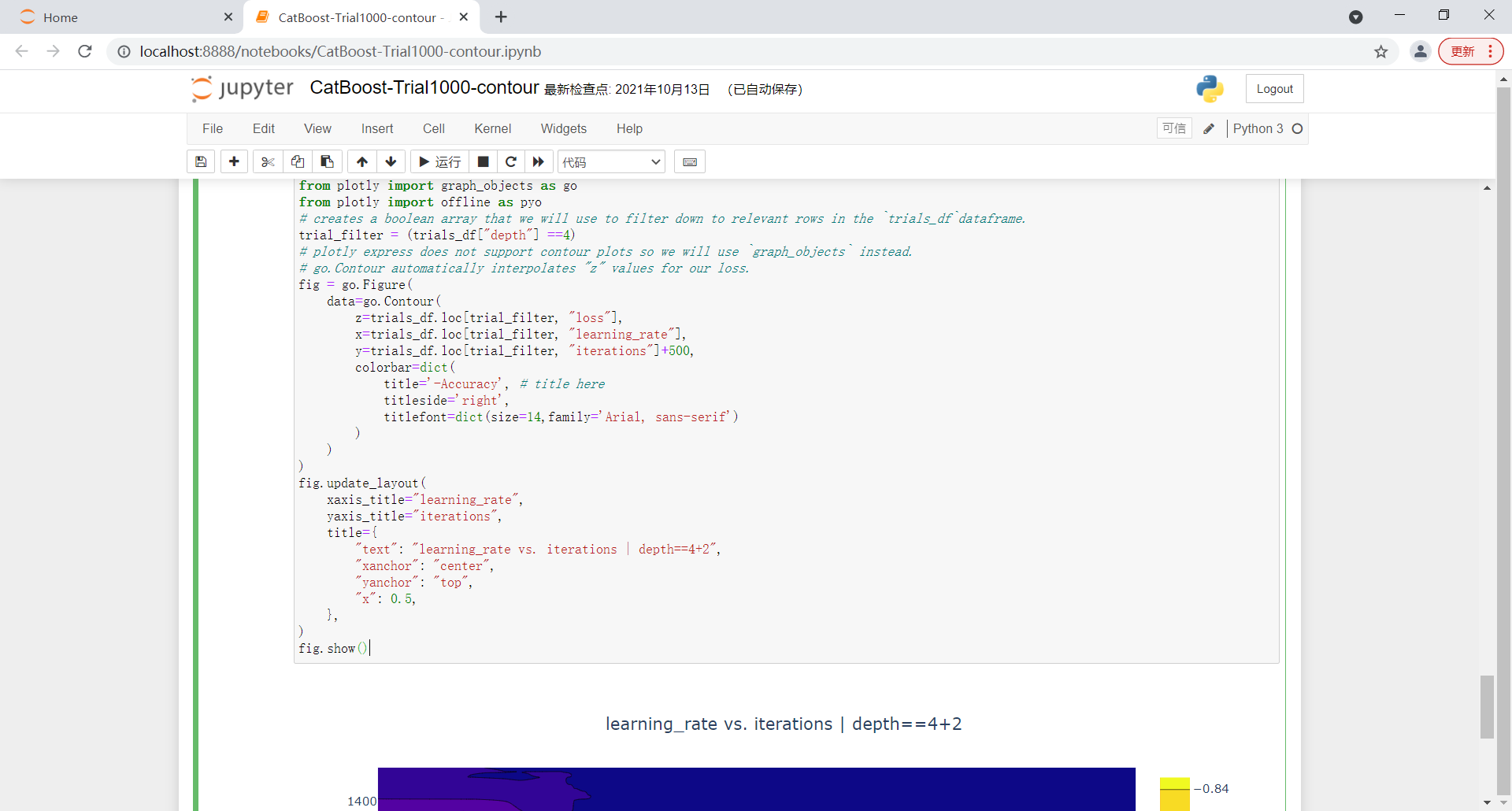
"yanchor": "top",

"x": 0.5,

},

)

fig.show()



CatBoost调参可以参阅以下的参考资料：

[普通参数](https://catboost.ai/en/docs/references/training-parameters/common)，[过拟合检测](https://catboost.ai/en/docs/references/training-parameters/overfitting-detection)，[性能参数](https://catboost.ai/en/docs/references/training-parameters/performance)， [参数优化](https://catboost.ai/en/docs/concepts/parameter-tuning)，[提高训练速度](https://catboost.ai/en/docs/concepts/speed-up-training)，以及[这篇文章](https://affine.ai/catboost-a-new-game-of-machine-learning/)，里面有一些几种GBDT算法在不同的Kaggle竞赛数据集上准确率比较的数据可以了解一下。

五、模型解释

画特征重要性看看，shap包作图的可解释性很好，是解释模型的利器。

##% Feature Importance

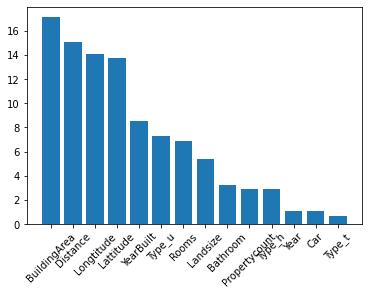
df\_importance = pd.DataFrame({"Feature":train\_X.columns,"importance":cat\_model\_train.feature\_importances\_})

df\_importance.sort\_values(["importance"],ascending = False, inplace=True)

plt.bar(df\_importance["Feature"], df\_importance["importance"])

plt.xticks(rotation=45)

plt.show()



##% Feature Importance using shap package

# https://zhuanlan.zhihu.com/p/83412330

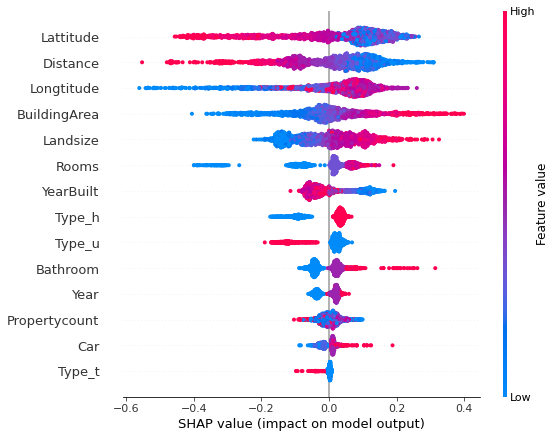
# https://zhuanlan.zhihu.com/p/106320452

# Should use TreeExplainer for LightGBM tree base algorithms

shap\_values = shap.TreeExplainer(cat\_best).shap\_values(valid\_X)

shap.summary\_plot(shap\_values, valid\_X)

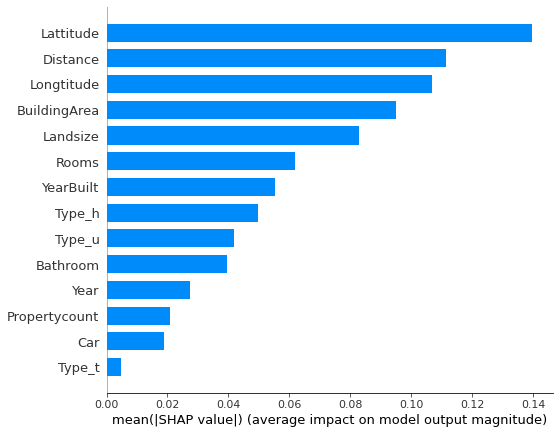
# 这个图可以看到各特征变量各取值区间对拟合值正面或负面影响的大小。



# a different importance plot of shap\_values

shap.summary\_plot(shap\_values, valid\_X, plot\_type="bar")

# shap 值计算的特征重要性与模型自己计算的稍有不同，是另一个参考。



# Feature importance for a single sample

# shap.initjs() # notebook环境下，加载用于可视化的JS代码

# 如果不想用JS,传入matplotlib=True,

row = 0

shap.force\_plot(valid\_y.mean(), shap\_values[row], np.round(valid\_X.iloc[row],4),matplotlib=True,text\_rotation=30)

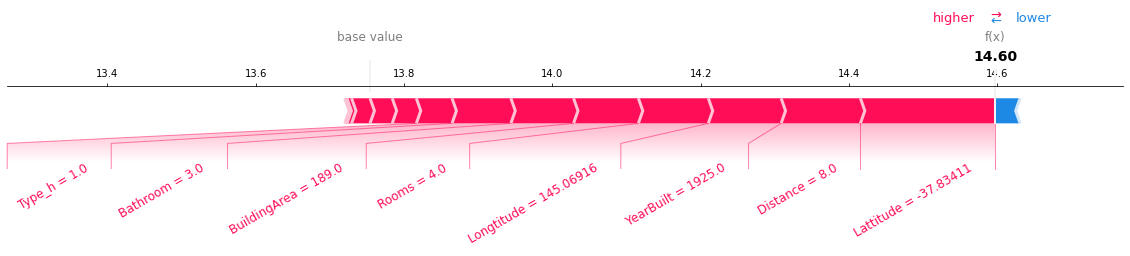
# better to plot with original feature values

shap.force\_plot(valid\_y.mean(), shap\_values[row], valid\_X2.iloc[row],matplotlib=True,text\_rotation=30)

# matplotlib = True is not yet supported for force plots with multiple samples!

# shap.force\_plot(valid\_y.mean(), shap\_values[0:3], X[0:3],matplotlib=True)

#该图反映的是每一个拟合值(row=0为例)中各特征变量正面或负面的贡献。



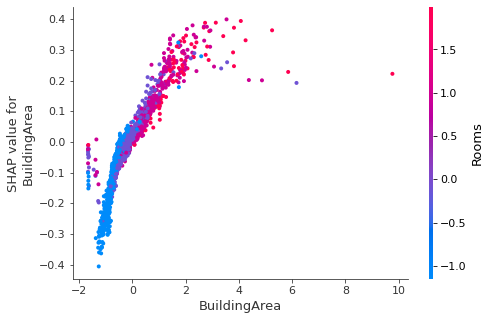
# create a SHAP dependence plot to show the effect of a single feature across the whole dataset

shap.dependence\_plot("BuildingArea", shap\_values, valid\_X)

#该图反映的是某个特征变量在整个数据集中对拟合值影响的变化，

#可以看到建筑面积越大，shap值越大，然后用另一个特征变量来着色，

#以增强可视化的效果。



六、进一步排除异常点

受线性回归RANSAC算法及[这篇论文](https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-7-123)的启发，现在把训练集排除异常点后再训练看看效果如何。7212个训练样本，如果按正负20%误差划分，有158个范围之外的异常点，占2.2%，不算多。如果按正负30%划分，有35个异常点，占0.5%。所以训练集选择更严格一点的20%也是可以的。

# ---按上下误差20%以内的标准排除训练集异常点，有158个，占比2.2%，不算高，如果是30%，则有35个，占0.5%。

train\_X2 = train\_X.copy()

train\_y2 = train\_y.copy()

predictions = cat\_model\_train.predict(train\_X)

df = pd.DataFrame({"real":train\_y,"predicts":predictions})

indexs = (df["predicts"]<=df["real"]+np.log(1.2)) & (df["predicts"]>=df["real"]+np.log(0.8))

indexs.value\_counts()

train\_X = train\_X[indexs==True]

train\_y = train\_y[indexs==True]

outliers\_X = train\_X2[indexs!=True]

outliers\_y = train\_y2[indexs!=True]

# 排除异常点后，从上次最优参数开始，重新用贝叶斯优化训练寻找最优超参数

# 参阅论文 https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-7-123

# Set initial values, start searching from the best point of GridSearchCV(), and default values

trials = generate\_trials\_to\_calculate([{

'border\_count':112, # default CPU 254 GPU 128

'iterations':989, # default 1000

'depth': 4, # default 6

'random\_strength':6.6489521372262645, # default 1.0, CPU only

'learning\_rate': 0.07811835381238333, # default 0.03

'subsample':0.9484820488113903, # default 0.8

'l2\_leaf\_reg': 8.070279328038293, # default 3

'rsm':0.7188098046587024, # default 1.0 CPU only

'fold\_len\_multiplier': 6.034216410528531, # default 2.0

'bagging\_temperature':0.47787665340753926 # default 1.0

}])

t1 = time.time()

# 1000trial [4:15:39, 15.34s/trial, best loss: -0.9247947954437636]

best\_params = fmin(f\_cat, space\_cat, algo=tpe.suggest, max\_evals=999, trials=trials)

t2 = time.time()

# Time elapsed: 15340.029090881348

print("Time elapsed: ", t2-t1)

print('best:')

# {'bagging\_temperature': 0.6730567766972233, 'border\_count': 114, 'depth': 4, 'fold\_len\_multiplier': 6.746920236421965, 'iterations': 871, 'l2\_leaf\_reg': 8.071698998237064, 'learning\_rate': 0.10378896820797036, 'random\_strength': 7.84645871379397, 'rsm': 0.8514600128978964, 'subsample': 0.9603810112084823}

print(best\_params)

# 按最新的最优超参数评估模型的性能，

# 训练集交叉验证有2%的提高，超过92.5%，不过测试集没有改善，略有下降，比LGBM好的是保持超过90%。

params = {'bagging\_temperature': 0.6730567766972233, 'border\_count': 114, 'depth': 4, 'fold\_len\_multiplier': 6.746920236421965, 'iterations': 871, 'l2\_leaf\_reg': 8.071698998237064, 'learning\_rate': 0.10378896820797036, 'random\_strength': 7.84645871379397, 'rsm': 0.8514600128978964, 'subsample': 0.9603810112084823}

params['border\_count'] = params['border\_count']+150

params['depth'] = params['depth']+2

params['iterations'] = params['iterations']+500

print(params)

cat\_best = CatBoostRegressor(task\_type='CPU',

random\_seed=0,

one\_hot\_max\_size=3,

leaf\_estimation\_iterations=1,

#max\_ctr\_complexity=0,

verbose=False, \*\*params) # CPU 44.64s/trial

acc = cross\_val\_score(cat\_best, train\_X, train\_y).mean()

# 0.9247947954437636

print(acc)

# predict

cat\_model\_train = cat\_best.fit(train\_X, train\_y)

# 0.974539467468884

print(cat\_model\_train.score(train\_X,train\_y))

#0.9032257307932621

print(cat\_model\_train.score(valid\_X,valid\_y))

用贝叶斯优化重新训练找到最优超参数组合后，训练集的准确率接近97.5%，提高了约1.2%，交叉验证准确率接近92.5%，提高了约2.1%，测试集准确率90.32%，比原来90.38%略有下降（因为测试集没有排除异常点），不过保持了90%以上优秀的水准。

训练集准确率>交叉验证>测试集，这是正常的逻辑。然后训练集与交叉验证准确率的差距在5%以内，说明过拟合问题已经较好的解决了。然后测试集>90%，泛化能力优秀，说明这样得到的模型已经非常优秀，接近完美，投产使用应该是没有问题了。

七、其它算法优化

1、LightGBM，可以参考下面的资料。

[参数及优化建议](https://www.pythonf.cn/read/6998)，[参数](https://lightgbm.readthedocs.io/en/latest/Parameters.html)， [过拟合优化](https://lightgbm.readthedocs.io/en/latest/Parameters-Tuning.html#deal-with-over-fitting)， [GPU优化](https://lightgbm.readthedocs.io/en/latest/GPU-Performance.html)。

# Auto search for better hyper parameters with hyperopt, only need to give a range

# Reference: https://www.pythonf.cn/read/6998

# https://lightgbm.readthedocs.io/en/latest/Parameters.html

# https://lightgbm.readthedocs.io/en/latest/Parameters-Tuning.html#deal-with-over-fitting

# https://lightgbm.readthedocs.io/en/latest/GPU-Performance.html

# 处理过拟合

# 设置较少的直方图数目 max\_bin

# 设置较小的叶节点数 num\_leaves

# 使用 min\_child\_samples（min\_data\_in\_leaf） 和 min\_child\_weight（= min\_sum\_hessian\_in\_leaf）

# 通过设置 subsample（bagging\_fraction） 和 subsample\_freq（= bagging\_freq） 来使用 bagging

# 通过设置 colsample\_bytree（feature\_fraction） 来使用特征子抽样

# 使用更大的训练数据

# 使用 reg\_alpha（lambda\_l1） , reg\_lambda（lambda\_l2） 和 min\_split\_gain（min\_gain\_to\_split） 来使用正则

# 尝试 max\_depth 来避免生成过深的树

# Try extra\_trees

# Try increasing path\_smooth

# trials = generate\_trials\_to\_calculate([{'max\_bin':63-8, # default CPU 255 GPU 63

# 'max\_depth':5-3, # default -1

# 'num\_leaves':31-20, # default 31

# 'min\_child\_samples':20-10, # default 20

# 'subsample\_freq':1-1, # default 1

# 'n\_estimators':6000-1000, # default 10

# 'learning\_rate':0.01, # default 0.1

# 'subsample':0.75, # default 1.0

# 'colsample\_bytree':0.8, # default 1.0

# 'lambda\_l1':0.0, # default 0.0

# 'lambda\_l2':0.0, # default 0.0

# 'min\_child\_weight':0.001, # default 0.001

# 'min\_split\_gain':0.0, # default 0.0

# #'path\_smooth':0.0 # default 0.0

# }])

# 缩小参数取值范围，搜索会快很多

from lightgbm import LGBMRegressor

space\_lgbm = {

'max\_bin': hp.choice('max\_bin', range(50, 501)), # CPU 50-501 GPU 8-128

'max\_depth': hp.choice('max\_depth', range(3, 31)),

'num\_leaves': hp.choice('num\_leaves', range(10, 256)),

'min\_child\_samples': hp.choice('min\_child\_samples', range(10, 51)),

'subsample\_freq': hp.choice('subsample\_freq', range(1, 6)),

'n\_estimators': hp.choice('n\_estimators', range(500, 6001)),

'learning\_rate': hp.uniform('learning\_rate', 0.005, 0.15),

'subsample': hp.uniform('subsample', 0.5, 0.99),

'colsample\_bytree': hp.uniform('colsample\_bytree', 0.5, 0.99),

'reg\_alpha': hp.uniform('reg\_alpha', 0, 5), # lambda\_l1

'reg\_lambda': hp.uniform('reg\_lambda', 0, 3), # lambda\_l2

'min\_child\_weight': hp.uniform('min\_child\_weight',0.0001, 50),

'min\_split\_gain': hp.uniform('min\_split\_gain',0.0, 1),

#'path\_smooth': hp.uniform('path\_smooth',0.0, 3)

}

def f\_lgbm(params):

# Set extra\_trees=True to avoid overfitting

lgbm = LGBMRegressor(seed=0,verbose=-1, \*\*params) # CPU 4.96s/trial

# lgbm = LGBMRegressor(device='gpu', num\_threads =3, \*\*params) # GPU 65.93s/trial

#lgb\_model = lgbm.fit(train\_X, train\_y)

#acc = lgb\_model.score(valid\_X,valid\_y)

acc = cross\_val\_score(lgbm, train\_X, train\_y).mean() # CPU

# acc = cross\_val\_score(lgbm, train\_X, train\_y, n\_jobs=6).mean() # GPU

return -acc

# trials = Trials()

# Set initial values, start searching from the best point of GridSearchCV(), and default values

trials = generate\_trials\_to\_calculate([{

'max\_bin':278, # default CPU 255 GPU 63

'max\_depth':17, # default -1

'num\_leaves':12, # default 31

'min\_child\_samples':14, # default 20

'subsample\_freq':0, # default 1

'n\_estimators':2647, # default 10

'learning\_rate':0.0203187560767722, # default 0.1

'subsample':0.788703175392162, # default 1.0

'colsample\_bytree':0.5203150334508861, # default 1.0

'reg\_alpha': 0.988139501870491, # default 0.0

'reg\_lambda':2.789779486137205, # default 0.0

'min\_child\_weight':21.813225361674828, # default 0.001

'min\_split\_gain':0.00039636685518264865, # default 0.0

#'path\_smooth':0.0 # default 0.0

}])

t1 = time.time()

# 1000trial [5:23:09, 19.39s/trial, best loss: -0.9082183160929432] CPU

# 1000trial [1:22:39, 4.96s/trial, best loss: -0.9079837941918502] CPU

# 1000trial [5:39:51, 20.39s/trial, best loss: -0.9068431932173453] GPU

# 1000trial [1:02:28, 3.75s/trial, best loss: -0.9080477825539048] CPU

# 1000trial [18:55, 1.14s/trial, best loss: -0.9029308764279137] CPU

# 1000trial [1:14:40, 4.48s/trial, best loss: -0.9253639597148784] CPU

best\_params = fmin(f\_lgbm, space\_lgbm, algo=tpe.suggest, max\_evals=999, trials=trials)

t2 = time.time()

# 19390.56170320511 4960.896098852158 20392.74730038643GPU 3749.3419647216797 1136.1463103294373 4481.4692125320435

print("Time elapsed: ", t2-t1)

print('best:')

# CPU {'colsample\_bytree': 0.5000455292913467, 'reg\_alpha': 0.15545644376537782, 'reg\_lambda': 1.4080091797633087, 'learning\_rate': 0.007561841813178302, 'max\_bin': 419, 'max\_depth': 39, 'min\_child\_weight': 9.330764246889554, 'n\_estimators': 4716, 'num\_leaves': 12, 'subsample': 0.5887399629302962}

# CPU {'colsample\_bytree': 0.5740856868933041, 'reg\_alpha': 0.4978659241908678, 'reg\_lambda': 2.9895546493896226, 'learning\_rate': 0.01664367400440669, 'max\_bin': 86, 'max\_depth': 5, 'min\_child\_samples': 29, 'min\_child\_weight': 22.863111407056216, 'min\_split\_gain': 0.0003417086853309451, 'n\_estimators': 4428, 'num\_leaves': 10, 'subsample': 0.9538662288625716, 'subsample\_freq': 1}

# GPU {'colsample\_bytree': 0.7103345027555479, 'reg\_alpha': 1.0133278908262167, 'reg\_lambda': 0.42903027120676573, 'learning\_rate': 0.014545969324488227, 'max\_bin': 96, 'max\_depth': 3, 'min\_child\_samples': 10, 'min\_child\_weight': 15.450385729945399, 'min\_split\_gain': 0.0016071587697570192, 'n\_estimators': 4528, 'num\_leaves': 8, 'subsample': 0.7242389855352034, 'subsample\_freq': 4}

# CPU {'colsample\_bytree': 0.5203150334508861, 'reg\_alpha': 0.988139501870491, 'reg\_lambda': 2.789779486137205, 'learning\_rate': 0.0203187560767722, 'max\_bin': 278, 'max\_depth': 17, 'min\_child\_samples': 14, 'min\_child\_weight': 21.813225361674828, 'min\_split\_gain': 0.00039636685518264865, 'n\_estimators': 2647, 'num\_leaves': 12, 'subsample': 0.788703175392162, 'subsample\_freq': 0}

# CPU {'colsample\_bytree': 0.5142540541056978, 'learning\_rate': 0.014284678929509775, 'max\_bin': 161, 'max\_depth': 4, 'min\_child\_samples': 5, 'min\_child\_weight': 4.534457967283932, 'min\_split\_gain': 0.0006363777341674458, 'n\_estimators': 2006, 'num\_leaves': 93, 'reg\_alpha': 0.0037820689583625278, 'reg\_lambda': 2.947360470949046, 'subsample': 0.9448608935296047, 'subsample\_freq': 2}

# CPU {'colsample\_bytree': 0.5482469765978001, 'learning\_rate': 0.02138706516193863, 'max\_bin': 211, 'max\_depth': 25, 'min\_child\_samples': 9, 'min\_child\_weight': 26.29858787655885, 'min\_split\_gain': 0.00038968894882169256, 'n\_estimators': 4731, 'num\_leaves': 19, 'reg\_alpha': 0.8257383468656769, 'reg\_lambda': 1.3981267479316741, 'subsample': 0.750429969832067, 'subsample\_freq': 2}

print(best\_params)

# verify

params = best\_params.copy()

# restore best hyper parameters

params = {'colsample\_bytree': 0.5142540541056978, 'learning\_rate': 0.014284678929509775, 'max\_bin': 161, 'max\_depth': 4, 'min\_child\_samples': 5, 'min\_child\_weight': 4.534457967283932, 'min\_split\_gain': 0.0006363777341674458, 'n\_estimators': 2006, 'num\_leaves': 93, 'reg\_alpha': 0.0037820689583625278, 'reg\_lambda': 2.947360470949046, 'subsample': 0.9448608935296047, 'subsample\_freq': 2}

params['max\_bin'] = params['max\_bin']+50

params['max\_depth'] = params['max\_depth']+3

params['num\_leaves'] = params['num\_leaves']+20

params['min\_child\_samples'] = params['min\_child\_samples']+10

params['subsample\_freq'] = params['subsample\_freq']+1

params['n\_estimators'] = params['n\_estimators']+1000

print(params)

# Original best parameters of GridSearchCV()

# Set extra\_trees=True to avoid overfitting

lgbm\_best = LGBMRegressor(seed=0, \*\*params)

acc = cross\_val\_score(lgbm\_best, train\_X, train\_y).mean()

# 0.9082183160929432 0.9069692181747394 0.9066523433423663GPU 0.9078336217424837 0.9075054302960665 0.9248809113205866

print(acc)

# predict

lgb\_model\_full\_data = lgbm\_best.fit(train\_X, train\_y)

# 0.9727223028335408 0.9749978871820015 0.9662169060275624GPU 0.9701937780242721 0.9790954758709947 0.9840188541116298

print(lgb\_model\_full\_data.score(train\_X,train\_y))

# 0.9010664296217231 0.9005194608968802 0.8994218457293254GPU 0.9005790300507339 0.9031159392716986 0.8983448264278595

print(lgb\_model\_full\_data.score(valid\_X,valid\_y))

# evalute model using the entire dataset from Train.csv

evaluateRegressor(train\_y,lgb\_model\_full\_data.predict(train\_X),"Train set ")

evaluateRegressor(valid\_y,lgb\_model\_full\_data.predict(valid\_X),"Valid set ")

2、XGB，可以参考下面的资料。

[参数](https://xgboost.readthedocs.io/en/latest/parameter.html)， [参数调优](https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/)，与LightGBM差不多。

# Auto search for better hyper parameters with hyperopt, only need to give a range

# 设置较少的直方图数目 max\_bin

# 设置较小的叶节点数 num\_leaves

# 使用 min\_child\_samples（min\_data\_in\_leaf） 和 min\_child\_weight（= min\_sum\_hessian\_in\_leaf）

# 通过设置 subsample（bagging\_fraction） 和 subsample\_freq（= bagging\_freq） 来使用 bagging

# 通过设置 colsample\_bytree（feature\_fraction） 来使用特征子抽样

# 使用更大的训练数据

# 使用 reg\_alpha（lambda\_l1） , reg\_lambda（lambda\_l2） 和 min\_split\_gain（min\_gain\_to\_split） 来使用正则

# 尝试 max\_depth 来避免生成过深的树

# Try extra\_trees

# Try increasing path\_smooth

# trials = generate\_trials\_to\_calculate([{'max\_bin':63-8, # default CPU 255 GPU 63

# 'max\_depth':5-3, # default -1

# 'num\_leaves':31-20, # default 31

# 'min\_child\_samples':20-10, # default 20

# 'subsample\_freq':1-1, # default 1

# 'n\_estimators':6000-1000, # default 10

# 'learning\_rate':0.01, # default 0.1

# 'subsample':0.75, # default 1.0

# 'colsample\_bytree':0.8, # default 1.0

# 'lambda\_l1':0.0, # default 0.0

# 'lambda\_l2':0.0, # default 0.0

# 'min\_child\_weight':0.001,# default 0.001

# 'min\_split\_gain':0.0, # default 0.0

# #'path\_smooth':0.0 # default 0.0

# }])

# 缩小参数取值范围，搜索会快很多

from xgboost import XGBRegressor

space\_xgb = {

'max\_bin': hp.choice('max\_bin', range(50, 501)), # CPU 50-501 GPU 8-128

'max\_depth': hp.choice('max\_depth', range(3, 11)),

'n\_estimators': hp.choice('n\_estimators', range(100, 1001)),

'learning\_rate': hp.uniform('learning\_rate', 0.01, 0.3),

'subsample': hp.uniform('subsample', 0.5, 0.99),

'colsample\_bytree': hp.uniform('colsample\_bytree', 0.5, 0.99),

'reg\_alpha': hp.uniform('reg\_alpha', 0, 5), # lambda\_l1

'reg\_lambda': hp.uniform('reg\_lambda', 0, 3), # lambda\_l2

'gamma': hp.uniform('gamma',0.0, 10), # min\_split\_loss, min\_split\_gain

'min\_child\_weight': hp.uniform('min\_child\_weight',0.0001, 50),

}

def f\_xgb(params):

# Set extra\_trees=True to avoid overfitting

xgb = XGBRegressor(objective ='reg:squarederror', seed = 0,verbosity=0, \*\*params) # CPU 4.96s/trial

#xgb = XGBRegressor(tree\_method='gpu\_hist', objective ='reg:squarederror', seed = 0,verbosity=0,\*\*params) # CPU 4.96s/trial

#xgb\_model = xgb.fit(train\_X, train\_y)

#acc = xgb\_model.score(valid\_X,valid\_y)

# acc = cross\_val\_score(xgb, train\_X, train\_y).mean() # CPU

acc = cross\_val\_score(xgb, train\_X, train\_y, n\_jobs=6).mean() # GPU

return -acc

# trials = Trials()

# Set initial values, start searching from the best point of GridSearchCV(), and default values

trials = generate\_trials\_to\_calculate([{

'max\_bin':4, # default 256

'max\_depth':5, # default 6

'n\_estimators':578 # default 100

'learning\_rate':0.05508679239402551, # default 0.3

'subsample':0.8429852720715357, # default 1.0

'colsample\_bytree':0.8413894273173292, # default 1.0

'reg\_alpha': 0.809791155072757, # default 0.0

'reg\_lambda':1.4490119256389808, # default 1.0

'gamma':0.008478702584417519, # default 0.0

'min\_child\_weight':24.524635200338793, # default 1

}])

t1 = time.time()

# 1000trial [2:24:41, 8.68s/trial, best loss: -0.9080128034320879]

best\_params = fmin(f\_xgb, space\_xgb, algo=tpe.suggest, max\_evals=999, trials=trials)

t2 = time.time()

# 8681.310757875443

print("Time elapsed: ", t2-t1)

print('best:')

print(best\_params)

# verify

params = best\_params.copy()

# restore best hyper parameters

# {'colsample\_bytree': 0.8413894273173292, 'gamma': 0.008478702584417519, 'learning\_rate': 0.05508679239402551, 'max\_bin': 4, 'max\_depth': 5, 'min\_child\_weight': 24.524635200338793, 'n\_estimators': 578, 'reg\_alpha': 0.809791155072757, 'reg\_lambda': 1.4490119256389808, 'subsample': 0.8429852720715357}

params = {'colsample\_bytree': 0.8413894273173292, 'gamma': 0.008478702584417519, 'learning\_rate': 0.05508679239402551, 'max\_bin': 4, 'max\_depth': 5, 'min\_child\_weight': 24.524635200338793, 'n\_estimators': 578, 'reg\_alpha': 0.809791155072757, 'reg\_lambda': 1.4490119256389808, 'subsample': 0.8429852720715357}

params['max\_bin'] = params['max\_bin']+50

params['max\_depth'] = params['max\_depth']+3

#params['num\_leaves'] = params['num\_leaves']+20

#params['min\_child\_samples'] = params['min\_child\_samples']+10

#params['subsample\_freq'] = params['subsample\_freq']+1

params['n\_estimators'] = params['n\_estimators']+100

print(params)

# Original best parameters of GridSearchCV()

# Set extra\_trees=True to avoid overfitting

xgb\_best = XGBRegressor(objective ='reg:squarederror', seed = 0,verbosity=0, \*\*params) # CPU 4.96s/trial

acc = cross\_val\_score(xgb\_best, train\_X, train\_y).mean()

# 0.9080128034320879

print(acc)

# predict

xgb\_model\_full\_data = xgb\_best.fit(train\_X, train\_y)

# 0.9720045729712171

print(xgb\_model\_full\_data.score(train\_X,train\_y))

# 0.9012739170573821

print(xgb\_model\_full\_data.score(valid\_X,valid\_y))

# evalute model using the entire dataset from Train.csv

evaluateRegressor(train\_y,xgb\_model\_full\_data.predict(train\_X),"Train set ")

evaluateRegressor(valid\_y,xgb\_model\_full\_data.predict(valid\_X),"Valid set ")

3、GBR，可以参考下面的资料。

[参数](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html?highlight=gradientboostingregressor%20tuning)，[调参](https://zhuanlan.zhihu.com/p/55524425)。Scikit-Learn自带的GBR调参后的改善是比较大的，89.88%，只差微弱的一点点就跨过90%的门槛。

'''

trials = generate\_trials\_to\_calculate([{

'learning\_rate':0.1, # default 0.1

'n\_estimators':100-50, # default 100

'subsample':1.0, # default 1.0

'min\_samples\_split':2-2, # default 2

'min\_samples\_leaf':1-1, # default 1

'max\_depth':6-3, # default 3

'max\_features':6-3, # default sqrt(N)

'alpha':0.9, # default 0.9

'max\_leaf\_nodes':1501-2, # default None

}])

'''

# Auto search for better hyper parameters with hyperopt, only need to give a range

# 缩小参数取值范围，搜索会快很多

from sklearn.ensemble import GradientBoostingRegressor

space\_gbr = {

'learning\_rate': hp.uniform('learning\_rate', 0.003, 0.3),

'n\_estimators': hp.choice('n\_estimators', range(50, 501)),

'subsample': hp.uniform('subsample', 0.5, 1.0),

'min\_samples\_split': hp.choice('min\_samples\_split', range(2, 51)),

'min\_samples\_leaf': hp.choice('min\_samples\_leaf', range(1, 7)),

'max\_depth': hp.choice('max\_depth', range(3, 31)),

'max\_features': hp.choice('max\_features', range(3, 10)),

'alpha': hp.uniform('alpha', 0.001, 1),

'max\_leaf\_nodes': hp.choice('max\_leaf\_nodes', range(2, 2001)),

'min\_impurity\_decrease': hp.uniform('min\_impurity\_decrease', 0.0, 0.1),

}

def f\_gbr(params):

gbr = GradientBoostingRegressor(random\_state=0,verbose=0, \*\*params)

gbr\_model = gbr.fit(train\_X, train\_y)

acc = gbr\_model.score(valid\_X,valid\_y)

#acc = cross\_val\_score(gbr, train\_X, train\_y, n\_jobs=4).mean()

return -acc

# trials = Trials()

# Set initial values, start searching from the best point of GridSearchCV(), and default values

trials = generate\_trials\_to\_calculate([{

'learning\_rate':0.03189246699910192, # default 0.1

'n\_estimators':418, # default 100

'subsample':0.5162824805097872, # default 1.0

'min\_samples\_split':36, # default 2

'min\_samples\_leaf':1, # default 1

'max\_depth':7, # default 3

'max\_features':2, # default sqrt(N)

'alpha':0.29293429775771895, # default 0.9

'max\_leaf\_nodes':695, # default None

'min\_impurity\_decrease':0.0, # default 0

}])

t1 = time.time()

# 1000trial [1:11:27, 4.29s/trial, best loss: -0.9064136111667468]

# 1000trial [30:06, 1.81s/trial, best loss: -0.90169312112898]

best\_params = fmin(f\_gbr, space\_gbr, algo=tpe.suggest, max\_evals=999, trials=trials)

t2 = time.time()

# 4288.144340276718 1806.3821663856506

print("Time elapsed: ", t2-t1)

print('best:')

print(best\_params)

# verify

params = best\_params.copy()

# restore best hyper parameters

# {'alpha': 0.29293429775771895, 'learning\_rate': 0.03189246699910192, 'max\_depth': 7, 'max\_features': 2, 'max\_leaf\_nodes': 695, 'min\_samples\_leaf': 1, 'min\_samples\_split': 36, 'n\_estimators': 418, 'subsample': 0.5162824805097872}

# {'alpha': 0.9014933457984278, 'learning\_rate': 0.035668343067947715, 'max\_depth': 6, 'max\_features': 5, 'max\_leaf\_nodes': 943, 'min\_impurity\_decrease': 0.015183480929538314, 'min\_samples\_leaf': 2, 'min\_samples\_split': 30, 'n\_estimators': 440, 'subsample': 0.6109167820106534}

params = {'alpha': 0.9014933457984278, 'learning\_rate': 0.035668343067947715, 'max\_depth': 6, 'max\_features': 5, 'max\_leaf\_nodes': 943, 'min\_impurity\_decrease': 0.015183480929538314, 'min\_samples\_leaf': 2, 'min\_samples\_split': 30, 'n\_estimators': 440, 'subsample': 0.6109167820106534}

params['n\_estimators'] = params['n\_estimators']+50

params['min\_samples\_split'] = params['min\_samples\_split']+2

params['min\_samples\_leaf'] = params['min\_samples\_leaf']+1

params['max\_depth'] = params['max\_depth']+3

params['max\_features'] = params['max\_features']+3

params['max\_leaf\_nodes'] = params['max\_leaf\_nodes']+2

print(params)

# Original best parameters of GridSearchCV()

gbr\_best = GradientBoostingRegressor(random\_state=0,verbose=0, \*\*params)

acc = cross\_val\_score(gbr\_best, train\_X, train\_y).mean()

# 0.9064136111667468

print(acc)

# predict

gbr\_model\_full\_data = gbr\_best.fit(train\_X, train\_y)

# 0.9783374964600373

print(gbr\_model\_full\_data.score(train\_X,train\_y))

# 0.8988111539391862

print(gbr\_model\_full\_data.score(valid\_X,valid\_y))

# evalute model using the entire dataset from Train.csv

evaluateRegressor(train\_y,gbr\_model\_full\_data.predict(train\_X),"Train set ")

evaluateRegressor(valid\_y,gbr\_model\_full\_data.predict(valid\_X),"Valid set ")

如果把损失函数改为直接用测试集准确率的负值，则GBR可以达到90.17%，也跨过了90%的门槛。这时训练集的准确率是97.88%，交叉验证的准确率是90.49%，也非常好。这是个取巧的方法，泛化能力是不稳定的，因为与特定的测试集相关。

4、随机森林，可以参考下面的资料。

[随机森林学习与调参](https://blog.csdn.net/qq_37334135/article/details/86766014)，[A Beginner’s Guide to Random Forest Hyperparameter Tuning](https://www.analyticsvidhya.com/blog/2020/03/beginners-guide-random-forest-hyperparameter-tuning/)。

# Auto search for better hyper parameters with hyperopt, only need to give a range

'''

1: max\_depth，树深，3~50性能较好，过了10后测试集性能迅速下降，过拟合明显。有些问题可能需要较大的深度，如本例。

2: min\_sample\_split，分裂阈值，树的结点要分裂生长需要具有超过阈值的样本数，增大可以克服过拟合，过大会欠拟合。

3：max\_leaf\_nodes，树的最大叶子数，超过该值，树不再分裂生长。小值欠拟合，过大过拟合。本例要设置在1000~2000之间。

4: min\_samples\_leaf，叶子结点最少样本数，少于该阈值则树不再分裂生长。增大可以防止过拟合。

<100，容易过拟合，100~400性能较好，>500欠拟合明显。本例需要设置较小的值，如=1。

5: n\_estimators，树数，越大越好，但计算开销增大， >300无明显改进。

6: max\_features，树采用的特征数，3~10，经过一定的值后开始过拟合，默认特征数的平方根或6最佳。

7: max\_samples，数据采样率，(0,1],越低速度越快，但精度越低。超过一定的采样率，精度改善不明显。0.2以上合适。

'''

# 缩小参数取值范围，搜索会快很多

from sklearn.ensemble import RandomForestRegressor

space\_rf = {

'max\_depth': hp.choice('max\_depth', range(3, 51)),

#'max\_depth': hp.quniform('max\_depth', 3, 51, 1), # 用hp.quniform更方便一点

'min\_samples\_split': hp.choice('min\_samples\_split', range(2, 11)),

'max\_leaf\_nodes': hp.choice('max\_leaf\_nodes', range(2, 2001)),

'min\_samples\_leaf': hp.choice('min\_samples\_leaf', range(1, 501)),

'n\_estimators': hp.choice('n\_estimators', range(50, 501)),

'max\_features': hp.choice('max\_features', range(3, 10)),

'max\_samples': hp.uniform('max\_samples', 0.2, 1.0),

}

def f\_rf(params):

rf = RandomForestRegressor( random\_state = 0,verbose=0, n\_jobs = -1, \*\*params)

rf\_model = rf.fit(train\_X, train\_y)

acc = rf\_model.score(valid\_X,valid\_y)

#acc = cross\_val\_score(rf, train\_X, train\_y).mean()

return -acc

# trials = Trials()

# Set initial values, start searching from the best point of GridSearchCV(), and default values

trials = generate\_trials\_to\_calculate([{

'max\_depth':29-3, # default None

#'max\_depth':29 , # default None

'min\_samples\_split':2-2, # default 2 , >1

'max\_leaf\_nodes':1783-2, # default None

'min\_samples\_leaf':1-1, # default 1

'n\_estimators':431-50, # default 100

'max\_features':7-3, # default auto, sqrt(n)

'max\_samples': 0.9703002612349153, # default None

}])

t1 = time.time()

# 1000trial [28:31, 1.71s/trial, best loss: -0.8866269201223274]

# 1000trial [28:11, 1.69s/trial, best loss: -0.8868872465184857]

# 1000trial [28:56, 1.74s/trial, best loss: -0.8875152912944326]

best\_params = fmin(f\_rf, space\_rf, algo=tpe.suggest, max\_evals=999, trials=trials)

t2 = time.time()

# 1711.636605501175 1691.8947985172272 1736.71750998497

print("Time elapsed: ", t2-t1)

print('best:')

print(best\_params)

# verify

# params = best\_params.copy()

# restore best hyper parameters

# {'max\_depth': 22, 'max\_features': 5, 'max\_leaf\_nodes': 1513, 'max\_samples': 0.9945641511490839, 'min\_samples\_leaf': 0, 'min\_samples\_split': 4, 'n\_estimators': 368}

# {'max\_depth': 22, 'max\_features': 5, 'max\_leaf\_nodes': 1466, 'max\_samples': 0.9543521595369525, 'min\_samples\_leaf': 0, 'min\_samples\_split': 2, 'n\_estimators': 90}

# {'max\_depth': 26, 'max\_features': 4, 'max\_leaf\_nodes': 1781, 'max\_samples': 0.9703002612349153, 'min\_samples\_leaf': 0, 'min\_samples\_split': 0, 'n\_estimators': 381}

params = {'max\_depth': 26, 'max\_features': 4, 'max\_leaf\_nodes': 1781, 'max\_samples': 0.9703002612349153, 'min\_samples\_leaf': 0, 'min\_samples\_split': 0, 'n\_estimators': 381}

params['max\_depth'] = params['max\_depth']+3

params['min\_samples\_split'] = params['min\_samples\_split']+2

params['max\_leaf\_nodes'] = params['max\_leaf\_nodes']+2

params['min\_samples\_leaf'] = params['min\_samples\_leaf']+1

params['n\_estimators'] = params['n\_estimators']+50

params['max\_features'] = params['max\_features']+3

print(params)

# Original best parameters of GridSearchCV()

# Set extra\_trees=True to avoid overfitting

rf\_best = RandomForestRegressor( random\_state = 0,verbose=0, \*\*params)

acc = cross\_val\_score(rf\_best, train\_X, train\_y).mean()

#0.8866269201223276 0.8868872465184857 0.8875152912944326

print(acc)

# predict

rf\_model\_full\_data = rf\_best.fit(train\_X, train\_y)

# 0.9736106053470043 0.9757008947201352 0.9806938145048832

print(rf\_model\_full\_data.score(train\_X,train\_y))

# 0.8813185467662507 0.8820373113117788 0.8834725572033766

print(rf\_model\_full\_data.score(valid\_X,valid\_y))

# evalute model using the entire dataset from Train.csv

evaluateRegressor(train\_y,rf\_model\_full\_data.predict(train\_X),"Train set ")

evaluateRegressor(valid\_y,rf\_model\_full\_data.predict(valid\_X),"Valid set ")

5、SVM，可以参考下面的资料。

[SVM调优详解](https://7125messi.github.io/post/svm%E8%B0%83%E4%BC%98%E8%AF%A6%E8%A7%A3/)，[参数](https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html?highlight=svr#sklearn.svm.SVR)。因为迭代速度慢，只迭代了10次。

# Auto search for better hyper parameters with hyperopt, only need to give a range

# 缩小参数取值范围，搜索会快很多

from sklearn import svm

space\_SVR = {

'C': hp.uniform('C', 0.01, 100),

'gamma': hp.uniform('gamma', 0.00003, 0.3),

}

def f\_svr(params):

svr = svm.SVR(\*\*params)

svr\_model = svr.fit(train\_X, train\_y)

acc = svr\_model.score(valid\_X,valid\_y)

# acc = cross\_val\_score(svr, train\_X, train\_y).mean()

return -acc

# trials = Trials()

# Set initial values, start searching from the best point of GridSearchCV(), and default values

trials = generate\_trials\_to\_calculate([{

'C':1.0, # default 1.0

'gamma':0.03, # scale, auto or float, default scale

}])

t1 = time.time()

# 10trial [05:48, 34.84s/trial, best loss: -0.863203160989779]

# 1000trial [1:50:29, 6.63s/trial, best loss: -0.8659740738531577]

best\_params = fmin(f\_svr, space\_SVR, algo=tpe.suggest, max\_evals=999, trials=trials)

t2 = time.time()

# 348.4295656681061 6629.133600950241

print("Time elapsed: ", t2-t1)

print('best:')

# {'C': 27.458491133607748, 'gamma': 0.039507390135406495}

# {'C': 6.673350889023755, 'gamma': 0.05106238973376298}

print(best\_params)

# verify

# params = best\_params.copy()

params = {'C': 6.673350889023755, 'gamma': 0.05106238973376298}

print(params)

# Original best parameters of GridSearchCV()

# Set extra\_trees=True to avoid overfitting

svr\_best = svm.SVR(\*\*params)

acc = cross\_val\_score(svr\_best, train\_X, train\_y).mean()

# 0.863203160989779 0.8676735289704947

print(acc)

# predict

svr\_model\_full\_data = svr\_best.fit(train\_X, train\_y)

# 0.916419509431368 0.9109218210115129

print(svr\_model\_full\_data.score(train\_X,train\_y))

# 0.8641190047694843 0.8659740738531577

print(svr\_model\_full\_data.score(valid\_X,valid\_y))

# evalute model using the entire dataset from Train.csv

evaluateRegressor(train\_y,svr\_model\_full\_data.predict(train\_X),"Train set ")

evaluateRegressor(valid\_y,svr\_model\_full\_data.predict(valid\_X),"Valid set ")

上面这些算法既可以用于回归，也可以用于分类，通过贝叶斯优化可以高效的在有限的步数内找到准确率比较好的超参数组合，尤其是几个基于梯度提升决策树(GBDT)的算法，准确率和泛化能力都非常优秀，对于落地解决各种实际问题是很有帮助的。