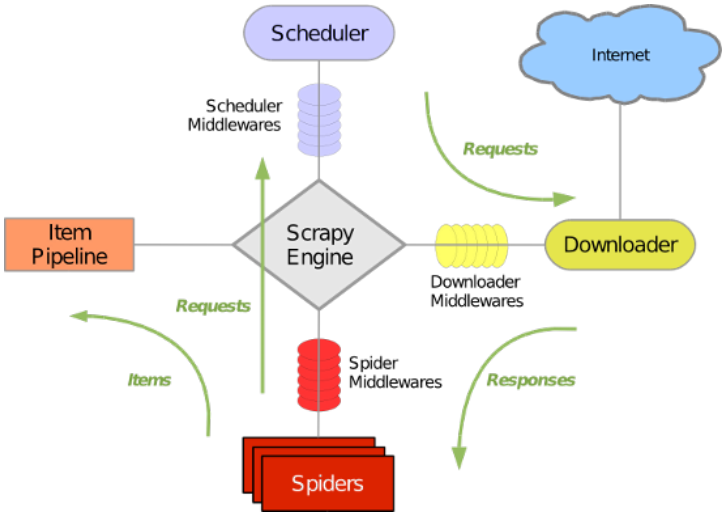
# 链家网上海地区二手房价格分析建模报告

我们小组选取的数据集来自于链家网上海地区二手房信息，在爬取数据和数据预处理后，对数据进行了建模与调试。

## 一、互联网数据爬取

### 1、爬虫框架说明

首先，我们采用了 python 的 scrapy 框架来爬取互联网数据，框架结构如图所示。




Scrapy 框架是一个为了爬取网站数据，提取结构性数据而编写的应用框架。 可以应用在包括数据挖掘，信息处理或存储历史数据等一系列的程序中。而网络爬虫，则是指抓取特定网站网页的 HTML 数据。

Scrapy 使用 Twisted 异步网络库来处理网络通讯，架构清晰，并且包含了各种中间件接口，可以灵活的完成各种需求。

从 scrapy 框架中可以看出，它主要包括了以下组件：

引擎(Scrapy)：用来处理整个系统的数据流处理，触发事务，是框架的核心。

调度器(Scheduler)：用来接收引擎发过来的请求，压入队列中，并在引擎再次请求的时候返回。可以想像成一个 URL（抓取网页的网址或者说是链接）的优先队列，由它来决定下一个要抓取的网址是什么，同时去除重复的网址。

下载器(Downloader)：用于下载网页内容，并将网页内容返回给 Spiders。(Scrapy 下载器是建立在 twisted 这个高效的异步模型上的。)

爬虫(Spiders)：爬虫用于从特定的网页中提取自己需要的信息，即所谓的实体(Item)。用户也可以从中提取出链接，让 Scrapy 继续抓取下一个页面。

项目管道(Pipeline)：负责处理爬虫从网页中抽取的实体，主要的功能是持久化实体、验证实体的有效性、清除不需要的信息。当页面被爬虫解析后，将被发送到项目管道，并经过几个特定的次序处理数据。

下载器中间件(Downloader Middlewares)：位于 Scrapy 引擎和下载器之间的框架，主要是处理 Scrapy 引擎与下载器之间的请求及响应。

爬虫中间件(Spider Middlewares)：介于 Scrapy 引擎和爬虫之间的框架，主要工作是处理爬虫的响应输入和请求输出。

调度中间件(Scheduler Middewares)：介于 Scrapy 引擎和调度之间的中间件，从 Scrapy 引擎发送到调度的请求和响应。

在使用 scrapy 框架进行爬虫时，其大致运行流程如下：

（1）首先，引擎从调度器中取出一个链接(URL)用于接下来的抓取。

（2）引擎把 URL 封装成一个请求(Request)传给下载器，下载器把资源下载下来，并封装成应答包(Response)。

（3）然后，爬虫解析 Response。

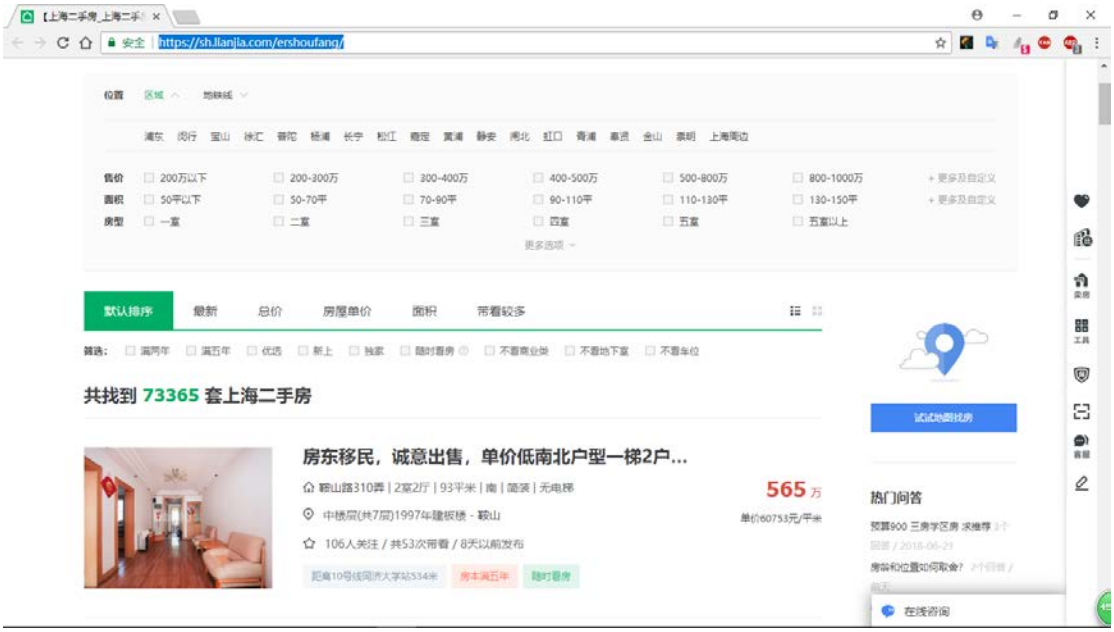
（4）如果解析出实体（Item），则交给实体管道进行进一步的处理。

（5）如果解析出的是链接（URL），则把 URL 交给 Scheduler 等待抓取。

**2、数据爬取部分流程及代码说明**

数据爬取部分主要代码由 class LianjiaSpider 完成。

①首先，确定了数据所在页面的 URL 为 https://sh.lianjia.com/ershoufang/。




```
# -*- coding: utf-8 -*-
import scrapy
from scrapy_redis.spiders import RedisSpider
from copy import deepcopy
from ..items import FangziItem
# from pprint import pprint
import re


class LianjiaSpider(scrapy.Spider):
    name = 'lianjia'
    allowed_domains = ['lianjia.com']
    start_urls = ['http://sh.lianjia.com/ershoufang/']
    # redis_key = 'lianjia'
```

②从链家上海二手房的网页上能够看出，这些房子的信息可以根据区域来划分。因此根据区域找到相应的子 url 网址，此部分功能由类 LianjiaSpider 中的函数 parse 完成。

```
def parse(self, response):
    # 区域列表    不限大分类
    a_list= response.xpath("//div[@id='plateList']/div/a")[1:]
    for a in a_list:
        item={}
        item["district"]= a.xpath("./@title").extract_first()
        item["zone_href"]= a.xpath("./@href").extract_first()
        if item["zone_href"] is not None:
            item["zone_href"]= 'http://sh.lianjia.com'+item["zone_href"]
            # pprint(item)
            # 请求小分类数据

            yield scrapy.Request(
                item["zone_href"],
                meta={"item":deepcopy(item)},
                callback=self.parse_s_cate

            )
```

③对网址中分类的数据进行解析，由类 LianjiaSpider 中函数 parse_s_cate 完成。

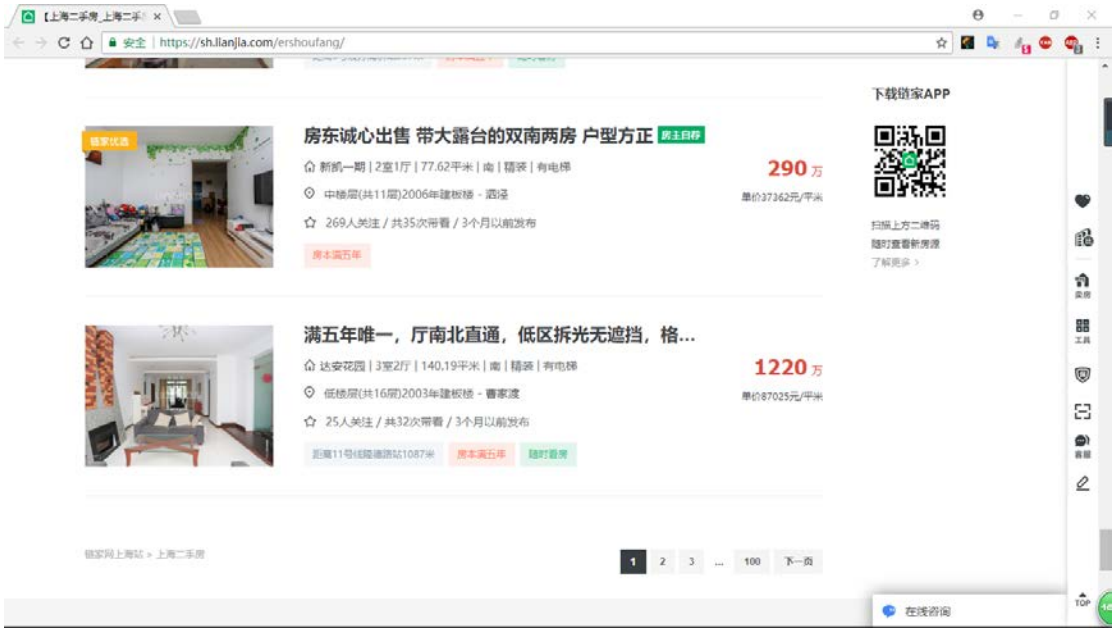
```
def parse_s_cate(self,response):
    # 解析出网址中分类的数据
    item= response.meta["item"]
    div_list=response.xpath("//div[@id='plateList']/div[@class='level2 gio_plate']/div")[1:]
    for div in div_list:
        item["s_cate"]= div.xpath("./a/text()").extract_first()
        item["s_cate_href"]= div.xpath("./a/@href").extract_first()
        # pprint(item)
        if item["s_cate_href"] is not None:
            item["s_cate_href"]= 'http://sh.lianjia.com'+item["s_cate_href"]

            # 请求具体数据
            yield scrapy.Request(
                item["s_cate_href"],
                meta={"item":deepcopy(item)},
                callback=self.parse_detail_info
            )
```

其中，请求具体数据步骤中调用了函数 parse_detail_info。

④从网页中提取到的信息分为各种数据项。因此，要将数据项放到 item 的对应成员中。针对每条二手房数据，设置 item 来分项存储数据。对于 item 的定义在类 FangziItem 中完成，具体定义如图所示。

```
import scrapy

class FangziItem(scrapy.Item):

    house_img = scrapy.Field()
    house_href = scrapy.Field()
    house_title = scrapy.Field()
    house_desc = scrapy.Field()
    house_price = scrapy.Field()
    singel_price = scrapy.Field()
    house_time = scrapy.Field()
    house_detail = scrapy.Field()
    district = scrapy.Field()
    zone_href = scrapy.Field()
    s_cate = scrapy.Field()
    s_cate_href = scrapy.Field()
```

⑤步骤④已经提到，从网页中提取到的数据分为各种数据项，要将数据项放到 item 的对应成员中。这一功能由类 LianjiaSpider 中的函数 parse_detail_info 完成。由于所有二手房信息并不在同一页面上，因此还要在函数中设置自动分页。具体代码如图所示。



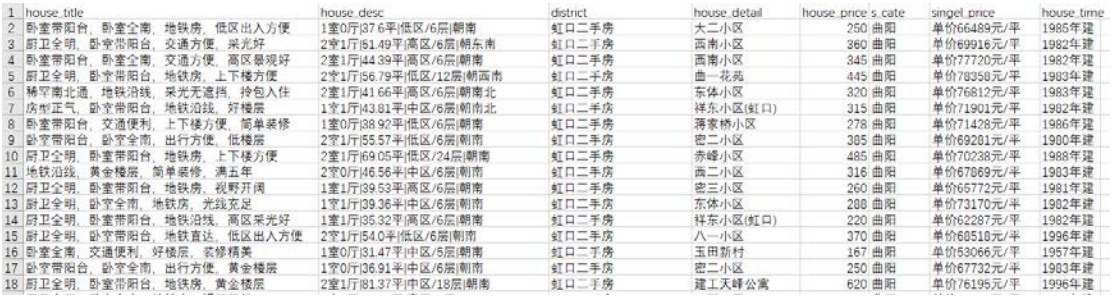

```
def parse_detail_info(self,response):
    item=response.meta["item"]
    # 提取具体信息
    li_list = response.xpath("//ul[@class='js_fang_list']/li")
    # 每套二手房
    for li in li_list:
        item["house_img"]= li.xpath("./a/img/@data-img-real").extract()
        # item["house_img"]= re.findall('(.*?)_\d+x\d+.jpg',item["house_img"])[0]
        item["house_href"]= li.xpath("./a/@href").extract_first()
        if item["house_href"] is not None:
            item["house_href"]= 'http://sh.lianjia.com'+item["house_href"]

        item["house_title"]= li.xpath("./div[@class='info']/div[1]/a/@title").extract()
        item["house_desc"]= li.xpath(".//div[@class='info-table']/div[1]/span/text()").extract()
        item["house_desc"]= [re.sub(r'\s+',"",i) for i in item["house_desc"]]
        item["house_desc"]= [i for i in item["house_desc"] if len(i)>0]
        item["house_price"]= li.xpath(".//div[@class='info-col price-item main']/span[1]/text()").extract()
        item["singel_price"]= li.xpath(".//div[@class='info-table']/div[2]/span[2]/text()").extract()
        item["singel_price"] =[ i.strip() for i in item["singel_price"] if item["singel_price"] is not None]

        # item["singel_price"] = item["singel_price"].strip() if item["singel_price"] is not None else None

        item["house_time"]= li.xpath(".//span[@class='info-col row2-text']/text()").extract()
        item["house_time"]= [re.sub(r'\s+|\|',"",i) for i in item["house_time"]]
        item["house_time"]= [i for i in item["house_time"] if len(i)>0]
        item["house_detail"] = li.xpath(".//span[@class='info-col row2-text']/a[@class='laisuzhou']/span/text()").extract()
        # print(response.request.meta['proxy'])
        # print(item)
        yield item
    # 设置自动翻页
    next_url_temp = response.xpath("//a[text()='下一页']/@href").extract_first()
    if next_url_temp is not None:
        print("*" * 100)
        yield scrapy.Request(
            "http://sh.lianjia.com" + next_url_temp,
            callback=self.parse_detail_info,
            meta={"item": deepcopy(item)}
        )
```

⑥将得到的数据存入 xlsx 文件中，共从网页上爬取到数据 7 万余条。每条数据包括七个属性，分别是：house_title、house_desc、district、house_detail、house_price　s_cate　、singel_price 和 house_time，如下表所示：

| 属性 | 定义 |
| --- | --- |
| house_title | 房源信息标题 |

| | |
|---|---|
| house_desc | 户型描述 |
| district | 所在区域 |
| house_detail | 所在小区 |
| house_price | 房价，单位：万 |
| s_cate | 所属街道 |
| singel_price | 单价 |
| house_time | 建房时间，精确到年 |

表 1 xlsx 文件属性描述



| | house_title | house_desc | district | house_detail | house_price | s_cate | singel_price | house_time |
|---|---|---|---|---|---|---|---|---|
| 2 | 卧室带阳台，卧室全南，地铁房，低区出入方便 | 1室0厅|37.6平|低区/6层|朝南 | 虹口二手房 | 大二小区 | 250 | 曲阳 | 单价66489元/平 | 1985年建 |
| 3 | 厨卫全明，卧室带阳台，交通方便，采光好 | 2室1厅|51.49平|高区/6层|朝东南 | 虹口二手房 | 西南小区 | 360 | 曲阳 | 单价69916元/平 | 1982年建 |
| 4 | 卧室带阳台，卧室全南，交通方便，高区景观好 | 2室1厅|44.39平|高区/6层|朝南 | 虹口二手房 | 西南小区 | 345 | 曲阳 | 单价77720元/平 | 1982年建 |
| 5 | 厨卫全明，卧室带阳台，地铁房，上下楼方便 | 2室1厅|56.79平|低区/12层|朝西南 | 虹口二手房 | 曲一花苑 | 445 | 曲阳 | 单价78358元/平 | 1983年建 |
| 6 | 精华南北通，地铁沿线，采光无遮挡，拎包入住 | 2室1厅|41.66平|高区/6层|朝南北 | 虹口二手房 | 东体小区 | 320 | 曲阳 | 单价76812元/平 | 1983年建 |
| 7 | 房型正气，卧室带阳台，地铁沿线，好楼层 | 1室1厅|43.81平|中区/6层|朝南北 | 虹口二手房 | 祥东小区(虹口) | 315 | 曲阳 | 单价71901元/平 | 1982年建 |
| 8 | 卧室带阳台，交通便利，上下楼方便，简单装修 | 1室0厅|38.92平|低区/6层|朝南 | 虹口二手房 | 蒋家桥小区 | 278 | 曲阳 | 单价71428元/平 | 1986年建 |
| 9 | 卧室带阳台，卧室全南，出行方便，低楼层 | 2室1厅|55.57平|低区/6层|朝南 | 虹口二手房 | 密二小区 | 385 | 曲阳 | 单价69281元/平 | 1900年建 |
| 10 | 厨卫全明，卧室带阳台，地铁房，上下楼方便 | 2室1厅|69.05平|低区/24层|朝南 | 虹口二手房 | 赤峰小区 | 485 | 曲阳 | 单价70238元/平 | 1988年建 |
| 11 | 地铁沿线，黄金楼层，简单装修，满五年 | 2室0厅|46.56平|中区/6层|朝南 | 虹口二手房 | 西二小区 | 316 | 曲阳 | 单价67869元/平 | 1983年建 |
| 12 | 厨卫全明，卧室带阳台，地铁房，视野开阔 | 1室1厅|39.53平|高区/6层|朝南 | 虹口二手房 | 密三小区 | 260 | 曲阳 | 单价65772元/平 | 1981年建 |
| 13 | 厨卫全明，卧室全南，地铁房，光线充足 | 1室1厅|39.36平|中区/6层|朝南 | 虹口二手房 | 东体小区 | 288 | 曲阳 | 单价73170元/平 | 1982年建 |
| 14 | 厨卫全明，卧室带阳台，地铁沿线，高区采光好 | 1室1厅|35.32平|高区/6层|朝南 | 虹口二手房 | 祥东小区(虹口) | 220 | 曲阳 | 单价62287元/平 | 1982年建 |
| 15 | 厨卫全明，卧室带阳台，地铁直达，低区出入方便 | 2室1厅|54.0平|低区/6层|朝南 | 虹口二手房 | 八一小区 | 370 | 曲阳 | 单价68518元/平 | 1996年建 |
| 16 | 卧室全南，交通便利，好楼层，装修精美 | 1室0厅|31.47平|中区/5层|朝南 | 虹口二手房 | 玉田新村 | 167 | 曲阳 | 单价53066元/平 | 1957年建 |
| 17 | 卧室带阳台，卧室全南，出行方便，黄金楼层 | 1室0厅|36.91平|中区/6层|朝南 | 虹口二手房 | 密二小区 | 250 | 曲阳 | 单价67732元/平 | 1983年建 |
| 18 | 厨卫全明，卧室带阳台，地铁房，黄金楼层 | 2室1厅|81.37平|中区/18层|朝南 | 虹口二手房 | 建工天峰公寓 | 620 | 曲阳 | 单价76195元/平 | 1996年建 |

# 二、中文分词和预处理

以上步骤所得的数据是中文数据，为了便于分析和数据建模，在正式建模前还需要对数据进行处理，将自然语言数据转化为数字数据，将各个特征规范化，并进行缺失值填充和数据清洗等操作，除去重复数据、不全不足数据，以简化建模过程，提高建模效率。在清洗掉重复数据之后，得到 73306 份有效数据。

在数据预处理过程中，又从每条数据中提取出了十六条特征，反映每套二手房的特征描述。

为了简单的反映房子的地理位置，将其所在区县定义为离散的整数值，具体定义如下：

```
def foo(var):
    return {
            '虹口': '1',
            '奉贤': '2',
            '金山': '3',
            '浦东': '4',
            '静安': '5',
            '长宁': '6',
            '黄浦': '7',
            '青浦': '8',
            '嘉定': '9',
            '徐汇': '10',
            '闵行': '11',
            '普陀': '12',
            '杨浦': '13',
            '宝山': '14',
            '崇明': '15',
            '松江': '16',
            '闸北': '17',
            '上海周边': '18',
    }.get(var,'error')
```

在对数据进行处理之前，先将 xlsx 文件复制到.txt 文件中。再遍历每条数据，根据属性内容给对应特征赋值，并在 house_price.txt 文件中填入相应列。

```
datal=open('C:\Users\jiatu\Desktop\数据挖掘\data.txt','r')
house_price=open('C:\Users\jiatu\Desktop\数据挖掘\house_price.txt','w')
lines=datal.readlines()
#lines[0]=lines[0].decode("cp936")
#print lines[0]
#.decode('unicode_escape')


# 遍历每一条数据
for i in range(73306):
    linedetail = lines[i].split('\t')
    house_price.write(linedetail[4]) #print house price
    house_price.write('\t')

    house_desc=linedetail[1].split('|')
    house_desc1=house_desc[0].split('室')
    house_price.write(house_desc1[0]) #print x shi
    house_price.write('\t')

    house_price.write(house_desc1[1][0]) #print x ting
    house_price.write('\t')

    house_desc2=house_desc[1].split('平') #print x ping
    house_price.write(house_desc2[0])
    house_price.write('\t')

    house_desc3=house_desc[2].split('区')  #print high/medium/low
    house_price.write(('0' if(house_desc3[0]=="低") else '1' if(house_desc3[0]=='中') else '2'))
    house_price.write('\t')

    house_desc4=house_desc[2].split('层') #print floor
    house_desc5=house_desc4[0].split('/')
    house_price.write(house_desc5[-1])
    house_price.write('\t')

    #house_desc4=house_desc[3].split('朝')  #print direction
    if ('南' in house_desc[-1]):  #print direction
        house_price.write('1')
    else:
        house_price.write('0')#数据填充
    house_price.write('\t')

    district=linedetail[2].split('二')  #print district
    house_price.write(foo(district[0]))
    house_price.write('\t')

    house_time=linedetail[7].split('年') #print house time
    if (house_time[0] != '\n'):
        house_price.write(house_time[0])
    else:
        house_price.write('2000')#数据填充
    house_price.write('\t')

    single_price=linedetail[6].split('价') #print single price
    single_price1=single_price[1].split('元')
    house_price.write(single_price1[0])
    house_price.write('\t')

    if ('地铁' in linedetail[0] or '交通' in linedetail[0] or '出行' in linedetail[0] or '出入' in linedetail[0]):  #print subway
        house_price.write('1')
    else:
        house_price.write('0')#数据填充
    house_price.write('\t')

    if ('阳台' in linedetail[0]):  #print balcony
        house_price.write('1')
    else:
        house_price.write('0')#数据填充
    house_price.write('\t')

    if ('光' in linedetail[0] or '明' in linedetail[0]):  #print light
        house_price.write('1')
    else:
        house_price.write('0')#数据填充
    house_price.write('\t')

    if ('飘窗' in linedetail[0]):  #print bay window
        house_price.write('1')
    else:
        house_price.write('0')#数据填充
    house_price.write('\t')
```

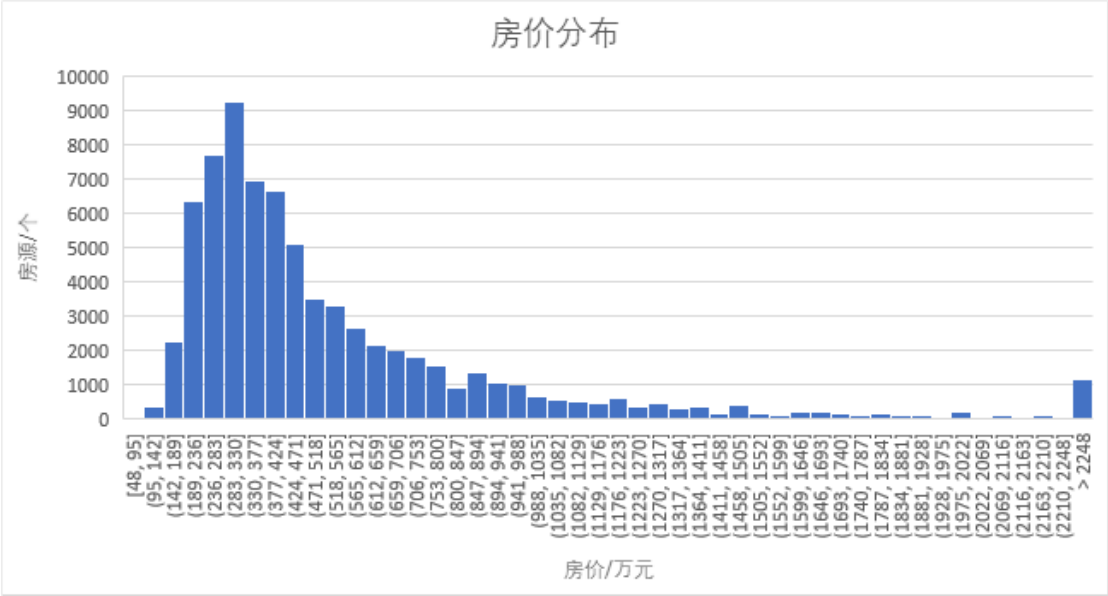
```
    if ('简单装修' in linedetail[0] or '中等装修' in linedetail[0] or '省装修费' in linedetail[0] or '拎包入住' in linedetail[0]): #pr
        house_price.write('1')
    elif ('精装修' in linedetail[0] or '品质装修' in linedetail[0] or '装修精' in linedetail[0] or '品味装修' in linedetail[0] or '装修
        house_price.write('2')
    else:
        house_price.write('0')#数据填充
    house_price.write('\t')

    if ('上下楼方便' in linedetail[0]):  #print elevator
        house_price.write('1')
    else:
        house_price.write('0')#数据填充
    house_price.write('\t')

    house_price.write('\n')

linedetail=lines[0].split('\t')

data1.close()
house_price.close()
```

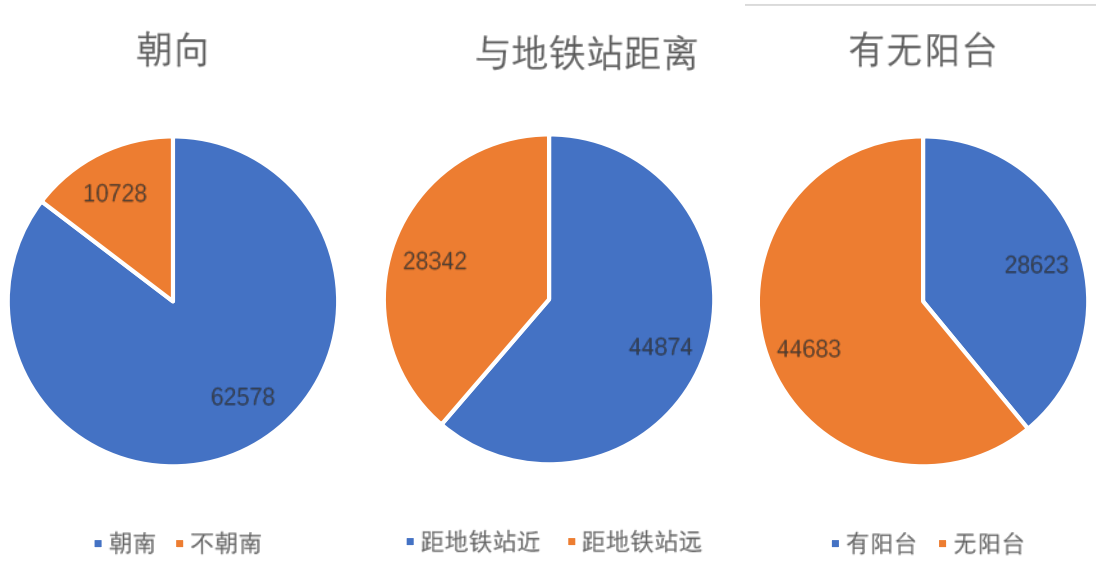
（遍历每条数据，提取特征并写入 house_price.txt）

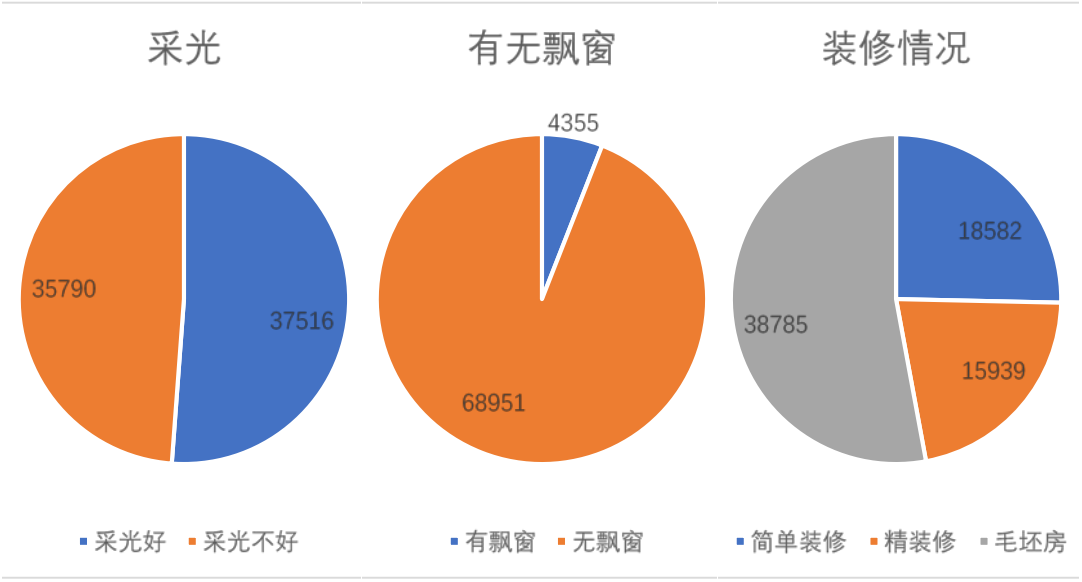
对 house_price.txt 中的十六列特征的数值的解释说明如下：

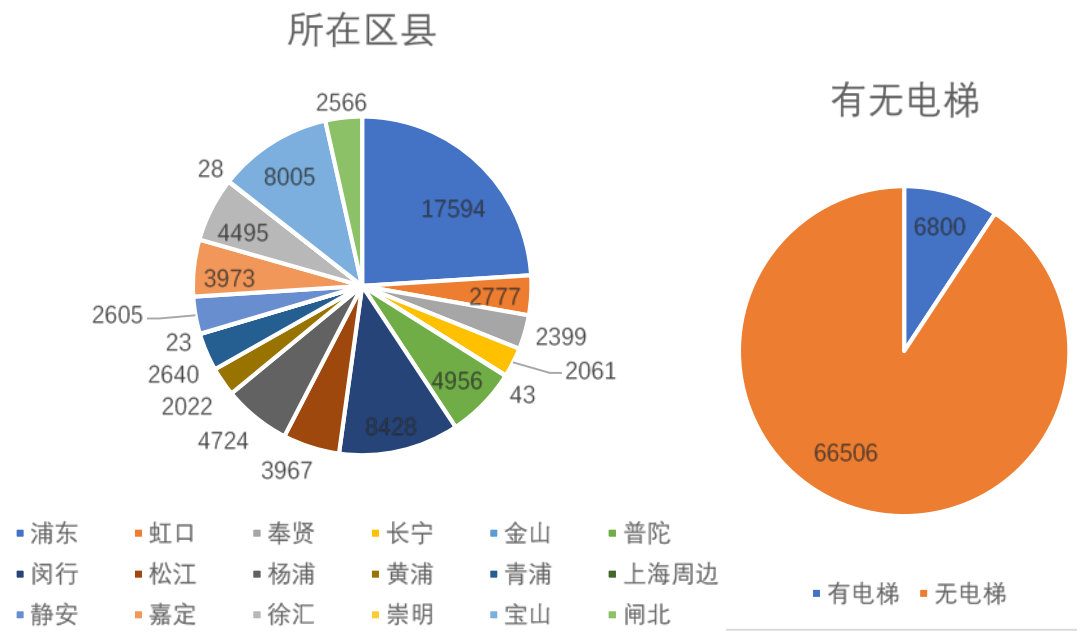
| 列数 | 说明 |
|---|---|
| 1 | 总房价，单位：万元 |
| 2 | 房间数 |
| 3 | 厅数 |
| 4 | 面积，单位：平米 |
| 5 | 楼层高度<br>0：低区楼层<br>1：中区楼层<br>2：高区楼层 |
| 6 | 楼层数 |
| 7 | 朝向<br>0：不朝南<br>1：朝南 |
| 8 | 所在区县，赋值为整数<br>1：虹口 2：奉贤 3：金山 4：浦东 5：静安 6：长宁<br>7：黄埔 8：青浦 9：嘉定 10：徐汇 11：闵行 12：普陀<br>13：杨浦 14：宝山 15：崇明 16：松江 17：闸北<br>18：上海周边 |
| 9 | 建房时间，单位：年 |
| 10 | 单价，单位：元/平米 |
| 11 | 距离地铁站远近<br>0：远<br>1：近 |
| 12 | 有无阳台<br>0：无<br>1：有 |
| 13 | 采光情况<br>0：不好<br>1：好 |
| 14 | 有无飘窗<br>0：无<br>1：有 |
| 15 | 装修情况<br>0：无装修<br>1：简单装修<br>2：精装修 |
| 16 | 有无电梯<br>0：无<br>1：有 |

为了更加直观地显示出所爬取数据的特征分布情况，对一些重要特征进行了统计作图，所做分布图如下所示。






# 三、数据建模、调优和测试

**1、导入数据集 house_price.txt**

**①导入数据集后并展示数据集部分数值及数据集信息**

```
#用pandas导入数据集house_price.txt
import pandas as pd
data = pd.read_table("E:\DM2\house_price.txt",delim_whitespace=True)
#展示数据集前几个数值
print (data.head)
#展示数据集信息
print(data.info())
```

部分输出结果如下图所示：

```
<bound method NDFrame.head of        total_price  room_num  hall_num    area  district  layer  orientation  \
0              250         1         0   37.60         0      6            1
1              360         2         1   51.49         2      6            1
2              345         2         1   44.39         2      6            1
3              445         2         1   56.79         0     12            1
4              320         2         1   41.66         2      6            1
5              315         1         1   43.81         1      6            1
6              278         1         0   38.92         0      6            1
7              385         2         1   55.57         0      6            1
8              485         2         1   69.05         0     24            1
9              316         2         0   46.56         1      6            1
10             260         1         1   39.53         2      6            1
11             288         1         1   39.36         1      6            1
12             220         1         1   35.32         2      6            1
13             370         2         1   54.00         0      6            1
14             167         1         0   31.47         1      5            1
15             250         1         0   36.91         1      6            1
16             620         2         1   81.37         1     18            1
17             205         1         1   33.47         2      5            0
18             511         3         0   72.96         2      6            1
19             190         1         1   32.66         2      6            1
20             300         1         1   48.46         0     21            0
21             400         2         1   61.42         2      6            1
22             180         1         0   31.04         1      5            0
23             515         2         1   66.64         2     25            1
24             572         3         0   88.07         2     19            1
25             988         3         2  146.64         1      9            1
26             340         2         1   52.34         2      6            1
27             280         1         1   39.81         1      5            1
28             520         2         1   66.04         1     24            0
29             185         1         0   32.75         1      5            1
...            ...       ...       ...     ...       ...    ...          ...

[73306 rows x 16 columns]>
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 73306 entries, 0 to 73305
Data columns (total 16 columns):
total_price    73306 non-null int64
room_num       73306 non-null int64
hall_num       73306 non-null int64
area           73306 non-null float64
district       73306 non-null int64
layer          73306 non-null int64
orientation    73306 non-null int64
location       73306 non-null int64
build_time     73306 non-null int64
price_per      73306 non-null int64
subway         73306 non-null int64
balcony        73306 non-null int64
light          73306 non-null int64
bay_window     73306 non-null int64
Decoration     73306 non-null int64
elevator       73306 non-null int64
dtypes: float64(1), int64(15)
memory usage: 8.9 MB
None
```

从上述输出结果来看, 数据集中共有 73306 条有效数据, 16 个特征值, 特征值中除 area 属性为 float 类型外，其余为 int 类型

**②再观察每列数据的标准差、最大值、最小值以及分位值。**输入：

```
print(data.describe())
```

获得关于数据集的描述：

| | total_price | room_num | hall_num | area | district | \ |
|---|---|---|---|---|---|---|
| count | 73306.000000 | 73306.000000 | 73306.000000 | 73306.000000 | 73306.000000 | |
| mean | 557.157149 | 2.261779 | 1.476714 | 95.735702 | 1.135419 | |
| std | 563.546757 | 0.949057 | 0.664317 | 67.258458 | 0.820984 | |
| min | 48.000000 | 1.000000 | 0.000000 | 20.100000 | 0.000000 | |
| 25% | 290.000000 | 2.000000 | 1.000000 | 58.560000 | 0.000000 | |
| 50% | 400.000000 | 2.000000 | 2.000000 | 82.710000 | 1.000000 | |
| 75% | 630.000000 | 3.000000 | 2.000000 | 111.430000 | 2.000000 | |
| max | 35000.000000 | 9.000000 | 9.000000 | 4588.000000 | 2.000000 | |

| | layer | orientation | location | build_time | price_per | \ |
|---|---|---|---|---|---|---|
| count | 73306.000000 | 73306.000000 | 73306.000000 | 73306.000000 | 73306.000000 | |
| mean | 10.501296 | 0.853655 | 8.908575 | 1999.309620 | 57316.623128 | |
| std | 7.836873 | 0.353455 | 4.563774 | 9.624029 | 21328.489238 | |
| min | 1.000000 | 0.000000 | 1.000000 | 1911.000000 | 7628.000000 | |
| 25% | 6.000000 | 1.000000 | 4.000000 | 1994.000000 | 42121.000000 | |
| 50% | 6.000000 | 1.000000 | 10.000000 | 2000.000000 | 54993.000000 | |
| 75% | 14.000000 | 1.000000 | 13.000000 | 2006.000000 | 68548.750000 | |
| max | 80.000000 | 1.000000 | 18.000000 | 2017.000000 | 199879.000000 | |

| | subway | balcony | light | bay_window | Decoration | \ |
|---|---|---|---|---|---|---|
| count | 73306.000000 | 73306.000000 | 73306.000000 | 73306.000000 | 73306.000000 | |
| mean | 0.612146 | 0.390459 | 0.511773 | 0.059409 | 0.688347 | |
| std | 0.487264 | 0.487857 | 0.499865 | 0.236389 | 0.805851 | |
| min | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | |
| 25% | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | |
| 50% | 1.000000 | 0.000000 | 1.000000 | 0.000000 | 0.000000 | |
| 75% | 1.000000 | 1.000000 | 1.000000 | 0.000000 | 1.000000 | |
| max | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 2.000000 | |

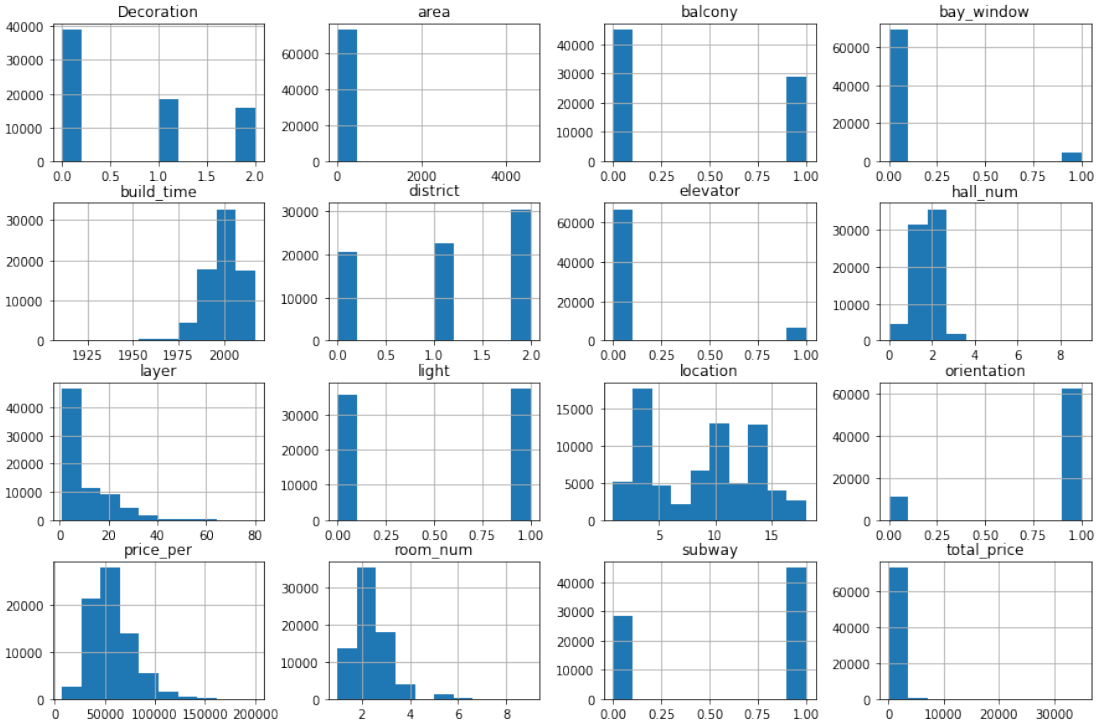
| | elevator |
|---|---|
| count | 73306.000000 |
| mean | 0.092762 |
| std | 0.290100 |
| min | 0.000000 |
| 25% | 0.000000 |
| 50% | 0.000000 |
| 75% | 0.000000 |
| max | 1.000000 |

从以上输出值可以获得一些统计信息，比如我们要预测的房价总值，均值为 557 万，最小值为 48 万，最大值为 35000 万，有 25%的房价小于 290 万，50%的房子总价小于 400 万，75%的房子总价小于 630 万。

**2、特征分析之独立分析**

**①分析每个特征的取值范围，并用直方图作可视化视图**

```
#分析每个特征的取值范围以及分布情况，用直方图可视化
import matplotlib.pyplot as plt
data.hist(figsize=(15,10))#bins 柱子个数
plt.savefig('a.jpg')  #保存图片
plt.show()
```




由生活常识不难得知，列 price_per 和列 total_price 具有很强的相关性，根据验证，total_price=price_per*area，因此可以在数据集中删除 price_per 这一列。

**②删去列 price_per**

```
#删去每平方米价格一列
import numpy as np
from pandas import Series, DataFrame
data_ind=data.drop(['price_per'],axis=1,inplace=False)
print(data_ind.head)
```

剩下 15 列：

```
[73306 rows x 15 columns]>
```

**3、划分测试集和训练集，以及分出属性和预测 label**

```
#划分测试集
import sklearn
from sklearn.model_selection import train_test_split
data_train,data_test=train_test_split(data_ind,test_size=0.3)
X_train = data_train.iloc[:,1:15]
X_test = data_test.iloc[:,1:15]
y_train = data_train.iloc[:,0]
y_test = data_test.iloc[:,0]
#print(X_train)
##X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.3)
```

这里将数据集的 70%作为训练集，30%作为测试集。

**4、特征相关性分析**

**①通过相关系数法计算各个特征与 total_price 之间的相关系数：**

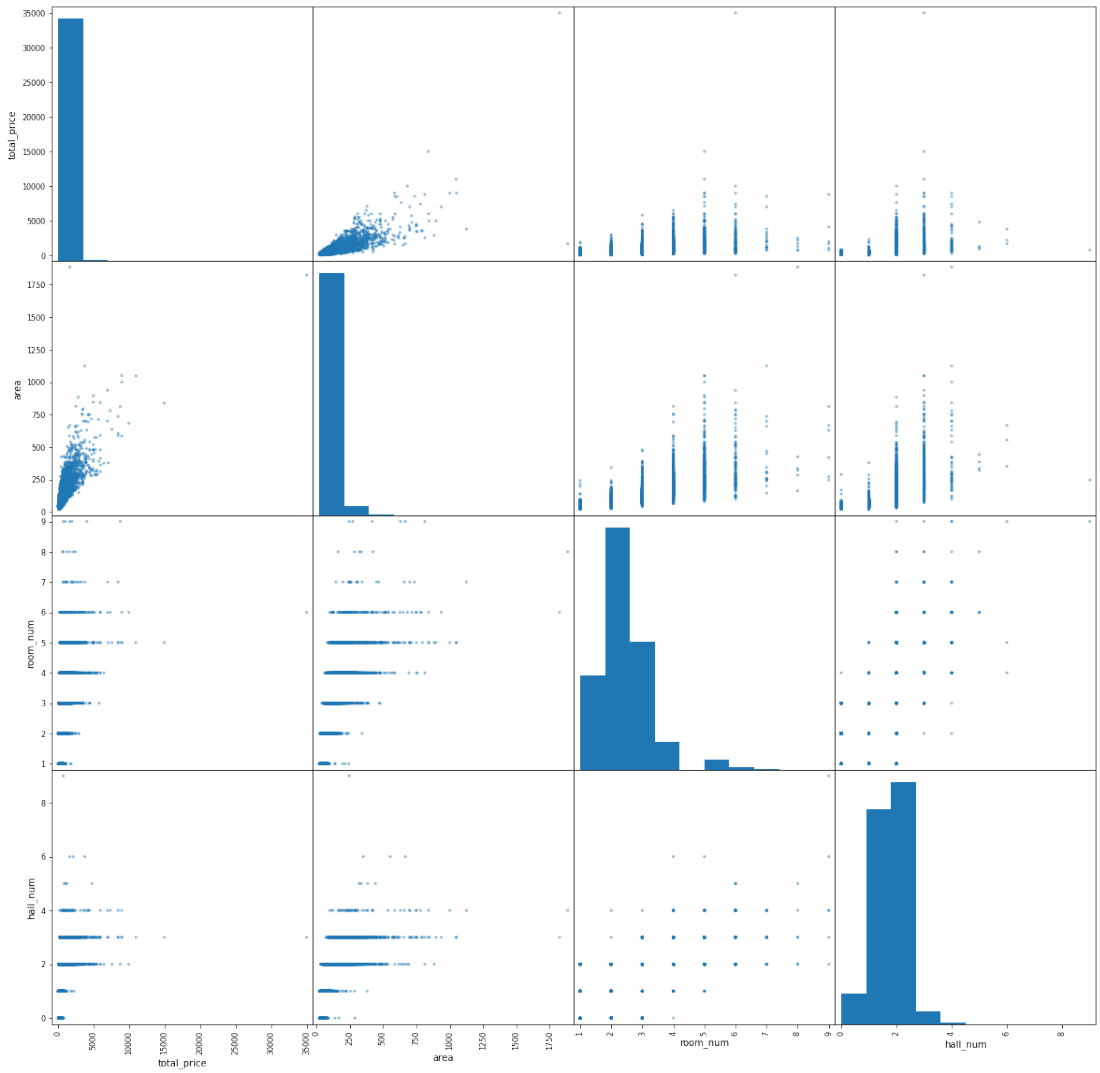
```
##通过计算相关系数矩阵找出特征之间的相关性
corr_matrix = data_train.corr()
print(corr_matrix['total_price'].sort_values(ascending=False))
```

```
total_price    1.000000
area           0.781092
room_num       0.551551
hall_num       0.444651
layer          0.259405
build_time     0.191311
subway         0.081135
Decoration     0.037278
district       0.026189
bay_window     0.004229
elevator       0.000056
orientation   -0.024573
location      -0.073273
light         -0.174412
balcony       -0.186570
Name: total_price, dtype: float64
```

从输出结果看出，total_price、area、room_num、hall_num 相关性最强。

**②将相关性最强的四个特征之间的分布做可视化**，结果如下：

```
##绘制相关系数最大的四个特征之间分布
from pandas.plotting import scatter_matrix
attribute = ['total_price','area','room_num','hall_num']
scatter_matrix(data_test[attribute],figsize=(20,20))
```




从上图得知，area 与 total_price 呈正相关。

**5、数据预处理**

在对数据集进行训练之前，需要对数据进行预处理，将数据标准化并进行编码，使特征的各种行为更加突出。

**①离散特征独热编码**

使用简单的序列对分类值进行表示后，进行模型训练时可能会产生一个问题，即特征因为数字值的不同而影响模型的训练效果。在模型训练的过程中不同的值会使得同一特征在样本中的权重发生变化，假如直接编码成 1000，比编码成 1 对模型的的影响更大。为了解决上述的问题，使训练过程中不受到因为分类值表示的问题对模型产生的负面影响，引入独热码对分类型的特征进行独热码编码。

```
from sklearn.preprocessing import OneHotEncoder
a=['district','orientation','location','subway','balcony','light','bay_window','Decoration','elevator']
encoder = OneHotEncoder(sparse=False)
train_cat_1hot = encoder.fit_transform(X_train[a])
test_cat_1hot = encoder.transform(X_test[a])
print(train_cat_1hot.dtype)
print(train_cat_1hot.shape)
print(train_cat_1hot)
print(test_cat_1hot.dtype)
print(test_cat_1hot.shape)
print(test_cat_1hot)
```

```
float64
(51314, 36)
[[0. 0. 1. ... 0. 1. 0.]
 [1. 0. 0. ... 1. 1. 0.]
 [0. 1. 0. ... 1. 1. 0.]
 ...
 [0. 0. 1. ... 0. 1. 0.]
 [0. 0. 1. ... 1. 1. 0.]
 [1. 0. 0. ... 0. 1. 0.]]
float64
(21992, 36)
[[0. 1. 0. ... 0. 1. 0.]
 [0. 0. 1. ... 0. 1. 0.]
 [0. 1. 0. ... 0. 1. 0.]
 ...
 [1. 0. 0. ... 0. 0. 1.]
 [0. 0. 1. ... 1. 1. 0.]
 [0. 0. 1. ... 0. 1. 0.]]
```

**②连续特征标准化**

对连续特征进行缩放，便于后续模型的训练与测试。

```
from sklearn.preprocessing import StandardScaler
#print(X_train)
train_cat=pd.concat([X_train.iloc[:,0:3],X_train.iloc[:,4],X_train.iloc[:,7]],axis=1)
test_cat=pd.concat([X_test.iloc[:,0:3],X_test.iloc[:,4],X_test.iloc[:,7]],axis=1)
#print(train_cat)
scaler =StandardScaler()
train_cat_scaled = scaler.fit_transform(train_cat)
test_cat_scaled = scaler.transform(test_cat) #这里是transform，，StandardScaler只使用训练集fit一次，这样保证训练集和测试集使用相同的标准进行的特征缩放。
print(train_cat_scaled.shape)
print(train_cat_scaled.dtype)
print(train_cat_scaled)
print(test_cat_scaled.shape)
print(test_cat_scaled.dtype)
print(test_cat_scaled)
```

```
(51314, 5)
float64
[[-0.27390966  0.79009419  0.31744546  0.95220288  0.17745588]
 [-0.27390966  0.79009419 -0.74231748 -0.700498   -1.68360642]
 [-0.27390966  0.79009419  1.10246604  0.44367953 -7.78375506]
 ...
 [ 0.77924486 -0.71472703 -0.18763494 -0.57336716 -0.44289822]
 [ 0.77924486  0.79009419  0.20324174 -0.57336716 -0.44289822]
 [-1.32706417 -0.71472703 -0.53960218  2.47777292  0.48763293]]
(21992, 5)
float64
[[-1.32706417 -0.71472703 -0.88147077 -0.700498   -1.68360642]
 [ 0.77924486  0.79009419 -0.11828106 -0.44623632  1.21137938]
 [-0.27390966 -0.71472703 -0.31208581  1.46072623  0.07406353]
 ...
 [-0.27390966 -0.71472703 -0.4371307   0.95220288 -1.06325232]
 [ 0.77924486  0.79009419  0.62188968  0.95220288  0.07406353]
 [-1.32706417 -2.21954824 -0.96805174 -0.57336716 -0.54629057]]
```

**③合并处理好的数据集作为最终数据集**

```
#数据处理完合并成为最终的数据集
# print(np.array(X_train[a]).shape)
# print(np.array(X_train[a]).ndim)
# print(np.array(X_train[a]).dtype)
# c=np.array(X_train[a])
# print(c.astype(np.float64).dtype)
#X_train_final=np.concatenate((c.astype(np.float64),train_cat_scaled),axis=1)
X_train_final=np.concatenate((train_cat_1hot,train_cat_scaled),axis=1)
X_test_final=np.concatenate((test_cat_1hot,test_cat_scaled),axis=1)
print(X_train_final.shape)
print(X_train_final)
print(X_test_final.shape)
print(X_test_final)
```

```
(51314, 41)
[[ 0.          0.          1.          ...  0.31744546  0.95220288
   0.17745588]
 [ 1.          0.          0.          ... -0.74231748 -0.700498
  -1.68360642]
 [ 0.          1.          0.          ...  1.10246604  0.44367953
  -7.78375506]
 ...
 [ 0.          0.          1.          ... -0.18763494 -0.57336716
  -0.44289822]
 [ 0.          0.          1.          ...  0.20324174 -0.57336716
  -0.44289822]
 [ 1.          0.          0.          ... -0.53960218  2.47777292
   0.48763293]]
(21992, 41)
[[ 0.          1.          0.          ... -0.88147077 -0.700498
  -1.68360642]
 [ 0.          0.          1.          ... -0.11828106 -0.44623632
   1.21137938]
 [ 0.          1.          0.          ... -0.31208581  1.46072623
   0.07406353]
 ...
 [ 1.          0.          0.          ... -0.4371307   0.95220288
  -1.06325232]
 [ 0.          0.          1.          ...  0.62188968  0.95220288
   0.07406353]
 [ 0.          0.          1.          ... -0.96805174 -0.57336716
  -0.54629057]]
```

**6、训练模型选择**

本阶段共选取了四个模型进行对比，分别是线性回归模型、非线性回归模型、随机森林模型以及 XGBoost 模型。对每个模型计算了方差、平均绝对误差、R 平方值三个指标，根据这三个指标来进行选择。

**①线性回归模型**

```
#线性回归
from sklearn import metrics
from sklearn.linear_model import LinearRegression
model1 = LinearRegression()
model1.fit(X_train_final,y_train)
# 对测试集进行预测
ans = model1.predict(X_test_final)

#模型评估
from sklearn.metrics import mean_squared_error,mean_absolute_error,r2_score
#方差
print ("MSE",mean_squared_error(ans,y_test))
#平均绝对误差
print("MAE",mean_absolute_error(ans,y_test))
#R2分数
print("R2",r2_score(ans,y_test))

#可视化
print('Slope:%.3f' %model1.coef_[0])
print('Intercept: %.3f' %model1.intercept_)
```

```
MSE 97075.38913795358
MAE 132.44900986722445
R2 0.546683425181091
```

②非线性回归模型

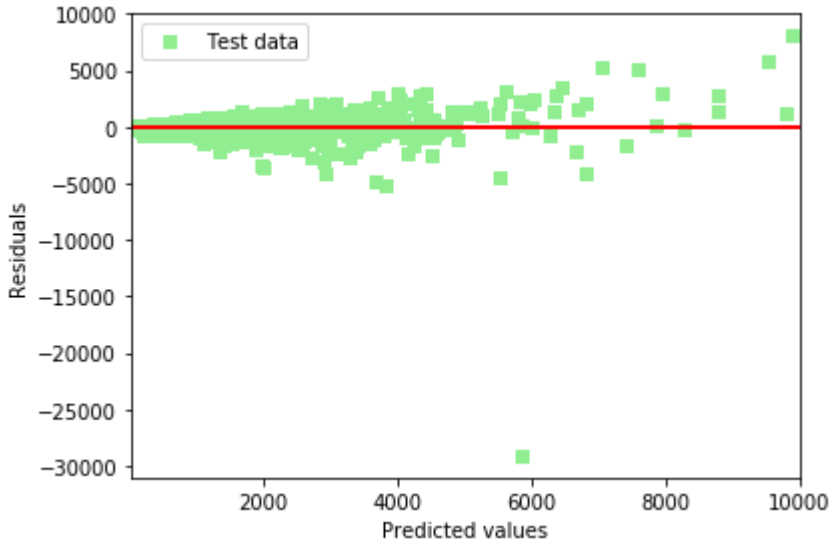
```
#非线性回归
from sklearn.tree import DecisionTreeRegressor
model2 = DecisionTreeRegressor()
model2.fit(X_train_final,y_train)

# 对测试集进行预测
ans = model2.predict(X_test_final)

#模型评估
from sklearn.metrics import mean_squared_error,mean_absolute_error,r2_score
#方差
print ("MSE",mean_squared_error(ans,y_test))
#平均绝对误差
print("MAE",mean_absolute_error(ans,y_test))
#R2分数
print("R2",r2_score(ans,y_test))
```

```
MSE 100813.08822273756
MAE 112.72736298047774
R2 0.6817544365083501
```

③随机森林模型

```
#随机森林
from sklearn.ensemble import RandomForestRegressor
forest=RandomForestRegressor()
forest.fit(X_train_final,y_train)
# 对测试集进行预测
ans = forest.predict(X_test_final)

#模型评估
from sklearn.metrics import mean_squared_error,mean_absolute_error,r2_score
#方差
print ("MSE",mean_squared_error(ans,y_test))
#平均绝对误差
print("MAE",mean_absolute_error(ans,y_test))
#R2分数
print("R2",r2_score(ans,y_test))

plt.scatter(ans,ans - y_test,c='lightgreen',marker='s',label='Test data')
plt.xlabel('Predicted values')
plt.ylabel('Residuals')
plt.legend(loc='upper left')
plt.hlines(y=0,xmin=40,xmax=10000,lw=2,color='red')
plt.xlim([40,10000])
plt.tight_layout()
plt.show()
```

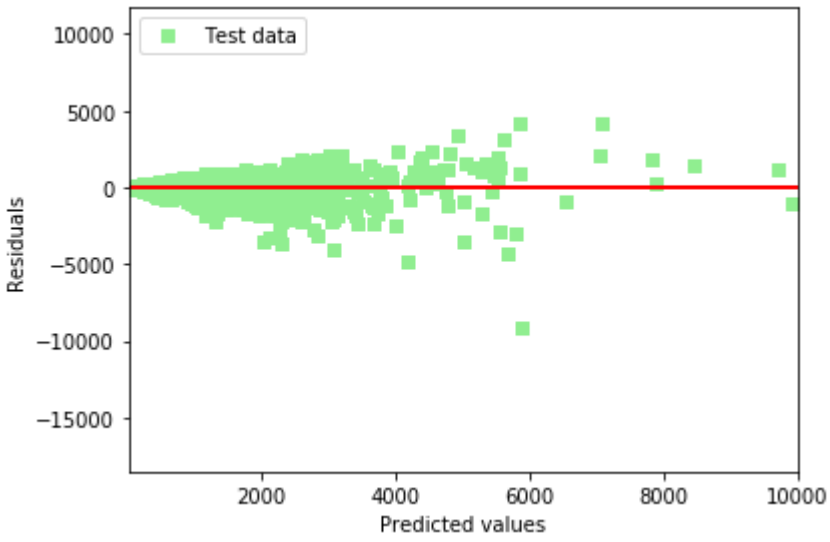
```
MSE 87013.72988908715
MAE 91.14940809284184
R2 0.6789189665599741
```



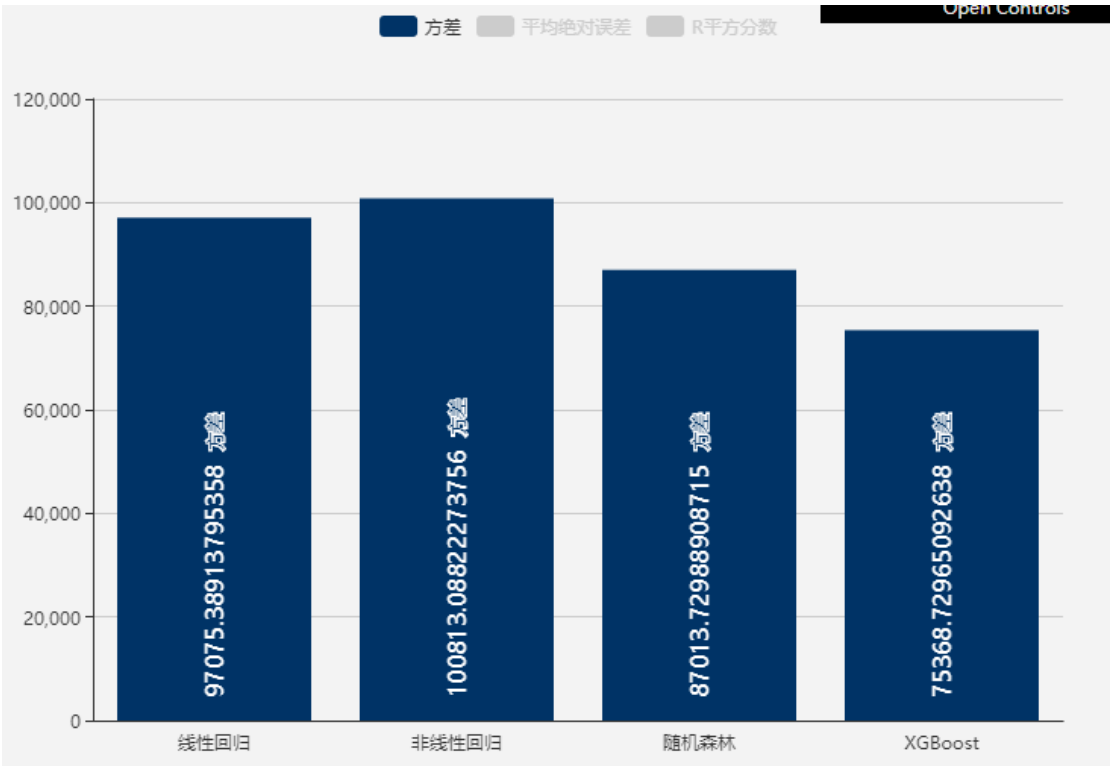

**④XGBoost 模型**

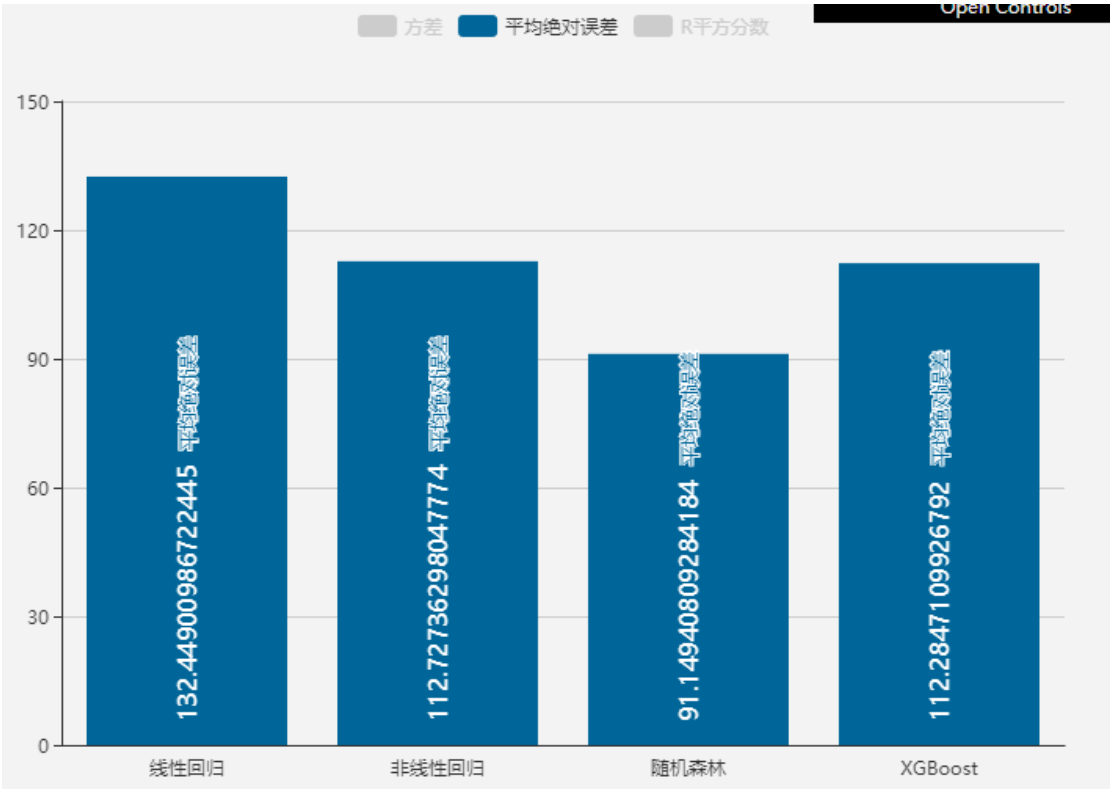
```
#XGBoost模型
import xgboost as xgb
xlf = xgb.XGBRegressor()
xlf.fit(X_train_final,y_train)
# 对测试集进行预测
ans = xlf.predict(X_test_final)

#模型评估
from sklearn.metrics import mean_squared_error,mean_absolute_error,r2_score
#方差
print ("MSE",mean_squared_error(ans,y_test))
#平均绝对误差
print("MAE",mean_absolute_error(ans,y_test))
#R2分数
print("R2",r2_score(ans,y_test))

plt.scatter(ans,ans - y_test,c='lightgreen',marker='s',label='Test data')
plt.xlabel('Predicted values')
plt.ylabel('Residuals')
plt.legend(loc='upper left')
plt.hlines(y=0,xmin=40,xmax=10000,lw=2,color='red')
plt.xlim([40,10000])
plt.tight_layout()
plt.show()
```

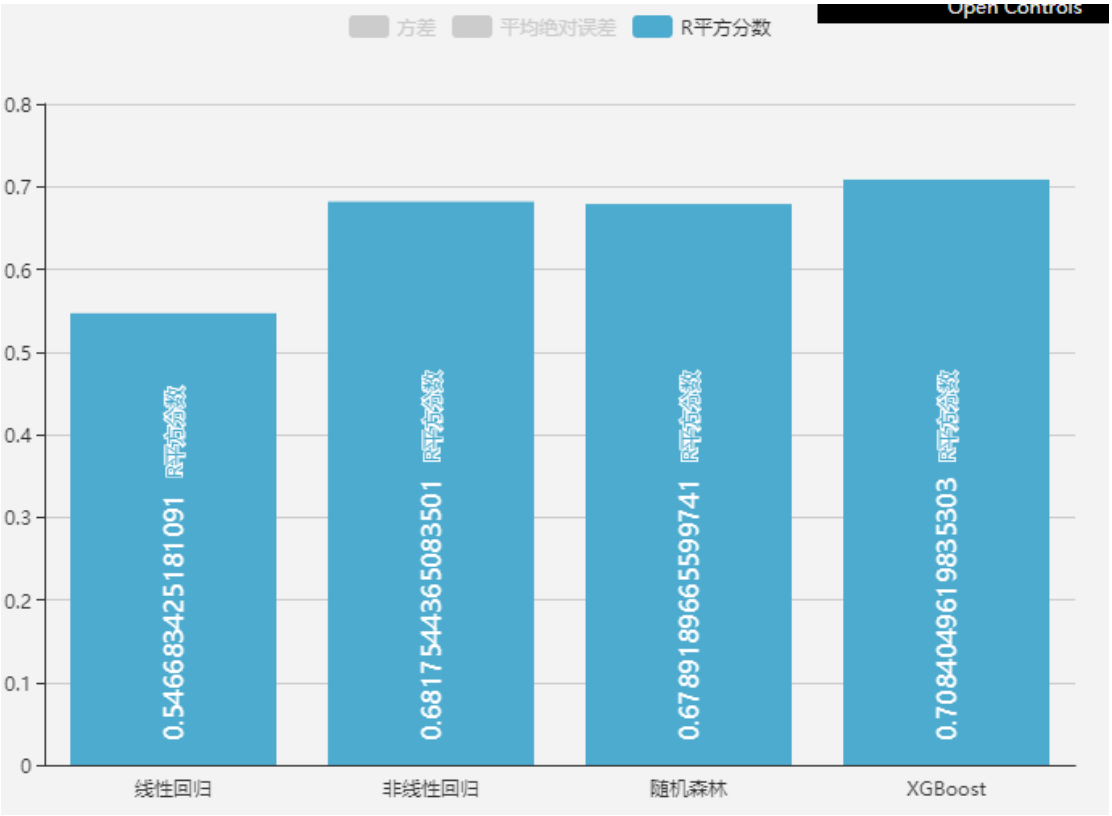
```
MSE 75368.72965092638
MAE 112.2847109926792
R2 0.7084049619835303
```




将三种评估参数做柱状图比较：


方差
平均绝对误差
R平方分数
120,000
100,000
80,000
60,000
40,000
20,000
0
97075.38913795358 方差
100813.08822273756 方差
87013.72988908715 方差
75368.72965092638 方差
线性回归
非线性回归
随机森林
XGBoost


方差
平均绝对误差
R平方分数
150
120
90
60
30
0
132.44900986722445 平均绝对误差
112.72736298047774 平均绝对误差
91.14940809284184 平均绝对误差
112.28471099926792 平均绝对误差
线性回归
非线性回归
随机森林
XGBoost




根据 r2 参数来看，XGBoost 的训练效果相对好一些，随机森林的平均误差好一些，Xgboost 的方差好一些，考虑到上次作业使用了随机森林，这次考虑使用 XGBoost 的模型，下面调整参数以获得更好的效果。

**7、用训练集对 XGBoost 模型调参**

**①基本参数设置**

```
 # XGBoost训练过程
import xgboost as xgb
from xgboost import plot_importance
from matplotlib import pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.grid_search import GridSearchCV
#模型参数设置
xlf = xgb.XGBRegressor(max_depth=5,
                       learning_rate=0.1,
                       n_estimators=100,
                       silent=True,
                       objective='reg:linear',
                       nthread=-1,
                       gamma=0,
                       min_child_weight=1,
                       max_delta_step=0,
                       subsample=0.85,
                       colsample_bytree=0.8,
                       colsample_bylevel=1,
                       reg_alpha=0,
                       reg_lambda=1,
                       scale_pos_weight=1,
                       seed=0,
                       missing=None)

xlf.fit(X_train_final, y_train)
print(xlf)
print("R2",r2_score(ans,y_test))
```

```
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
       colsample_bytree=0.8, gamma=0, learning_rate=0.1, max_delta_step=0,
       max_depth=5, min_child_weight=1, missing=None, n_estimators=100,
       n_jobs=1, nthread=-1, objective='reg:linear', random_state=0,
       reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=0, silent=True,
       subsample=0.85)
R2 0.70840496198353O3
```

**②最佳迭代次数 n_estimators**

将迭代次数按范围划分，并找出每个范围内的最佳取值和最佳模型得分。

考虑最佳迭代次数 n_estimators,范围在 400-800，结果 800 时效果最好，最佳模型得分为 0.8305197979055752，但运行时间较长，为 6.3min。

```
from sklearn.grid_search import GridSearchCV
cv_params = {'n_estimators': [400, 500, 600, 700, 800]}
other_params = {'learning_rate': 0.1, 'n_estimators': 500, 'max_depth': 5, 'min_child_weight': 1, 'seed': 0,
                'subsample': 0.8, 'colsample_bytree': 0.8, 'gamma': 0, 'reg_alpha': 0, 'reg_lambda': 1}
model = xgb.XGBRegressor(**other_params)
optimized_GBM = GridSearchCV(estimator=model, param_grid=cv_params, scoring='r2', cv=5, verbose=1, n_jobs=4)
optimized_GBM.fit(X_train_final, y_train)
evalute_result = optimized_GBM.grid_scores_
print('每轮迭代运行结果:{0}'.format(evalute_result))
print('参数的最佳取值：{0}'.format(optimized_GBM.best_params_))
print('最佳模型得分:{0}'.format(optimized_GBM.best_score_))
```

```
Fitting 5 folds for each of 5 candidates, totalling 25 fits
[Parallel(n_jobs=4)]: Done  25 out of  25 | elapsed:  6.3min finished
每轮迭代运行结果:[mean: 0.82805, std: 0.04405, params: {'n_estimators': 400}, mean: 0.82904, std: 0.04402, params: {'n_estimators': 500}, mean: 0.82978, std: 0.04416, params: {'n_estimators': 600}, mean: 0.83020, std: 0.04404, params: {'n_estimators': 700}, mean: 0.83052, std: 0.04395, params: {'n_estimators': 800}]
参数的最佳取值：{'n_estimators': 800}
最佳模型得分:0.8305197979055752
```

继续考虑最佳迭代次数 n_estimators，范围在 10-50，结果 50 时效果最好，最佳模型得分为 0.8041958455635226，运行时间为 20.2s。

```
#最佳迭代次数n_estimators
from sklearn.grid_search import GridSearchCV
cv_params = {'n_estimators': [10,20,30,40,50]}
other_params = {'learning_rate': 0.1, 'n_estimators': 300, 'max_depth': 5, 'min_child_weight': 1, 'seed': 0,
                'subsample': 0.8, 'colsample_bytree': 0.8, 'gamma': 0, 'reg_alpha': 0, 'reg_lambda': 1}
model = xgb.XGBRegressor(**other_params)
optimized_GBM = GridSearchCV(estimator=model, param_grid=cv_params, scoring='r2', cv=5, verbose=1, n_jobs=4)
optimized_GBM.fit(X_train_final, y_train)
evalute_result = optimized_GBM.grid_scores_
print('每轮迭代运行结果:{0}'.format(evalute_result))
print('参数的最佳取值：{0}'.format(optimized_GBM.best_params_))
print('最佳模型得分:{0}'.format(optimized_GBM.best_score_))
```

```
Fitting 5 folds for each of 5 candidates, totalling 25 fits
```

```
[Parallel(n_jobs=4)]: Done  25 out of  25 | elapsed:   20.2s finished
```

```
每轮迭代运行结果:[mean: 0.50590, std: 0.01223, params: {'n_estimators': 10}, mean: 0.73438, std: 0.01819, params: {'n_estimators': 20}, mean: 0.78422, std: 0.02617, params: {'n_estimators': 30}, mean: 0.79990, std: 0.03185, params: {'n_estimators': 40}, mean: 0.80420, std: 0.03689, params: {'n_estimators': 50}]
参数的最佳取值：{'n_estimators': 50}
最佳模型得分:0.8041958455635226
```

继续考虑最佳迭代次数 n_estimators,范围在 100-300，结果 300 时效果最好，最佳模型得分为 0.8262946825068367，运行时间为 1.2min。

```
#最佳迭代次数n_estimators
from sklearn.grid_search import GridSearchCV
#from sklearn import grid_search
cv_params = {'n_estimators': [100,200,300]}
other_params = {'learning_rate': 0.1, 'n_estimators': 300, 'max_depth': 5, 'min_child_weight': 1, 'seed': 0,
                'subsample': 0.8, 'colsample_bytree': 0.8, 'gamma': 0, 'reg_alpha': 0, 'reg_lambda': 1}
model = xgb.XGBRegressor(**other_params)
optimized_GBM = GridSearchCV(estimator=model, param_grid=cv_params, scoring='r2', cv=5, verbose=1, n_jobs=4)
optimized_GBM.fit(X_train_final, y_train)
evalute_result = optimized_GBM.grid_scores_
print('每轮迭代运行结果:{0}'.format(evalute_result))
print('参数的最佳取值：{0}'.format(optimized_GBM.best_params_))
print('最佳模型得分:{0}'.format(optimized_GBM.best_score_))
```

```
Fitting 5 folds for each of 3 candidates, totalling 15 fits
```

```
[Parallel(n_jobs=4)]: Done  15 out of  15 | elapsed:  1.2min finished
```

```
每轮迭代运行结果:[mean: 0.81367, std: 0.04591, params: {'n_estimators': 100}, mean: 0.82289, std: 0.04466, params: {'n_estimators': 200}, mean: 0.82629, std: 0.04388, params: {'n_estimators': 300}]
参数的最佳取值：{'n_estimators': 300}
最佳模型得分:0.8262946825068367
```

因此，综合考虑运行时间和训练效果，选用 300 为迭代次数。

### ③参数 max_depth

```
cv_params = {'max_depth': [3, 4, 5, 6, 7, 8]}
other_params = {'learning_rate': 0.1, 'n_estimators': 300, 'max_depth': 5, 'min_child_weight': 1, 'seed': 0,
                'subsample': 0.8, 'colsample_bytree': 0.8, 'gamma': 0, 'reg_alpha': 0, 'reg_lambda': 1}
model = xgb.XGBRegressor(**other_params)
optimized_GBM = GridSearchCV(estimator=model, param_grid=cv_params, scoring='r2', cv=5, verbose=1, n_jobs=4)
optimized_GBM.fit(X_train_final, y_train)
evalute_result = optimized_GBM.grid_scores_
print('每轮迭代运行结果:{0}'.format(evalute_result))
print('参数的最佳取值：{0}'.format(optimized_GBM.best_params_))
print('最佳模型得分:{0}'.format(optimized_GBM.best_score_))
```

```
Fitting 5 folds for each of 6 candidates, totalling 30 fits
```

```
[Parallel(n_jobs=4)]: Done  30 out of  30 | elapsed:  4.0min finished
```

```
每轮迭代运行结果:[mean: 0.80291, std: 0.04976, params: {'max_depth': 3}, mean: 0.81784, std: 0.04662, params: {'max_depth': 4}, mean: 0.82629, std: 0.04388, params: {'max_depth': 5}, mean: 0.83198, std: 0.04469, params: {'max_depth': 6}, mean: 0.83332, std: 0.04205, params: {'max_depth': 7}, mean: 0.83718, std: 0.03997, params: {'max_depth': 8}]
参数的最佳取值：{'max_depth': 8}
最佳模型得分:0.837181166857593
```

最终 max_depth 取 8 时，获得最佳模型，得分为 0.837181166857593。

### ④参数 min_child_weight

```
cv_params = {'min_child_weight': [1, 2, 3, 4, 5]}
other_params = {'learning_rate': 0.1, 'n_estimators': 300, 'max_depth': 8, 'min_child_wei
ght': 1, 'seed': 0,
                    'subsample': 0.8, 'colsample_bytree': 0.8, 'gamma': 0, 'reg_alpha': 0
, 'reg_lambda': 1}
model = xgb.XGBRegressor(**other_params)
optimized_GBM = GridSearchCV(estimator=model, param_grid=cv_params, scoring='r2', cv=5, v
erbose=1, n_jobs=4)
optimized_GBM.fit(X_train_final, y_train)
evalute_result = optimized_GBM.grid_scores_
print('每轮迭代运行结果:{0}'.format(evalute_result))
print('参数的最佳取值：{0}'.format(optimized_GBM.best_params_))
print('最佳模型得分:{0}'.format(optimized_GBM.best_score_))
```

```
Fitting 5 folds for each of 5 candidates, totalling 25 fits
[Parallel(n_jobs=4)]: Done  25 out of  25 | elapsed:  5.8min finished
每轮迭代运行结果:[mean: 0.83718, std: 0.03997, params: {'min_child_weight': 1}, mean:
0.83661, std: 0.02129, params: {'min_child_weight': 2}, mean: 0.83436, std: 0.01578, pa
rams: {'min_child_weight': 3}, mean: 0.83541, std: 0.01803, params: {'min_child_weigh
t': 4}, mean: 0.83604, std: 0.00841, params: {'min_child_weight': 5}]
参数的最佳取值：{'min_child_weight': 1}
最佳模型得分:0.837181166857593
```

最终当 min_child_weight 取 1 时，获得最佳模型，得分为 0.837181166857593。

**⑤调参数 gamma**

```
cv_params = {'gamma': [0,0.1, 0.2, 0.3, 0.4, 0.5, 0.6]}
other_params = {'learning_rate': 0.1, 'n_estimators': 300, 'max_depth': 8, 'min_child_wei
ght': 1, 'seed': 0,
                    'subsample': 0.8, 'colsample_bytree': 0.8, 'gamma': 0, 'reg_alpha': 0
, 'reg_lambda': 1}
model = xgb.XGBRegressor(**other_params)
optimized_GBM = GridSearchCV(estimator=model, param_grid=cv_params, scoring='r2', cv=5, v
erbose=1, n_jobs=4)
optimized_GBM.fit(X_train_final, y_train)
evalute_result = optimized_GBM.grid_scores_
print('每轮迭代运行结果:{0}'.format(evalute_result))
print('参数的最佳取值：{0}'.format(optimized_GBM.best_params_))
print('最佳模型得分:{0}'.format(optimized_GBM.best_score_))
```

```
Fitting 5 folds for each of 7 candidates, totalling 35 fits
[Parallel(n_jobs=4)]: Done  35 out of  35 | elapsed:  8.4min finished
每轮迭代运行结果:[mean: 0.83718, std: 0.03997, params: {'gamma': 0}, mean: 0.83718, st
d: 0.03997, params: {'gamma': 0.1}, mean: 0.83730, std: 0.04001, params: {'gamma': 0.
2}, mean: 0.83735, std: 0.04001, params: {'gamma': 0.3}, mean: 0.83733, std: 0.04005, p
arams: {'gamma': 0.4}, mean: 0.83733, std: 0.04005, params: {'gamma': 0.5}, mean: 0.837
33, std: 0.04005, params: {'gamma': 0.6}]
参数的最佳取值：{'gamma': 0.3}
最佳模型得分:0.837353062772207
```

最终当 gamma 取 0.3 时，获得最佳模型，得分为 0.837353062772207。

**⑥调参数 subsample，colsample_bytree**

```
cv_params = {'subsample': [0.6, 0.7, 0.8, 0.9], 'colsample_bytree': [0.6, 0.7, 0.8, 0.9]}
other_params = {'learning_rate': 0.1, 'n_estimators': 300, 'max_depth': 8, 'min_child_weight': 1, 'seed': 0,
                'subsample': 0.8, 'colsample_bytree': 0.8, 'gamma': 0.3, 'reg_alpha': 0, 'reg_lambda': 1}
model = xgb.XGBRegressor(**other_params)
optimized_GBM = GridSearchCV(estimator=model, param_grid=cv_params, scoring='r2', cv=5, verbose=1, n_jobs=4)
optimized_GBM.fit(X_train_final, y_train)
evalute_result = optimized_GBM.grid_scores_
print('每轮迭代运行结果:{0}'.format(evalute_result))
print('参数的最佳取值：{0}'.format(optimized_GBM.best_params_))
print('最佳模型得分:{0}'.format(optimized_GBM.best_score_))
```

```
Fitting 5 folds for each of 16 candidates, totalling 80 fits

[Parallel(n_jobs=4)]: Done  42 tasks       | elapsed:  9.1min
[Parallel(n_jobs=4)]: Done  80 out of  80 | elapsed: 18.0min finished

每轮迭代运行结果:[mean: 0.84530, std: 0.02454, params: {'colsample_bytree': 0.6, 'subsample': 0.6}, mean: 0.8430
8, std: 0.01987, params: {'colsample_bytree': 0.6, 'subsample': 0.7}, mean: 0.84200, std: 0.01924, params: {'col
sample_bytree': 0.6, 'subsample': 0.8}, mean: 0.84485, std: 0.02138, params: {'colsample_bytree': 0.6, 'subsampl
e': 0.9}, mean: 0.84248, std: 0.03393, params: {'colsample_bytree': 0.7, 'subsample': 0.6}, mean: 0.84398, std:
0.02800, params: {'colsample_bytree': 0.7, 'subsample': 0.7}, mean: 0.84490, std: 0.02862, params: {'colsample_b
ytree': 0.7, 'subsample': 0.8}, mean: 0.84482, std: 0.02763, params: {'colsample_bytree': 0.7, 'subsample': 0.
9}, mean: 0.83613, std: 0.04785, params: {'colsample_bytree': 0.8, 'subsample': 0.6}, mean: 0.84033, std: 0.0445
5, params: {'colsample_bytree': 0.8, 'subsample': 0.7}, mean: 0.83735, std: 0.04001, params: {'colsample_bytre
e': 0.8, 'subsample': 0.8}, mean: 0.84471, std: 0.03068, params: {'colsample_bytree': 0.8, 'subsample': 0.9}, me
an: 0.83211, std: 0.04849, params: {'colsample_bytree': 0.9, 'subsample': 0.6}, mean: 0.83488, std: 0.04721, par
ams: {'colsample_bytree': 0.9, 'subsample': 0.7}, mean: 0.83611, std: 0.03904, params: {'colsample_bytree': 0.9,
'subsample': 0.8}, mean: 0.83671, std: 0.04671, params: {'colsample_bytree': 0.9, 'subsample': 0.9}]
参数的最佳取值：{'colsample_bytree': 0.6, 'subsample': 0.6}
最佳模型得分:0.8452982893855446
```

最终当 subsample=0.6， colsample_bytree=0.6 时，获得最佳模型，得分为 0.8452982893855446。

### ⑦调参数 reg_alpha 以及 reg_lambda

```
#调参数reg_alpha以及reg_lambdacv_params = {'reg_alpha': [0.05, 0.1, 1, 2, 3], 'reg_lambda': [0.05, 0.1, 1, 2, 3]}
other_params = {'learning_rate': 0.1, 'n_estimators': 300, 'max_depth': 8, 'min_child_weight': 1, 'seed': 0,
                'subsample': 0.6, 'colsample_bytree': 0.6, 'gamma': 0.3, 'reg_alpha': 0, 'reg_lambda': 1}
model = xgb.XGBRegressor(**other_params)
optimized_GBM = GridSearchCV(estimator=model, param_grid=cv_params, scoring='r2', cv=5, verbose=1, n_jobs=4)
optimized_GBM.fit(X_train_final, y_train)
evalute_result = optimized_GBM.grid_scores_
print('每轮迭代运行结果:{0}'.format(evalute_result))
print('参数的最佳取值：{0}'.format(optimized_GBM.best_params_))
print('最佳模型得分:{0}'.format(optimized_GBM.best_score_))
```

```
Fitting 5 folds for each of 25 candidates, totalling 125 fits

[Parallel(n_jobs=4)]: Done  42 tasks       | elapsed:  8.6min
[Parallel(n_jobs=4)]: Done 125 out of 125 | elapsed: 25.7min finished

每轮迭代运行结果:[mean: 0.83607, std: 0.03469, params: {'reg_alpha': 0.05, 'reg_lambda': 0.05}, mean: 0.83655, std: 0.03052, params: {'reg_a
lpha': 0.05, 'reg_lambda': 0.1}, mean: 0.84606, std: 0.02388, params: {'reg_alpha': 0.05, 'reg_lambda': 1}, mean: 0.84359, std: 0.02227, par
ams: {'reg_alpha': 0.05, 'reg_lambda': 2}, mean: 0.84682, std: 0.02280, params: {'reg_alpha': 0.05, 'reg_lambda': 3}, mean: 0.83506, std: 0.
03609, params: {'reg_alpha': 0.1, 'reg_lambda': 0.05}, mean: 0.83620, std: 0.03136, params: {'reg_alpha': 0.1, 'reg_lambda': 0.1}, mean: 0.8
4548, std: 0.02393, params: {'reg_alpha': 0.1, 'reg_lambda': 1}, mean: 0.84280, std: 0.02255, params: {'reg_alpha': 0.1, 'reg_lambda': 2}, m
ean: 0.84667, std: 0.02274, params: {'reg_alpha': 0.1, 'reg_lambda': 3}, mean: 0.83554, std: 0.03596, params: {'reg_alpha': 1, 'reg_lambda':
0.05}, mean: 0.83835, std: 0.02888, params: {'reg_alpha': 1, 'reg_lambda': 0.1}, mean: 0.84470, std: 0.02490, params: {'reg_alpha': 1, 'reg_
lambda': 1}, mean: 0.84362, std: 0.02197, params: {'reg_alpha': 1, 'reg_lambda': 2}, mean: 0.84463, std: 0.02266, params: {'reg_alpha': 1,
'reg_lambda': 3}, mean: 0.83477, std: 0.03460, params: {'reg_alpha': 2, 'reg_lambda': 0.05}, mean: 0.83687, std: 0.03083, params: {'reg_alph
a': 2, 'reg_lambda': 0.1}, mean: 0.84550, std: 0.02270, params: {'reg_alpha': 2, 'reg_lambda': 1}, mean: 0.84211, std: 0.02175, params: {'re
g_alpha': 2, 'reg_lambda': 2}, mean: 0.84726, std: 0.02244, params: {'reg_alpha': 2, 'reg_lambda': 3}, mean: 0.83214, std: 0.03691, params:
{'reg_alpha': 3, 'reg_lambda': 0.05}, mean: 0.83562, std: 0.03245, params: {'reg_alpha': 3, 'reg_lambda': 0.1}, mean: 0.84656, std: 0.02331,
params: {'reg_alpha': 3, 'reg_lambda': 1}, mean: 0.84229, std: 0.02153, params: {'reg_alpha': 3, 'reg_lambda': 2}, mean: 0.84814, std: 0.022
77, params: {'reg_alpha': 3, 'reg_lambda': 3}]
参数的最佳取值：{'reg_alpha': 3, 'reg_lambda': 3}
最佳模型得分:0.8481385945575455
```

当 reg_alpha=3，reg_lambda=3 时，获得最佳模型，得分为 0.8481385945575455。

### ⑧最后是 learning_rate，这时候一般要调小学习率来测试

```
cv_params = {'learning_rate': [0.01, 0.05, 0.07, 0.1, 0.2]}
other_params = {'learning_rate': 0.1, 'n_estimators': 300, 'max_depth': 8, 'min_child_weight': 1, 'seed': 0,
                'subsample': 0.6, 'colsample_bytree': 0.6, 'gamma': 0.3, 'reg_alpha': 3, 'reg_lambda': 3}
model = xgb.XGBRegressor(**other_params)
optimized_GBM = GridSearchCV(estimator=model, param_grid=cv_params, scoring='r2', cv=5, verbose=1, n_jobs=4)
optimized_GBM.fit(X_train_final, y_train)
evalute_result = optimized_GBM.grid_scores_
print('每轮迭代运行结果:{0}'.format(evalute_result))
print('参数的最佳取值：{0}'.format(optimized_GBM.best_params_))
print('最佳模型得分:{0}'.format(optimized_GBM.best_score_))
```

```
Fitting 5 folds for each of 5 candidates, totalling 25 fits
[Parallel(n_jobs=4)]: Done  25 out of  25 | elapsed:  5.3min finished
每轮迭代运行结果:[mean: 0.79787, std: 0.01979, params: {'learning_rate': 0.01}, mean: 0.84382, std: 0.02121, par
ams: {'learning_rate': 0.05}, mean: 0.84597, std: 0.02289, params: {'learning_rate': 0.07}, mean: 0.84814, std:
0.02277, params: {'learning_rate': 0.1}, mean: 0.83611, std: 0.02401, params: {'learning_rate': 0.2}]
参数的最佳取值：{'learning_rate': 0.1}
最佳模型得分:0.8481385945575455
```

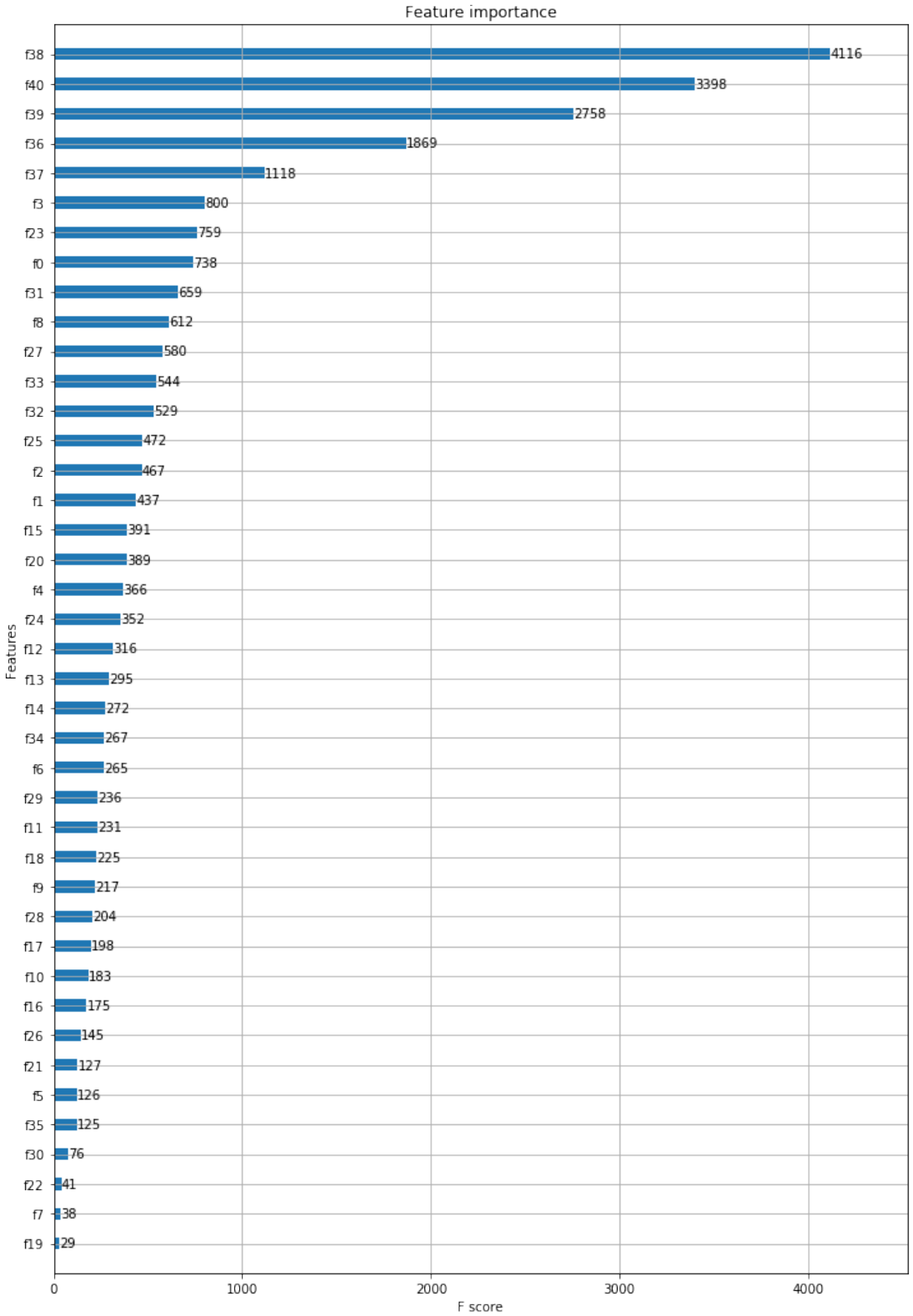
当 learning_rate=0.1 时，获得最佳模型，得分为 0.8481385945575455。

## 8、最终训练模型及模型基本评估

选取前一步调试出的最佳参数组合进行训练，并用方差、平均绝对误差和 R 平方分数对模型进行评估，打印出模型的重要特征。

```
#最终训练模型
# XGBoost训练过程，下面的参数就是刚才调试出来的最佳参数组合
model = xgb.XGBRegressor(learning_rate=0.1, n_estimators=300, max_depth=8, min_child_weight=1, seed=0,
                         subsample=0.6, colsample_bytree=0.6, gamma=0.3, reg_alpha=3, reg_lambda=3)
model.fit(X_train_final, y_train)

# 对测试集进行预测
ans = model.predict(X_test_final)
ans_train = model.predict(X_train_final)

#模型评估
from sklearn.metrics import mean_squared_error,mean_absolute_error,r2_score
#方差
print ("MSE",mean_squared_error(ans,y_test))
#平均绝对误差
print("MAE",mean_absolute_error(ans,y_test))
#R2分数
print("R2",r2_score(ans,y_test))

# 显示重要特征
plot_importance(model)
plt.show()
```

```
MSE 71263.26206587603
MAE 92.22237317576847
R2 0.7378048738018894
```

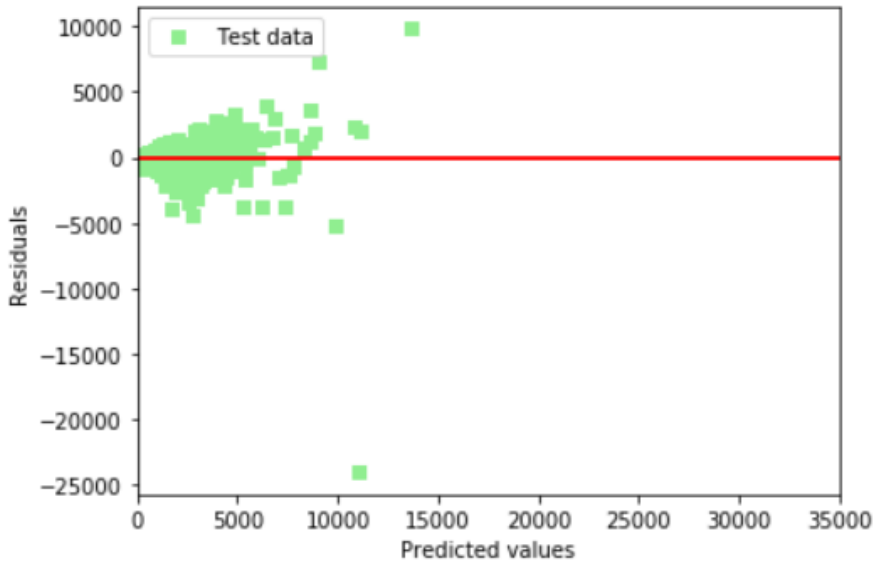



由于对离散变量做了独热编码，模型中显示出的特征有 41 个。其中，area 所占比重最多。

下面对模型的预测值与实际值作残差图：

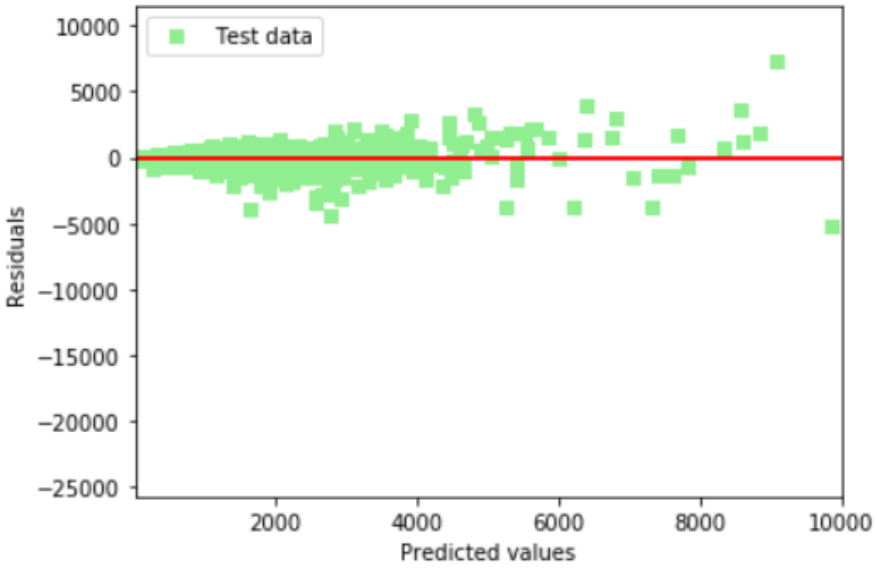
首先设定横轴（预测值）坐标范围为[40, 35000]。

```
plt.scatter(ans,ans - y_test,c='lightgreen',marker='s',label='Test data')
plt.xlabel('Predicted values')
plt.ylabel('Residuals')
plt.legend(loc='upper left')
plt.hlines(y=0,xmin=40,xmax=35000,lw=2,color='red')
plt.xlim([40,35000])
plt.tight_layout()
plt.show()
```




从上图看出，数据点主要分布在 10000 之前。再设置横轴坐标范围为[40, 10000]。

```
plt.scatter(ans,ans - y_test,c='lightgreen',marker='s',label='Test data')
plt.xlabel('Predicted values')
plt.ylabel('Residuals')
plt.legend(loc='upper left')
plt.hlines(y=0,xmin=40,xmax=10000,lw=2,color='red')
plt.xlim([40,10000])
plt.tight_layout()
plt.show()
```




从上图可以看出，模型在较低房价处的拟合效果要比高价处拟合效果好。

**9、模型总结**

本次调整完参数的最终模型的 r2 值也只能达到 0.73，模型得分 0.848，与调参之前相比（r2=0.708，score=0.813）有所提升，但是并没有取得很大的提升。而且模型在高房价区的效果很差。我们认为这与我们的数据集有关。

第一，我们只是爬取了 14 个用于预测房价的属性值，而实际房价的取决因素有很多，我们的属性明显比较少。

第二，我们的部分属性处理并不很准确。比如 location 属性值，我们在预处理阶段对数据按照上海市实际行政区划划分为离散属性值，并采用了独热编码。但实际的房价不仅与行政区划有关，还与其实际所处的位置有关，比如周遭环境等。

第三，对于高房价部分，我们缺少决定高房价的重要属性值，比如静安区别墅等等。