一、问题分析：

本题要求根据已知的商品各种描述和价钱来预测新的产品的价钱，与之前练习的题目不同，本题的各类特征大部分都是文本。故重点在于对商品的特征进行合适的特征工程，以及利用适当的方法进行文本向量化。最后选择模型进行训练，进行测评。

二、模型选择：

小组经过讨论后选择使用 Ridge，LGBM，MLP 三种模型。分别将调参好的 Ridge，LGBM，和不同节点数的 MLP 模型的结果进行加权平均来得到最后的结果。

三、使用的库：

lightgbm	2.3.0
numpy	1.16.5
pandas	0.25.1
scikit-learn	0.21.3
scipy	1.3.1
tensorflow	2.0.0
tensorflow-estimator	2.0.1
Keras	2.3.1
Keras-Applications	1.0.8
Keras-Preprocessing	1.1.
Nltk PorterStemmer	3.4.5

四、尝试的方法：

有效的：

1) Porter Stemmer:

- PorterStemmer

```
'''
from nltk.stem import PorterStemmer

stemmer = PorterStemmer()
train_data=train_data.applymap(stemmer.stem)
train_data.info()
'''
```

考虑到是英文文本，我们一开始想对文本进行预处理，经过搜索后找到了PorterStemmer方法。它可以实现还原英文单词原型。然而在实际使用效果不是很明显，还会出现品牌VANS变成了van的情况。故选择在MLP其中一个训练集下使用，LGBM，Ridge模型不使用。

2) 正则表达式:

```
import re

my_file_path = 'test_new.csv'
save_file_path = 'test_clean_new.csv'

# 打开文件
my_file = open(my_file_path, 'r', encoding='utf-8')

# 只保留中英文 数字 . _ 回车 | 空格的正则表达式
cop = re.compile("[^\u4e00-\u9fa5^.^a-z^A-Z^0-9\n/\_\s]")

for line in my_file.readlines():
    string = cop.sub("", line)
    save_file = open(save_file_path, 'a', encoding='utf-8')
    save_file.write(string)
    save_file.flush()
    save_file.close()

# 关闭文件
my_file.close()
```

在查看训练集的txt之后，我们发现文本描述里出现了大量的表情符号，例如“◀◀✕✕”等。我们一致认为这种表情会影响最终模型的训练，在预处理时利用正则表达式将测试集仅保留英文，中文，数字，空格，回车，\，.，_这些字符。为了使用方便，将处理好的训练集和测试集保存为新的文件。

3) *拼接特征*:

将两个或多个特征进行拼接后作为一个整体比单个特征更为有效。我们在特征工程方面进行了大量的尝试。对不同的模型使用不同的特征工程。具体的特征分析见后。

4) *CountVectorizer， Tf-idf*

针对不同的属性我们选择了不同的处理方法，具体见后文的特征处理

5) *one_hot编码*

针对包邮（shipping）和商品状况(item_condition_id)这两个类别变量。我们选择了使用one_hot编码将其向量化。利用pandas提供的get_dummies。

6) *加权平均*

在使用stacking失败后，我们选择了加权平均对最后的结果进行整合。结果mse确实变的比使用单个模型要更好。最终选择加权平均。

7）特征缩放 StandardScaler

```
y_scaler=StandardScaler()# 缩放 标准化y值
y_train=y_scaler.fit_transform(y_train.values.reshape(-1,1))
```

无效的：

1) *标签二值化（LabelBinarizer）*

对品牌进行处理时，我们首先考虑了对品牌进行二值化，再和不进行二值

化而是进行Tfidf处理的结果进行对比。进行tfidf处理的结果更好，最终选择对name进行tfidf处理。

lb = LabelBinarizer(sparse_output=True)

2) *交叉验证KFold*

针对新的训练集，我们最开始选择了KFold进行训练和测试。但是小组内部为了模型运行的更加高效，选择分别在两个电脑上运行不同的模型，KFold的测试集和数据集划分在两台电脑上可能不一样，且LGBM和MLP的运行速度较慢，用KFold会更加影响速度。于是，我们选择了使用train_test_split方法，将11%的数据作为测试集。且为了在两台电脑上数据同步，将训练集和测试集分别保存为文件。

```
from sklearn.model_selection import train_test_split
X_train,X_test,Y_train,Y_test=train_test_split(train_data,y_train,test_size=0.11,random_state=5)
```

```
X_train.shape,X_test.shape,Y_train.shape,Y_test.shape
```

```
X_train.to_csv('X_train.csv',index=False)
```

```
X_test.to_csv('X_test.csv',index=False)
```

```
Y_train.to_csv('Y_train.csv',index=False)
```

```
Y_test.to_csv('Y_test.csv',index=False)
```

3) *集成学习stacking*

由于我们使用了多个不同的模型，在后期考虑将多个模型进行融合，以期得到更好的结果。在查找资料后找到了stacking方法。我们尝试利用stacking对已经调参好的Ridge和LGBM进行融合。然而结果并不理想（可能是参数没调好？）。最后选择了放弃。

4) *Sklearn 的 MLPRegresson*

最开始训练 MLP 使用的 SKlearn 的库，但由 *Sklearn 的 MLPRegresson 适用*

*于较小的数据集，对*于大的数据集训练太慢，故改用 keras 的 MLP。

mlp

```
In [30]: from sklearn.neural_network import MLPRegressor

In [39]: mlp=MLPRegressor(activation='logistic',hidden_layer_sizes=(100,),early_stopping=True,verbose=True)

In [40]: mlp.fit(X_train_sparse, y_train)
```

5）TruncatedSVD 降维

由于我们认为是特征维数过多导致 MLP 训练过慢，故使用 TruncatedSVD 对稀疏矩阵降维。但降到 100 样本特征丢失过多，导致结果不佳；降到 1000 维无法运行出结果，会 memory error，故不使用 TruncatedSVD 降维。

```
from sklearn.decomposition import TruncatedSVD
```

```
svd=TruncatedSVD(n_components=100)#Desired dimensionality of output data.For LSA潜在语义分析, a value of 100 is recommended.
X_PCA=hstack()
```

*6）*高斯降维

在sklearn库调取MLPRegressor运行时，由于特征的维数太多，运行速度非常缓慢。尝试使用了高斯降维来处理数据，将数据维数降低到1000维。然而降维后运行速度虽然加快了，但是mse很大，最终放弃了降维处理，改用keras搭建的MLP

- 降维

```
from sklearn.random_projection import GaussianRandomProjection
gauss_proj = GaussianRandomProjection(n_components=1000)
```

```
vector_t = gauss_proj.fit_transform(sparse_merge)
```

```
print(vector_t.shape)
```

```
(349924, 1000)
```

五、模型分析

（一）MLP

1.预处理:

去除文本中非英文字符和数字和部分标点，将category_name中的'\'替换为空格，缺失值填充为''，将所有特征格式转换为字符串。

将name、brand_name拼接作为一个新特征name，表述商品最主要的特征。原样本的name字段描述了商品主要信息（例如详细种类等），而brand对同一种类的物品价格影响很大，故将两者结合能更好表达商品最主要的特征。

item_description、name、category_name拼接作为新特征text。

```
df['name']=df['name'].fillna('')+' '+ df['brand_name'].fillna('')
df['text']=df['item_description'].fillna('')+' '+df['name']+' '+df['category_name'].fillna('')
df=df[['name','text','shipping','item_condition_id']]
```

2.PorterStemmer

使用对合并以后的文本提取词干，减小因为单词形式不同而在TFIDF分析时被分为不同的词而带来的误差。

```
from nltk.stem import PorterStemmer
stemmer = PorterStemmer()
df=df.applymap(stemmer.stem)
```

3. TF-IDF

使用TF-IDF处理所有文本:

```
Tfidf=TfidfVectorizer(max_features=100000, stop_words='english',token_pattern='\w+')
X_name=Tfidf.fit_transform(df['name'])
Tfidf=TfidfVectorizer(max_features=100000, ngram_range=(1, 2), stop_words='english',token_pattern='\w+')
X_text=Tfidf.fit_transform(df['text'])
```

4.one hot编码

使用表示item_condition_id和shipping。

```
X_dummies = csr_matrix(pd.get_dummies(df[['item_condition_id', 'shipping']], sparse=True).values)
```

5.MLP 分析

尝试了使用多组不同的训练样本处理方法处理得到的稀疏矩阵训练神经网络，调整了多组神经网络节点数、隐层数，最终选取了几组较好的结果用于最终合并。

模型设置：

```
model = Sequential([
    Dense(256, input_shape=(X_train_sparse.shape[1],),activation='relu'),
    Dense(64 ,activation='relu'),
    Dense(1)
])
model.compile(loss='mean_squared_error',optimizer=Adam(lr=3e-3))
```

训练模型：迭代三次，batch_size初值设为2^10，每次迭代增大batch_size，经过多次实验表明第二次迭代结束结果较好，第一次迭代结束欠拟合，第三次迭代结束过拟合。

```
for i in range(3):
    model.fit(x=X_train_sparse, y=y_train,batch_size=2**(10+i),epochs=1,verbose=1)
```

测试结果：

隐层节点数/数据集	192/64/64	256/64	256/64/32	192/64/32	192/64	128/64	256/128/64	256/128
数据集1	0.194603（batch size2^11）	0.193094	0.193342	0.194985（batch size2^11）	0.194122	0.19601	0.19694	0.213876(不使用StandardScaler)
	0.19533	0.193301						
数据集2	0.194363	0.1949						
数据集3	0.195655	0.193873						

注：1)数据集1：详见LGBM和岭回归的特征工程。

2)数据集2：不使用Porterstemmer和去除'/'。缺失值填充为''。将原始文本拼接为name和text两个特征，再加上shipping和item_condition的one_hot编码，丢弃原始所有特征，对文本只使用TFIDF

3)数据集3：用Porterstemmer 的数据集

4)表中没有注释的默认batch_size=2^10，随迭代次数增加batch_size增大;

默认使用StandardScaler。每次运行结果稍有误差。

实验结果表明，设置两个隐层、隐层节点数取256和64时，在不同训练集上表现均较好，多次测试结果MSLE大约为0.1933 。

如果隐层结点数过少，网络不能具有必要的学习能力和信息处理能力，模型拟合结果不够好。例如节点数取128/64/32时第一个隐层节点过少，在测试集上表现较差，MSLE=0.196010 。

反之，若隐层节点数过多，不仅会大大增加网络结构的复杂性，网络在学习过程中更易陷入局部极小点，而且会使网络的训练速度变得很慢，同时也可能导致过拟合，使模型在测试集上表现不好。在隐层取256/128/64时，节点过多，训练极为缓慢，在测试集上表现较差，MSLE=0.196940 。

在隐层节点总数相同的情况下，增加单层隐层节点数比增加隐层层数表现更好。隐层第一层取更多节点可以更好地学习到训练集的特征。例如，隐层节点数取256/64比192/64/64结果更好。对标签y使用了standard scaler的训练集运行结果更好。

（二）Ridge

1.缺失值处理

训练集:

利用train_data.info()查看发现有缺失数据的属性为name，category_name，brand_name，item_description

其中item_description缺失的行数较少（100行左右）。经过组内讨论，我们认为item_description对预测结果的影响较大，缺失item_description会影响模型训练的准确性，而其缺失的行数较少。选择将缺失item_description的行删除。

对于name,category_name,brand_name属性，对缺失值填充missing。

```
train_data['item_description'] = train_data['item_description'].fillna('missing').astype(str)
train_data.info()
```

```
dele=[]
for i in range(len(train_data['item_description'])):
    if train_data['item_description'][i] == 'missing':
        dele.append(i)

for i in range(len(dele)):
    train_data.drop([dele[i]],inplace=True)
train_data.info()
```

```
train_data['name'] = train_data['name'].fillna('missing').astype(str)
train_data['category_name'] = train_data['category_name'].fillna('missing').astype(str)
train_data['brand_name'] = train_data['brand_name'].fillna('missing').astype(str)
```

- 强制类型转换

```
train_data['train_id'] = train_data['train_id'].astype(str)
train_data['item_condition_id'] = train_data['item_condition_id'].astype(str)
train_data['shipping'] = train_data['shipping'].astype(str)
train_data['price'] = train_data['price'].astype(str)
train_data.info()
```

对合并训练集和测试集后的数据:

category_name,brand_name,item_description分别填充MISS，missing，NO。缺失值填充后，将所有的数据强制转换为字符型。

```
df = df.drop(['price', 'test_id', 'train_id'], axis=1)
df['category_name'] = df['category_name'].fillna('MISS').astype(str)
df['brand_name'] = df['brand_name'].fillna('missing').astype(str)
df['item_description'] = df['item_description'].fillna('No')

df['shipping'] = df['shipping'].astype(str)
df['item_condition_id'] = df['item_condition_id'].astype(str)
df['name'] = df['name'].astype(str)

# 训练数据的行数
nrow_train = train_data.shape[0]

df.head()
```

2.CountVectorizer + Tfidf

对合并后的数据集进行向量化处理。针对每个特征有不同的处理方式。

name特征:

```
# name特征向量化，输出稀疏矩阵
cv = CountVectorizer(min_df=10)
X_name = cv.fit_transform(df['name'])
```

category_name特征

```
# category_name特征向量化，输出稀疏矩阵
cv = CountVectorizer()
X_category = cv.fit_transform(df['category_name'])
```

item_description特征：

```
# 利用tfidf向量化item_description属性，使用停词
tv = TfidfVectorizer(max_features=60000, ngram_range=(1, 2), stop_words='english')
X_description = tv.fit_transform(df['item_description'])
```

brand_name特征：

```
# brand_name二值化
# lb = LabelBinarizer(sparse_output=True)
X_brand = tv.fit_transform(df['brand_name'])
```

item_condition_id，shipping特征：

这两个类别变量进行**one_hot编码**，输出为一个新的稀疏矩阵

```
# 合并item_condition_id，shipping属性形成一个新属性
X_dummies = csr_matrix(pd.get_dummies(df[['item_condition_id', 'shipping']], sparse=True).values)
```

在分别将单个属性向量化之后，考虑将两个或多个属性合并起来，作为一个新的属性加入训练。

3.属性合并

结果	合并的属性
0.241	df['brand_name'] +df['category_name']
0.24731811722829436	df['item_description'] +df['category_name']
0.23686654378532468	1.df['item_description']+df['name'] 2.df['brand_name']+df['category_name']
0.24108560740325458	1.df['item_description']+df['name'] 2.df['item_description']+df['category_name'] 3.df['item_description']+df['item_condition_id']
0.23634671796262421	1.df['item_description']+df['name'] 2.df['brand_name']+df['category_name']
0.23277704301764082	1.df['brand_name'] +df[''category _name'] 2.df['name'] +df[''category _name'] 3.df['item_description'] +df['name']

0.23501340567423148	df['name']+df['category_name']+df['item_description']
0.22947310231523774	1.df['name']+df['category_name']+df[' brand_name'] 2.df['item_description'] +df['name']
0.211916004081646	1.df['brand_name'] +df['name'] 2.df['item_description'] +df['name'] 3.df['name']+df['category_name']+df['brand_name']

经过各种尝试，最后选择使用将brand_name,name 合并；item_description, name 合并;name,category_name,brand_name 合并

4.Ridge 的参数如下:

```
def ridgeClassify(train_data, train_label):
    ridgeClf = Ridge(
        solver='auto',
        fit_intercept=True,
        alpha=0.5,
        max_iter=500,
        normalize=False,
        tol=0.05)
    # 训练
    ridgeClf.fit(train_data, train_label)
    return ridgeClf
```

经过各种尝试，Ridge模型在划分的测试集的mse为0.211916004081646

5.调试思路：

在三个模型中，Ridge 模型的训练与预测速度是最快的。在进行特征处理时我们先在 Ridge 模型中尝试特征的合并，运行 Ridge 模型并观察结果。若结果变好，再证明对特征的处理有效。可以尝试将特征处理运用到另外两个模型中去。

(三)LGBM

1.缺失值处理同Ridge。

2.CountVectorizer + Tfidf

单个特征的处理同Ridge，在特征合并上，二者不同:

```
#                           +品牌+种类    描述+名字              +tv  brand_name            +名称+种类              +3500
# 0.3097490417597127    0.8
# 0.3033324592965209    0.75    0.24971677027174807
# 0.30297409282684656   0.7
# 0.2924852529344102    0.5
#                       0.435                                   0.23116733540797127
#                       0.4                                     0.23048268999359142                                 0.22972760045798618
# 0.2909902128620731    0.37    0.23071333494801358             0.23063115440336257         0.23435643863541147
# 0.29328001723848024   0.25

# 0.15963062734630298
```

```
# 0.4   品牌+种类 描述+名字                +tv  brand_name   3800       0.22927714889704326
#       品牌+名字 描述+名字 名字+种类+品牌  +tv  brand_name   3800   4   0.2288647837812071 0.2284 ✓
#       描述+名字  名称+种类+牌子           +tv  brand_name   3800       0.2314978669614828
```

采用不同的学习率，深度，迭代次数和特征值的运行结果如上

3.LGBM 的参数如下：

```
params = {
        'learning_rate': 0.4,
        'max_depth': 4,
        'num_leaves': 100,
        'metric': 'mse',
    }
```

```
gbm = lgb.train(params, train_set=train_X, num_boost_round=3800, verbose_eval=100)
```

MSLE=0.215

五、最终结果