Junior quant task Report

by Ivan Pavlovich

Contents

Task description

Вам предлагается решить задачу восстановления торговой стратегии наших конкурентов по данным. В файле train.txt даны 9000 пар (стакан; цена). Ниже каждого стакана указана та цена, по которой алгоритм, глядя на стакан, готов купить или продать. Обратите внимание, что алгоритм может выбрать цену не только из 20 предложенных в тестовом стакане, но и цену которой там нет, кратную 5. То, что с заявкой происходит далее - значения не имеет. Все стаканы между собой не связаны и были выбраны в случайные моменты времени. Вам надо придумать свой алгоритм, который бы на контрольной выборке стаканов выбрал те же цены, что и исходный алгоритм. Результат будет оцениваться по числу точных совпадений ваших цен с искомыми. В качестве результата мы ожидаем увидеть: 1) 25 пар (стакан; цена) - нужно заполнить файл test.txt в таком же формате как train.txt (просто добавить пропущенные цены) 2) Отчет, где изложены ваши мысли в свободной форме и сам алгоритм

Почитать про стакан можно тут https://ru.wikipedia.org/wiki/Биржевой_стакан и тут https://www.machow.ski/posts/2021-07-18-introduction-to-limit-order-books/

Data preprocessing

First of all, let's import the required libraries

In [15]:
```
import numpy as np # https://numpy.org
import pandas as pd # https://pandas.pydata.org

import matplotlib.pyplot as plt # https://matplotlib.org
from tqdm import tqdm # https://tqdm.github.io

from sklearn.preprocessing import StandardScaler, RobustScaler # https://
from sklearn.model_selection import train_test_split # https://scikit-lea

from sklearn.linear_model import LinearRegression, Lasso, Ridge # https:/
from sklearn.metrics import mean_squared_error, mean_absolute_error # htt

import warnings # this helps to get rid of plt warnings
import matplotlib.cbook
warnings.filterwarnings("ignore",category=matplotlib.cbook.mplDeprecation

pd.set_option('display.max_columns', None) # this let us see all columns
```

Now we can read the data and move on

In [16]:
```
col_names = ['Price','Amount','Order']
train = pd.read_csv('train.txt', sep='\t', names=col_names) # reading the
test = pd.read_csv('test.txt',sep='\t', names=col_names) # reading the da
```

In [17]:
```
train.shape # raw data size
```

Out[17]:
```
(378000, 3)
```

In [18]:
```
len(train[train['Price'].str.contains('=')]) # => 9000 independent order
```

Out[18]:
```
9000
```

In [19]:
```
train = train[~train['Price'].str.contains('=')] # getting rid of the sep
test = test[~test['Price'].str.contains('=')]
```

In [20]:
```
train['Order'] = train['Order'].apply(lambda x: 0 if x=='Sell' else 1) #
test['Order'] = test['Order'].apply(lambda x: 0 if x=='Sell' else 1)
```

The each order book contains 40 tuples - price | amount | order. Due to the fact that model is meant to predict only one target variable for these 40 rows, we should modify the dataset and make vector conversions, so that each vector(row) will contain 40 * 3 features, and its final dimension would be **1x120**.

Hence, final matrix dimension will be **9000x120**. We also should include the predicted order(buy/sell), so **9000x121**.

In [21]:
```
X_train, y_train = np.zeros((9000,1+120)), np.zeros(9000) # creating empt
```

In [22]:
```
orderbook_counter = 0 #  two counters for orderbooks and orders respectiv
order_counter = 0

for i in tqdm(range(train.shape[0])): # filling the np.arrays
    row = train.iloc[i]
    if 'price' not in row['Price']:
        X_train[orderbook_counter,order_counter:order_counter+3] = row.to
        order_counter += 3
    elif 'price' in row['Price']:
        pred_order = row['Price'].split(':')[0]
        c = [0 if 'Sell' in pred_order else 1]
        X_train[orderbook_counter,-1] = c[0]
        y_train[orderbook_counter] = float(row['Price'].split(':')[1])
        orderbook_counter += 1
        order_counter = 0
```

```
100%|██████████████████████████████████| 369000/369000 [00:18<00:00, 20113.
14it/s]
```

In [23]:
```
columns_ = [] # creating the title for the final matrix
for i in range(40):
    price, amount, order = 'price_' + str(i), 'amount_' + str(i), 'order_
    columns_.extend([price, amount, order])
```

In [24]:
```
data = pd.DataFrame(X_train, columns=columns_ + ['pred_order'])
```

In [25]:
```
data # the dataset is ready
```

Out[25]:

	price_0	amount_0	order_0	price_1	amount_1	order_1	price_2	amount_2
0	130990.0	41.0	0.0	130985.0	16.0	0.0	130980.0	22.0
1	130995.0	34.0	0.0	130990.0	22.0	0.0	130985.0	4.0
2	131030.0	11.0	0.0	131025.0	11.0	0.0	131020.0	42.0
3	131045.0	19.0	0.0	131040.0	5.0	0.0	131035.0	3.0
4	131035.0	3.0	0.0	131030.0	11.0	0.0	131025.0	11.0
...	...	...	...	...	...	...	...	...
8995	131695.0	45.0	0.0	131690.0	29.0	0.0	131685.0	63.0
8996	131680.0	36.0	0.0	131675.0	26.0	0.0	131670.0	23.0
8997	131675.0	26.0	0.0	131670.0	23.0	0.0	131665.0	48.0
8998	131675.0	26.0	0.0	131670.0	23.0	0.0	131665.0	35.0
8999	131670.0	29.0	0.0	131665.0	35.0	0.0	131660.0	29.0

9000 rows × 121 columns

Data split and Metrics function

In [26]:
```
X_tt,X_val,y_tt,y_val = train_test_split(X_train,y_train, test_size=0.3,r
```

Below we will often use different models, so we will use this function to avoid making a mess of the code. It takes the model and the loss function as arguments. Then it fits the model and returns an error. It is also important to note that, according to the assignment, the predicted value must be a multiple of 5. So the function makes the model predict values divided by 5, and then multiplies the results by 5.

In [27]:
```
def metrics(model, loss): # function to make the code look nice and make

    model.fit(X_tt, y_tt / 5) # according to the task the price should be

    pred_tt = 5 * model.predict(X_tt).astype(np.int32) # it's important t
    pred_val = 5 * model.predict(X_val).astype(np.int32)

    error_train = loss(y_tt,pred_tt)
    error_test = loss(y_val,pred_val)

    print('Results on train set: {}'.format(error_train))
    print('Results on test set: {}'.format(error_test))

    return [error_train,error_test]
```

Applying linear regression

In [28]:
```
linreg = LinearRegression()
```

Let's estimate regression loss using **Mean Abslute Error(MAE)**. We're predicting a price, hence MAE is quite suitable for quantitative feature and the results are easy to interpret.

$$MeanAbsluteError(MAE) = \frac{\sum_{i=1}^{n} |y_i - \hat{y}_i|}{n}$$

In [29]:
```
metrics(linreg,mean_absolute_error)
```

```
Results on train set: 16.56031746031746
Results on test set: 16.54259259259259
```

Out[29]:
```
[16.56031746031746, 16.54259259259259]
```

Feature engineering

As the best practice quants use multiple features to fit the models, however most of them require very precise and varied data such as tick size, strict timing, different asset valuations on different markets e.t.c

All order books have the same size and were taken during periods of low volatility, when (effective) spreads are really small.

Let's try to create few fetures based only on orderbook liqidity.

The inside quotes

$$Spread = \frac{p_a - p_b}{p_a} * 100$$

Best ask price - p_a

Best bid price - p_b

The inside quotes, which are also known as the Best Bid and Offer or BBO, are the highest bid, and lowest ask, in the order book. They are the prices at which the next market buy (with the best offer) or market sell (with the best bid) will transact. Traders constantly reshuffling their bids and asks, and other traders interacting with orders, is what causes prices to move. In an actively traded stock, the bid and ask prices—and the quantities of shares available at those prices—will change by the second.

In [80]:
```
data['spread %'] = ((data['price_19'] - data['price_20'])/data['price_19'
```

Orderbook Imbalances

$$IMB_t^{a,i} = \frac{p_{a,t}^i(N_i)}{p_{a,t}^i(1)} * 1000$$

$$IMB_t^{b,i} = \frac{p_{b,t}^i(N_i)}{p_{b,t}^i(1)} * 1000$$

$(1) p_{a,t}^i(x)$

$(2) p_{a,t}^i(1)$

$(3) p_{b,t}^i(x)$

$(4) p_{b,t}^i(1)$

Let us define the following quantities for each market i = 1, . . . , 14 and for fixed quantities N1, . . . , N14 ∊ R≥1.Where **(1)** denotes the average price one would pay at time **t** for a market buy order of size x on market **i** and **(2)** is simply the top ask price. Similarly **(3)** is the average price for a market sell order of size x ∊ R≥1, so that **(4)** is just the top bid price. The quantity IMBa,i can be described as the difference in basis points between the top ask price on market i and the average price of a market order of size Ni, and analogously for the bid version IMBb,i.

What are sensible choices of N ∊ [1,∞) for i = 1,...,14? Note that N = 1 always yields IMBa,i = 0, while letting Ni → ∞ we have IMBa,i → ∞; thus the two extreme ends of the spectrum are void of any signal. **We set it N_i to be the median liquidity within the top five basis points** of the top of the book on market i.

In [83]:
```
data['IMBa']= ((data.iloc[0][:15:3].median()/data['price_0'])-1)*1000
```

In [84]:
```
data['IMBb']= ((data.iloc[0][:15:3].median()/data['price_20'])-1)*1000
```

Normalization

In [190…
```
scaler = StandardScaler()
```

In [198…
```
X_train_features = data.values # updated data set

X_tt,X_val,y_tt,y_val = train_test_split(X_train_features,y_train, test_s
```

In [205…

```
scaler.fit(X_tt)
X_tt = scaler.transform(X_tt)
X_val = scaler.transform(X_val)
```

Regularization

L1 Lasso regression

L1 regularization adds a penalty that is equal to the absolute value of the magnitude of the coefficient. This regularization type can result in sparse models with few coefficients. Some coefficients might become zero and get eliminated from the model. Larger penalties result in coefficient values that are closer to zero (ideal for producing simpler models)

$$Loss = Error(Y - \widehat{Y}) + \lambda \sum_{1}^{n} |w_i|$$

- λ denotes the amount of shrinkage.
- λ = 0 implies all features are considered and it is equivalent to the linear regression where only the residual sum of squares is considered to build a predictive model
- λ = ∞ implies no feature is considered i.e, as λ closes to infinity it eliminates more and more features
- The bias increases with increase in λ
- variance increases with decrease in λ

In [ ]:

```
alphas = np.arange(0.001, 1, 0.01) # hyperparameter matching
df_lasso = pd.DataFrame({"alpha_l1": alphas,
                         "MAE_train_l1": np.zeros(len(alphas)),
                         "MAE_val_l1": np.zeros(len(alphas))}) # creating

for a in tqdm(range(len(alphas))):
    alpha = alphas[a]
    lasso = Lasso(alpha = alpha)
    df_lasso.iloc[a] = [alpha] + metrics(lasso, mean_absolute_error) # fi
```

In [31]:

```
df_lasso.sort_values(by = "MAE_val_l1").head(3) # So the best shot is 15.
```

Out[31]:

	alpha_l1	MAE_train_l1	MAE_val_l1
77	0.771	15.915873	15.955556
78	0.781	15.910317	15.962963
79	0.791	15.897619	15.974074

Let's look at the weights and how many of them have been turned to 0

In [78]:
```
L1 = Lasso(alpha = 0.771)
metrics(L1,mean_absolute_error)
L1.coef_
```

```
Results on train set: 15.915873015873016
Results on test set: 15.955555555555556
```

```
/Users/ivanpavlovich/opt/anaconda3/lib/python3.8/site-packages/sklearn/linear_model/_coordinate_descent.py:647: ConvergenceWarning: Objective did not converge. You might want to increase the number of iterations, check the scale of the features or consider increasing regularisation. Duality gap: 5.373e+04, tolerance: 9.201e+03
  model = cd_fast.enet_coordinate_descent(
```

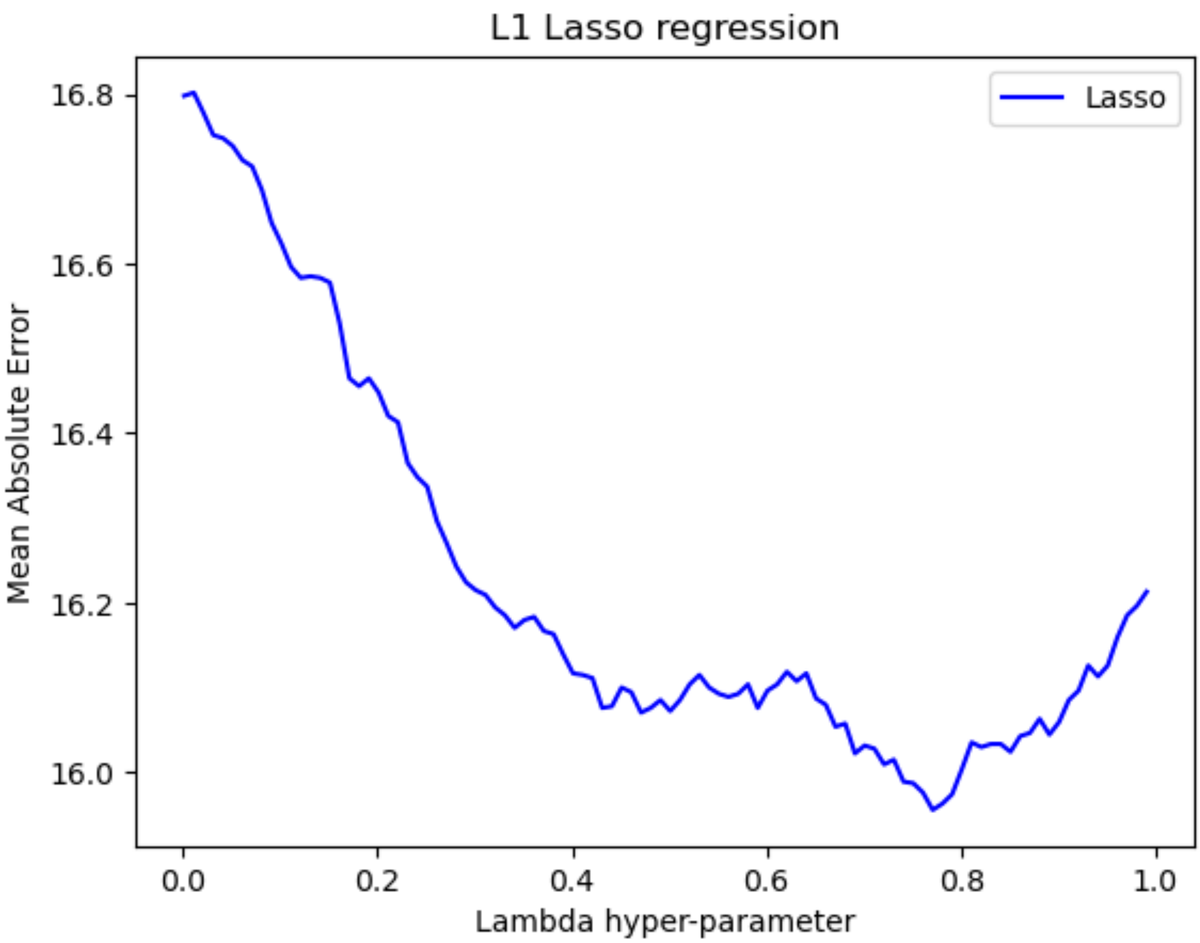
Out[78]:
```
array([ 1.91715640e-01,  7.04639838e-04,  0.00000000e+00,  1.84155239e-04
,
       -0.00000000e+00,  0.00000000e+00,  4.32481263e-05, -5.43282686e-04
,
        0.00000000e+00,  5.78209616e-05, -1.58656460e-04,  0.00000000e+00
,
        3.19560099e-05, -7.75002013e-04,  0.00000000e+00,  1.30661191e-04
,
       -6.36891726e-04,  0.00000000e+00,  0.00000000e+00, -2.71472935e-03
,
        0.00000000e+00,  9.54042404e-05, -7.09715953e-04,  0.00000000e+00
,
        1.18232437e-05, -1.42175029e-03,  0.00000000e+00,  0.00000000e+00
,
       -1.11722427e-03,  0.00000000e+00,  0.00000000e+00, -2.56476419e-03
,
        0.00000000e+00,  0.00000000e+00, -1.24715256e-03,  0.00000000e+00
,
        0.00000000e+00, -4.57798056e-03,  0.00000000e+00,  0.00000000e+00
,
       -2.72903740e-03,  0.00000000e+00,  0.00000000e+00, -4.58994713e-03
,
        0.00000000e+00,  0.00000000e+00, -1.36811981e-03,  0.00000000e+00
,
        0.00000000e+00, -2.83587919e-03,  0.00000000e+00,  0.00000000e+00
,
       -1.86358019e-04,  0.00000000e+00,  0.00000000e+00,  1.90623185e-05
,
        0.00000000e+00,  0.00000000e+00,  1.08294291e-02,  0.00000000e+00
,
        4.06210939e-03, -1.13011108e-02,  0.00000000e+00,  0.00000000e+00
,
       -1.07662335e-03,  0.00000000e+00,  9.50401196e-05,  0.00000000e+00
,
        0.00000000e+00,  2.10353242e-04,  7.15893749e-03,  0.00000000e+00
,
        4.90186886e-04,  6.59380799e-03,  0.00000000e+00,  1.39065288e-04
,
        2.70538891e-03,  0.00000000e+00,  5.93727238e-04,  4.36197088e-03
,
        0.00000000e+00,  5.57734211e-04,  3.48224841e-03,  0.00000000e+00
,
```

```
        3.97153403e-04,  2.19404973e-03,  0.00000000e+00,  2.07913312e-04
,
        3.30910844e-03,  0.00000000e+00,  1.22611128e-04,  4.55328298e-03
,
        0.00000000e+00,  1.72949262e-04,  1.00744853e-03,  0.00000000e+00
,
        0.00000000e+00,  1.15295690e-03,  0.00000000e+00,  1.03797923e-04
,
        3.08264891e-03,  0.00000000e+00,  0.00000000e+00,  1.24025887e-03
,
        0.00000000e+00,  6.18573435e-05,  2.90024059e-04,  0.00000000e+00
,
        2.61658934e-04,  3.85480404e-04,  0.00000000e+00,  1.11748783e-04
,
        3.00817594e-03,  0.00000000e+00,  1.13225341e-04,  2.21640883e-03
,
        0.00000000e+00,  0.00000000e+00,  4.88851552e-04,  0.00000000e+00
,
       -7.36597793e+00])
```

We all know that the data scientists like pretty charts, so let's see how λ relates to the results

In [80]:
```
fig = plt.figure()
ax1 = plt.plot(df_lasso['alpha_l1'],df_lasso['MAE_val_l1'],c='b',label='L
plt.title('L1 Lasso regression')
plt.xlabel('Lambda hyper-parameter')
plt.ylabel('Mean Absolute Error')
plt.legend()
```

Out[80]:
```
<matplotlib.legend.Legend at 0x7ff5f0a59d90>
```



There seems to be no extreme outliers and no local extremes, so we can use these weights with the L1 model. We got better results using the X_train dataset without normalization, so let's remember the results and compare with the Ridge regression.

L2 Ridge regression

Ridge regression adds “squared magnitude” of coefficient as penalty term to the loss function. Here the highlighted part represents L2 regularization element.

$$Loss = ||y - X\hat{w}||^2 + \frac{1}{2}\lambda||w||^2$$

- if λ is zero then you can imagine we get back ordinary least squares(OLS)
- if λ is very large then it will add too much weight and it will lead to under-fitting

In [ ]:
```
alphas = np.arange(1, 700, 1)  # hyperparameter matching
df_ridge = pd.DataFrame({"alpha_l2": alphas,
                         "MAE_train_l2": np.zeros(len(alphas)),
                         "MAE_val_l2": np.zeros(len(alphas))})# creating

for a in tqdm(range(len(alphas))):
    alpha = alphas[a]
    ridge = Ridge(alpha = alpha)
    df_ridge.iloc[a] = [alpha] + metrics(ridge, mean_absolute_error)
```

In [47]:
```
df_ridge.sort_values(by = "MAE_val_l2").head(5) # the best shot is 15.74
```

Out[47]:

	alpha_l2	MAE_train_l2	MAE_val_l2
634	635	15.753968	15.738889
662	663	15.761111	15.742593
636	637	15.753175	15.742593
658	659	15.757143	15.744444
663	664	15.765079	15.744444

Knowing that large L2 coefficients can lead to overfitting, we must check for abnormally large coefficients

In [76]:
```
L2 = Ridge(alpha = 635)
metrics(L2,mean_absolute_error)
L2.coef_
```

```
Results on train set: 15.753968253968255
Results on test set: 15.738888888888889
```

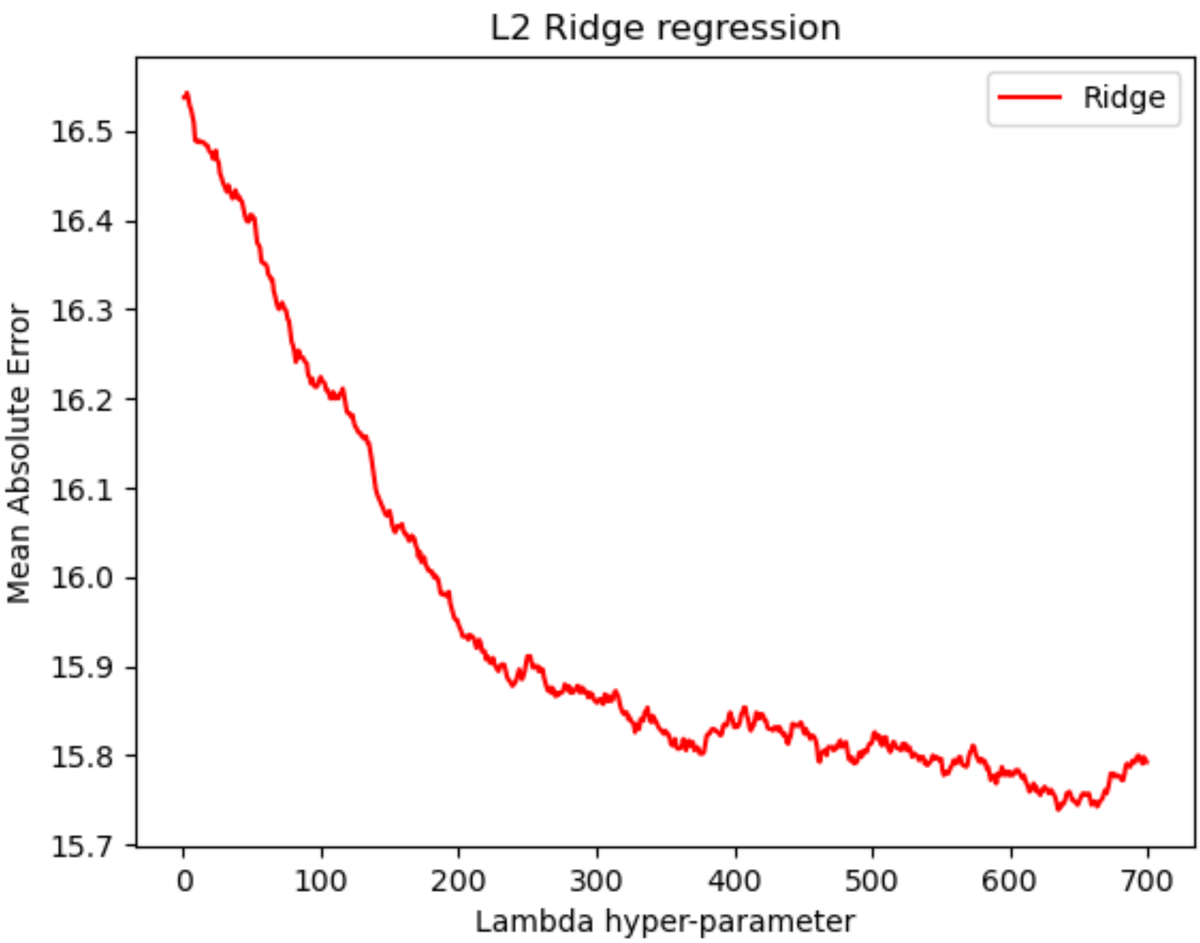
Out[76]:
```
array([ 1.87585645e-02,  6.09792099e-04,  0.00000000e+00,  8.01821449e-02
,
       -6.82548481e-04,  0.00000000e+00, -1.28113348e-02, -8.67033310e-04
,
        0.00000000e+00,  7.48397327e-03, -5.19884554e-04,  0.00000000e+00
,
       -7.88169169e-02, -7.12516456e-04,  0.00000000e+00,  1.16898916e-01
,
       -6.53861030e-04,  0.00000000e+00, -1.92592939e-02, -2.50000545e-03
,
        0.00000000e+00,  5.71960938e-02, -7.33609088e-05,  0.00000000e+00
,
        5.35598655e-02, -9.42939006e-04,  0.00000000e+00, -1.73133711e-02
,
       -8.98087913e-04,  0.00000000e+00,  1.44721478e-01, -2.35754994e-03
,
        0.00000000e+00, -1.17032935e-01, -1.18889270e-03,  0.00000000e+00
,
        3.86114281e-02, -4.49268831e-03,  0.00000000e+00, -3.25262970e-02
,
       -2.45250184e-03,  0.00000000e+00, -1.77388483e-02, -4.49088492e-03
,
        0.00000000e+00, -7.27081948e-03, -1.39885719e-03,  0.00000000e+00
,
       -2.78834610e-02, -2.89712439e-03,  0.00000000e+00, -1.27347608e-01
,
       -7.50139134e-04,  0.00000000e+00,  7.38037892e-02,  1.44823722e-03
,
        0.00000000e+00, -8.98893608e-03,  1.25390909e-02,  0.00000000e+00
,
        4.93826785e-02, -1.17268616e-02,  0.00000000e+00, -7.00299640e-02
,
       -1.39062658e-03,  0.00000000e+00, -1.19247574e-01,  4.86369503e-04
```

```
,
        0.00000000e+00, -8.87382274e-02,  7.82178413e-03,  0.00000000e+00
,
        1.07913801e-01,  7.78186237e-03,  0.00000000e+00, -1.38778637e-01
,
        2.95668498e-03,  0.00000000e+00,  4.72286931e-02,  4.42329855e-03
,
        0.00000000e+00, -4.15736612e-02,  3.33281411e-03,  0.00000000e+00
,
        6.22208476e-02,  2.05006156e-03,  0.00000000e+00, -9.60748980e-03
,
        2.97553001e-03,  0.00000000e+00,  2.36736405e-02,  4.60593297e-03
,
        0.00000000e+00,  1.96304668e-01,  8.94593589e-04,  0.00000000e+00
,
       -1.04537991e-01,  1.04610850e-03,  0.00000000e+00,  1.41802368e-01
,
        2.88690867e-03,  0.00000000e+00, -1.01805030e-01,  9.68529938e-04
,
        0.00000000e+00, -1.59443003e-01,  2.08052939e-04,  0.00000000e+00
,
        1.22861915e-01,  1.77981297e-04,  0.00000000e+00,  3.49065094e-02
,
        2.66018374e-03,  0.00000000e+00,  1.61107321e-01,  1.88461168e-03
,
        0.00000000e+00, -3.79228627e-02,  1.11324248e-03,  0.00000000e+00
,
       -7.43742992e+00])
```

In [49]:

```python
fig = plt.figure()
ax2 = plt.plot(df_ridge['alpha_l2'],df_ridge['MAE_val_l2'],c='r',label='R
plt.title('L2 Ridge regression')
plt.xlabel('Lambda hyper-parameter')
plt.ylabel('Mean Absolute Error')
plt.legend()
```

Out[49]: <matplotlib.legend.Legend at 0x7ff611f272e0>



We achieved an average absolute error of 15.74, which is the best result we got. However, λ is really big here, but in any case we checked the weights and they seem to be fine.

Test prices forecasting

In [58]:
```
test[test['Price'].str.contains(':')].shape[0] # => we have 25 orderbooks
```

Out[58]:
```
25
```

In [62]:
```
X_test, y_test = np.zeros((25, 1 + 3 * 40)), np.zeros(25) # creating empt
```

In [64]:
```
orderbook_counter = 0 #  two counters for orderbooks and orders respectiv
order_counter = 0

for i in tqdm(range(test.shape[0])): # filling the np.arrays
    row = test.iloc[i]
    if 'price' not in row['Price']:
        X_test[orderbook_counter,order_counter:order_counter+3] = row.to_
        order_counter += 3
    elif 'price' in row['Price']:
        pred_order = row['Price'].split(':')[0]
        c = [0 if 'Sell' in pred_order else 1]
        X_test[orderbook_counter,-1] = c[0]
        orderbook_counter += 1
        order_counter = 0
```

```
100%|████████████████████████████████████| 1025/1025 [00:00<00:00, 11397.
23it/s]
```

In [79]:
```
Results = 5 * L2.predict(X_test).astype(np.int32) # predicted prices
```

We received the final values, let's write them into the test.txt file

In [ ]:
```
file = open("test.txt", "r")
lines = file.readlines()

new_lines=[]
order_num = 0
for line in lines:
            if not 'price' in line:
                new_lines.append(line)
            if 'price' in line:
                new_lines.append((line.strip()+' '+str(Results[order_num]
                order_num += 1

file = open("test.txt", "w")
file.writelines(new_lines)
```

Conclusion

We tested different linear regression models on different datasets, basic and updated with features, and we tried each set separately with and without normalization. In the end, we got the best results with the base non normalized dataset, which contains only prices, amounts, and orders, using the L2 model. The final average absolute error is 15.738889, which is 0.8037 less than what we got with the very first model. The regularization showed no significant improvement, although, predicting an accurate result, such values might be critical.

References

- Fragmentation, Price Formation, and Cross-Impact in Bitcoin Markets - Jakob Albers Department of Statistics, University of Oxford, Sam Howison, Mathematical Institute, University of Oxford Oxford, UK
- Aurélien Alfonsi, Antje Fruth, and Alexander Schied. Optimal execution strategies in limit order books with general shape functions. Quantitative Finance, 10(2):143–157, Jun 2009.
- Jean-Philippe Bouchaud, J. Farmer, and F. Lillo. How markets slowly digest changes in supply and demand. Capital Markets: Market Microstructure, 2008.
- Jean-Philippe Bouchaud, Marc Mézard, and Marc Potters. Statistical properties of stock order books: empirical results and models. Quantitative Finance, 2(4):251–256, Aug 2002.
- Rama Cont, Arseniy Kukanov, and Sasha Stoikov. The price impact of order book events. Journal of financial econometrics, 12(1):47–88, 2014.