

Junior quant task Report

by Ivan Pavlovich

Contents

- Task description
- Data preprocessing
- Data split and Metrics function
- Applying linear regression
- Feature engineering
 - The inside quotes
 - Orderbook Imbalances
- Normalization
- Regularization
 - L1 Lasso regression
 - L2 Ridge regression
- Test prices forecasting
- Conclusion
- References

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
(<https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D1%80%D0%B6%D0%B5%D0%B2%>
и тут <https://www.machow.ski/posts/2021-07-18-introduction-to-limit-order-books/>
(<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 # ht
from sklearn.model_selection import train_test_split # https://scik

from sklearn.linear_model import LinearRegression, Lasso, Ridge # h
from sklearn.metrics import mean_squared_error, mean_absolute_error

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

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

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) # readi
test = pd.read_csv('test.txt', sep='\t', names=col_names) # reading
```

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

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

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

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

```
In [19]: train = train[~train['Price'].str.contains('=')] # getting rid of t
test = test[~test['Price'].str.contains('=')]
```

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

The each orderbook 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
```

```
In [22]: orderbook_counter = 0 # two counters for orderbooks and orders res
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] =
        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), 'o'
    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	ord
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=

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
    model.fit(X_tt, y_tt / 5) # according to the task the price sho
    pred_tt = 5 * model.predict(X_tt).astype(np.int32) # it's impor
    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 Absolute Error(MAE)**. We're predicting a price, hence MAE is quite suitable for quantitative feature and the results are easy to interpret.

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

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 liquidity.

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['pri
```

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 $i = 1, \dots, 14$ and for fixed quantities $N_1, \dots, N_{14} \in \mathbb{R}_{\geq 1}$. 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 $x \in \mathbb{R}_{\geq 1}$, so that **(4)** is just the top bid price. The quantity $IMB_{a,i}$ 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 N_i , and analogously for the bid version $IMB_{b,i}$.

What are sensible choices of $N \in [1, \infty)$ for $i = 1, \dots, 14$? Note that $N = 1$ always yields $IMB_{a,i} = 0$, while letting $N_i \rightarrow \infty$ we have $IMB_{a,i} \rightarrow \infty$; 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)*100
```

```
In [84]: data['IMBb'] = ((data.iloc[0][:15:3].median()/data['price_20'])-1)*100
```

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,
```

```
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)

$$Loss = Error(Y - \hat{Y}) + \lambda \sum_1^n |w_i|$$

- λ denotes the amount of shrinkage.
- $\lambda = 0$ 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
- $\lambda = \infty$ 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))}) # create df

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

```
In [31]: df_lasso.sort_values(by = "MAE_val_l1").head(3) # So the best shot .
```

```
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

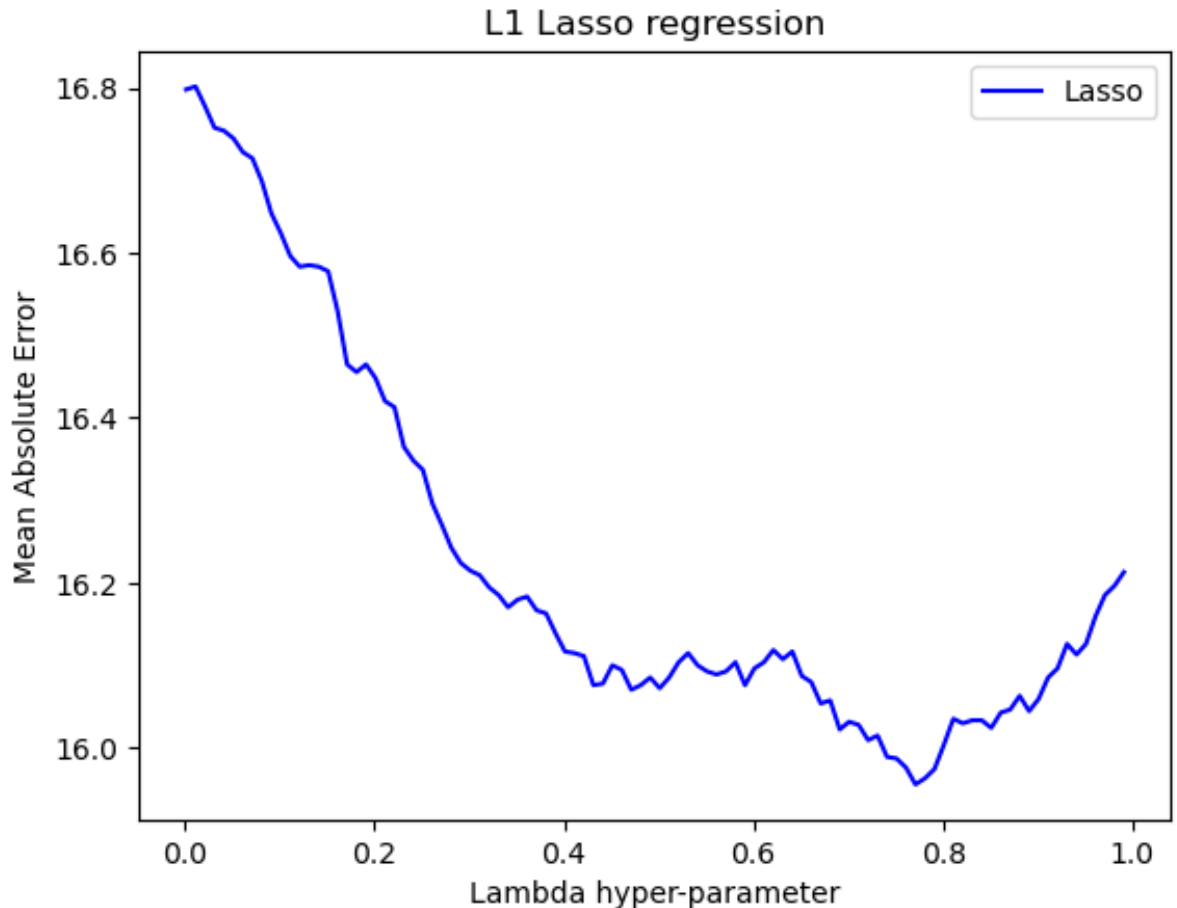
```
In [78]: L1 = Lasso(alpha = 0.771)
metrics(L1,mean_absolute_error)
L1.coef_
岭正则化: duality gap: 9.575e-04, tolerance: 9.201e-05
model = cd_fast.enet_coordinate_descent(

Out[78]: array([ 1.91715640e-01,  7.04639838e-04,  0.00000000e+00,  1.84155
239e-04,
               -0.00000000e+00,  0.00000000e+00,  4.32481263e-05, -5.43282
686e-04,
               0.00000000e+00,  5.78209616e-05, -1.58656460e-04,  0.00000
000e+00,
               3.19560099e-05, -7.75002013e-04,  0.00000000e+00,  1.30661
191e-04,
               -6.36891726e-04,  0.00000000e+00,  0.00000000e+00, -2.71472
935e-03,
               0.00000000e+00,  9.54042404e-05, -7.09715953e-04,  0.00000
000e+00,
               1.18232437e-05, -1.42175029e-03,  0.00000000e+00,  0.00000
000e+00,
               -1.11722427e-03,  0.00000000e+00,  0.00000000e+00, -2.56476
419e-03,
               0.00000000e+00,  0.00000000e+00, -1.24715256e-03,  0.00000
000e+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='Lasso')
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))})# create df

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
```

```
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,  8.01821
449e-02,
        -6.82548481e-04,  0.00000000e+00, -1.28113348e-02, -8.67033
310e-04,
         0.00000000e+00,  7.48397327e-03, -5.19884554e-04,  0.00000
000e+00,
        -7.88169169e-02, -7.12516456e-04,  0.00000000e+00,  1.16898
916e-01,
        -6.53861030e-04,  0.00000000e+00, -1.92592939e-02, -2.50000
545e-03,
         0.00000000e+00,  5.71960938e-02, -7.33609088e-05,  0.00000
000e+00,
         5.35598655e-02, -9.42939006e-04,  0.00000000e+00, -1.73133
711e-02,
        -8.98087913e-04,  0.00000000e+00,  1.44721478e-01, -2.35754
994e-03,
         0.00000000e+00, -1.17032935e-01, -1.18889270e-03,  0.00000
000e+00,
         3.86114281e-02, -4.49268831e-03,  0.00000000e+00, -3.25262
```

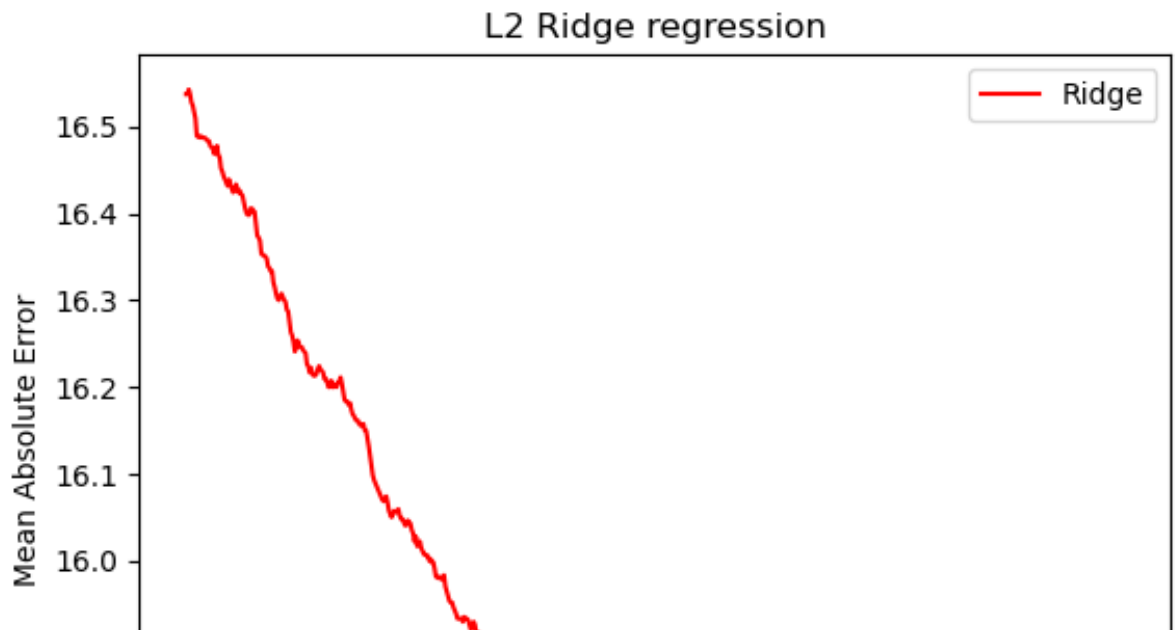
```

970e-02,
    -2.45250184e-03,  0.00000000e+00, -1.77388483e-02, -4.49088
492e-03,
    0.00000000e+00, -7.27081948e-03, -1.39885719e-03,  0.00000
000e+00,
    -2.78834610e-02, -2.89712439e-03,  0.00000000e+00, -1.27347
608e-01,
    -7.50139134e-04,  0.00000000e+00,  7.38037892e-02,  1.44823
722e-03,
    0.00000000e+00, -8.98893608e-03,  1.25390909e-02,  0.00000
000e+00,
    4.93826785e-02, -1.17268616e-02,  0.00000000e+00, -7.00299
640e-02,
    -1.39062658e-03,  0.00000000e+00, -1.19247574e-01,  4.86369
503e-04,
    0.00000000e+00, -8.87382274e-02,  7.82178413e-03,  0.00000
000e+00,
    1.07913801e-01,  7.78186237e-03,  0.00000000e+00, -1.38778
637e-01,
    2.95668498e-03,  0.00000000e+00,  4.72286931e-02,  4.42329
855e-03,
    0.00000000e+00, -4.15736612e-02,  3.33281411e-03,  0.00000
000e+00,
    6.22208476e-02,  2.05006156e-03,  0.00000000e+00, -9.60748
980e-03,
    2.97553001e-03,  0.00000000e+00,  2.36736405e-02,  4.60593
297e-03,
    0.00000000e+00,  1.96304668e-01,  8.94593589e-04,  0.00000
000e+00,
    -1.04537991e-01,  1.04610850e-03,  0.00000000e+00,  1.41802
368e-01,
    2.88690867e-03,  0.00000000e+00, -1.01805030e-01,  9.68529
938e-04,
    0.00000000e+00, -1.59443003e-01,  2.08052939e-04,  0.00000
000e+00,
    1.22861915e-01,  1.77981297e-04,  0.00000000e+00,  3.49065
094e-02,
    2.66018374e-03,  0.00000000e+00,  1.61107321e-01,  1.88461
168e-03,
    0.00000000e+00, -3.79228627e-02,  1.11324248e-03,  0.00000
000e+00,
    -7.43742992e+00] )

```

```
In [49]: fig = plt.figure()
ax2 = plt.plot(df_ridge['alpha_l2'], df_ridge['MAE_val_l2'], c='r', label='Ridge')
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 orders
```

Out[58]: 25

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

```
In [64]: orderbook_counter = 0 # two counters for orderbooks and orders res
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] = r
        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 price.
```

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.