KnutKnut devlog & data visualizer

October 3, 2024

1 KnutKnut devlog

Vlad Oleksik, Wasik Mahir, Constantin-Marius Achim

1.1 Visualizing the data

In order to estimate the time taken to perform the deliveries, we shuld first attempt to visualize the correlation between the time of departure and the total duration for each possible path to be taken. The (raw) data can be seen below:

[1]:
```
import pandas as pd
jsonObj = pd.read_json(path_or_buf='traffic.jsonl', lines=True)
print(jsonObj)
```

```
         road depature arrival
0     B->C->E    13:17   15:25
1     A->C->E    07:07   08:47
2     A->C->E    07:59   09:32
3     B->C->E    14:21   16:29
4     B->C->D    10:09   11:13
...       ...      ...     ...
1026  A->C->D    15:44   17:32
1027  A->C->E    15:24   17:00
1028  B->C->E    10:23   12:26
1029  A->C->D    11:18   12:40
1030  A->C->D    13:54   15:13

[1031 rows x 3 columns]
```

1.1.1 'Annotating' the dataset

In order to better process the data above, it would be useful to include a column containing the departure time in minutes passed since midnight (luckily, since the drivers don't do night shifts - yet - this convention is enough).
Another column we need is the interval between the departure and the arrival times, represented in minutes (the precision we have is of the order of a minute).

[2]:

```
from datetime import datetime
jsonObj["timeTaken"]=([(datetime.strptime(arr, '%H:%M') - datetime.strptime(dep,␣
↪'%H:%M')).total_seconds()/60.0 for [arr,dep] in␣
↪zip(jsonObj["arrival"],jsonObj["depature"]) ])
jsonObj["depTime"]=([(datetime.strptime(dep, '%H:%M') - datetime.strptime("00:
↪00", '%H:%M')).total_seconds()/60.0 for dep in jsonObj["depature"] ])
```

[3]:
```
print(jsonObj)
```

```
          road depature arrival  timeTaken  depTime
0      B->C->E    13:17   15:25      128.0    797.0
1      A->C->E    07:07   08:47      100.0    427.0
2      A->C->E    07:59   09:32       93.0    479.0
3      B->C->E    14:21   16:29      128.0    861.0
4      B->C->D    10:09   11:13       64.0    609.0
...        ...      ...     ...        ...      ...
1026   A->C->D    15:44   17:32      108.0    944.0
1027   A->C->E    15:24   17:00       96.0    924.0
1028   B->C->E    10:23   12:26      123.0    623.0
1029   A->C->D    11:18   12:40       82.0    678.0
1030   A->C->D    13:54   15:13       79.0    834.0

[1031 rows x 5 columns]
```

We are then separating the dataset per road taken, for later use:

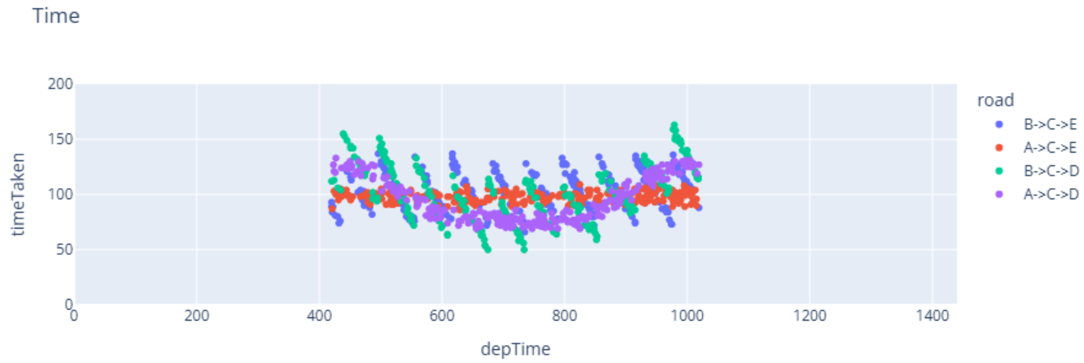
[4]:
```
filterACE = jsonObj.query('road == "A->C->E"')
filterACD = jsonObj.query('road == "A->C->D"')
filterBCE = jsonObj.query('road == "B->C->E"')
filterBCD = jsonObj.query('road == "B->C->D"')
```

1.2 Visualization

Below, we can see the time it takes to take each delivery route, depending on the departure time.

[5]:
```
import plotly.express as px
```

[6]:
```
fig = px.scatter(jsonObj, x="depTime", y="timeTaken", title="Time", color='road')
fig.update_layout(xaxis_range=[0,1440], yaxis_range=[0,200])
fig.show()
```



1.3 Some observations

As it can be seen, for some route choices, the time varies widely - and not continuously.
Intuitively, it can be seen that both alternatives that contain route B->C have their time vary discontinuously along 80-minute intervals. Conversely, the routes that contain the segment A->C are smoother but have more low-amplitude 'noise' regarding the travel time.
Regarding the roads exiting node C, the segment C->D acn be seen to drive a higher travel time until 9:00 and from 14:45, with a decreased travel time in this interval. The routes containing segment C->E are more constant in their travel time over a several hour window.

We would like to see whether the data could support the training of a model that would predict, for each possible departure time in the range, the best road to take and its associated travel time.

[7]:
```
print(jsonObj.sort_values(by=['depTime']))
```

```
        road depature arrival  timeTaken  depTime
365  B->C->D    07:00   08:52      112.0    420.0
391  B->C->E    07:00   08:29       89.0    420.0
610  B->C->E    07:00   08:33       93.0    420.0
782  B->C->E    07:01   08:25       84.0    421.0
539  B->C->D    07:01   08:53      112.0    421.0
..       ...      ...     ...        ...      ...
904  B->C->D    16:56   18:51      115.0   1016.0
521  B->C->D    16:58   18:54      116.0   1018.0
407  B->C->D    16:58   18:52      114.0   1018.0
765  A->C->D    16:59   19:06      127.0   1019.0
486  B->C->E    16:59   18:27       88.0   1019.0

[1031 rows x 5 columns]
```

[8]:
```
jsonObj["arrTime"]=([(datetime.strptime(arr, '%H:%M') - datetime.strptime("00:
↪00", '%H:%M')).total_seconds()/60.0 for arr in jsonObj["arrival"] ])
print(jsonObj.sort_values(by=['arrTime']))
```

```
        road depature arrival  timeTaken  depTime  arrTime
782  B->C->E    07:01   08:25       84.0    421.0    505.0
951  B->C->E    07:12   08:26       74.0    432.0    506.0
972  B->C->E    07:06   08:27       81.0    426.0    507.0
321  A->C->E    07:01   08:28       87.0    421.0    508.0
391  B->C->E    07:00   08:29       89.0    420.0    509.0
..       ...      ...     ...        ...      ...      ...
189  B->C->D    16:39   18:59      140.0    999.0   1139.0
704  B->C->D    16:21   18:59      158.0    981.0   1139.0
838  B->C->D    16:40   19:00      140.0   1000.0   1140.0
536  B->C->D    16:19   19:02      163.0    979.0   1142.0
765  A->C->D    16:59   19:06      127.0   1019.0   1146.0

[1031 rows x 6 columns]
```

1.4 Discrete data

However, as it can be seen (and as it is obvious) there is not enough data to simply compare the alternatives by departure time and use the 'database' model.

Another approach that we would be interested in taking is *isolating* the relative effect of each track segment by subtracting the travel times from routes differing in only one segment and having the same departure/arrival time. However, the overlap for each departure/arrival time is limited and, although the outcome of modelling the travel time on each road segment is still much desired for the scalability of the app, we should devise a different strategy.

Ulimately, one of the sound approaches we've considered is initially modelling each path (A->C->D, A->C->E, B->C->D, B->C->E) individually to compare the travel times for this particular road configuration.

1.5 Decision tree model

Since the data has multiple discontinuities, one of the attempts would be to create a decision tree that would segment the departure time axis into segments, each predicted by a linear model. However as it can be seen below, training such a model using a pure, unoptimized fortuna algorithm proves to be very inefficient.

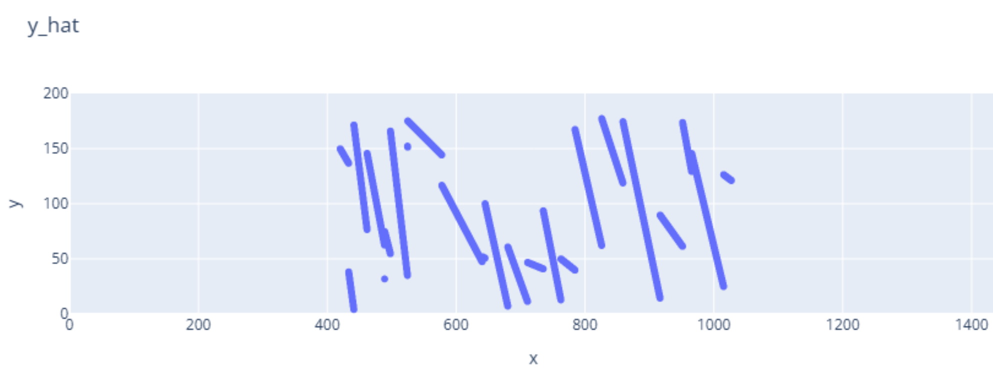
[72]:
```
%%time
import tqdm
import numpy as np

def predict_time(xs, theta):
    thresh, a_s, b_s = theta
    times = []
    for x in xs:
        ok = 0
        for i in range(thresh.shape[0]):
            if x<thresh[i]:
                times.append(a_s[i]*(thresh[i]-x) + b_s[i])
```

```
                ok=1
                break
        if(ok==0):
            times.append(a_s[thresh.shape[0]]*x+b_s[thresh.shape[0]])
    return times

def sample_bs(size_of_theta):
    theta = np.random.uniform(0, 200, size=size_of_theta)
    return theta

def sample_as(size_of_theta):
    theta = np.random.uniform(0, 5, size=size_of_theta)
    return theta

def sample_thresh(size_of_theta):
    theta = np.sort(np.random.uniform(420, 1030, size=size_of_theta))
    return theta


def get_loss(ys, ys_hat):
    loss = 0.0
    for y_hat,y in zip(ys_hat,ys):
        #print(jsonObj.loc[(jsonObj['road']==road) &␣
 ↪(jsonObj['depTime']==x)]['timeTaken'].tolist()[0])
        loss += ((y_hat - y)**2)
    return loss


no_intervals = 24
best_theta_ACE = sample_thresh(no_intervals-1), sample_as(no_intervals),␣
 ↪sample_bs(no_intervals)
best_theta_ACD = sample_thresh(no_intervals-1), sample_as(no_intervals),␣
 ↪sample_bs(no_intervals)
best_theta_BCD = sample_thresh(no_intervals-1), sample_as(no_intervals),␣
 ↪sample_bs(no_intervals)
best_theta_BCE = sample_thresh(no_intervals-1), sample_as(no_intervals),␣
 ↪sample_bs(no_intervals)
best_loss_ACE = float('inf')
best_loss_ACD = float('inf')
best_loss_BCD = float('inf')
best_loss_BCE = float('inf')

#Training ACD
road="A->C->D"
xs_ACD =np.array(filterACD['depTime'])
ys_ACD = np.array([jsonObj.loc[(jsonObj['road']==road) &␣
 ↪(jsonObj['depTime']==x)]['timeTaken'].tolist()[0] for x in xs_ACD])
```

```
for _ in tqdm.tqdm(range(100000)):
    curr_theta = sample_thresh(no_intervals-1), sample_as(no_intervals),␣
 ↪sample_bs(no_intervals)
    ys_hat = predict_time(xs_ACD,curr_theta)
    loss = get_loss(ys_ACD, ys_hat)
    if loss < best_loss_ACD:
        best_loss_ACD=loss
        best_theta_ACD = curr_theta

print(best_loss_ACD)
```

```
100%||
100000/100000 [01:01<00:00, 1613.44it/s]

478843.8115061116
CPU times: total: 41.3 s
Wall time: 1min 2s
```

[73]:
```
xs = np.arange(420,1030,0.1)
fig = px.scatter(x=xs, y=predict_time(xs, best_theta_ACD), title="y_hat")
fig.update_layout(xaxis_range=[0,1440], yaxis_range=[0,200])
fig.show()
```



[28]:
```
print(best_theta_ACD)
```

```
(array([ 426.77579872,  432.47477775,  449.91023443,  496.32599549,
         548.79569556,  633.06009962,  759.52846795,  860.14312735,
         921.87368205,  966.70453491, 1016.89142874]), array([0.00624798,
1.06574447, 4.58278226, 2.43339224, 4.18097506,
       2.25461541, 0.52819191, 1.58460408, 1.99890074, 1.30844123,
       4.6858917 , 4.40228072]), array([120.42502762, 169.42543956,
65.12621806, 180.90877239,
```

```
 21.77851521,  38.31511336,  83.12520948,  85.44822229,
100.92189711, 129.43510873,   9.05181367, 172.04401769]))
```

1.6 A more suitable model

Another model, more adapted to the data we have, is $\hat{y} = a * (x \mod m) + b + c * (x - d) * (x - e)$. This model represents the sum between a linear, periodic function of x (mod m) and a quadratic function of x. The $(x - x_1) * (x - x_2) * c_1 + c_2$ formulation for the quadratic function has been chosen since the parameters have a clearer dependence on the root (and, thus minimum/maximum) position.

1.6.1 A->C->D Route

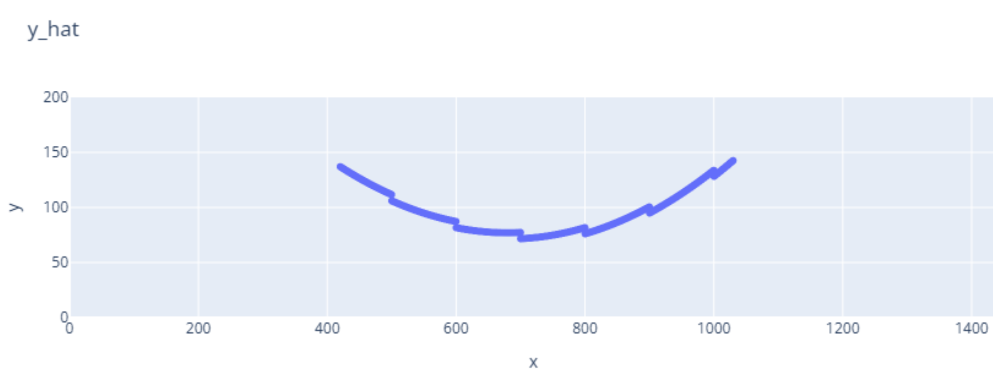
[99]:
```
%%time
import tqdm
import random
import numpy as np

def predict_time(xs, a,b,c,d,e,m):
    times = []
    for x in xs:
        times.append(a*(x%m)+b + c*(x-d)*(x-e))
    return times

def sample_theta(size_of_theta):
    theta = np.random.uniform(-1000, 1000, size=size_of_theta)
    theta[2]=np.random.uniform(-0.01,0.01)
    return theta

def get_loss(ys, ys_hat, theta):
    loss = 0.0
    for y_hat,y in zip(ys_hat,ys):
        loss += ((y_hat - y)**2)
    loss+=1000000000.0/(theta[5]*theta[5])
    return loss


best_theta_ACD = sample_theta(6)
best_loss_ACD = float('inf')

#Training ACD
road="A->C->D"
xs =np.array(filterACD['depTime'])
ys = np.array([jsonObj.loc[(jsonObj['road']==road) &␣
 ↪(jsonObj['depTime']==x)]['timeTaken'].tolist()[0] for x in xs])
for _ in tqdm.tqdm(range(1000000)):
    curr_theta = sample_theta(6)
    if(curr_theta[5]<0):
```

```
            curr_theta[5]=-curr_theta[5]
        curr_theta[5]/=10
        i = random.randrange(6)
        for j in range(6):
            if(j!=i):
                curr_theta[j]=best_theta_ACD[j]
        [a,b,c,d,e,m] = curr_theta
        ys_hat = predict_time(xs,a,b,c,d,e,m)
        loss = get_loss(ys, ys_hat,curr_theta)
        if loss < best_loss_ACD:
            best_loss_ACD=loss
            best_theta_ACD = curr_theta

print(best_loss_ACD)
```

```
100%||
1000000/1000000 [04:10<00:00, 3984.37it/s]

107547.27015406975
CPU times: total: 2min 45s
Wall time: 4min 11s
```

[102]:

```
xs = np.arange(420,1030,0.1)
[a,b,c,d,e,m]=best_theta_ACD
#[a,b,c,d,e,m]=[5.58504856e-02, 9.35050080e+01, 7.21302751e-04, 5.44132759e+02,␣
 ↪8.94329003e+02, 9.99987763e+01]
fig = px.scatter(x=xs, y=predict_time(xs, a,b,c,d,e,m), title="y_hat")
fig.update_layout(xaxis_range=[0,1440], yaxis_range=[0,200])
fig.show()
```



[101]:

```
print(best_theta_ACD)
```

```
[5.58504856e-02 9.35050080e+01 7.21302751e-04 5.44132759e+02
 8.94329003e+02 9.99987763e+01]
```

1.6.2 B->C->D Route

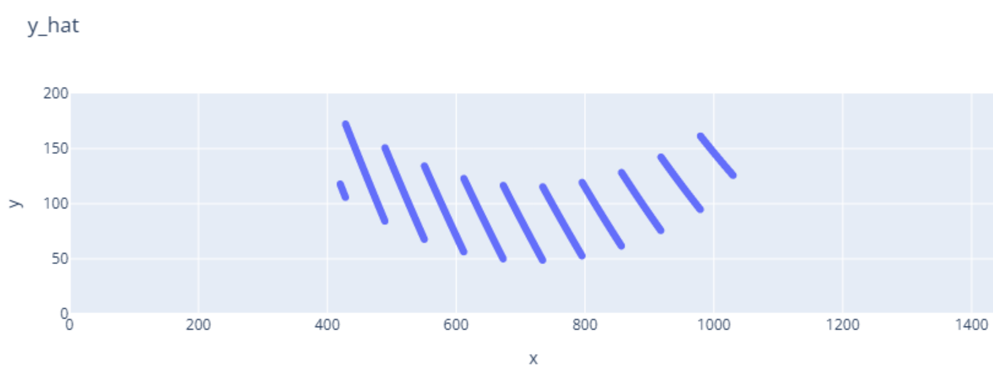
[73]:
```
%%time
import tqdm
import random
import numpy as np

def predict_time(xs, a,b,c,d,e,m):
    times = []
    for x in xs:
        times.append(a*(x%m)+b + c*(x-d)*(x-e))
    return times

def sample_theta(size_of_theta):
    theta = np.random.uniform(-1000, 1000, size=size_of_theta)
    theta[2]=np.random.uniform(-0.01,0.01)
    return theta

def get_loss(ys, ys_hat, theta):
    loss = 0.0
    for y_hat,y in zip(ys_hat,ys):
        loss += ((y_hat - y)**2)
    loss+=1000000000.0/(theta[5]*theta[5])
    return loss


best_theta_BCD = sample_theta(6)
best_loss_BCD = float('inf')

#Training BCD
road="B->C->D"
xs = np.array(filterBCD['depTime'])
ys = np.array([jsonObj.loc[(jsonObj['road']==road) &␣
↪(jsonObj['depTime']==x)]['timeTaken'].tolist()[0] for x in xs])
for _ in tqdm.tqdm(range(100000)):
    curr_theta = sample_theta(6)
    if(curr_theta[5]<0):
        curr_theta[5]=-curr_theta[5]
    curr_theta[5]/=10
    i = random.randrange(6)
    for j in range(6):
        if(j!=i):
            curr_theta[j]=best_theta_BCD[j]
    [a,b,c,d,e,m] = curr_theta
```

```
    ys_hat = predict_time(xs,a,b,c,d,e,m)
    loss = get_loss(ys, ys_hat,curr_theta)
    if loss < best_loss_BCD:
        best_loss_BCD = loss
        best_theta_BCD = curr_theta

print(best_loss_BCD)
```

```
100%||
100000/100000 [00:26<00:00, 3817.16it/s]

309307.3907470814
CPU times: total: 18.2 s
Wall time: 26.3 s
```

[96]:

```
xs = np.arange(420,1030,0.1)
[a,b,c,d,e,m]=best_theta_BCD
#[a,b,c,d,e,m]=[-1.42409584e+00,  1.59847935e+02,  6.01965721e-04,  9.
↪58748083e+02, 4.95124415e+02,  6.11806397e+01]
fig = px.scatter(x=xs, y=predict_time(xs, a,b,c,d,e,m), title="y_hat")
fig.update_layout(xaxis_range=[0,1440], yaxis_range=[0,200])
fig.show()
```



[75]:

```
print(best_theta_BCD)
```

```
[-1.09210764e+00  1.69155299e+02  6.78624516e-04  4.36610227e+02
  9.99696472e+02  6.11842595e+01]
```

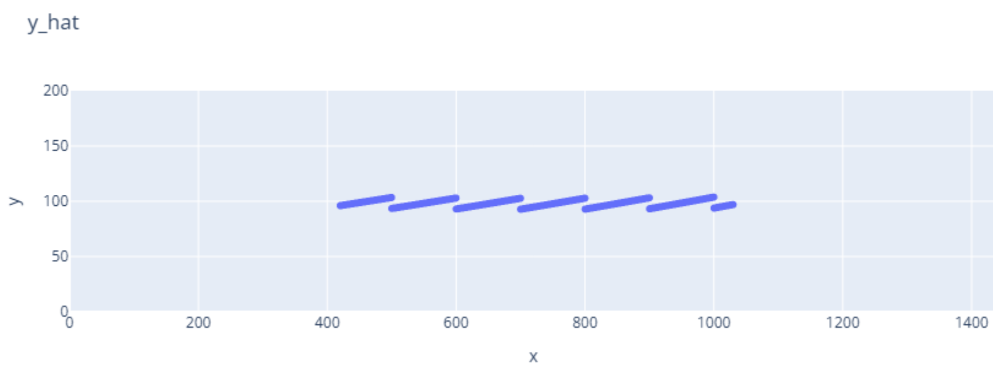
1.6.3 A->C->E Route

[90]:
```
%%time
import tqdm
import random
import numpy as np

def predict_time(xs, a,b,c,d,e,m):
    times = []
    for x in xs:
        times.append(a*(x%m)+b + c*(x-d)*(x-e))
    return times

def sample_theta(size_of_theta):
    theta = np.random.uniform(-1000, 1000, size=size_of_theta)
    theta[2]=np.random.uniform(-0.01,0.01)
    return theta

def get_loss(ys, ys_hat, theta):
    loss = 0.0
    for y_hat,y in zip(ys_hat,ys):
        loss += ((y_hat - y)**2)
    loss+=1000000000.0/(theta[5]*theta[5])
    return loss


best_theta_ACE = sample_theta(6)
best_loss_ACE = float('inf')

#Training ACE
road="A->C->E"
xs = np.array(filterACE['depTime'])
ys = np.array([jsonObj.loc[(jsonObj['road']==road) &␣
 ↪(jsonObj['depTime']==x)]['timeTaken'].tolist()[0] for x in xs])
for _ in tqdm.tqdm(range(1000000)):
    curr_theta = sample_theta(6)
    if(curr_theta[5]<0):
        curr_theta[5]=-curr_theta[5]
    curr_theta[5]/=10
    i = random.randrange(6)
    for j in range(6):
        if(j!=i):
            curr_theta[j]=best_theta_ACE[j]
    [a,b,c,d,e,m] = curr_theta
    ys_hat = predict_time(xs,a,b,c,d,e,m)
    loss = get_loss(ys, ys_hat,curr_theta)
    if loss < best_loss_ACE:
```

```
        best_loss_ACE = loss
        best_theta_ACE = curr_theta

print(best_loss_ACE)
```

```
100%||
1000000/1000000 [04:20<00:00, 3844.31it/s]

106492.51617131643
CPU times: total: 2min 52s
Wall time: 4min 20s
```

[95]:
```
xs = np.arange(420,1030,0.1)
[a,b,c,d,e,m]=best_theta_ACE
#[a,b,c,d,e,m]=[9.93855518e-02, 9.27820981e+01, 1.39691014e-05, 7.22904073e+02,␣
↪7.25313453e+02, 9.99997793e+01]
fig = px.scatter(x=xs, y=predict_time(xs, a,b,c,d,e,m), title="y_hat")
fig.update_layout(xaxis_range=[0,1440], yaxis_range=[0,200])
fig.show()
```



[120]:
```
print(best_theta_ACE)
```

```
[9.93855518e-02 9.27820981e+01 1.39691014e-05 7.22904073e+02
 7.25313453e+02 9.99997793e+01]
```

1.6.4 B->C->E Route

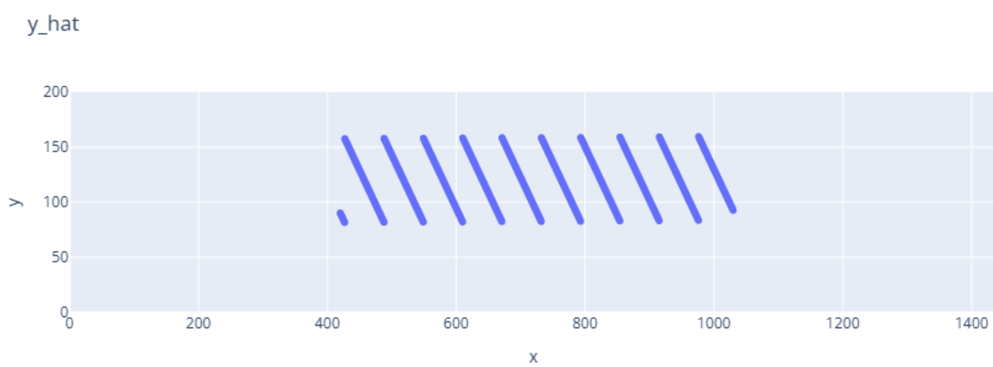
[113]:
```
%%time
import tqdm
import random
import numpy as np
```

```
def predict_time(xs, a,b,c,d,e,m):
    times = []
    for x in xs:
        times.append(a*(x%m)+b + c*(x-d)*(x-e))
    return times

def sample_theta(size_of_theta):
    theta = np.random.uniform(-1000, 1000, size=size_of_theta)
    theta[2]=np.random.uniform(-0.01,0.01)
    return theta

def get_loss(ys, ys_hat, theta):
    loss = 0.0
    for y_hat,y in zip(ys_hat,ys):
        loss += ((y_hat - y)**2)
    loss+=1000000000.0/(theta[5]*theta[5])
    return loss


best_theta_BCE = sample_theta(6)
best_loss_BCE = float('inf')

#Training BCE
road="B->C->E"
xs = np.array(filterBCE['depTime'])
ys = np.array([jsonObj.loc[(jsonObj['road']==road) &␣
 ↪(jsonObj['depTime']==x)]['timeTaken'].tolist()[0] for x in xs])
for _ in tqdm.tqdm(range(100000)):
    curr_theta = sample_theta(6)
    if(curr_theta[5]<0):
        curr_theta[5]=-curr_theta[5]
    curr_theta[5]/=10
    i = random.randrange(6)
    for j in range(6):
        if(j!=i):
            curr_theta[j]=best_theta_BCE[j]
    [a,b,c,d,e,m] = curr_theta
    ys_hat = predict_time(xs,a,b,c,d,e,m)
    loss = get_loss(ys, ys_hat,curr_theta)
    if loss < best_loss_BCE:
        best_loss_BCE = loss
        best_theta_BCE = curr_theta

print(best_loss_BCE)
```

```
100%||
100000/100000 [00:25<00:00, 3916.67it/s]
```

```
179408.0213594903
CPU times: total: 16 s
Wall time: 25.7 s
```

[118]:
```
xs = np.arange(420,1030,0.1)
[a,b,c,d,e,m]=best_theta_BCE
#[a,b,c,d,e,m]=[-1.24989828e+00,  1.55888252e+02,  1.06261846e-06, -8.
↪59365605e+02, -9.50812914e+02,  6.10084942e+01]
fig = px.scatter(x=xs, y=predict_time(xs, a,b,c,d,e,m), title="y_hat")
fig.update_layout(xaxis_range=[0,1440], yaxis_range=[0,200])
fig.show()
```



[119]:
```
print(best_theta_BCE)
```

```
[-1.24989828, 155.888252, 1.06261846e-06, -859.365605, -950.812914, 61.0084942]
```

1.7 Saving the models

Since we have obtained a satisfactory model of the travel time for each of the roads choices, we can save these models to use as-is in our application to predict the best road and estimate the travel time for each case.

[121]:
```
import pickle

thetas = {}
thetas["ACD"]=best_theta_ACD
thetas["ACE"]=best_theta_ACE
thetas["BCD"]=best_theta_BCD
thetas["BCE"]=best_theta_BCE

savefile = open('knutknut_thetas.bin', 'wb')
```

```
pickle.dump(thetas, savefile)
savefile.close()
```

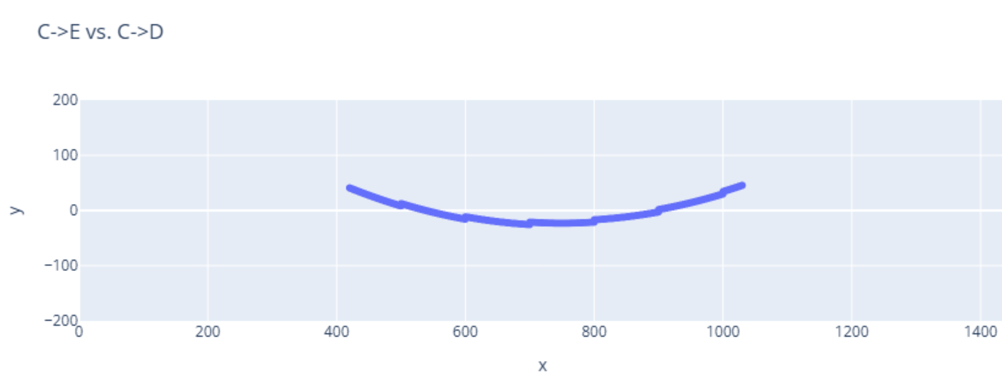
[7]:
```
import pickle

savefile = open('knutknut_thetas.bin', 'rb')

thetas = pickle.load(savefile)
savefile.close()
```

1.8 Isolating each road segment

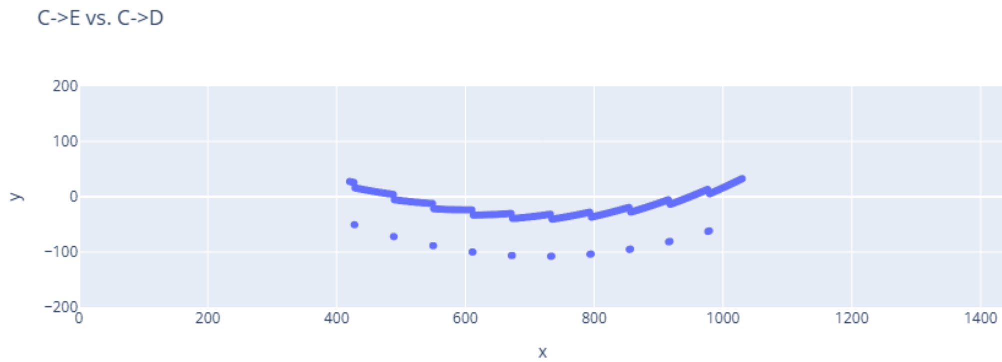
However, the scalability of this approach depends on isolating each road (so that Knut Knut won't have to drive over every single road combination every time a new road opens.
For this, now, that we have a model to predict the travel time for a road choice for each of the (continuously-varying) departure time, we can plot the difference in travel times over time for roads only differing in the second raod segment (the one departing node C).

[8]:
```
import numpy as np
def predict_times(xs, thetas):
    [a,b,c,d,e,m] = thetas
    times = []
    for x in xs:
        times.append(a*(x%m)+b + c*(x-d)*(x-e))
    return times
```

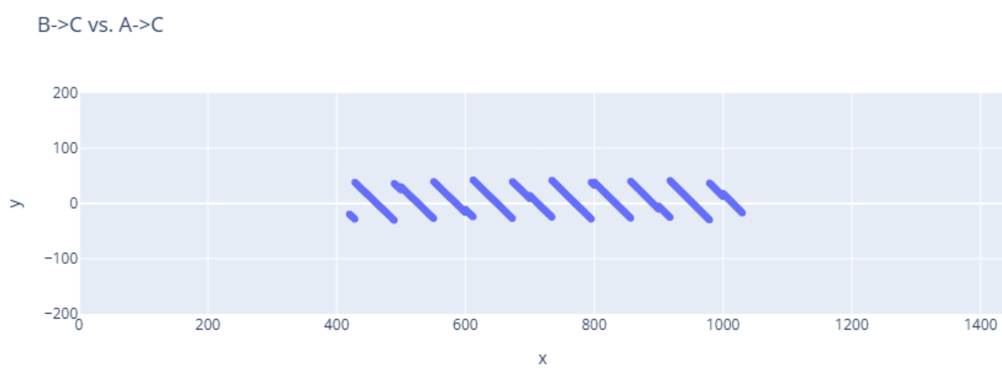
[23]:
```
xs = np.arange(420,1030,0.1)
fig = px.scatter(x=xs, y=np.array(predict_times(xs, thetas["ACD"]))-np.
 ↪array(predict_times(xs, thetas["ACE"])), title="C->E vs. C->D")
fig.update_layout(xaxis_range=[0,1440], yaxis_range=[-200,200])
fig.show()
```



[24]:
```
xs = np.arange(420,1030,0.1)
fig = px.scatter(x=xs, y=np.array(predict_times(xs, thetas["BCD"]))-np.
 ↪array(predict_times(xs, thetas["BCE"])), title="C->E vs. C->D")
fig.update_layout(xaxis_range=[0,1440], yaxis_range=[-200,200])
fig.show()
```

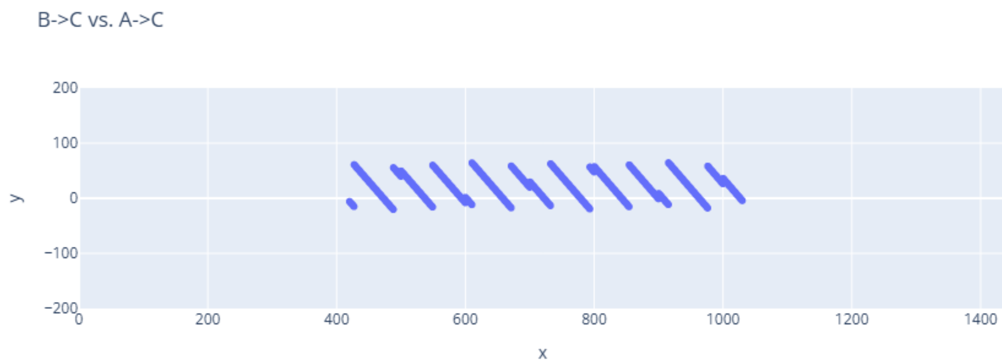


[26]:
```
xs = np.arange(420,1030,0.1)
fig = px.scatter(x=xs, y=np.array(predict_times(xs, thetas["BCD"]))-np.
 ↪array(predict_times(xs, thetas["ACD"])), title="B->C vs. A->C")
fig.update_layout(xaxis_range=[0,1440], yaxis_range=[-200,200])
fig.show()
```



[27]:
```
xs = np.arange(420,1030,0.1)
fig = px.scatter(x=xs, y=np.array(predict_times(xs, thetas["BCE"]))-np.
 ↪array(predict_times(xs, thetas["ACE"])), title="B->C vs. A->C")
fig.update_layout(xaxis_range=[0,1440], yaxis_range=[-200,200])
```

```
fig.show()
```



1.9 Evaluating the model

In order to estimate how much time we save Knut Knut using this model, we can add up all the time taken for the routes in the traffic providing, as seen below, the routes in the dataset are chosen with equal probability, and compare that to the time taken to follow the route recommended by Knut Knut.

[9]:
```
print("ACE: {}".format(filterACE.size))
print("ACD: {}".format(filterACD.size))
print("BCD: {}".format(filterBCD.size))
print("BCE: {}".format(filterBCE.size))
```

```
ACE: 1315
ACD: 1260
BCD: 1295
BCE: 1285
```

[10]:
```
s1 = 0
s2 = 0
for idx,r in jsonObj.iterrows():
    s1 += int(r["timeTaken"])
    dt = int(r["depTime"])
    s2 += int(min(predict_times([dt], thetas["ACE"])[0],predict_times([dt],␣
 ↪thetas["ACD"])[0],predict_times([dt], thetas["BCE"])[0],predict_times([dt],␣
 ↪thetas["BCD"])[0]))

print("Reference drive-time using random approach: {}".format(s1))
print("Reference drive-time using AI model: {}".format(s2))
```

```
Reference drive-time using random approach: 102643
Reference drive-time using AI model: 86784
```

[11]:
```
print("The AI model is saving Knut Knut {}% of their time, increasing their␣
↪revenue by {}%.".format(100.0-s2*100.0/s1,s1*100.0/s2-100.0))
```

```
The AI model is saving Knut Knut 15.450639595491168% of their time, increasing
their revenue by 18.2741058259587%.
```

[16]:
```
inf_rate = 3.5
print("Considering every delivery takes {}% less time on average, taking the␣
↪inflation rate of {}% per year, a further loss of {}% due to the inflation␣
↪while in transit is prevented.".format(100.0-s2*100.0/s1,inf_rate,inf_rate/365.
↪0/24.0/60.0*(s1-s2)/float(s1)))
```

```
Considering every delivery takes 15.450639595491168% less time on average,
taking the inflation rate of 3.5% per year, a further loss of
1.0288667919371974e-06% due to the inflation while in transit is prevented.
```