

```
In [0]: import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from scipy.stats import norm
import datetime
import re
import cvxpy as cp
import warnings

plt.rcParams["figure.figsize"] = [21, 10]
plt.rcParams['axes.labelsize'] = 14
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12
plt.rcParams['text.color'] = 'k'

warnings.filterwarnings("ignore")
plt.style.use('fivethirtyeight')
```

```
In [0]: path = "/content/drive/My Drive/AMS Smart Energy/data/HomeB-meter1_2014.csv"
df_meter = pd.read_csv(path, float_precision='round_trip')
```

```
In [0]: df_meter.shape
```

```
Out[0]: (17520, 18)
```

```
In [0]: df_meter.drop_duplicates('Date & Time', inplace = True)
df_meter['Date & Time'] = pd.to_datetime(df_meter['Date & Time'])
```

```
In [0]: df_meter.loc[df_meter['Date & Time'] == '2014-01-11 00:30:00']
```

```
Out[0]:
```

	Date & Time	use [kW]	gen [kW]	Grid [kW]	AC [kW]	Furnace [kW]	Cellar Lights [kW]	Washer [kW]	First Floor lights [kW]	Util Bas
14591	2014-01-11 00:30:00	0.508599	0	0.508599	0.000022	0.009703	0.006969	0.00031	0.003818	0.0

◀ ▶

```
In [6]: df_meter = df_meter['use [kW]']
df_meter=df_meter[14591:(14591+672)]
len(df_meter)
```

```
Out[6]: 672
```

```
In [0]: backup = df_meter
```

```
In [0]: df_meter.index = np.arange(1, len(df_meter) + 1)
```

```
In [0]: def plot_graph_offline(prov_type, x):
    plt.title("Actual vs Optimal values for Offline " + prov_type + " provisioning")
    plt.plot(x, 'b', label="Optimal Values")
    plt.plot(df_meter, 'r', label="True Values")
    plt.legend()
    plt.ylabel("Electricity Units in kW")
    plt.xlabel("Time step t(1 unit = 15 minutes)")
```

```
In [0]: #Store algorithms, optimal values and decision values for objective function
opt_dict = {}
decision_dict = {}
```

## OFFLINE STATIC OPTIMIZATION

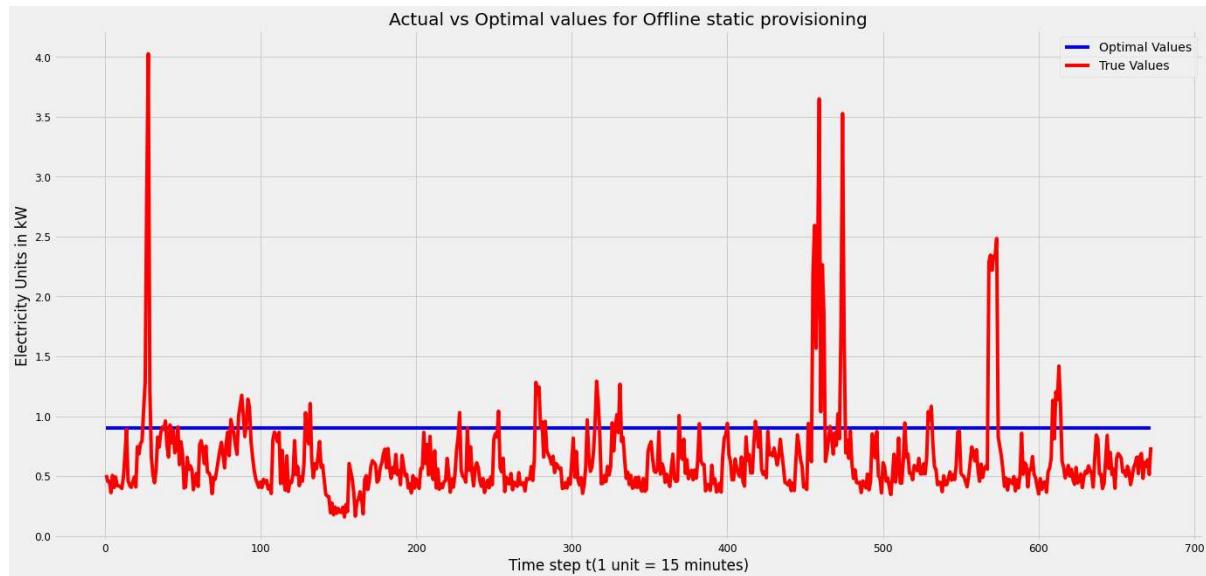
```
In [0]: def offline_static_provision(df, verbose):
    p = 0.4/2
    a = 4/2
    b = 4/2
    y = df.to_list()
    x = cp.Variable(1)

    cost = p*x + a*cp.maximum(0, y - x)
    objective = cp.Minimize(cp.sum(cost) + b*x)
    constraints = [0 <= x]
    problem = cp.Problem(objective, constraints)
    result = problem.solve()
    opt = pd.Series(np.full((672), x.value), index=df.index)
    opt_dict['Offline Static'] = result
    decision_dict['Offline Static'] = opt

    if (verbose == True):
        print("\nThe optimal value is", result)
        print("The optimal x is")
        print(x.value)
        plot_graph_offline('static', opt)
    else:
        return opt
```

```
In [14]: offline_static_provision(df_meter, True)
```

```
The optimal value is 194.313703523238  
The optimal x is  
[0.90273111]
```



Thus as shown above, the static offline optimization gives the value of 0.902 kW and the objective function cost as 194.31. Since this is static optimization, we get a straight line as optimal on graph as above.

## OFFLINE DYNAMIC OPTIMIZATION

```
In [0]: def offline_dynamic_provision(df, verbose):
    p = 0.4/2
    a = 4/2
    b = 4/2
    y = df.to_list()
    x = cp.Variable(672)
    cost = 0

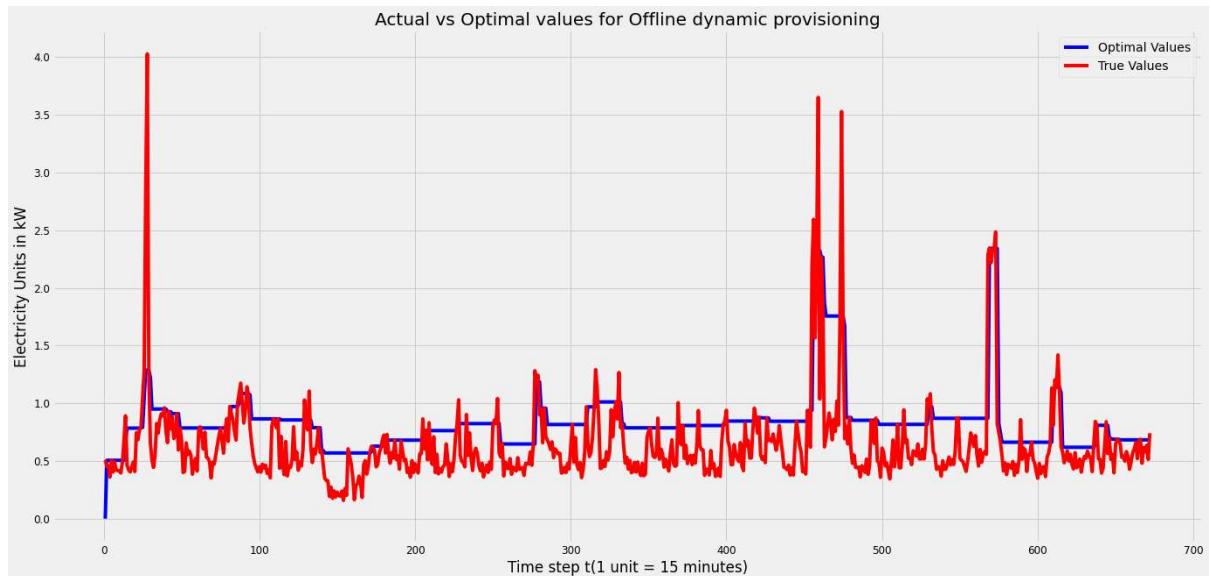
    for i in range(1,672):
        cost += p*x[i] + a*cp.maximum(0, y[i-1] - x[i]) + b*cp.abs(x[i]-x[i-1])

    objective = cp.Minimize(cost)
    constraints = [x[0] == 0, x[1:] >= 0]
    problem = cp.Problem(objective, constraints)
    result = problem.solve()
    opt = pd.Series(np.array(x.value), index=df.index)
    opt_dict['Offline Dynamic'] = result
    decision_dict['Offline Dynamic'] = opt

    if (verbose == True):
        print("\nThe optimal value is", result)
        print("First 15 optimal values of x is")
        print(x.value[0:15])
        plot_graph_offline('dynamic', opt)
    else:
        return opt
```

```
In [16]: offline_dynamic_provision(df_meter, True)
```

The optimal value is 160.78457052256465  
 First 15 optimal values of x is  
 [-2.29303909e-15 5.05893333e-01 5.05893333e-01 5.05893333e-01  
 5.05893333e-01 5.05893333e-01 5.05893333e-01 5.05893333e-01  
 5.05893333e-01 5.05893333e-01 5.05893333e-01 5.05893333e-01  
 5.05893334e-01 6.72743333e-01 7.84803729e-01]



Thus as shown above, the static offline optimization different values of  $x$  for each time step and the objective function cost as 160.78 which is less than the static one. Since this is dynamic optimization, we get a boxed line as optimal on graph as above.

## ONLINE GRADIENT DESCENT

```
In [0]: def cost(method, x, y, verbose):
    cost = 0
    p = 0.4/2
    a = 4/2
    b = 4/2
    for i in range(1, len(y) + 1):
        cost += p * x[i] + a * max(0, y[i] - x[i] + b * abs(x[i] - x[i - 1]))
    if (verbose == True):
        print("\nThe objective value for " + method + " is", cost)
    else:
        return cost
```

```
In [0]: def gradient(x, y, t, p, a, b):
    slope = 0

    if (y[t] > x[t]):
        if (x[t] > x[t - 1]):
            slope = p - a + b;
        else:
            slope = p - a - b;
    else:
        if (x[t] > x[t - 1]):
            slope = p + b;
        else:
            slope = p - b;
    return slope;
```

```
In [0]: def online_gradient_descent(y, steps):
    n = len(y)
    x = [0.0] * (n + 1)
    x[1] = 0
    p = 0.4/2
    a = 4/2
    b = 4/2
    for t in range(1, n):
        x[t + 1] = x[t] - steps * gradient(x, y, t, p, a, b)
    return x
```

```
In [0]: x = online_gradient_descent(df_meter, steps = 0.2)
```

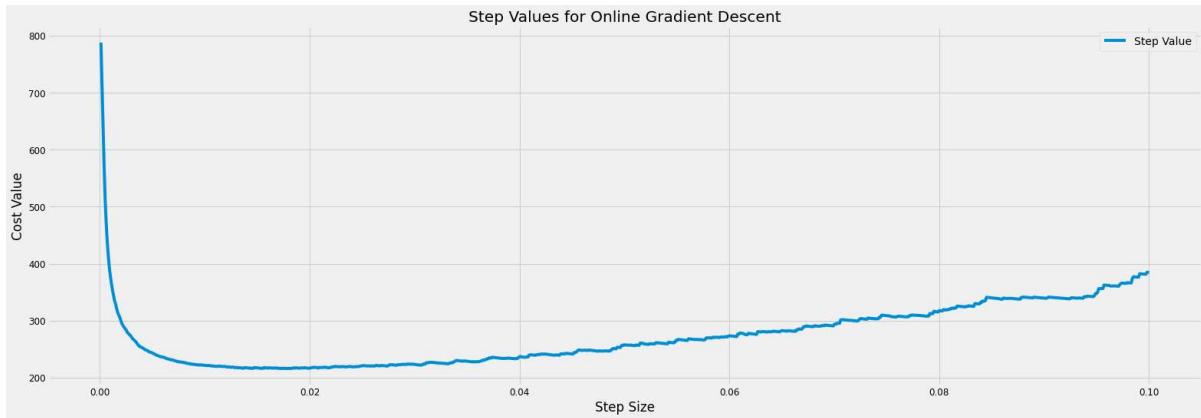
```
In [21]: cost('Online Gradient Descent', x, df_meter, True)
```

The objective value for Online Gradient Descent is 760.4470088960092

```
In [0]: costs = []
steps = []
step = 0
for i in range(1000):
    step += 0.0001
    x = online_gradient_descent(df_meter, steps = step)
    steps.append(step)
    costs.append(cost("Cost", x ,df_meter, False))
```

```
In [23]: fig = plt.figure(figsize=(24, 8))
plt.title("Step Values for Online Gradient Descent")
plt.plot(steps, costs, label = "Step Value")
plt.legend()
plt.ylabel("Cost Value")
plt.xlabel("Step Size")
```

Out[23]: Text(0.5, 0, 'Step Size')



The graph above shows the variation of objective function value w.r.t the step size. Initially, the cost is higher but it falls drastically later and becomes the least at step size of 0.01789. After this, the objective function value again increases gradually.

```
In [24]: steps[costs.index(min(costs))]
```

Out[24]: 0.01789999999999947

```
In [25]: x = online_gradient_descent(df_meter, steps = steps[costs.index(min(costs))])
print("Optimal cost found at step: ", steps[costs.index(min(costs))])
print("\nOptimal cost is:")
print(cost("Online Gradient Descent", x, df_meter, True))
opt_dict['OGD'] = cost("Online Gradient Descent", x, df_meter, False)
decision_dict['OGD'] = x
```

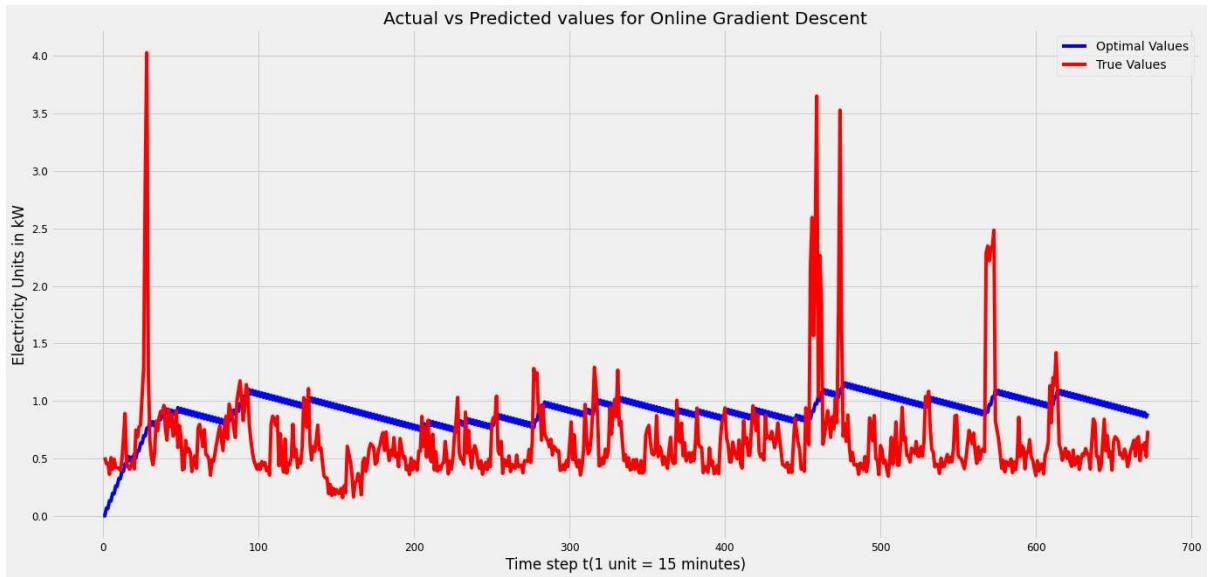
Optimal cost found at step: 0.0178999999999947

Optimal cost is:

The objective value for Online Gradient Descent is 215.90609956600005  
None

```
In [26]: plt.title("Actual vs Predicted values for Online Gradient Descent")
plt.plot(x, 'b', label="Optimal Values")
plt.plot(df_meter, 'r', label="True Values")
plt.legend()
plt.ylabel("Electricity Units in kW")
plt.xlabel("Time step t(1 unit = 15 minutes)")
```

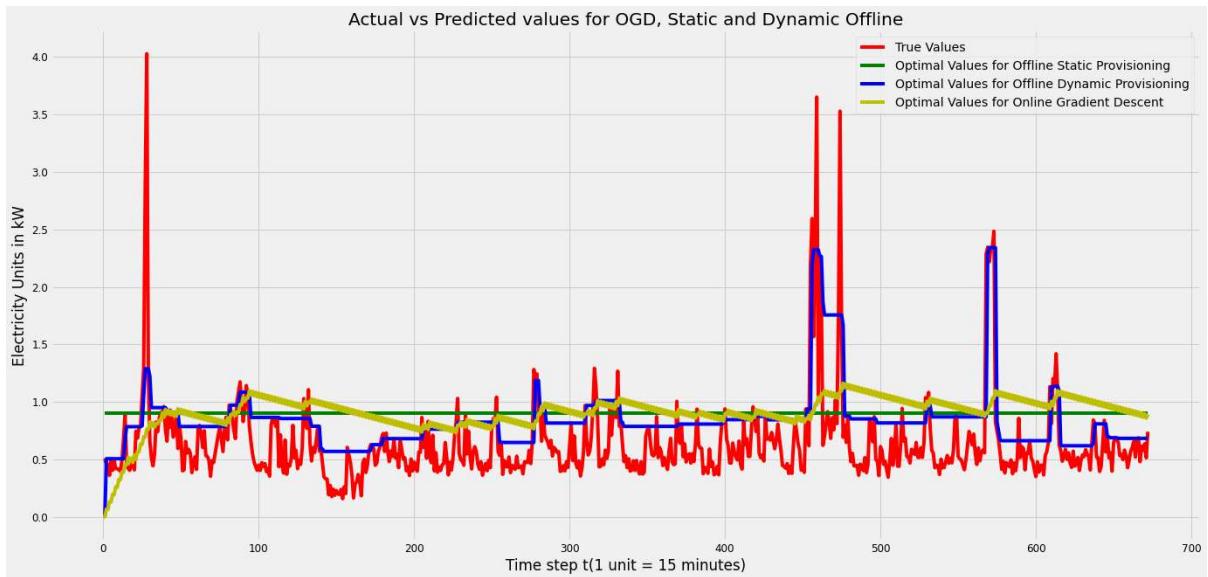
Out[26]: Text(0.5, 0, 'Time step t(1 unit = 15 minutes)')



Thus as shown above, the online gradient descent optimization gives different values of x for each time step T and the least objective function cost 215.906 is obtained at step size 0.0179.

```
In [27]: plt.title("Actual vs Predicted values for OGD, Static and Dynamic Offline")
plt.plot(df_meter, 'r', label="True Values")
plt.plot(offline_static_provision(df_meter, False), 'g', label="Optimal Values
for Offline Static Provisioning")
plt.plot(offline_dynamic_provision(df_meter, False), 'b', label="Optimal Value
s for Offline Dynamic Provisioning")
plt.plot(x, 'y', label="Optimal Values for Online Gradient Descent")
plt.legend()
plt.ylabel("Electricity Units in kW")
plt.xlabel("Time step t(1 unit = 15 minutes)")
```

Out[27]: Text(0.5, 0, 'Time step t(1 unit = 15 minutes)')



The above graph shows the comparison of provisioning by Static Offline, Dynamic Offline and OGD algorithms against the actual values. As discussed in lecture, the optimal values for Online Gradient Descent converges with the optimal value for Static Offline Provisioning but they don't converge with Dynamic Offline Provisioning values.

```
In [0]: decision_dict['Actual'] = df_meter
```

```
In [0]: path = "/content/drive/My Drive/AMS Smart Energy/data/prediction_results_B_ext
ratrees.csv"
y_extratrees = pd.read_csv(path, usecols = ['prediction'])
```

## RHC and CHC with ExtraTrees and SVR for fixed value of a and b

SVR and ExtraTrees were found out to be the best algorithms for House B for prediction in Assignment 1. Following section describes use of RHC and CHC with both of them separately

```
In [0]: def RHC(y, predictionHorizon, prediction_algo):
    T = 2*24*14;
    p = 0.4/2;
    a = 4/2;
    b = 4/2;
    optValues = np.zeros(T);
    for horizonStart in range(0, T):
        horizonEnd = horizonStart + predictionHorizon
        windowY = y[horizonStart: horizonEnd]

        obj = 0;
        x = cp.Variable(predictionHorizon)

        for i in range(0, predictionHorizon):
            obj += p * x[i] + a * cp.maximum(0, windowY[i] - x[i])
            if i == 0:
                obj += b * cp.abs(x[i]); #because x(0) is 0
            else:
                obj += b * cp.abs(x[i] - x[i - 1])

        objective = cp.Minimize(obj)
        problem = cp.Problem(objective)
        result = problem.solve()

        optValues[horizonStart] = x.value[0];

        obj = 0;
        for i in range(0, T):
            obj += p * optValues[i] + a * max(0, y[i] - optValues[i])
            if i == 0:
                obj += b * abs(optValues[i]); #because x(0) is 0
            else:
                obj += b * abs(optValues[i] - optValues[i - 1])

        opt_dict['RHC' + '_' + prediction_algo] = obj
        decision_dict['RHC' + '_' + prediction_algo] = optValues

    return obj
```

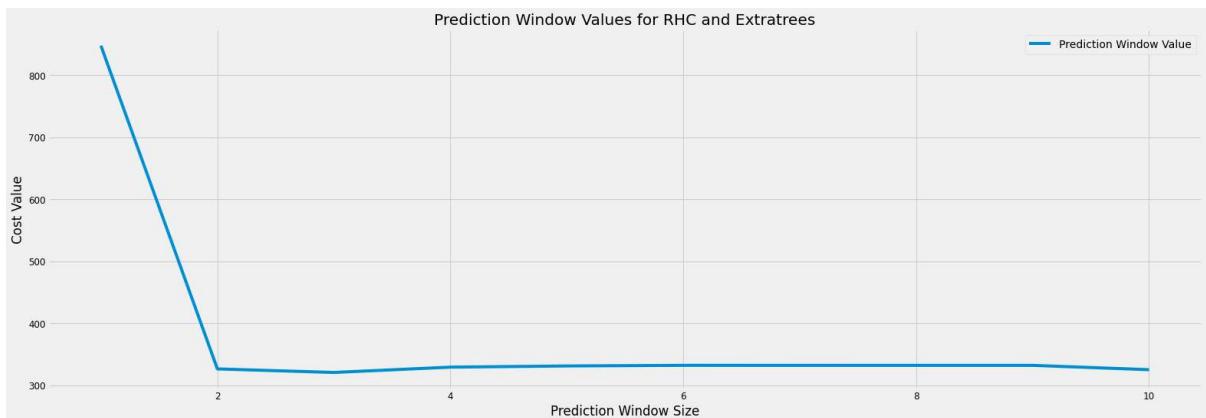
```
In [31]: RHC(y_extratrees['prediction'].to_numpy(), 5, 'ExtraTrees')
```

```
Out[31]: 331.4356744400349
```

```
In [0]: costs = []
windows = []
for i in range(1,11):
    windows.append(i)
    costs.append(RHC(y_extratrees['prediction'].to_numpy(), i, 'ExtraTrees'))
```

```
In [33]: fig = plt.figure(figsize=(24, 8))
plt.title("Prediction Window Values for RHC and Extratrees")
plt.plot(windows, costs, label = "Prediction Window Value")
plt.legend()
plt.ylabel("Cost Value")
plt.xlabel("Prediction Window Size")
```

Out[33]: Text(0.5, 0, 'Prediction Window Size')



The graph above shows the variation of objective function value for combination of RHC and ExtraTrees algorithm w.r.t the prediction window size. Initially, the cost is higher but it falls drastically later and becomes the least (320.96) at window size of 3. After this, the objective function value almost remains constant.

```
In [34]: x = RHC(y_extratrees['prediction'].to_numpy(), windows[costs.index(min(costs))], 'ExtraTrees')
print("Optimal cost for RHC and extratrees found at window size: ", windows[costs.index(min(costs))])
print("\nOptimal cost is: ", x)
```

Optimal cost for RHC and extratrees found at window size: 3

Optimal cost is: 320.96741974782117

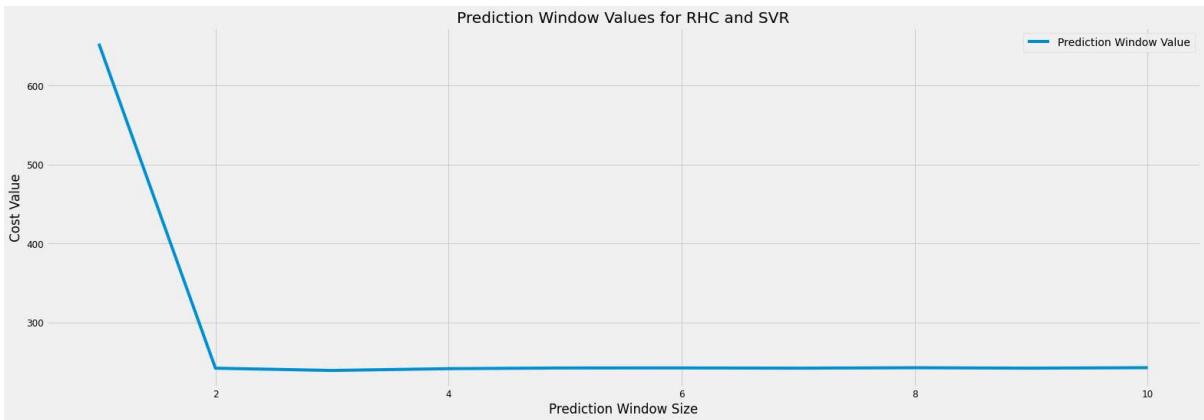
## RHC with SVR

```
In [0]: path = "/content/drive/My Drive/AMS Smart Energy/data/prediction_results_B_sv
r.csv"
y_svr = pd.read_csv(path, usecols = ['prediction'])
```

```
In [0]: costs = []
windows = []
for i in range(1,11):
    windows.append(i)
    costs.append(RHC(y_svr['prediction'].to_numpy(), i, 'SVR'))
```

```
In [37]: fig = plt.figure(figsize=(24, 8))
plt.title("Prediction Window Values for RHC and SVR")
plt.plot(windows, costs, label = "Prediction Window Value")
plt.legend()
plt.ylabel("Cost Value")
plt.xlabel("Prediction Window Size")
```

Out[37]: Text(0.5, 0, 'Prediction Window Size')



The graph above shows the variation of objective function value for combination of RHC and SVR algorithm w.r.t the prediction window size. Initially, the cost is higher but it falls drastically later and becomes the least (239.01) at window size of 3. After this, the objective function value almost remains constant.

```
In [0]: x = RHC(y_svr['prediction'].to_numpy(), windows[costs.index(min(costs))], 'SVR')
print("Optimal cost for RHC and SVR found at window size: ", windows[costs.index(min(costs))])
print("\nOptimal cost is: ", x)
```

Optimal cost for RHC and SVR found at window size: 3

Optimal cost is: 239.01666653802405

## CHC with ExtraTrees and SVR

```
In [0]: def CHC(y, predictionHorizon, commitmentHorizon, prediction_algo):
    T = 2*24*14;
    p = 0.4/2;
    a = 4/2;
    b = 4/2;
    optValues = np.zeros(T + 20);
    for horizonStart in range(0, T):
        horizonEnd = horizonStart + predictionHorizon
        windowY = y[horizonStart: horizonEnd]

        obj = 0;
        x = cp.Variable(predictionHorizon)

        for i in range(0, predictionHorizon):
            obj += p * x[i] + a * cp.maximum(0, windowY[i] - x[i])
            if i == 0:
                obj += b * cp.abs(x[i]); #because x(0) is 0
            else:
                obj += b * cp.abs(x[i] - x[i - 1])

        objective = cp.Minimize(obj)
        problem = cp.Problem(objective)
        result = problem.solve()

        for i in range(0, commitmentHorizon):
            optValues[horizonStart + i] += x.value[i];

    optValues = optValues/commitmentHorizon

    obj = 0;
    for i in range(0, T):
        obj += p * optValues[i] + a * max(0, y[i] - optValues[i])
        if i == 0:
            obj += b * abs(optValues[i]); #because x(0) is 0
        else:
            obj += b * abs(optValues[i] - optValues[i - 1])

    opt_dict['CHC' + '_' + prediction_algo] = obj
    decision_dict['CHC' + '_' + prediction_algo] = optValues

    return obj
```

```
In [0]: CHC(y_extratrees['prediction'].to_numpy(), 10, 1, 'ExtraTrees')
```

```
Out[0]: 325.4237322988211
```

## CHC with ExtraTrees

```
In [0]: costs = []
windows = []
for i in range(1,11):
    windows.append(i)
    costs.append(CHC(y_extratrees['prediction'].to_numpy(), 10, i, 'ExtraTrees'))
```

```
In [40]: fig = plt.figure(figsize=(24, 8))
plt.title("Commitment Horizon Values for CHC and ExtraTrees")
plt.plot(windows, costs, label = "Commitment Horizon Value")
plt.legend()
plt.ylabel("Cost Value")
plt.xlabel("Commitment Horizon Size")
```

Out[40]: Text(0.5, 0, 'Commitment Horizon Size')



The graph above shows the variation of objective function value for combination of CHC and ExtraTrees algorithm w.r.t the commitment horizon size. Initially, the cost is higher but it falls gradually later and becomes the least (188.25) at commitment horizon size of 10. After this, the objective function value almost remains constant.

```
In [41]: x = CHC(y_extratrees['prediction'].to_numpy(), 10, windows[costs.index(min(costs))], 'ExtraTrees')
print("Optimal cost for CHC found at commitment horizon size: ", windows[costs.index(min(costs))])
print("\nOptimal cost is: ", x)
```

Optimal cost for CHC found at commitment horizon size: 10

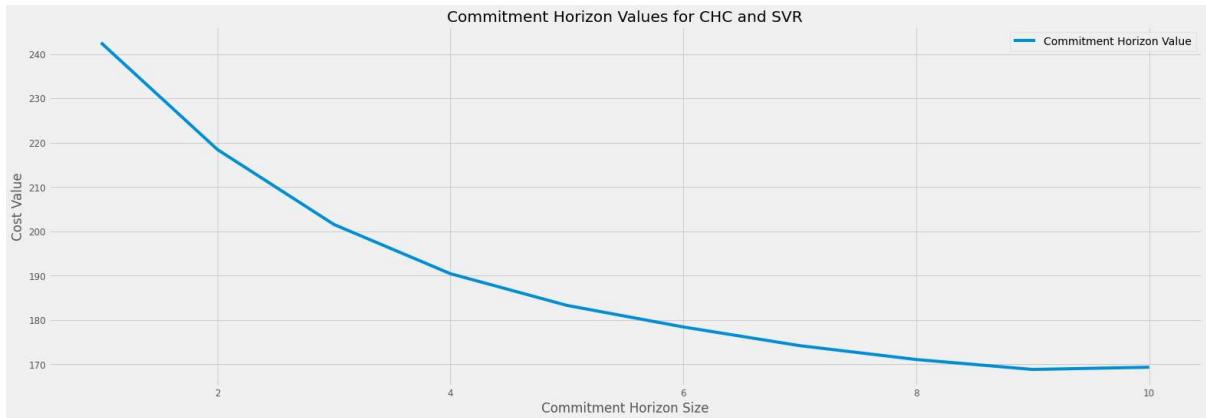
Optimal cost is: 188.25955132022318

## CHC with SVR

```
In [0]: costs = []
windows = []
for i in range(1,11):
    windows.append(i)
    costs.append(CHC(y_svr['prediction'].to_numpy(), 10, i, 'SVR'))
```

```
In [0]: fig = plt.figure(figsize=(24, 8))
plt.title("Commitment Horizon Values for CHC and SVR")
plt.plot(windows, costs, label = "Commitment Horizon Value")
plt.legend()
plt.ylabel("Cost Value")
plt.xlabel("Commitment Horizon Size")
```

Out[0]: Text(0.5, 0, 'Commitment Horizon Size')



The graph above shows the variation of objective function value for combination of CHC and SVR algorithm w.r.t the commitment horizon size. Initially, the cost is higher but it falls gradually later and becomes the least (168.78) at commitment horizon size of 9. After this, the objective function value almost remains constant.

```
In [43]: x = CHC(y_svr['prediction'].to_numpy(), 10, windows[costs.index(min(costs))], 'SVR')
print("Optimal cost for CHC and SVR found at commitment horizon size: ", windows[costs.index(min(costs))])
print("\nOptimal cost is: ", x)
```

Optimal cost for CHC and SVR found at commitment horizon size: 9

Optimal cost is: 168.7876517578396

## Comparison of OGD, CHC, RHC with Static Offline and Dynamic Offline Algorithms

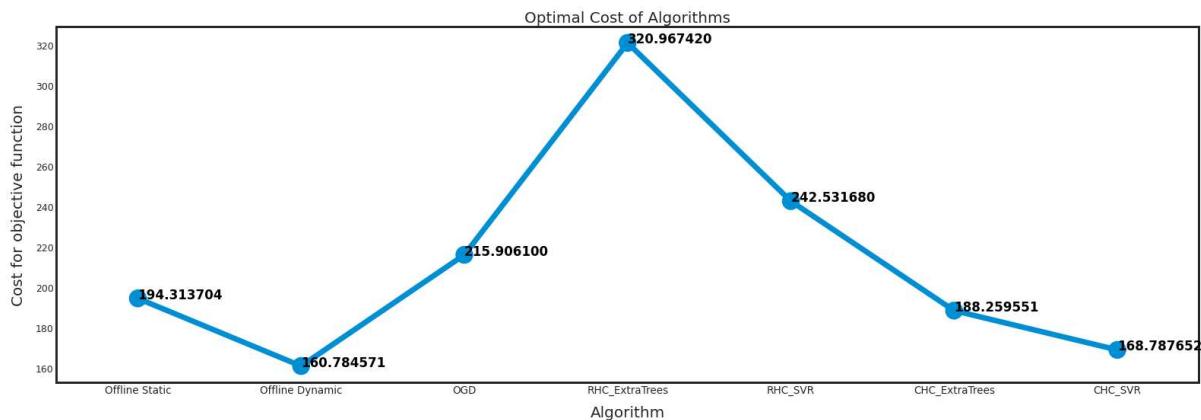
```
In [44]: sns.set_style("white")
fig = plt.figure(figsize=(24, 8))

ax = sns.pointplot(x=list(opt_dict.keys()), y=[score for score in opt_dict.values()], markers=['o'], linestyles=['-'])
for i, score in enumerate(opt_dict.values()):
    ax.text(i, score + 0.002, '{:.6f}'.format(score), horizontalalignment='left', size='large', color='black', weight='semibold')

plt.ylabel('Cost for objective function', size=20, labelpad=12.5)
plt.xlabel('Algorithm', size=20, labelpad=12.5)
plt.tick_params(axis='x', labelsize=13.5)
plt.tick_params(axis='y', labelsize=12.5)

plt.title('Optimal Cost of Algorithms', size=20)

plt.show()
```



The above graph compares the objective function value for Offline Static and Dynamic, OGD, RHC with ExtraTrees, RHC with SVR, CHC with ExtraTrees, CHC with SVR. RHC with ExtraTrees preforms the worst while Offline Dynamic Provisioning preforms the best. Lower the values, higher the preformance and better the accuracy.

```
In [46]: static_obj = opt_dict.pop('Offline Static')
dynamic_obj = opt_dict.pop('Offline Dynamic')
```

```
Out[46]: {'CHC_ExtraTrees': 188.25955132022318,
          'CHC_SVR': 168.7876517578396,
          'OGD': 215.90609956600005,
          'RHC_ExtraTrees': 320.96741974782117,
          'RHC_SVR': 242.53168030840706}
```

```
In [0]: static_dict = {}
for key in opt_dict:
    static_dict[key] = opt_dict[key] - static_obj

dynamic_dict = {}
for key in opt_dict:
    dynamic_dict[key] = opt_dict[key] - dynamic_obj
```

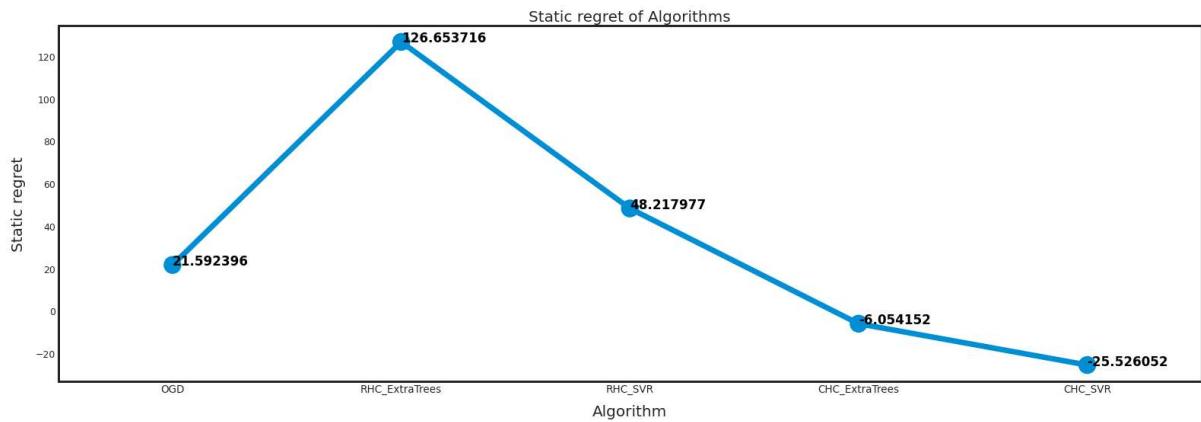
```
In [52]: sns.set_style("white")
fig = plt.figure(figsize=(24, 8))

ax = sns.pointplot(x=list(static_dict.keys()), y=[score for score in static_dict.values()], markers=['o'], linestyles=['-'])
for i, score in enumerate(static_dict.values()):
    ax.text(i, score + 0.002, '{:.6f}'.format(score), horizontalalignment='left', size='large', color='black', weight='semibold')

plt.ylabel('Static regret', size=20, labelpad=12.5)
plt.xlabel('Algorithm', size=20, labelpad=12.5)
plt.tick_params(axis='x', labelsize=13.5)
plt.tick_params(axis='y', labelsize=12.5)

plt.title('Static regret of Algorithms', size=20)

plt.show()
```



Thus, some algorithms have a positive static regret while some have negative. CHC with SVR has the most negative static regret meaning it performs better than static optimal.

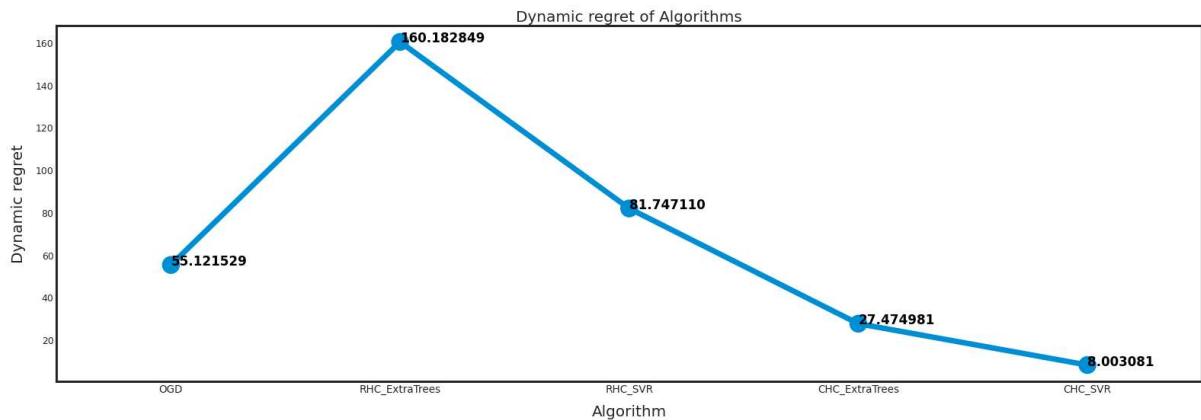
```
In [53]: sns.set_style("white")
fig = plt.figure(figsize=(24, 8))

ax = sns.pointplot(x=list(dynamic_dict.keys()), y=[score for score in dynamic_
dict.values()], markers=['o'], linestyles=['-'])
for i, score in enumerate(dynamic_dict.values()):
    ax.text(i, score + 0.002, '{:.6f}'.format(score), horizontalalignment='lef
t', size='large', color='black', weight='semibold')

plt.ylabel('Dynamic regret', size=20, labelpad=12.5)
plt.xlabel('Algorithm', size=20, labelpad=12.5)
plt.tick_params(axis='x', labelsize=13.5)
plt.tick_params(axis='y', labelsize=12.5)

plt.title('Dynamic regret of Algorithms', size=20)

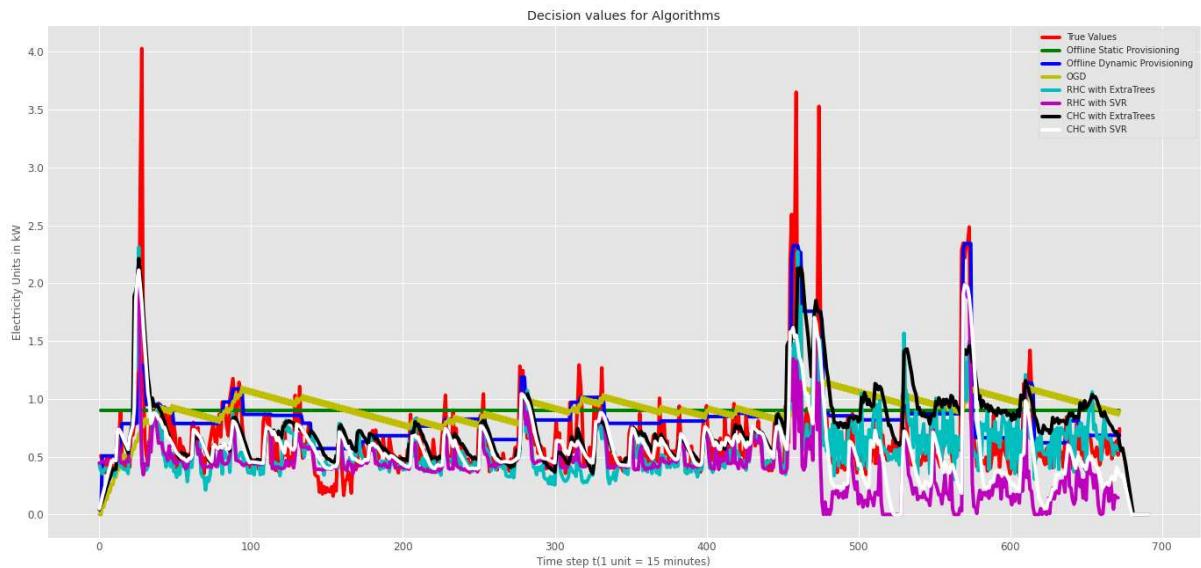
plt.show()
```



Thus all algorithms have a positive dynamic regret meaning dynamic optimal performs better than all.

```
In [0]: plt.style.use('ggplot')
plt.title("Decision values for Algorithms")
plt.plot(df_meter, 'r', label="True Values")
plt.plot(decision_dict['Offline Static'], 'g', label="Offline Static Provisioning")
plt.plot(decision_dict['Offline Dynamic'], 'b', label="Offline Dynamic Provisioning")
plt.plot(decision_dict['OGD'], 'y', label="OGD")
plt.plot(decision_dict['RHC_ExtraTrees'], 'c', label="RHC with ExtraTrees")
plt.plot(decision_dict['RHC_SVR'], 'm', label="RHC with SVR")
plt.plot(decision_dict['CHC_ExtraTrees'], 'k', label="CHC with ExtraTrees")
plt.plot(decision_dict['CHC_SVR'], 'w', label="CHC with SVR")
plt.legend()
plt.ylabel("Electricity Units in kW")
plt.xlabel("Time step t(1 unit = 15 minutes)")
```

Out[0]: Text(0.5, 0, 'Time step t(1 unit = 15 minutes)')



The above graph plots provisions by 7 different algorithms with the actual consumption values.

## Varying a and b for the best combination of control algorithm and prediction algorithm

Best combination is CHC with SVR with commitment horizon size of 9 with  $a=4$  and  $b=4$ . We keep commitment horizon size as constant and vary first  $a$  by keeping  $b$  constant and then vice-versa to see the impact of change.

```
In [0]: def chc_vary_and_b(y, predictionHorizon, commitmentHorizon, prediction_algo, a, b):
    T = 2*24*14;
    p = 0.4/2;
    a = a/2;
    b = b/2;
    optValues = np.zeros(T + 20);
    for horizonStart in range(0, T):
        horizonEnd = horizonStart + predictionHorizon
        windowY = y[horizonStart: horizonEnd]

        obj = 0;
        x = cp.Variable(predictionHorizon)

        for i in range(0, predictionHorizon):
            obj += p * x[i] + a * cp.maximum(0, windowY[i] - x[i])
            if i == 0:
                obj += b * cp.abs(x[i]); #because x(0) is 0
            else:
                obj += b * cp.abs(x[i] - x[i - 1])

        objective = cp.Minimize(obj)
        problem = cp.Problem(objective)
        result = problem.solve()

        for i in range(0, commitmentHorizon):
            optValues[horizonStart + i] += x.value[i];

    optValues = optValues/commitmentHorizon

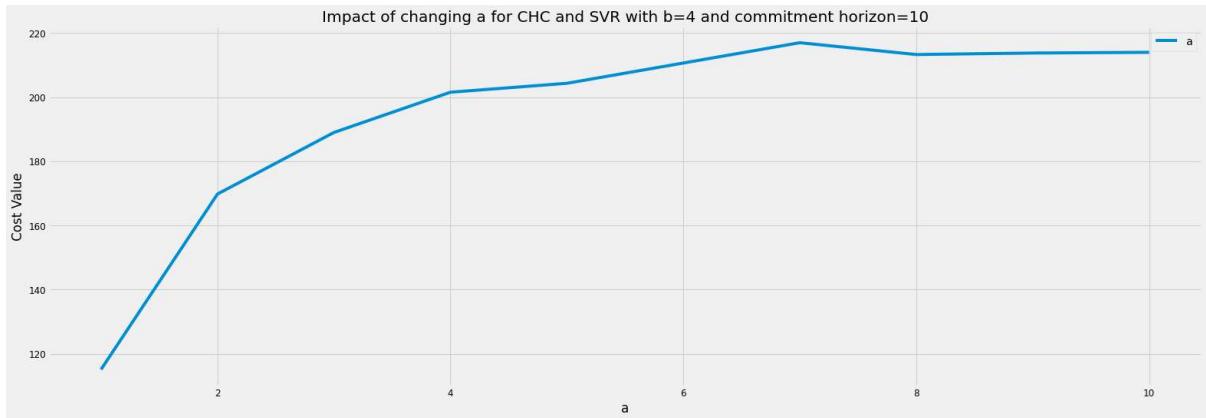
    obj = 0;
    for i in range(0, T):
        obj += p * optValues[i] + a * max(0, y[i] - optValues[i])
        if i == 0:
            obj += b * abs(optValues[i]); #because x(0) is 0
        else:
            obj += b * abs(optValues[i] - optValues[i - 1])

    return obj
```

```
In [0]: costs = []
windows = []
b = 4
for i in range(1,11):
    windows.append(i)
    costs.append(chc_vary_and_b(y_svr['prediction'].to_numpy(), 10, 3, 'SVR', i, b))
```

```
In [0]: fig = plt.figure(figsize=(24, 8))
plt.title("Impact of changing a for CHC and SVR with b=4 and commitment horizon=10")
plt.plot(windows, costs, label = "a")
plt.legend()
plt.ylabel("Cost Value")
plt.xlabel("a")
```

Out[0]: Text(0.5, 0, 'a')

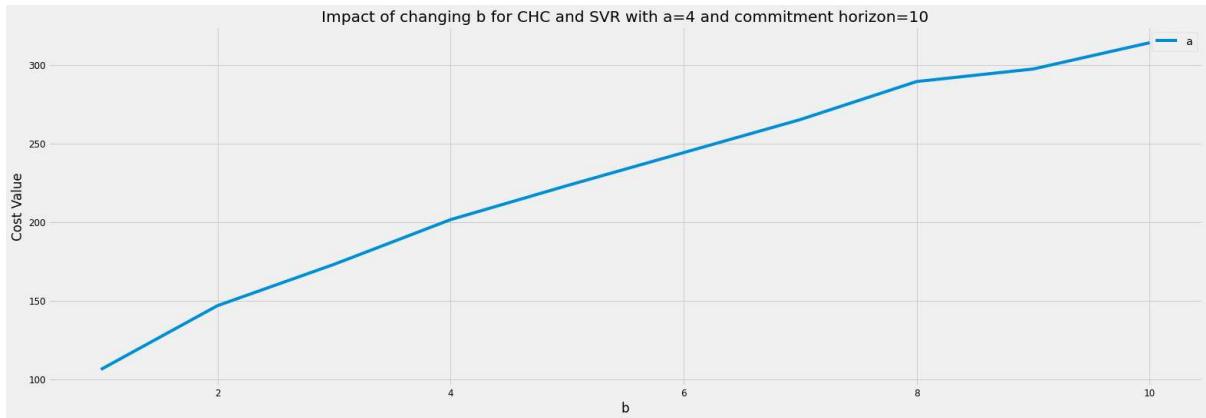


The above graph shows the impact of changing values for a for combination of CHC and SVR with commitment horizon of 10. The cost goes on increasing with increase in a with some exceptions.

```
In [0]: costs = []
windows = []
a = 4
for i in range(1,11):
    windows.append(i)
    costs.append(chc_vary_and_b(y_svr['prediction'].to_numpy(), 10, 3, 'SVR',
a, i))
```

```
In [0]: fig = plt.figure(figsize=(24, 8))
plt.title("Impact of changing b for CHC and SVR with a=4 and commitment horizon=10")
plt.plot(windows, costs, label = "a")
plt.legend()
plt.ylabel("Cost Value")
plt.xlabel("b")
```

Out[0]: Text(0.5, 0, 'b')



The above graph shows the impact of changing values for b for combination of CHC and SVR with commitment horizon of 10. The cost goes on increasing with increase in b.

## Algorithm Selection

In order to select the best algorithm out of multiple algorithms for online convex optimization, there should be constant performance criteria for all of them. I decided to use dynamic regret factor as the evaluation criteria. For each of the algorithms, we minimize the objective function.

```
In [0]: def online_gradient_descent_fixed_window(y, steps):
    n = 4
    x = [0.0] * (n + 1)
    x[1] = 0
    p = 0.4/2
    a = 4/2
    b = 4/2
    for t in range(1, n):
        if (y[t] > x[t]):
            if (x[t] > x[t - 1]):
                slope = p - a + b;
            else:
                slope = p - a - b;
        else:
            if (x[t] > x[t - 1]):
                slope = p + b;
            else:
                slope = p - b;
        x[t + 1] = x[t] - steps * slope
    return x
```

```
In [0]: def rhc_fixed_window(y, predictionHorizon, prediction_algo):
    T = 4;
    p = 0.4/2;
    a = 4/2;
    b = 4/2;
    optValues = np.zeros(T);
    for horizonStart in range(0, T):
        horizonEnd = horizonStart + predictionHorizon
        windowY = y[horizonStart: horizonEnd]

        obj = 0;
        x = cp.Variable(predictionHorizon)

        for i in range(0, predictionHorizon):
            obj += p * x[i] + a * cp.maximum(0, windowY[i] - x[i])
            if i == 0:
                obj += b * cp.abs(x[i]); #because x(0) is 0
            else:
                obj += b * cp.abs(x[i] - x[i - 1])

        objective = cp.Minimize(obj)
        problem = cp.Problem(objective)
        result = problem.solve()

        optValues[horizonStart] = x.value[0];

        obj = 0;
        for i in range(0, T):
            obj += p * optValues[i] + a * max(0, y[i] - optValues[i])
            if i == 0:
                obj += b * abs(optValues[i]); #because x(0) is 0
            else:
                obj += b * abs(optValues[i] - optValues[i - 1])

    return obj
```

```
In [0]: def chc_fixed_window(y, predictionHorizon, commitmentHorizon, prediction_algo):
    T = 4;
    p = 0.4/2;
    a = 4/2;
    b = 4/2;
    optValues = np.zeros(T + 20);
    for horizonStart in range(0, T):
        horizonEnd = horizonStart + predictionHorizon
        windowY = y[horizonStart: horizonEnd]

        obj = 0;
        x = cp.Variable(predictionHorizon)

        for i in range(0, predictionHorizon):
            obj += p * x[i] + a * cp.maximum(0, windowY[i] - x[i])
            if i == 0:
                obj += b * cp.abs(x[i]); #because x(0) is 0
            else:
                obj += b * cp.abs(x[i] - x[i - 1])

        objective = cp.Minimize(obj)
        problem = cp.Problem(objective)
        result = problem.solve()

        for i in range(0, commitmentHorizon):
            optValues[horizonStart + i] += x.value[i];

    optValues = optValues/commitmentHorizon

    obj = 0;
    for i in range(0, T):
        obj += p * optValues[i] + a * max(0, y[i] - optValues[i])
        if i == 0:
            obj += b * abs(optValues[i]); #because x(0) is 0
        else:
            obj += b * abs(optValues[i] - optValues[i - 1])

    return obj
```

```
In [0]: def cost_fixed_window(x, y):
    cost = 0
    p = 0.4/2
    a = 4/2
    b = 4/2
    for i in range(1, 5):
        cost += p * x[i] + a * max(0, y[i] - x[i] + b * abs(x[i] - x[i - 1]))

    return cost
```

```
In [0]: def offline_dynamic_provision_fixed_window(y):
    p = 0.4/2
    a = 4/2
    b = 4/2
    x = cp.Variable(4)
    cost = 0

    for i in range(1,4):
        cost += p*x[i] + a*cp.maximum(0, y[i-1] - x[i]) + b*cp.abs(x[i]-x[i-1])

    objective = cp.Minimize(cost)
    constraints = [x[0] == 0, x[1:] >= 0]
    problem = cp.Problem(objective, constraints)
    result = problem.solve()
    opt = np.array(x.value)
    opt = np.insert(opt, 0, 0., axis=0)

    return opt
```

## Deterministic Approach

The algorithm followed in the Deterministic approach is as follows:

- 1)Select a fixed window size(eg. 4) and step size(eg. 4)
- 2)Iterate over entire time horizon ( $T=672$ ) with step size = window size.
- 3)In each window, evaluate objective function value for each of the algorithms.
- 4)The algorithm which give the cost value closest to the cost value given by offline dynamic algorithm wins that round/window.
- 5)An account of the number of times each algorithm won is maintained.
- 6)At the end, the algorithm with highest number of wins is declared the winner.

In [0]: `import operator`

```
def deterministic():
    win_ogd = 0;
    win_rhc = 0;
    win_chc = 0;
    obj_diff_dict = {}
    y = y_svr['prediction'].to_numpy()
    for i in range(1, 673):
        obj_ogd = cost_fixed_window(online_gradient_descent_fixed_window(y, 0.017
), y[i:])
        obj_rhc = rhc_fixed_window(y[i:], 3, 'SVR')
        obj_chc = chc_fixed_window(y[i:], 10, 9, 'SVR')
        obj_offline_dynamic = cost_fixed_window(offline_dynamic_provision_fixed_wi
ndow(y[i:]), y[i:])
        obj_diff_dict['OGD'] = abs(obj_offline_dynamic - obj_ogd)
        obj_diff_dict['RHC'] = abs(obj_offline_dynamic - obj_rhc)
        obj_diff_dict['CHC'] = abs(obj_offline_dynamic - obj_chc)
        optimal_algo = min(obj_diff_dict.items(), key=operator.itemgetter(1))[0]

    if optimal_algo == 'OGD':
        win_ogd += 1
    elif optimal_algo == 'RHC':
        win_rhc += 1
    else:
        win_chc += 1

    obj_win_dict = {}
    obj_win_dict['OGD'] = win_ogd
    obj_win_dict['RHC'] = win_rhc
    obj_win_dict['CHC'] = win_chc
    print(obj_win_dict)
    return max(obj_win_dict.items(), key=operator.itemgetter(1))[0]
```

In [0]: `print('The winner for deterministic approach is: ',deterministic())`

```
{'OGD': 196, 'RHC': 30, 'CHC': 446}
The winner for deterministic approach is: CHC
```

As seen above, out of 672 windows, CHC wins the most number of times(446) and RHC the least(30). Thus, CHC is the winner.

## Randomized Approach

The algorithm followed in the Randomized approach is as follows:

1)Assign equal weights to each algorithm at the start 2) Pick a window size say 4 3)Iterate over entire time horizon (T=672). 4)In each window, evaluate objective function value for each of the algorithms. 5)Increase the weight of algorithm with least difference with objective function value of offline dynamic by 0.0005 and decrease the weight of the algorithm with max difference by 0.0005. 6)At the end, we get final weights of all algorithms which can be used to make a decision.

```
In [0]: import math

def randomized():
    wt_ogd = 1/3;
    wt_rhc = 1/3;
    wt_chc = 1/3;
    obj_diff_dict = {}
    y = y_svr['prediction'].to_numpy()

    for i in range(1, 673):
        obj_ogd = cost_fixed_window(online_gradient_descent_fixed_window(y, 0.017
), y[i:])
        obj_rhc = rhc_fixed_window(y[i:], 3, 'SVR')
        obj_chc = chc_fixed_window(y[i:], 10, 9, 'SVR')
        obj_offline_dynamic = cost_fixed_window(offline_dynamic_provision_fixed_wi
ndow(y[i:]), y[i:])
        obj_diff_dict['OGD'] = abs(obj_offline_dynamic - obj_ogd)
        obj_diff_dict['RHC'] = abs(obj_offline_dynamic - obj_rhc)
        obj_diff_dict['CHC'] = abs(obj_offline_dynamic - obj_chc)
        most_optimal_algo = min(obj_diff_dict.items(), key=operator.itemgetter(1))
[0]
        least_optimal_algo = max(obj_diff_dict.items(), key=operator.itemgetter(1
))[0]

        if i > 1:
            if most_optimal_algo == 'OGD':
                wt_ogd += 0.0005
            elif most_optimal_algo == 'RHC':
                wt_rhc += 0.0005
            else:
                wt_chc += 0.0005

            if least_optimal_algo == 'OGD':
                wt_ogd -= 0.0005
            elif least_optimal_algo == 'RHC':
                wt_rhc -= 0.0005
            else:
                wt_chc -= 0.0005

        last_winner = most_optimal_algo
        last_loser = least_optimal_algo

        obj_wt_dict = {}
        obj_wt_dict['OGD'] = wt_ogd
        obj_wt_dict['RHC'] = wt_rhc
        obj_wt_dict['CHC'] = wt_chc
        print('The final weights for randomized approach is: ', obj_wt_dict)
```

```
In [0]: randomized()
```

The final weights for randomized approach is: {'OGD': 0.4058333333333334, 'R  
HC': 0.0568333333333307, 'CHC': 0.5373333333333293}

The final weights obtained by Randomized approach is 0.41 for OGD, 0.05 for RHC and 0.54 for CHC.

```
In [0]: def compare():
    y = y_svr['prediction'].to_numpy()
    obj_deterministic = 0
    obj_offline_dynamic = 0
    obj_randomized = 0

    for i in range(1, 673):
        obj_deterministic += chc_fixed_window(y[i:], 10, 9, 'SVR')
        obj_offline_dynamic += cost_fixed_window(offline_dynamic_provision_fixed_window(y[i:]), y[i:])
        obj_randomized += 0.54*chc_fixed_window(y[i:], 10, 9, 'SVR')+0.05*rhc_fixed_window(y[i:], 3, 'SVR')+0.41*cost_fixed_window(online_gradient_descent_fixed_window(y, 0.017), y[i:])

    obj_dict = {}
    obj_dict['Deterministic'] = obj_deterministic
    obj_dict['Offline Dynamic'] = obj_offline_dynamic
    obj_dict['Randomized'] = obj_randomized
    print(obj_dict)
    return max(obj_dict.items(), key=operator.itemgetter(1))[0]
```

```
In [0]: compare()
```

```
{'Deterministic': 2257.927113529734, 'Offline Dynamic': 2385.4609087574013,
'Randomized': 2381.5017183527807}
```

```
Out[0]: 'Offline Dynamic'
```

The above part compares the objective function value for Offline Dynamic, Deterministic and Randomized approaches. If we consider Offline Dynamic as the baseline, then Randomized algorithm is the closest to it meaning it performs better.