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/HomeC-meter1_2015.csv"
df_meter = pd.read_csv(path, float_precision='round_trip')
```

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

Out[0]:
```
(39764, 19)
```

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'] == '2015-01-11 00:30:00']
```

Out[0]:

| | Date & Time | use [kW] | gen [kW] | House overall [kW] | Dishwasher [kW] | Furnace 1 [kW] | Furnace 2 [kW] | Home office [kW] | Fridge [kW] | |
|---|---|---|---|---|---|---|---|---|---|---|
| **481** | 2015-01-11 00:30:00 | 1.273613 | 0.0 | 1.273613 | 0.000063 | 0.253386 | 0.274793 | 0.018847 | 0.006168 | 0. |

In [6]:
```
df_meter = df_meter['use [kW]']
df_meter=df_meter[481:(481+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 + " provisioni
ng")
  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)")
```

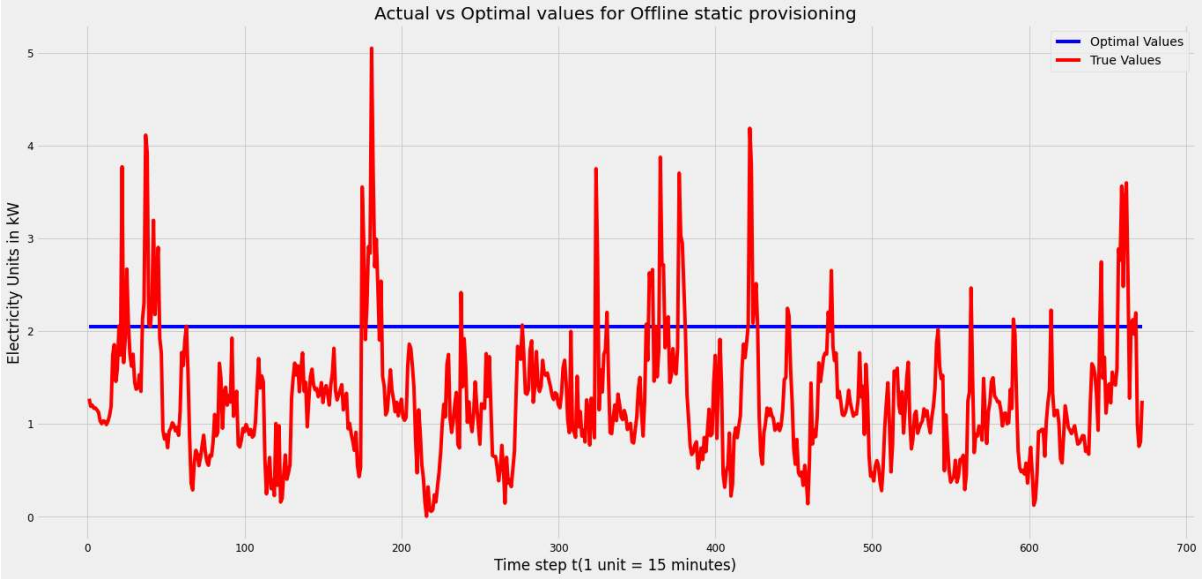
In [0]:
```
#Store algorithms, optimal values and decision values for objective function
opt_dict = {}
decision_dict = {}
```

# 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 374.17292733225565
The optimal x is
[2.05492111]
```



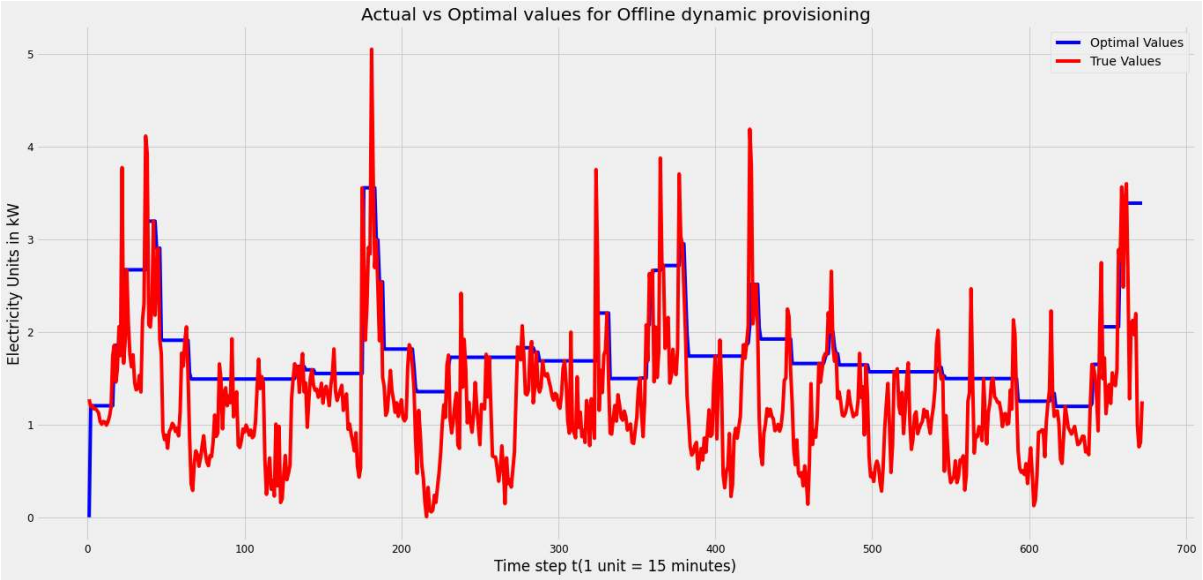

Thus as shown above, the static offline optimization gives the value of 2.054 kW and the objective function cost as 374.17. Since this is static optimization, we get a straight line as optimal on graph as above.

# 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 [17]:
```
offline_dynamic_provision(df_meter, True)
```

```
The optimal value is 317.8709558554409
First 15 optimal values of x is
[-4.92599939e-15  1.20364000e+00  1.20364000e+00  1.20364000e+00
  1.20364000e+00  1.20364000e+00  1.20364000e+00  1.20364000e+00
  1.20364000e+00  1.20364000e+00  1.20364000e+00  1.20364000e+00
  1.20364000e+00  1.20364000e+00  1.20364000e+00]
```



Thus as shown above, the static offline optimization different values of x for each time step and the objective function cost as 317.87 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)
```

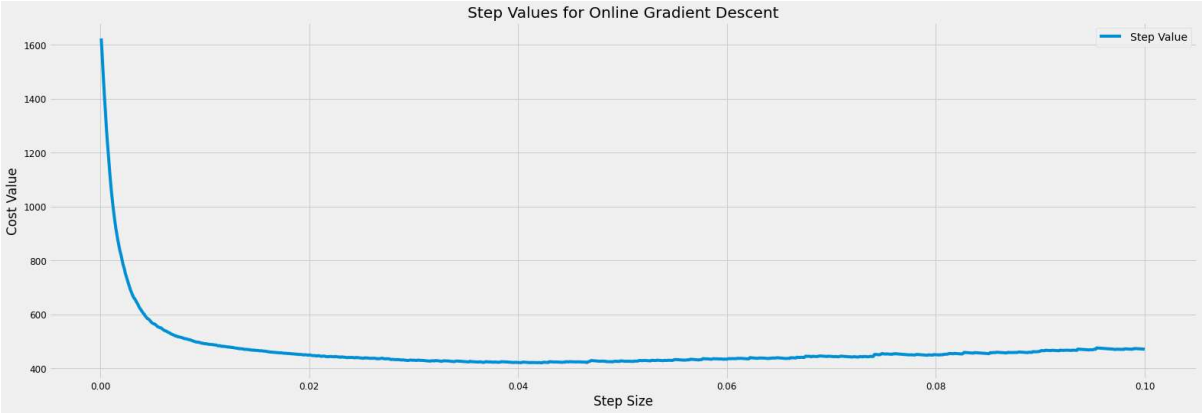
In [0]:
```
cost('Online Gradient Descent', x, df_meter, True)
```

```
The objective value for Online Gradient Descent is 715.1079033380046
```

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 [0]:
```
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[0]:
```
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 0.0423. After this, the objective function value again increases gradually.

In [0]:
```
steps[costs.index(min(costs))]
```

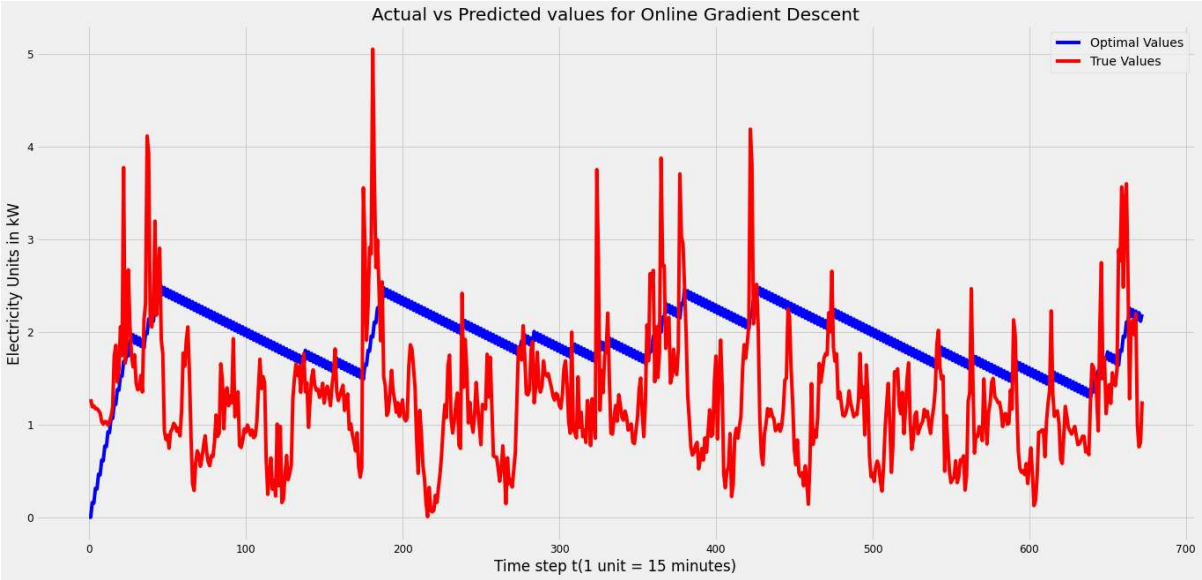
Out[0]:
```
0.042300000000000185
```

In [22]:
```
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:  0.042300000000000185

Optimal cost is:

The objective value for Online Gradient Descent is 420.21615355799804
None
```

In [0]:
```
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)")
```

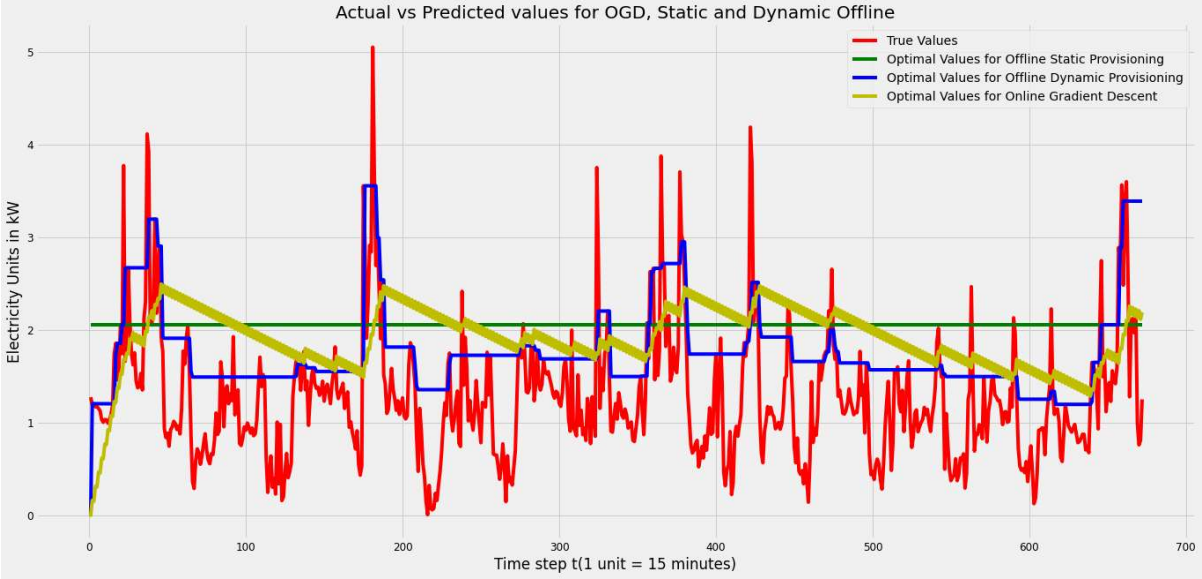
Out[0]:
```
Text(0.5, 0, 'Time step t(1 unit = 15 minutes)')
```




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 420.216 is obtained at step size 0.0423.

In [0]:
```
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 Values 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[0]:
```
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 agains 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
```

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

# RHC and CHC with Gradient Boost and XGBoost for fixed value of a and b

Gradient Boost and XGBoost were found out to be the best algorithms for prediction for House C 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
```

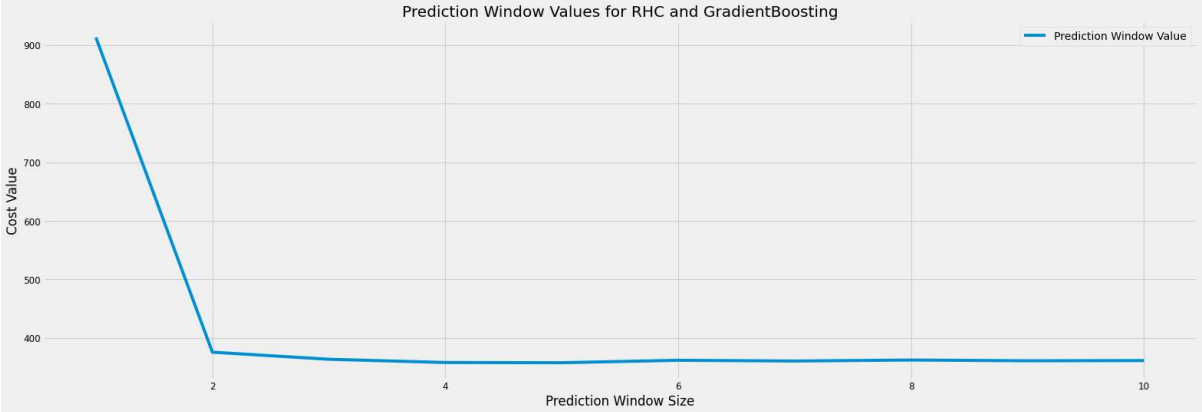
In [0]:
```
RHC(y_extratrees['prediction'].to_numpy(), 5, 'GradientBoosting')
```

Out[0]:
```
357.7614441520886
```

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

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

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




The graph above shows the variation of objective function value for combination of RHC and Gradient Boost algorithm w.r.t the prediction window size. Initially, the cost is higher but it falls drastically later and becomes the least (357.761) at window size of 5. After this, the objective function value almost remains constant.

In [27]:
```
x = RHC(y_extratrees['prediction'].to_numpy(), windows[costs.index(min(costs
))], 'GradientBoosting')
print("Optimal cost for RHC and GradientBoosting found at window size: ", wind
ows[costs.index(min(costs))])
print("\nOptimal cost is: ", x)
```

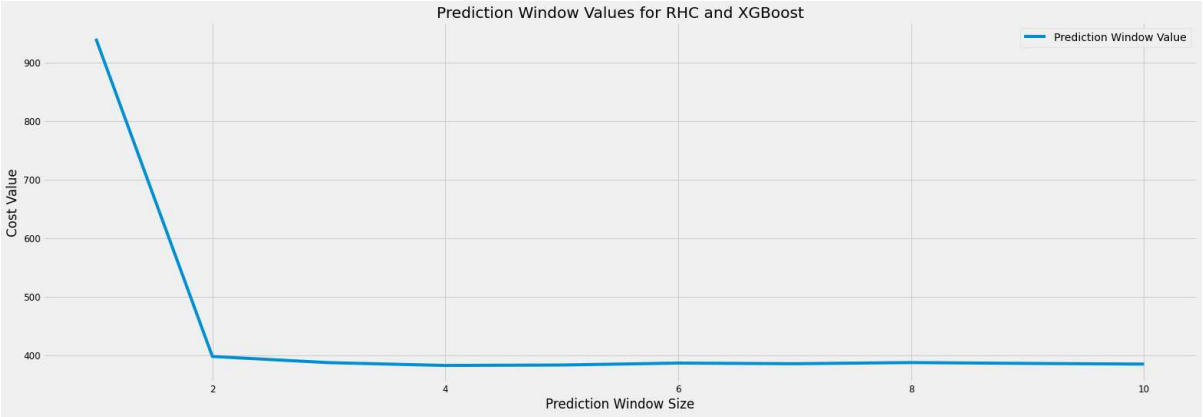
```
Optimal cost for RHC and GradientBoosting found at window size:  5

Optimal cost is:  357.7614441520886
```

In [0]:
```
path = "/content/drive/My Drive/AMS Smart Energy/data/prediction_results_C_xgb
oost.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, 'XGBoost'))
```

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

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




The graph above shows the variation of objective function value for combination of RHC and XGBoost algorithm w.r.t the prediction window size. Initially, the cost is higher but it falls drastically later and becomes the least (382.155) at window size of 4. After this, the objective function value almost remains constant.

In [30]:
```
x = RHC(y_svr['prediction'].to_numpy(), windows[costs.index(min(costs))], 'XGB
oost')
print("Optimal cost for RHC and XGBoost found at window size: ", windows[costs
.index(min(costs))])
print("\nOptimal cost is: ", x)
```

```
Optimal cost for RHC and XGBoost found at window size:  4

Optimal cost is:  382.1559258677289
```

# CHC with Gradient Boost and XGBoost

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, 'GradientBoost')
```

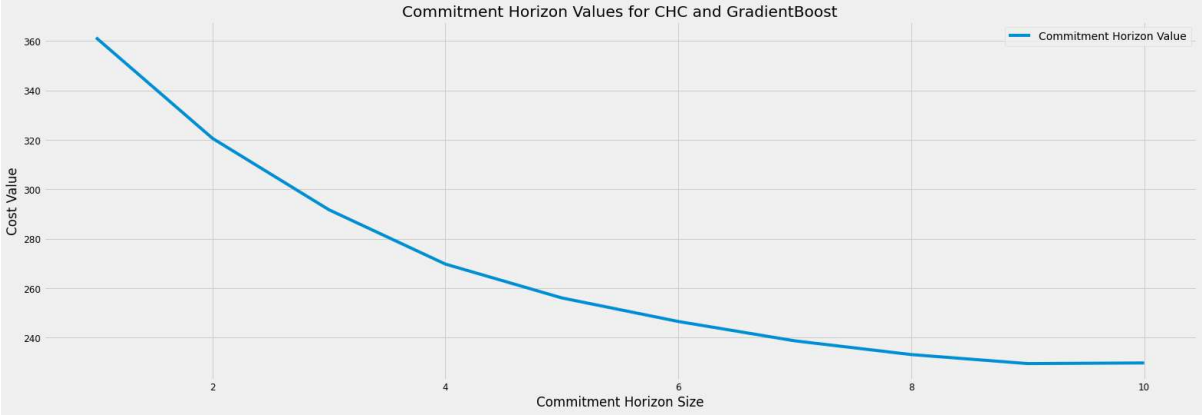
Out[0]:
```
361.40805265773105
```

# CHC with Gradient Boost

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

In [0]:
```
fig = plt.figure(figsize=(24, 8))
plt.title("Commitment Horizon Values for CHC and GradientBoost")
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 Gradient Boost algorithm w.r.t the commitment horizon size. Initially, the cost is higher but it falls gradually later and becomes the least (229.53) at commitment horizon size of 9. After this, the objective function value almost remains constant.

In [33]:
```
x = CHC(y_extratrees['prediction'].to_numpy(), 10, windows[costs.index(min(cos
ts))], 'GradientBoost')
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:  9

Optimal cost is:  229.5392016418284
```

# CHC with XGBoost

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

In [0]:
```
fig = plt.figure(figsize=(24, 8))
plt.title("Commitment Horizon Values for CHC and XGBoost")
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 XGBoost algorithm w.r.t the commitment horizon size. Initially, the cost is higher but it falls gradually later and becomes the least 240.642) at commitment horizon size of 10. After this, the objective function value almost remains constant.

In [35]:
```
x = CHC(y_svr['prediction'].to_numpy(), 10, windows[costs.index(min(costs))],
'XGBoost')
print("Optimal cost for CHC and XGBoost found at commitment horizon size: ", w
indows[costs.index(min(costs))])
print("\nOptimal cost is: ", x)
```

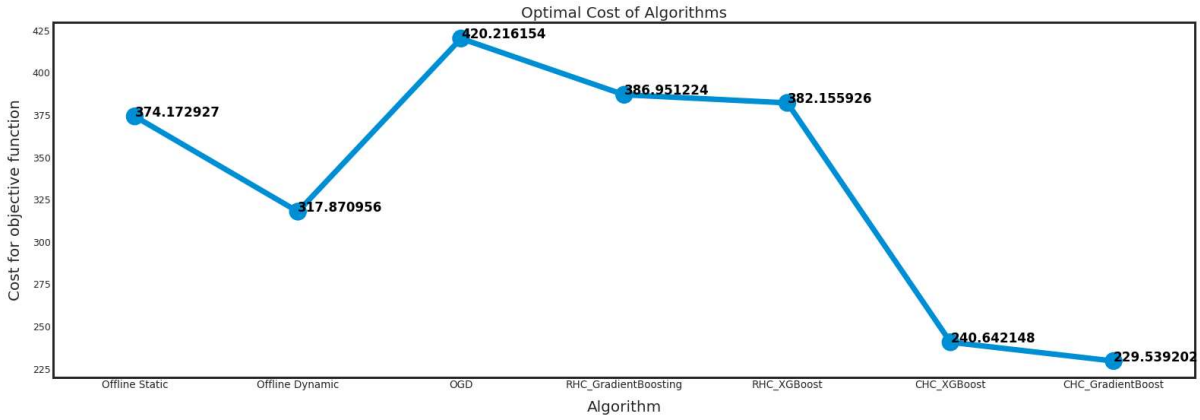
```
Optimal cost for CHC and XGBoost found at commitment horizon size:  10

Optimal cost is:  240.64214801084634
```

In [0]:
```
opt_dict
```

Out[0]:
```
{'CHC_GradientBoost': 229.5392016418284,
 'CHC_XGBoost': 240.64214801084634,
 'OGD': 420.21615355799804,
 'Offline Dynamic': 317.8709558554409,
 'Offline Static': 374.17292733225565,
 'RHC_GradientBoosting': 386.9512236088673,
 'RHC_XGBoost': 382.1559258677289}
```

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

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

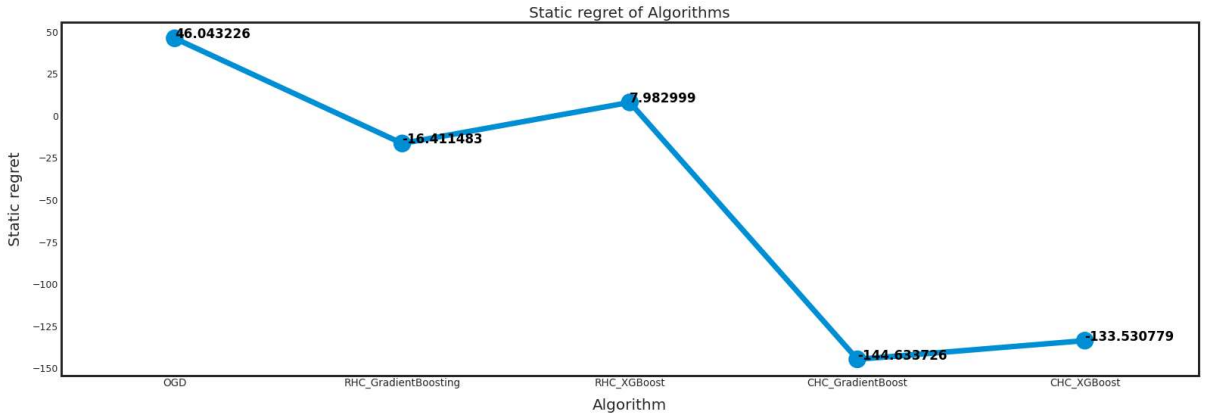



The above graph compares the objective function value for Offline Static and Dynamic, OGD, RHC with Gradient Boost, RHC with XGBoost, CHC with Gradient Boost, CHC with XGBoost. OGD preforms the worst while CHC with Gradient Boost preforms the best. Lower the values, higher the preformance and better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

In [0]:
```
static_obj = opt_dict.pop('Offline Static')
dynamic_obj = opt_dict.pop('Offline Dynamic')
```

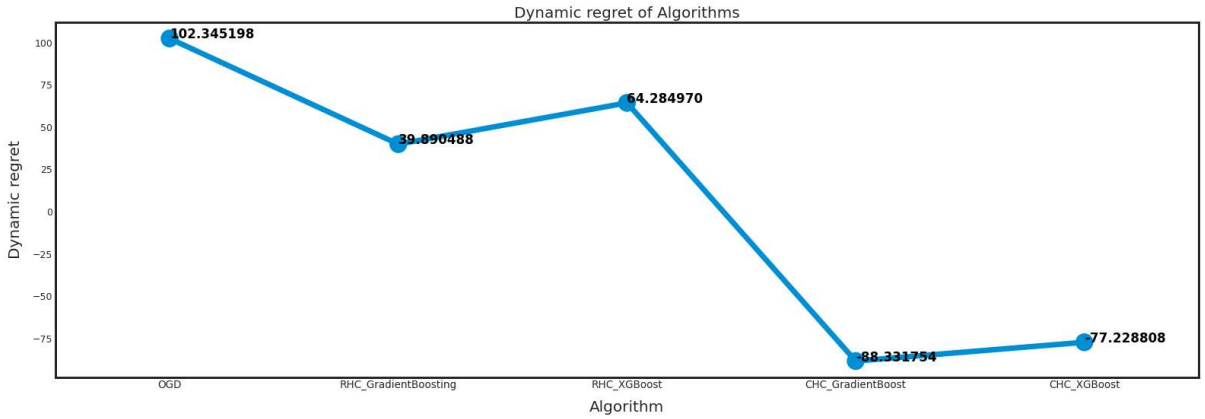
In [0]:
```
static_dict = {}
for key in opt_dict:
  static_dict[key] = opt_dict[key] - static_obj

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

In [43]:
```
sns.set_style("white")
fig = plt.figure(figsize=(24, 8))

ax = sns.pointplot(x=list(static_dict.keys()), y=[score for score in static_di
ct.values()], markers=['o'], linestyles=['-'])
for i, score in enumerate(static_dict.values()):
    ax.text(i, score + 0.002, '{:.6f}'.format(score), horizontalalignment='lef
t', 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 agorithms have a positive static regret while some have negative. CHC with GradientBoost has the most negative static regret meaning it performs better than static optimal.

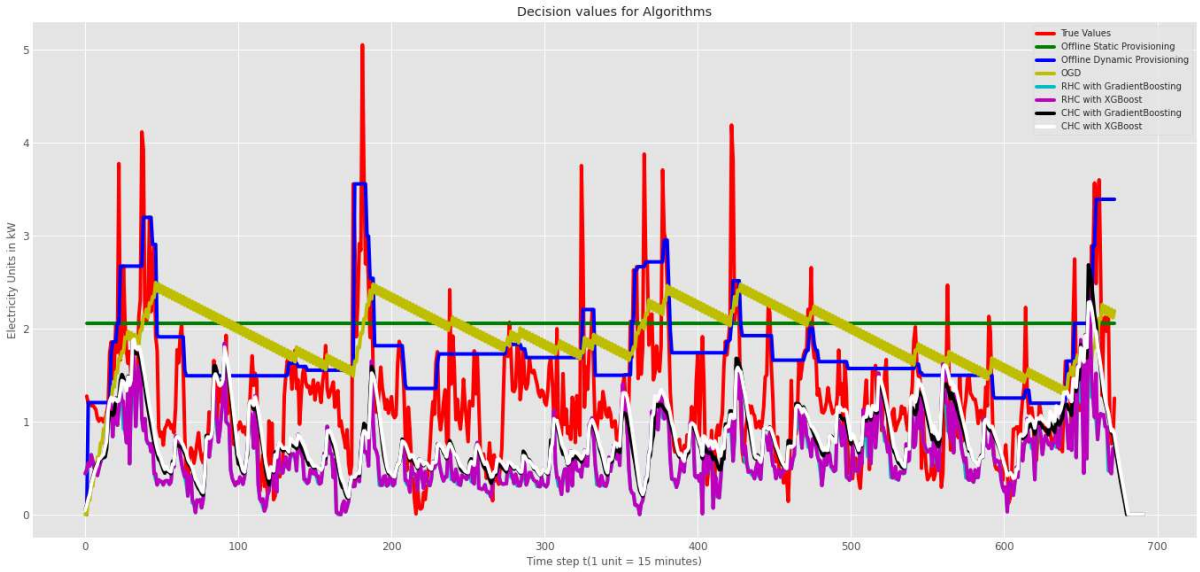
In [44]:
```
sns.set_style("white")
fig = plt.figure(figsize=(24, 8))

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()
```




In [0]:
```
Thus some agorithms have a negative dynamic regret meaning they perform better
than dynamic optimal.
```

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_GradientBoosting'], 'c', label="RHC with GradientBoosting")
plt.plot(decision_dict['RHC_XGBoost'], 'm', label="RHC with XGBoost")
plt.plot(decision_dict['CHC_GradientBoost'], 'k', label="CHC with GradientBoosting")
plt.plot(decision_dict['CHC_XGBoost'], 'w', label="CHC with XGBoost")
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

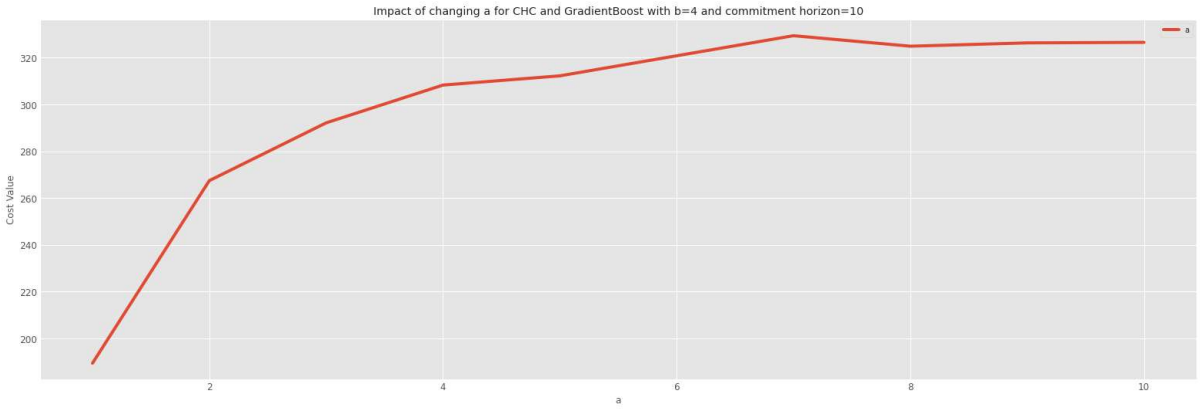
Best combination is CHC with GradientBoost with commitment horizon size of 9 with a=4 and b=4. We keep commitment horizon size as constant and vary first a by keeping b constant and then vice-versa to see the impact of change.

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, 'Gradie
ntBoost', i, b))
```

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

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

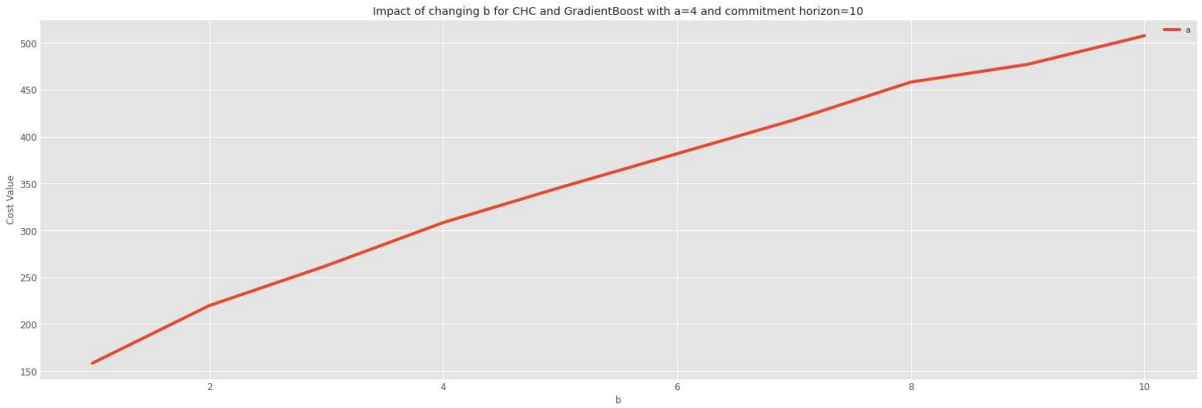



The above graph shows the impact of changing values for a for combination of CHC and Gradient Boost with commitment horizon of 10. The cost goes on increasing with increase in a with some exceptions.

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, 'Gradie
ntBoost', a, i))
```

In [0]:
```
fig = plt.figure(figsize=(24, 8))
plt.title("Impact of changing b for CHC and GradientBoost with a=4 and commitm
ent 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 Gradient Boost 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 givenby 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, 'GradientBoost')
    obj_chc = chc_fixed_window(y[i:], 10, 9, 'GradientBoost')
    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': 247, 'RHC': 23, 'CHC': 402}
The winner for deterministic approach is:  RHC
```

As seen above, out of 672 windows, CHC wins the most number of times(402) and RHC the least(23). 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': 0.4368333333333334, 'R
HC': 0.04583333333333306, 'CHC': 0.5173333333333315}
```

The final weights obtained by Randomized approach is 0.44 for OGD, 0.04 for RHC and 0.52 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, 'GradientBoost')
    obj_offline_dynamic += cost_fixed_window(offline_dynamic_provision_fixed_w
indow(y[i:]), y[i:])
    obj_randomized += 0.52*chc_fixed_window(y[i:], 10, 9, 'GradientBoost')+0.0
4*rhc_fixed_window(y[i:], 3, 'GradientBoost')+0.44*cost_fixed_window(online_gr
adient_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': 3248.884472960786, 'Offline Dynamic': 3492.428728049965, 'R
andomized': 3458.013061995692}
```

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.