Regression - Gradient Descent Overview

- Linear Model. Estimated Target = $w_0 + w_1x_1 + w_2x_2 + w_3x_3 + ... + w_nx_n$
  where, w is the weight and x is the feature
- Predicted Value: Numeric
- Algorithm Used: Linear Regression. Objective is to find the weights w
- Optimization: Gradient Descent. Seeks to minimize loss/cost so that predicted value is as close to actual as possible
- Cost/Loss Calculation: Squared loss function

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Input Feature: x

Target: 5*x + 8 + some noise

In [2]:
```
# True Function
def straight_line(x):
    return 5*x + 8
```

In [3]:
```
# Estimate predicted value for a given weight
def predicted_at_weight(weight0, weight1, x):
    return weight1*x + weight0
```

In [4]:
```
np.random.seed(5)

samples = 150
x = pd.Series(np.arange(0,150))
y = x.map(straight_line) + np.random.randn(samples)*10
```

In [5]:
```
df = pd.DataFrame({'x':x,'y':y})
```

In [6]:
```
# One Feature example
# Training Set - Contains several examples of feature 'x' and corresponding correct answer 'y'
```

```
# Objective is to find out the form y = w0 + w1*x1
df.head()
```

Out[6]:

	x	y
0	0	12.412275
1	1	9.691298
2	2	42.307712
3	3	20.479079
4	4	29.096098

In [7]:

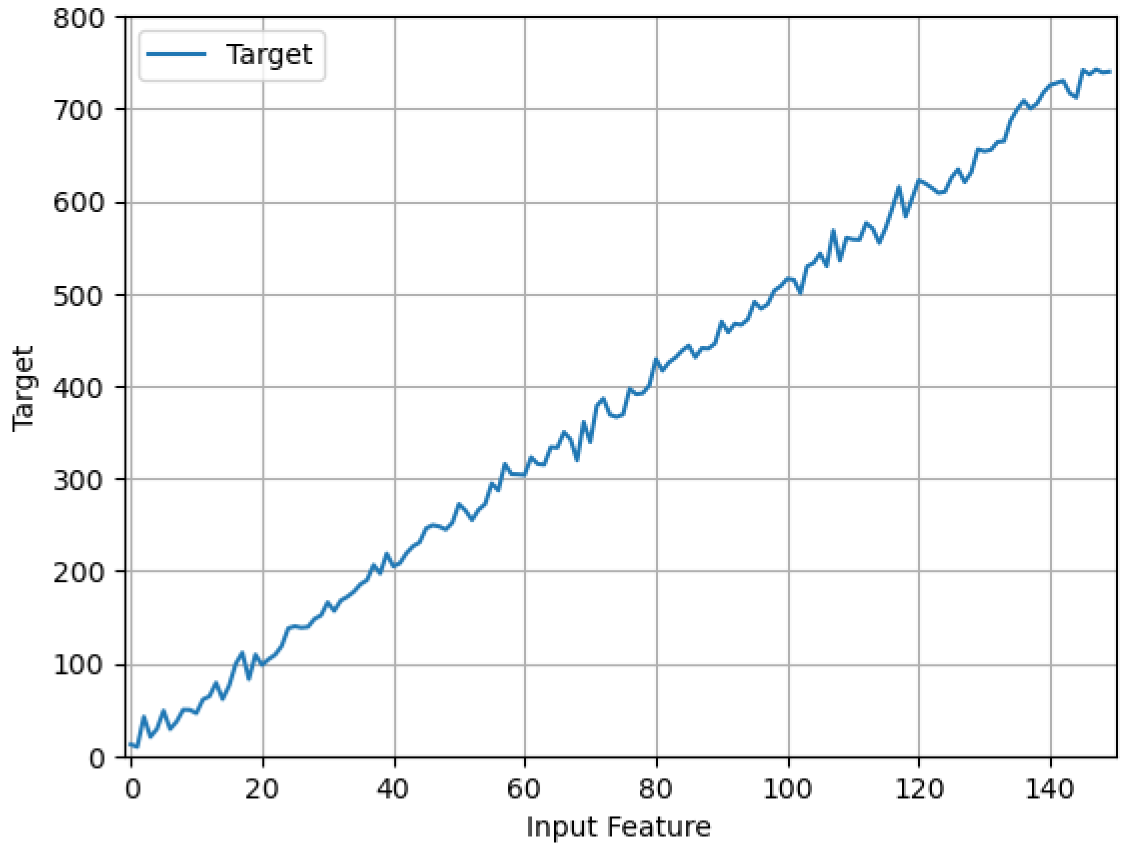
```
df.tail()
```

Out[7]:

	x	y
145	145	741.771528
146	146	737.061676
147	147	742.443290
148	148	739.105793
149	149	739.990485

In [8]:

```
plt.plot(df.x,df.y,label='Target')
plt.grid(True)
plt.xlim(-1,150)
plt.ylim(0,800)
plt.xlabel('Input Feature')
plt.ylabel('Target')
plt.legend()
plt.show()
```



In [9]:
```
# Linear Regression
import numpy as np
from sklearn.linear_model import LinearRegression
```

In [10]:
```
reg = LinearRegression()
```

In [11]:
```
reg.fit(df[['x']],df['y'])
```

Out[11]:
```
▾ LinearRegression
LinearRegression()
```

In [12]:
```
print('Coefficients:',reg.coef_,'Intercept:',reg.intercept_)
```

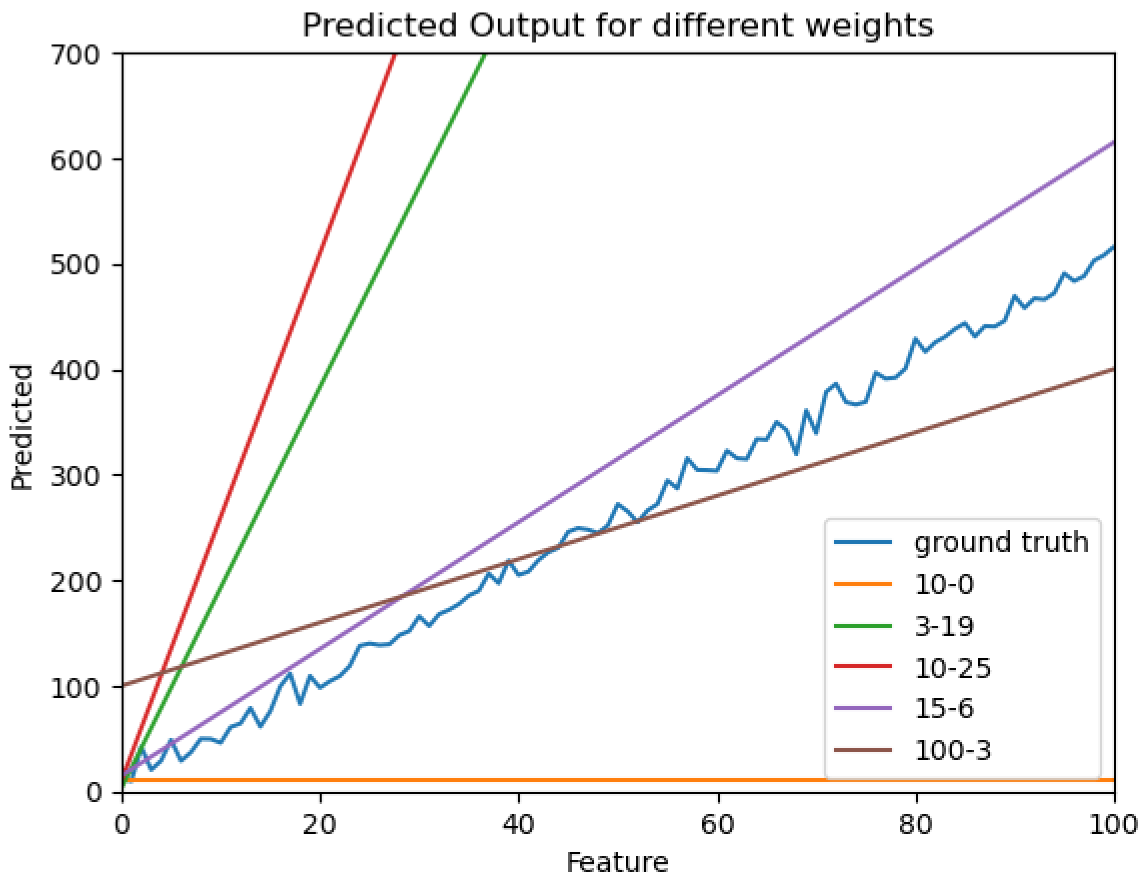
```
Coefficients: [4.99342639] Intercept: 9.095553826738524
```

Predict Y for different weights

In [13]:
```
# True function weight is  w1 = 5 and w0 = 8.  5*x + 8
w0 = [10,3,10,15,100]
w1 = [0,19,25,6,3]
```

In [14]:
```
y_predicted = {}
for i in range(len(w1)):
    y_predicted['{0}-{1}'.format(w0[i],w1[i])] = predicted_at_weight(w0[i],w1[i], x)
```

In [15]:
```
plt.plot(x,y,label='ground truth')

for w in y_predicted.keys():
    plt.plot(x,y_predicted[w],label=w)

plt.xlim(0,100)
plt.ylim(0,700)
plt.xlabel('Feature')
plt.ylabel('Predicted')
plt.title('Predicted Output for different weights')
plt.legend()
plt.show()
```



Squared Loss

In [16]:
```
for w in y_predicted.keys():
    squared_loss = (y-y_predicted[w])**2
    print('Weight:{0}\tLoss: {1:10.2f}'.format(w, squared_loss.mean()))
```

```
Weight:10-0     Loss:  184575.00
Weight:3-19     Loss: 1444121.26
Weight:10-25    Loss: 2974822.07
Weight:15-6     Loss:    8549.24
Weight:100-3    Loss:   10874.60
```

Plot Loss at different weights for x

In [17]:
```
# For a set of weights, let's find out loss or cost
# True Function: 5x+8
# Linear Regression algorithm iteratively tries to find the correct weight for x.
# Let's test how the lost changes at different weights for x.

# In this example, let's see how the "loss" changes for different weights
#DWB# that is, for different values of w1 (with w0 held at the correct 8)
weight = pd.Series(np.linspace(3,7,100))
```

In [18]:
```
print(weight[:5])
print()
print(weight[-5:])
```

```
0    3.000000
1    3.040404
2    3.080808
3    3.121212
4    3.161616
dtype: float64

95    6.838384
96    6.878788
97    6.919192
98    6.959596
99    7.000000
dtype: float64
```

Compute Loss using Squared Loss Function

loss = average((true - predicted)^2)

In [19]:
```
# Cost/Loss Calculation: Squared loss function...a measure of how far is predicted value from actual
# Steps :

#  For every weight for feature x, predict y
#  Now, find out loss by = average ((actual - predicted)**2)

#DWB# -v-Figuring out what's going on, here.
do_see_the_guts = True
this_count = 0
this_count_max = 5
```

```
#DWB# -v- Remember:
# # Estimate predicted value for a given weight
# def predicted_at_weight(weight0, weight1, x):
#     return weight1*x + weight0

loss_at_wt = []
for w1 in weight:
    y_predicted = predicted_at_weight(8,w1,x)

    if do_see_the_guts:
        this_count += 1
        if this_count == this_count_max:
            break
        ##endof:  if this_count == this_count_max

        print()
        print(f"  this_count:\n{this_count}")
        print(f"  this_count_max:\n{this_count_max}")

        print()
        print(f"  w1:\n{w1}")
        print(f"  type(w1):\n{type(w1)}")
        print()
        print(f"  x:\n{x}")
        print(f"  type(x):\n{type(x)}")
        print()
        print(f"  y_predicted = predicted_at_weight(8,{w1},x) =>")
        print(f"  y_predicted =\n{y_predicted}")
        print()
        print("  Yay 1!")
        input("  Press enter to continue.")
    ##endof:  if do_see_the_guts

    squared_error = (y - y_predicted)**2

    if do_see_the_guts:
        print()
        print(f"  y:\n{y}")
        print(f"  type(y):\n{type(y)}")
        print()
        print(f"  y_predicted:\n{y_predicted}")
```

```
        print(f"  type(y_predicted):\n{type(y_predicted)}")
        print()
        print(f"  squared_error:\n{squared_error}")
        print(f"  type(squared_error:\n{type(squared_error)})")
        print()
        print("  Yay 2!")
        input("  Press enter to continue.")
    ##endof:  if do_see_the_guts

    # Average Squared Error at weight w1
    loss_at_wt.append(squared_error.mean())

    if do_see_the_guts:
        print()
        print(f"  squared_error.mean():\n{squared_error.mean()}")
        print(f"  type(squared_error.mean()\n{type(squared_error.mean())})")
        print()
        print(f"  loss_at_wt:\n{loss_at_wt}")
        print(f"  type(loss_at_wt):\n{type(loss_at_wt)}")
        print()
        print("  Yay 3!")
        input("  Press enter to continue.")
    ##endof:  if do_see_the_guts
```

```
  this_count:
1
  this_count_max:
5

  w1:
3.0
  type(w1):
<class 'float'>

  x:
0        0
1        1
2        2
3        3
4        4
      ...
145    145
146    146
147    147
148    148
149    149
Length: 150, dtype: int64
  type(x):
<class 'pandas.core.series.Series'>

  y_predicted = predicted_at_weight(8,3.0,x) =>
  y_predicted =
0        8.0
1       11.0
2       14.0
3       17.0
4       20.0
       ...
145    443.0
146    446.0
147    449.0
148    452.0
149    455.0
Length: 150, dtype: float64

  Yay 1!
```

```
  Press enter to continue.

  y:
0       12.412275
1        9.691298
2       42.307712
3       20.479079
4       29.096098
          ...
145    741.771528
146    737.061676
147    742.443290
148    739.105793
149    739.990485
Length: 150, dtype: float64
  type(y):
<class 'pandas.core.series.Series'>

  y_predicted:
0        8.0
1       11.0
2       14.0
3       17.0
4       20.0
       ...
145    443.0
146    446.0
147    449.0
148    452.0
149    455.0
Length: 150, dtype: float64
  type(y_predicted):
<class 'pandas.core.series.Series'>

  squared_error:
0         19.468170
1          1.712700
2        801.326551
3         12.103989
4         82.739006
            ...
145    89264.426016
```

```
146    84716.898953
147    86108.964242
148    82429.736178
149    81219.576816
Length: 150, dtype: float64
  type(squared_error:
<class 'pandas.core.series.Series'>)

  Yay 2!
  Press enter to continue.

  squared_error.mean():
29938.07548415073
  type(squared_error.mean()
<class 'numpy.float64'>)

  loss_at_wt:
[29938.07548415073]
  type(loss_at_wt):
<class 'list'>

  Yay 3!
  Press enter to continue.

  this_count:
2
  this_count_max:
5

  w1:
3.04040404040404
  type(w1):
<class 'float'>

  x:
0        0
1        1
2        2
3        3
4        4
      ...
145    145
```

```
146    146
147    147
148    148
149    149
Length: 150, dtype: int64
  type(x):
<class 'pandas.core.series.Series'>

  y_predicted = predicted_at_weight(8,3.04040404040404,x) =>
  y_predicted =
0        8.000000
1       11.040404
2       14.080808
3       17.121212
4       20.161616
          ...
145    448.858586
146    451.898990
147    454.939394
148    457.979798
149    461.020202
Length: 150, dtype: float64

  Yay 1!
  Press enter to continue.

  y:
0       12.412275
1        9.691298
2       42.307712
3       20.479079
4       29.096098
          ...
145    741.771528
146    737.061676
147    742.443290
148    739.105793
149    739.990485
Length: 150, dtype: float64
  type(y):
<class 'pandas.core.series.Series'>
```

```
  y_predicted:
0        8.000000
1       11.040404
2       14.080808
3       17.121212
4       20.161616
          ...
145    448.858586
146    451.898990
147    454.939394
148    457.979798
149    461.020202
Length: 150, dtype: float64
  type(y_predicted):
<class 'pandas.core.series.Series'>

  squared_error:
0         19.468170
1          1.820086
2        796.758098
3         11.275268
4         79.824973
           ...
145    85797.991745
146    81317.757267
147    82658.490051
148    79031.824884
149    77824.419055
Length: 150, dtype: float64
  type(squared_error:
<class 'pandas.core.series.Series'>)

  Yay 2!
  Press enter to continue.

  squared_error.mean():
28747.51883105903
  type(squared_error.mean()
<class 'numpy.float64'>)

  loss_at_wt:
[29938.07548415073, 28747.51883105903]
```

```
  type(loss_at_wt):
<class 'list'>

  Yay 3!
  Press enter to continue.

  this_count:
3
  this_count_max:
5

  w1:
3.080808080808081
  type(w1):
<class 'float'>

  x:
0        0
1        1
2        2
3        3
4        4
      ...
145    145
146    146
147    147
148    148
149    149
Length: 150, dtype: int64
  type(x):
<class 'pandas.core.series.Series'>

  y_predicted = predicted_at_weight(8,3.080808080808081,x) =>
  y_predicted =
0        8.000000
1       11.080808
2       14.161616
3       17.242424
4       20.323232
          ...
145    454.717172
146    457.797980
```

```
147    460.878788
148    463.959596
149    467.040404
Length: 150, dtype: float64

  Yay 1!
  Press enter to continue.

  y:
0       12.412275
1        9.691298
2       42.307712
3       20.479079
4       29.096098
          ...
145    741.771528
146    737.061676
147    742.443290
148    739.105793
149    739.990485
Length: 150, dtype: float64
  type(y):
<class 'pandas.core.series.Series'>

  y_predicted:
0        8.000000
1       11.080808
2       14.161616
3       17.242424
4       20.323232
          ...
145    454.717172
146    457.797980
147    460.878788
148    463.959596
149    467.040404
Length: 150, dtype: float64
  type(y_predicted):
<class 'pandas.core.series.Series'>

  squared_error:
0        19.468170
```

```
1          1.930737
2        792.202704
3         10.475932
4         76.963179
             ...
145    82400.203531
146    77988.211745
147    79278.568659
148    75705.429558
149    74501.746960
Length: 150, dtype: float64
  type(squared_error:
<class 'pandas.core.series.Series'>)

  Yay 2!
  Press enter to continue.

  squared_error.mean():
27581.2051463719
  type(squared_error.mean()
<class 'numpy.float64'>)

  loss_at_wt:
[29938.07548415073, 28747.51883105903, 27581.2051463719]
  type(loss_at_wt):
<class 'list'>

  Yay 3!
  Press enter to continue.

  this_count:
4
  this_count_max:
5

  w1:
3.121212121212121
  type(w1):
<class 'float'>

  x:
0        0
```

```
1        1
2        2
3        3
4        4
      ...
145    145
146    146
147    147
148    148
149    149
Length: 150, dtype: int64
  type(x):
<class 'pandas.core.series.Series'>

  y_predicted = predicted_at_weight(8,3.121212121212121,x) =>
  y_predicted =
0        8.000000
1       11.121212
2       14.242424
3       17.363636
4       20.484848
          ...
145    460.575758
146    463.696970
147    466.818182
148    469.939394
149    473.060606
Length: 150, dtype: float64

  Yay 1!
  Press enter to continue.

  y:
0       12.412275
1        9.691298
2       42.307712
3       20.479079
4       29.096098
          ...
145    741.771528
146    737.061676
147    742.443290
```

```
148    739.105793
149    739.990485
Length: 150, dtype: float64
  type(y):
<class 'pandas.core.series.Series'>

  y_predicted:
0        8.000000
1       11.121212
2       14.242424
3       17.363636
4       20.484848
          ...
145    460.575758
146    463.696970
147    466.818182
148    469.939394
149    473.060606
Length: 150, dtype: float64
  type(y_predicted):
<class 'pandas.core.series.Series'>

  squared_error:
0         19.468170
1          2.044653
2        787.660370
3          9.705981
4         74.153625
           ...
145    79071.061373
146    74728.262386
147    75969.200069
148    72450.550200
149    71251.560528
Length: 150, dtype: float64
  type(squared_error:
<class 'pandas.core.series.Series'>)

  Yay 2!
  Press enter to continue.

  squared_error.mean():
```

```
26439.134430089372
  type(squared_error.mean()
<class 'numpy.float64'>)

  loss_at_wt:
[29938.07548415073, 28747.51883105903, 27581.2051463719, 26439.134430089372]
  type(loss_at_wt):
<class 'list'>

  Yay 3!
  Press enter to continue.
```

In [22]:

```python
#DWB#+ This doesn't really mean anything, since we
#DWB#+ stopped after 4 <strike>or 5</strike> iterations.
min(loss_at_wt)
```

Out[22]:

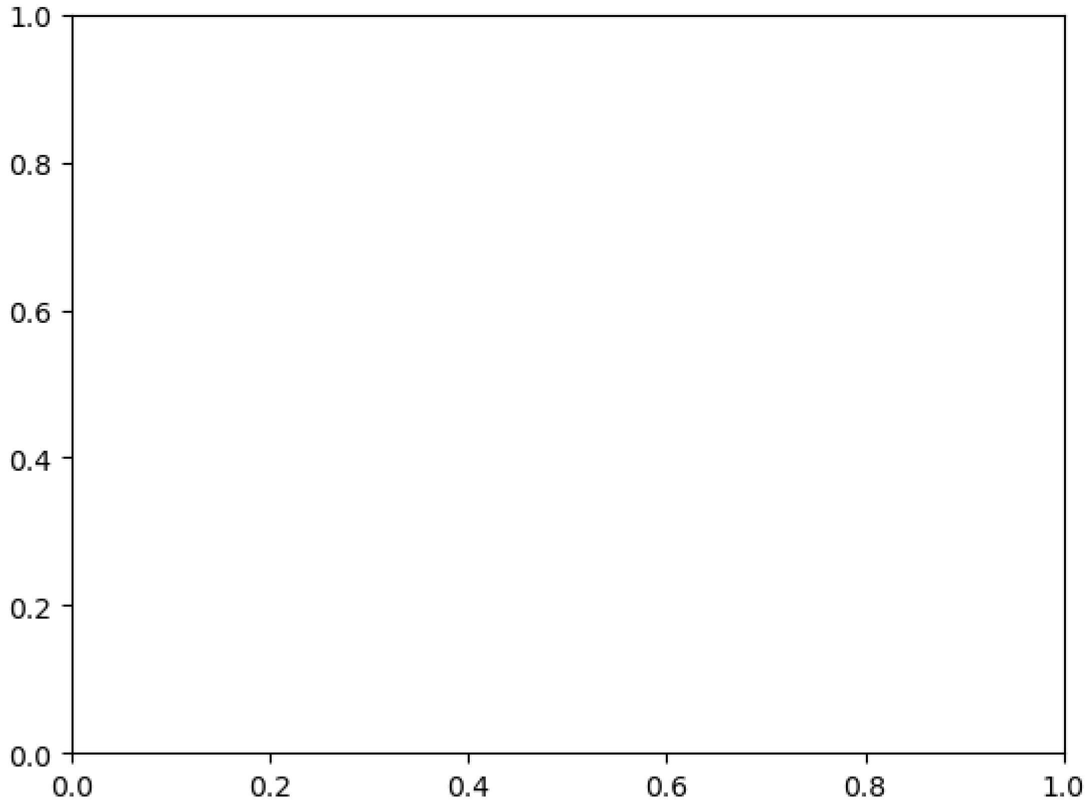
```
26439.134430089372
```

In [23]:

```python
#DWB#  This one shouldn't even work, once again since
#DWB#+ we stopped after the 4 <strike>5</strike> iterations. Let's see
#DWB#+ the error, just for fun.

#plt.scatter(x=weight, y=loss_at_wt)
plt.plot(weight,loss_at_wt)
plt.grid(True)
plt.xlabel('Weight for feature x')
plt.ylabel('Loss')
plt.title('Loss Curve - Loss at different weight')
plt.show()
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[23], line 6
      1 #DWB#  This one shouldn't even work, once again since
      2 #DWB#+ we stopped after the 4 <strike>5</strike> iterations. Let's see
      3 #DWB#+ the error, just for fun.
      4 
      5 #plt.scatter(x=weight, y=loss_at_wt)
----> 6 plt.plot(weight,loss_at_wt)
      7 plt.grid(True)
      8 plt.xlabel('Weight for feature x')

File ~/anaconda3/envs/python3/lib/python3.10/site-packages/matplotlib/pyplot.py:2812, in plot(scalex, scaley, data, *args, **kwargs)
   2810 @_copy_docstring_and_deprecators(Axes.plot)
   2811 def plot(*args, scalex=True, scaley=True, data=None, **kwargs):
-> 2812     return gca().plot(
   2813         *args, scalex=scalex, scaley=scaley,
   2814         **({"data": data} if data is not None else {}), **kwargs)

File ~/anaconda3/envs/python3/lib/python3.10/site-packages/matplotlib/axes/_axes.py:1688, in Axes.plot(self, scalex, scaley, data, *args, **kwargs)
   1445 """
   1446 Plot y versus x as lines and/or markers.
   1447 
   (...)
   1685 (``'green'``) or hex strings (``'#008000'``).
   1686 """
   1687 kwargs = cbook.normalize_kwargs(kwargs, mlines.Line2D)
-> 1688 lines = [*self._get_lines(*args, data=data, **kwargs)]
   1689 for line in lines:
   1690     self.add_line(line)

File ~/anaconda3/envs/python3/lib/python3.10/site-packages/matplotlib/axes/_base.py:311, in _process_plot_var_args._
_call__(self, data, *args, **kwargs)
    309     this += args[0],
    310     args = args[1:]
--> 311 yield from self._plot_args(
    312     this, kwargs, ambiguous_fmt_datakey=ambiguous_fmt_datakey)

File ~/anaconda3/envs/python3/lib/python3.10/site-packages/matplotlib/axes/_base.py:504, in _process_plot_var_args._
plot_args(self, tup, kwargs, return_kwargs, ambiguous_fmt_datakey)
```

```
    501     self.axes.yaxis.update_units(y)
    503 if x.shape[0] != y.shape[0]:
--> 504     raise ValueError(f"x and y must have same first dimension, but "
    505                      f"have shapes {x.shape} and {y.shape}")
    506 if x.ndim > 2 or y.ndim > 2:
    507     raise ValueError(f"x and y can be no greater than 2D, but have "
    508                      f"shapes {x.shape} and {y.shape}")

ValueError: x and y must have same first dimension, but have shapes (100,) and (4,)
```



Oh, yeah. I can run the loop through without the additions.

In [24]:
```
# Cost/Loss Calculation: Squared loss function...a measure of how far is predicted value from actual
# Steps :

#  For every weight for feature x, predict y
#  Now, find out loss by = average ((actual - predicted)**2)
```

```
#DWB# -v-Figuring out what's going on, here.
do_see_the_guts = False

if do_see_the_guts:
    this_count = 0
    this_count_max = 5

#DWB# -v- Remember:
# # Estimate predicted value for a given weight
# def predicted_at_weight(weight0, weight1, x):
#     return weight1*x + weight0

loss_at_wt = []
for w1 in weight:
    y_predicted = predicted_at_weight(8,w1,x)

    if do_see_the_guts:
        this_count += 1
        if this_count == this_count_max:
            break
        ##endof:  if this_count == this_count_max

        print()
        print(f"  this_count:\n{this_count}")
        print(f"  this_count_max:\n{this_count_max}")

        print()
        print(f"  w1:\n{w1}")
        print(f"  type(w1):\n{type(w1)}")
        print()
        print(f"  x:\n{x}")
        print(f"  type(x):\n{type(x)}")
        print()
        print(f"  y_predicted = predicted_at_weight(8,{w1},x) =>")
        print(f"  y_predicted =\n{y_predicted}")
        print()
        print("  Yay 1!")
        input("  Press enter to continue.")
    ##endof:  if do_see_the_guts

    squared_error = (y - y_predicted)**2
```

```
    if do_see_the_guts:
        print()
        print(f"  y:\n{y}")
        print(f"  type(y):\n{type(y)}")
        print()
        print(f"  y_predicted:\n{y_predicted}")
        print(f"  type(y_predicted):\n{type(y_predicted)}")
        print()
        print(f"  squared_error:\n{squared_error}")
        print(f"  type(squared_error:\n{type(squared_error)})")
        print()
        print("  Yay 2!")
        input("  Press enter to continue.")
    ##endof:  if do_see_the_guts

    # Average Squared Error at weight w1
    loss_at_wt.append(squared_error.mean())

    if do_see_the_guts:
        print()
        print(f"  squared_error.mean():\n{squared_error.mean()}")
        print(f"  type(squared_error.mean()\n{type(squared_error.mean())})")
        print()
        print(f"  loss_at_wt:\n{loss_at_wt}")
        print(f"  type(loss_at_wt):\n{type(loss_at_wt)}")
        print()
        print("  Yay 3!")
        input("  Press enter to continue.")
    ##endof:  if do_see_the_guts
```

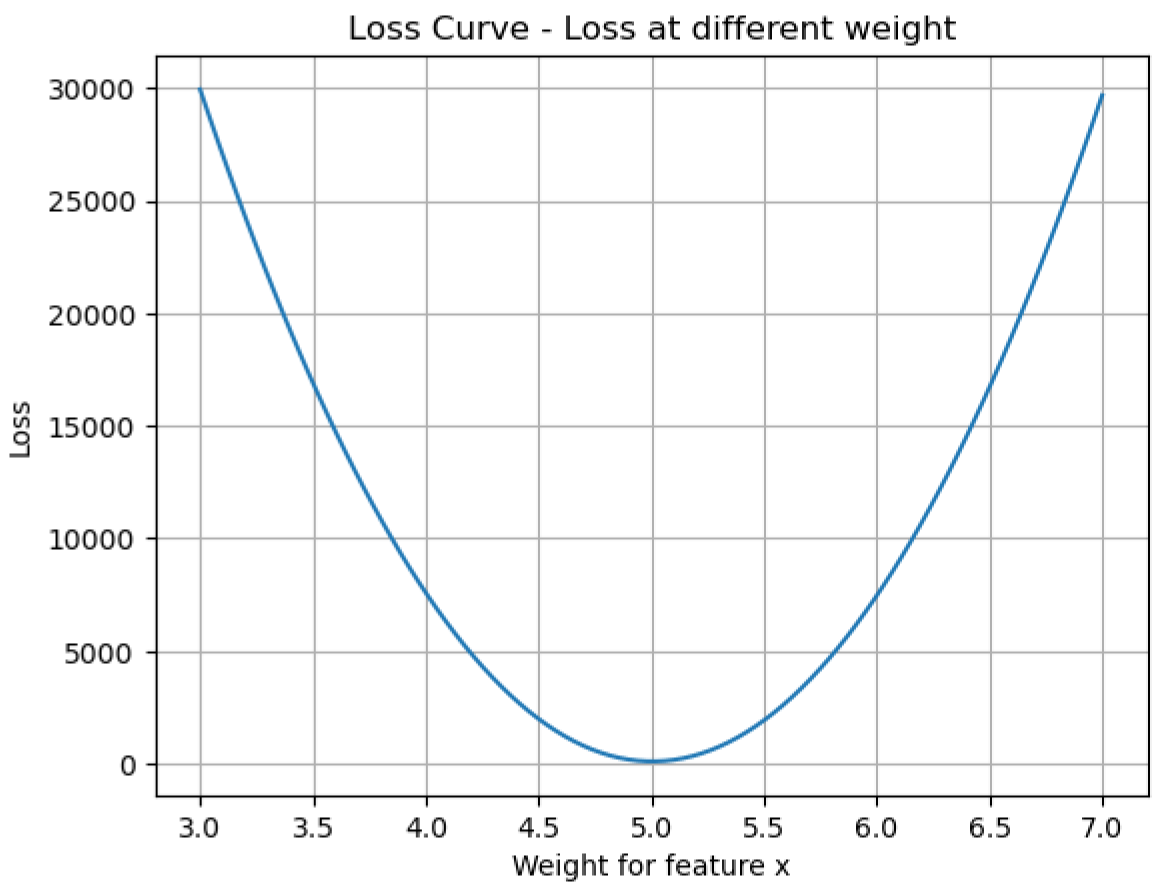
In [33]:

```
#DWB#  Since I ran the code a second time without the
#DWB#+ additions (I didn't want to look up the pre-
#DWB#+ additions code, so I just set the boolean for
#DWB#+ running it to False the second time), this
#DWB#+ cell and the next will mean something
#min(loss_at_wt)

#DWB#
loss_to_find = min(loss_at_wt)
print(loss_to_find)
```

```
107.87912518145518
```

In [26]:

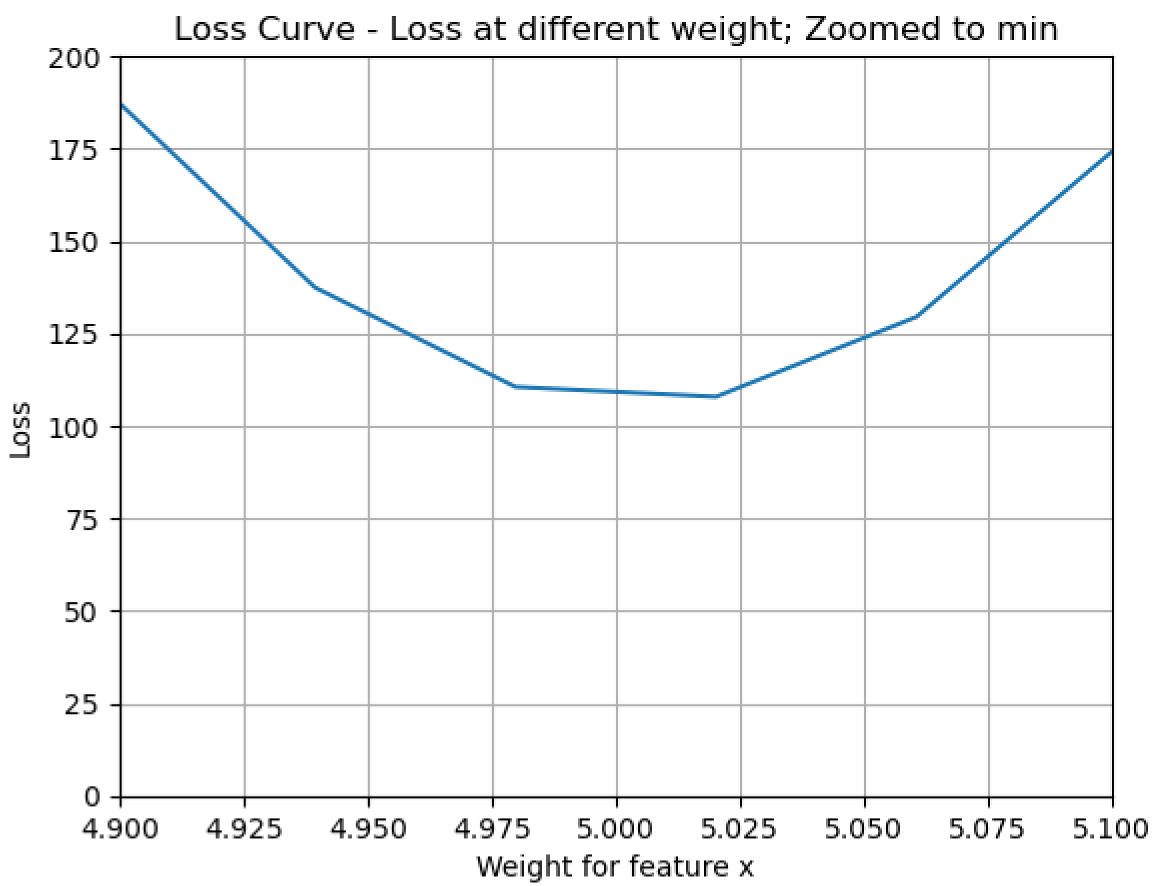
```
#plt.scatter(x=weight, y=loss_at_wt)
plt.plot(weight,loss_at_wt)
plt.grid(True)
plt.xlabel('Weight for feature x')
plt.ylabel('Loss')
plt.title('Loss Curve - Loss at different weight')
plt.show()
```



In [39]:

```
#DWB#
plt.plot(weight,loss_at_wt)
plt.grid(True)
plt.xlim(4.9, 5.1)
```

```
plt.ylim(0, 200)
plt.xlabel('Weight for feature x')
plt.ylabel('Loss')
plt.title('Loss Curve - Loss at different weight; Zoomed to min')
plt.show()
```



In [38]:
```
#DWB#  I guess this is very similar the original code, but I'm
#DWB#+ adding details to find out more about the minimum.

we_have_found_it = False

for w1 in weight:
    y_predicted = predicted_at_weight(8,w1,x)
    squared_error = (y - y_predicted)**2
```

```
    this_loss = squared_error.mean()

    if abs(this_loss - loss_to_find) < 0.1:
        we_have_found_it = True
        print( "  We found it!")
        print(f"  The minimum loss of: {this_loss}")
        print( "  came at the (numerically-found) weight of: " + \
              f"{w1}"
             )
        print("  (For 'numerically-found', you can read, 'approximate'.)")
    ##endof:  if abs(this_loss - loss_to_find) < 0.1

    if we_have_found_it:
        break

##endof:  for w1 in weight
```

```
We found it!
The minimum loss of: 107.87912518145518
came at the (numerically-found) weight of: 5.020202020202021
(For 'numerically-found', you can read, 'approximate'.)
```

Summary

Squared Loss Function

Squared Loss is the average of the squared difference between predicted and actual value. This loss function not only gives us loss at a given weight; it also tells us which direction to go to minimize loss.
For a given weight, the algorithm finds the slope

- If the slope is negative, then increase the weight
- If the slope is positive, then decrease the weight

Learning Rate

Learning Rate parameter controls how much the weight should be increased or decreased
Too big of a change, the algorithm will skip the point where loss is minimal
Too small of a change, the algorithm will take several iterations to find the optimal weight

Gradient Descent

Gradient Descent optimization computes the loss and slope, then adjusts the weights of all the features.
It iterates this process until it finds the optimal weight.
There are three flavors of Gradient descent:

Batch Gradient Descent

Batch gradient descent computes loss for all examples in the training set and then adjusts the weight
It repeats this process for every iteration.
This process can be slow to converge when you have a large training data set

Stochastic Gradient Descent

With Stochastic Gradient Descent, the algorithm computes loss for the next training example and immediately adjusts the weights.
This approach can help in converging to optimal weights for large data sets.
However, one problem with this approach is algorithm is adjusting weights based on a single example [our end objective is to find weight that works for all training examples and not for the immediate example], and this can result in wild fluctuation in weights.

Mini-Batch Gradient Descent

Mini-batch Gradient descent combines benefit of Stochastic and Batch Gradient descent.
It adjusts the weight by testing few samples. The number of samples is defined by mini-batch size, typically around 128.
The mini-batch approach can be used to compute loss in parallel.
This technique is prevalent in deep learning and other algorithms.

In [ ]:

```
```