CS 236756 - Technion - Intro to Machine Learning

Tal Daniel

Tutorial 06 - Optimization

Agenda

- Optimization Problems
- 1-dimensional Optimization
  - Least-Squares
  - Gradient Descent
  - Stochastic Gradient Descent (SGD)
- Mathematical Background
  - Gradient
  - Multi-dimensional Calculus
  - Chain Rule
- Multi-dimensional Optimization
- Constrained Optimization
  - Largrange Multipliers
  - Examples: Entropy

Optimization Problems

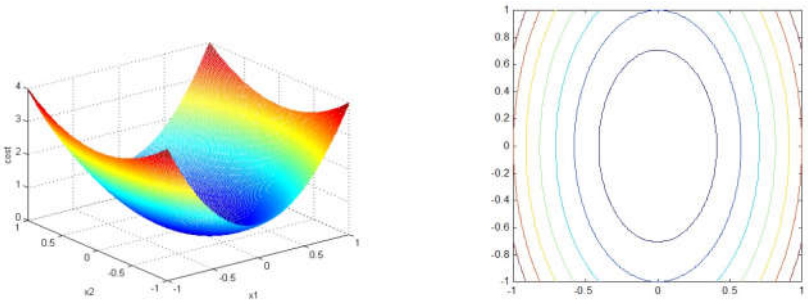
Definitions

- **Objective Function** - mathematical function which is optimized by changing the values of the design variables.
- **Design Variables** - variables that we, as designers, can change.
- **Constraints** - functions of the design variables which establish limits in individual variables or combinations of design.
  - For example - "Find θ that minimizes $f_\theta(x)$ s.t. (subject to) $\theta \leq 1$"

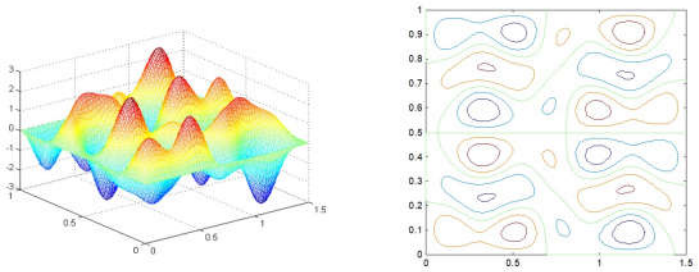
The main problem in optimization is *how* to search for the values of decision variables that minimze.

Types of Objective Functions

- **Unimodal** - only **one** optimum, that is, the *local* optimum is also global.



- **Multimodal** - more than one optimum



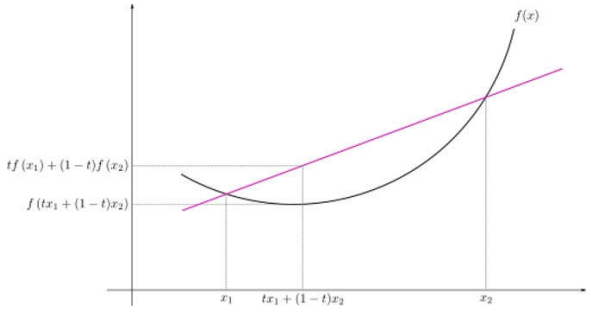
Most serach schemes are based on the assumption of **unimodal** surface. The optimum determined in such cases is called **local optimum design**.

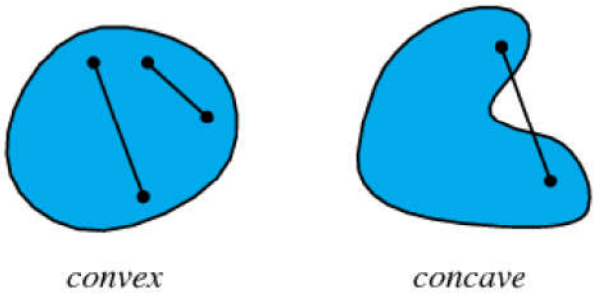
The **global optimum** is the best of all *local optimum* designs.

Convexity

- **Definition**:

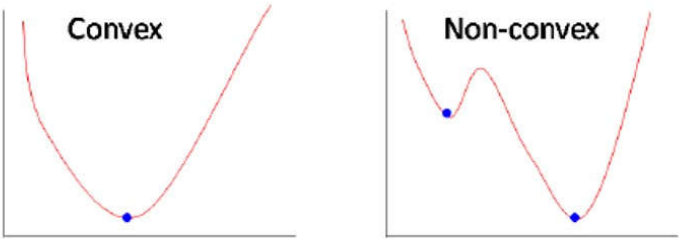
$$\forall x_1, x_2 \in X, \forall t \in [0,1]:$$
$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2)$$





(image from wolfram.com (http://mathworld.wolfram.com/Convex.html))

- Convex functions are **unimodal**

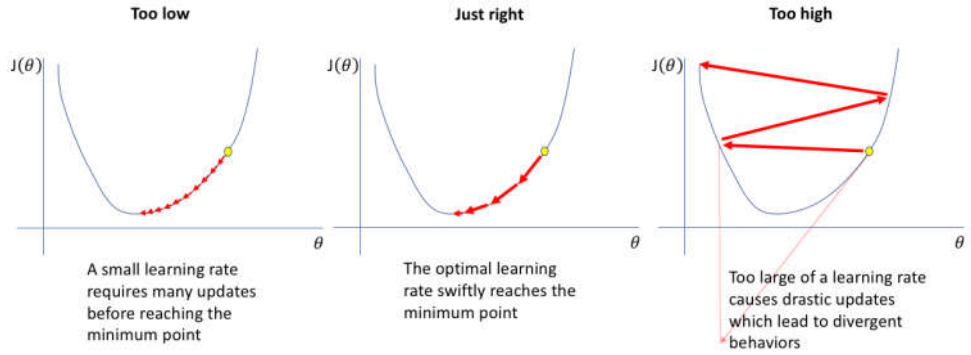


Continuous Optimization

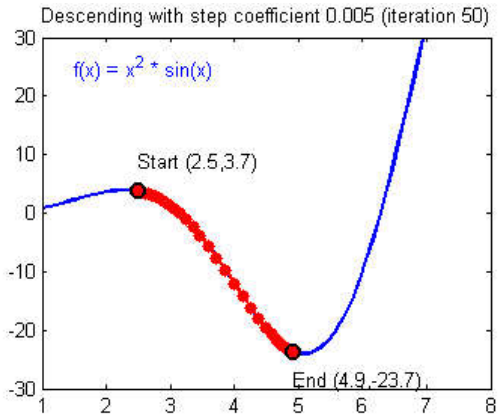
In [1]:
```
# imports for the tutorial
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
%matplotlib notebook
```

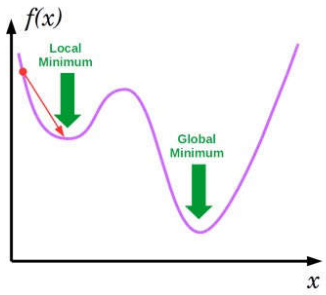
1-D Optimization

(Batch) Gradient Descent

- Generic optimization algorithm capable of finding optimal solutions to a wide range of problems.
- The general idea is to tweak parameters **iteratively** to minimize a cost function.
- It measures the local gradient of the error function with regards to the parameter vector (θ or w), and it goes down in the direction of the descending gradient. Once the gradient is zero - the minimum is reached (=convergence).
- **Learning Rate** hyperparameter - it is the size of step to be taken in each iteration.
  - Too *small* $\rightarrow$ the algorithm will have to go through many iterations to converge, which will take a long time
  - Too *high* $\rightarrow$ might make the algorithm diverge as it may miss the minimum
  - 

    (image from jeremyjordan.me (https://www.jeremyjordan.me/nn-learning-rate/))
- **Pseudocode**:
  - **Require**: Learning rate α_k
  - **Require**: Initial parameter w
  - **While** stopping criterion not met **do**
    - Compute gradient: $g \leftarrow f'(x, w)$ (more specifically, for M samples: $g \leftarrow \frac{1}{M} \sum_{i=1}^{M} f'(x_i, w)$, where f' is w.r.t θ or w)
    - Apply update: $w \leftarrow w - \alpha_k g$
    - $k \leftarrow k + 1$
  - **end while**
- **Visualization**:





- **Convergene**: When the cost function is *convex* and its slope does not change abruptly, (Batch) GD with a *fixed* learning rate will eventually converge to the optimal solution (but the time is depndent on the rate).

Example - Linear Least Squares

- **Problem Formulation**
  - $y \in \mathbb{R}^N$ - vector of values
  - $X \in \mathbb{R}^N$ - data matrix with *one feature* (= data *vector*)
  - $w \in \mathbb{R}$ - the *parameter* to be learnt
- **Goal**: find w that best fits the measurement y
- Mathematiacally:

$$\min_w f(w;x,y) = \min_w \sum_{i=1}^{N} (wx_i - y_i)^2$$

- In vector form:

$$\min_w f(w;x,y) = \min_w ||wX - Y||_2^2$$

LLS - Analytical Solution

- Mathematiacally:

$$\min_w f(w;x,y) = \min_w \sum_{i=1}^{N} (wx_i - y_i)^2 = \min_w \sum_{i=1}^{N} w^2x_i^2 - 2wx_iy_i + y_i^2$$

- The derivative:

$$\sum_{i=1}^{N} 2wx_i^2 - 2x_iy_i = 0 \rightarrow w = \frac{\sum_{i=1}^{N} y_ix_i}{\sum_{i=1}^{N} x_i^2}$$

- The second derivative, to ensure minimum:

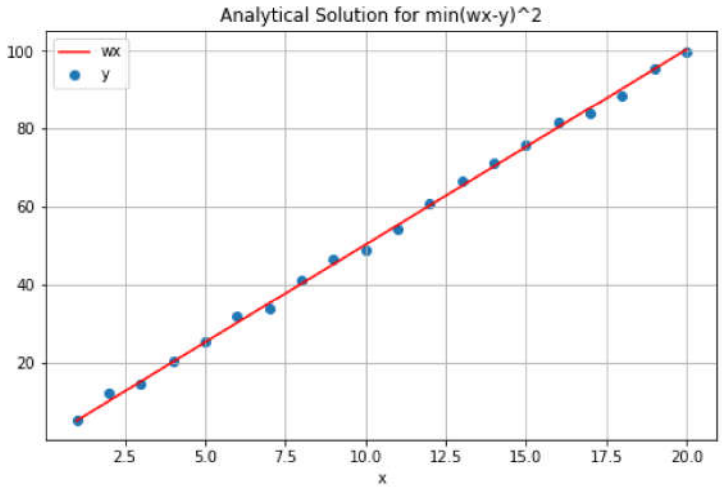
$$f''(w;x,y) = \sum_{i=1}^{N} 2x_i^2 > 0 \rightarrow good!$$

In [2]:
```
# generate some random data
N = 20
x = np.linspace(1, 20, N)
y = 5 * x + np.random.randn(N)
```

In [3]:
```
# lls - analytical solution
# we want to find w that minimizes (wx-y)^2
# by the above derivation
w = np.sum(y * x) / np.sum(np.square(x))
print("best w:", w)
# let's plot
fig = plt.figure(figsize=(8,5))
ax = fig.add_subplot(1,1,1)
ax.scatter(x, y, label="y")
ax.plot(x, w * x, label="wx", color='r')
ax.legend()
ax.grid()
ax.set_xlabel("x")
ax.set_title("Analytical Solution for min(wx-y)^2")
```

```
best w: 5.020283685183086
```

Out[3]: Text(0.5,1,'Analytical Solution for min(wx-y)^2')



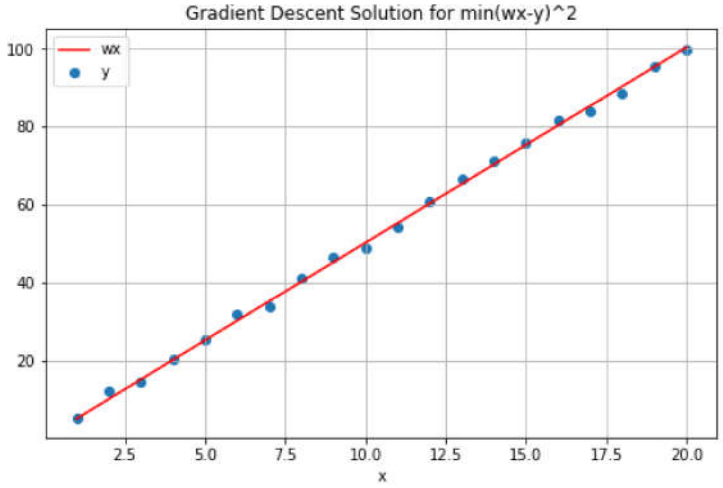
LLS - Gradient Descent Solution

- **Pseudocode**:
  - **Require**: Learning rate α_k
  - **Require**: Initial parameter w
  - **While** stopping criterion not met **do**
    - Compute gradient: $g \leftarrow \frac{1}{N} \sum_{i=1}^{N} 2wx_i^2 - 2x_i y_i$
    - Apply update: $w \leftarrow w - \alpha_k g$
    - $k \leftarrow k + 1$
  - **end while**

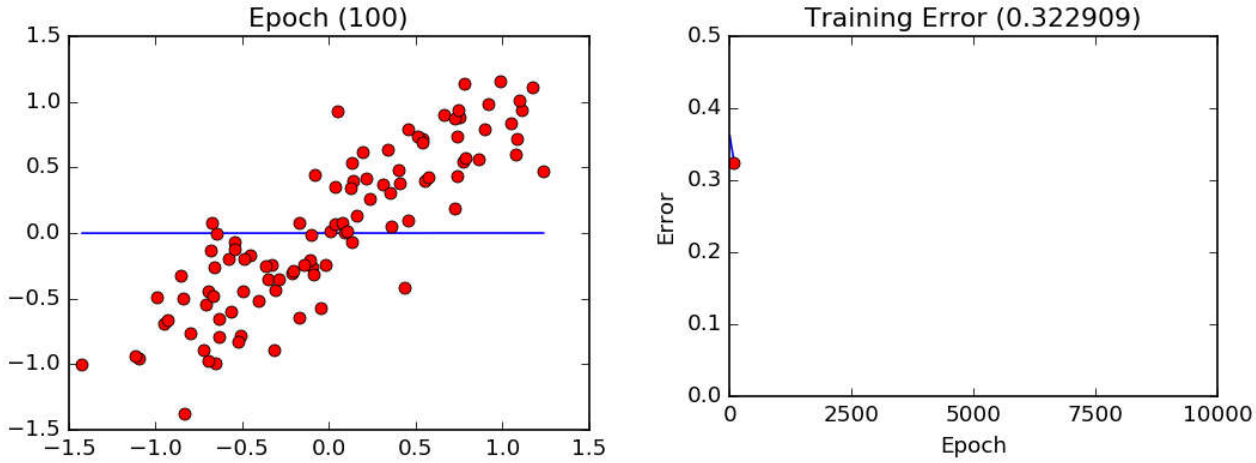
In [4]:
```
# lls - gradient descent solution
N = 20 # num samples
num_iterations = 20
alpha_k = 0.005
# we want to find w that minimizes (wx-y)^2
# initialize w
w = 0
for i in range(num_iterations):
    print("iter:", i, " w = ", w)
    gradient = np.sum(2 * w * np.square(x) - 2 * x * y) / N
    w = w - alpha_k * gradient
print("best w:", w)
# let's plot
fig = plt.figure(figsize=(8,5))
ax = fig.add_subplot(1,1,1)
ax.scatter(x, y, label="y")
ax.plot(x, w * x, label="wx", color='r')
ax.legend()
ax.grid()
ax.set_xlabel("x")
ax.set_title("Gradient Descent Solution for min(wx-y)^2")
```

```
iter: 0  w =  0
iter: 1  w =  7.204107088237729
iter: 2  w =  4.070320504854317
iter: 3  w =  5.433517668626101
iter: 4  w =  4.840526902385375
iter: 5  w =  5.098477885700091
iter: 6  w =  4.986269207958189
iter: 7  w =  5.035079982775916
iter: 8  w =  5.013847295730206
iter: 9  w =  5.02308351459509
iter: 10  w =  5.019065759388865
iter: 11  w =  5.020813482903573
iter: 12  w =  5.020053223174675
iter: 13  w =  5.020383936156746
iter: 14  w =  5.020240076009545
iter: 15  w =  5.020302655173577
iter: 16  w =  5.020275433237223
iter: 17  w =  5.0202872747795375
iter: 18  w =  5.020282123708631
iter: 19  w =  5.020284364424475
best w: 5.020283389713082
```

Out[4]: Text(0.5,1,'Gradient Descent Solution for min(wx-y)^2')

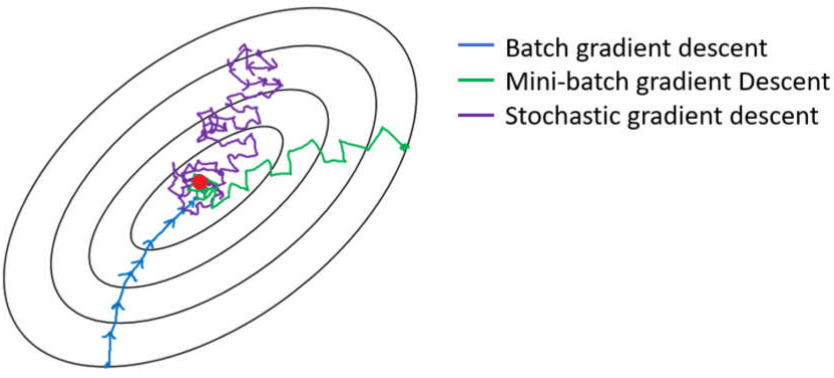


- **Least Squares Visualization**:



Stochastic Gradient Descent (Mini-Batch Gradient Descent)

- The main problem with (Batch) GD is that it uses the **whole** training set to compute the gradients. But what if that training set is huge? Computing the gradient can take a very long time.
- *Stochastic* Gradient Descent on the other hand, samples just one instance randomly at every step and computes the gradients based on that single instance. This makes the algorithm much faster but due to its randomness, it is much less stable. Instead of steady decreasing untill reaching the minimum, the cost function will bounce up and down, **decreasing only on average.** With time, it will get *very close* to the minimum, but once it is there it will continue to bounce around!
- The final parameters are good but **not optimal.**
- When the cost function is very irregular, this bouncing can actually help the algorithm escape local minima, so SGD has better chance to fund the *global* minimum.
- How to find optimal parameters using SGD?
  - **Reduce the learning rate gradually**: this is called *learning schedule*
    - But don't reduce too quickly or you will get stuck at a local minimum or even frozen!
- *Mini-Batch* Gradient Descent - same idea as SGD, but instead of one instance each step, m samples.
  - Get a little bit closer to the minimum than SGD but a little harder to escape local minima.
- **Pseudocode**:
  - **Require**: Learning rate α_k
  - **Require**: Initial parameter w
  - **While** stopping criterion not met **do**
    - Sample a minibatch of m examples from the training set ($m = 1$ for SGD)
    - Set $\{x_1, \ldots, x_m, \}$ with corresponding targets $\{y_1, \ldots, y_m\}$
    - Compute gradient: $g \leftarrow \frac{1}{m} \sum_{i=1}^{m} f'(x_i, w)$
    - Apply update: $w \leftarrow w - \alpha_k g$
    - $k \leftarrow k + 1$
  - **end while**



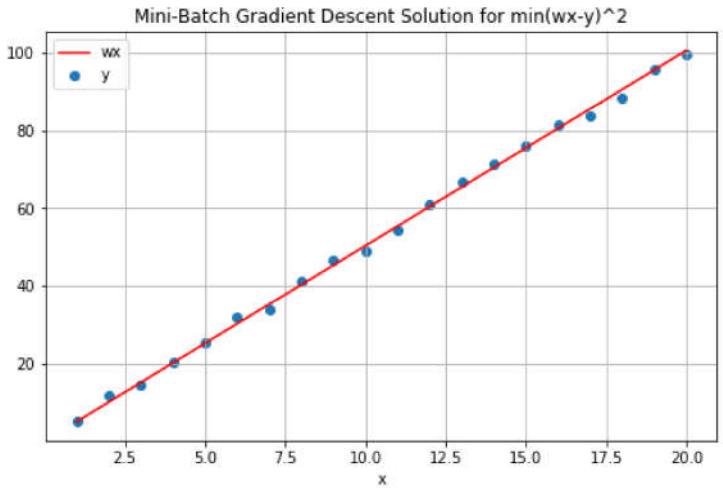
(image from Imad Dabbura @ towardsdatascience.com (https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3))

In [5]:
```
def batch_generator(x, y, batch_size, shuffle=True):
    """
    This function generates batches for a given dataset x.
    """
    N = len(x)
    num_batches = N // batch_size
    batch_x = []
    batch_y = []
    if shuffle:
        # shuffle
        rand_gen = np.random.RandomState(0)
        shuffled_indices = rand_gen.permutation(np.arange(len(x)))
        x = x[shuffled_indices]
        y = y[shuffled_indices]
    for i in range(N):
        batch_x.append(x[i])
        batch_y.append(y[i])
        if len(batch_x) == batch_size:
            yield np.array(batch_x), np.array(batch_y)
            batch_x = []
            batch_y = []
    if batch_x:
        yield np.array(batch_x), np.array(batch_y)


# mini-batch gradient descent
batch_size = 5
num_batches = N // batch_size
print("total batches:", num_batches)
num_iterations = 20
alpha_k = 0.001
batch_gen = batch_generator(x, y, batch_size, shuffle=True)
# we want to find w that minimizes (wx-y)^2
# initialize w
w = 0
for i in range(num_iterations):
    for batch_i, batch in enumerate(batch_gen):
        batch_x, batch_y = batch
        if batch_i % 5 == 0:
            print("iter:", i, "batch:", batch_i, " w = ", w)
        gradient = np.sum(2 * w * np.square(batch_x) - 2 * batch_x * batch_y) / len(batch_x)
        w = w - alpha_k * gradient
    batch_gen = batch_generator(x, y, batch_size, shuffle=True)
print("best w:", w)
# let's plot
fig = plt.figure(figsize=(8,5))
ax = fig.add_subplot(1,1,1)
ax.scatter(x, y, label="y")
ax.plot(x, w * x, label="wx", color='r')
ax.legend()
ax.grid()
ax.set_xlabel("x")
ax.set_title("Mini-Batch Gradient Descent Solution for min(wx-y)^2")
```

```
total batches: 4
iter: 0 batch: 0  w =  0
iter: 1 batch: 0  w =  3.7503746340220854
iter: 2 batch: 0  w =  4.704812127816367
iter: 3 batch: 0  w =  4.947708109806134
iter: 4 batch: 0  w =  5.009523009629727
iter: 5 batch: 0  w =  5.025254360181099
iter: 6 batch: 0  w =  5.02925785110119
iter: 7 batch: 0  w =  5.030276704472674
iter: 8 batch: 0  w =  5.030535993731299
iter: 9 batch: 0  w =  5.030601980576433
iter: 10 batch: 0  w =  5.030618773650523
iter: 11 batch: 0  w =  5.030623047340784
iter: 12 batch: 0  w =  5.030624134957521
iter: 13 batch: 0  w =  5.030624411746453
iter: 14 batch: 0  w =  5.030624482186813
iter: 15 batch: 0  w =  5.030624500113267
iter: 16 batch: 0  w =  5.030624504675393
iter: 17 batch: 0  w =  5.030624505836413
iter: 18 batch: 0  w =  5.0306245061318835
iter: 19 batch: 0  w =  5.030624506207078
best w: 5.030624506226214
```

```
Out[5]: Text(0.5,1,'Mini-Batch Gradient Descent Solution for min(wx-y)^2')
```



- Note: All of the Gradient Descent algorithms require **scaling** if the feaures are not within the same range!

GD Comparison Summary

Method	Accuracy	Update Speed	Memory Usage	Online Learning
Batch Gradient Descent	Good	Slow	High	No
Stochastic Gradient Descent	Good (with softening)	Fast	Low	Yes
Mini-Batch Gradient Descent	Good	Medium	Medium	Yes (depends on the MB size)

- *"Online"* - samples arrive while the algorithm runs (that is, when the algorithm starts running, not all samples exist)

Challenges

- Choosing a **learning rate**
  - Defining **learning schedule**
- Working with features of different scales (e.g. heights (cm), weights (kg) and age (scalar))
- Avoiding **local minima** (or *suboptimal* minima)

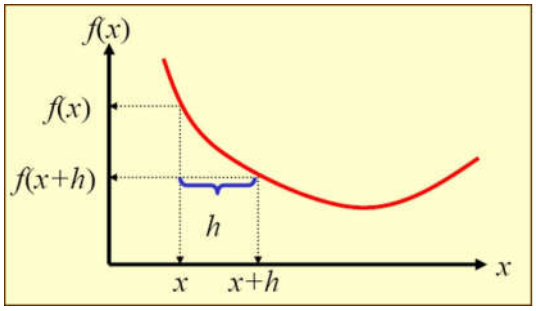
Mathematical Background

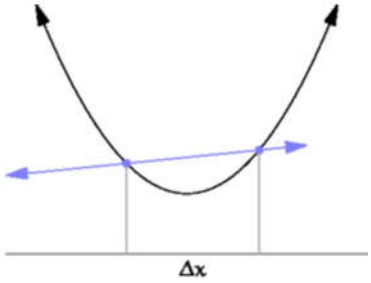
Multivariate Calculus

- **The Derivative** - the derivative of $f : \mathbb{R} \to \mathbb{R}$ is a *function* $f' : \mathbb{R} \to \mathbb{R}$ given by:

$$f'(x) = \frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

  - Illustration:

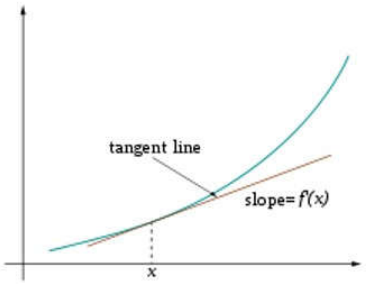




  - Rewrite the above:

$$\lim_{h \to 0} \frac{f(x+h) - f(x) - f'(x) \cdot h}{h} = 0$$

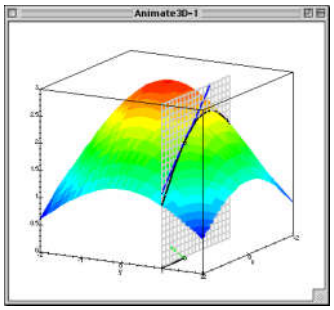
    -



- **The Gradient** - the gradient of $f : \mathbb{R}^N \to \mathbb{R}$ is a *function* $\nabla f : \mathbb{R}^N \to \mathbb{R}^N$ given by:

$$\lim_{h \to 0} \frac{||f(\overline{x} + \overline{h}) - f(\overline{x}) - \nabla f(\overline{x}) \cdot \overline{h}||}{||\overline{h}||} = 0$$

  -



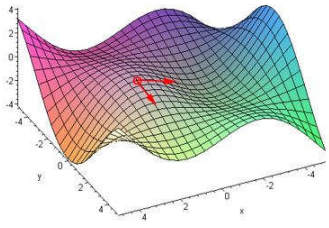
  - The gradient can be expressed in terms of the function's **partial derivatives**: $$\nabla f(x) = \begin{bmatrix}

```
\frac{\partial f}{\partial x_1} \\
\frac{\partial f}{\partial x_2} \\
\vdots \\
\frac{\partial f}{\partial x_n}
```

\end{bmatrix} $$

  - Illustration:

$$\nabla f(x) \triangleq \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix}$$



- **The Hessian Matrix**
  - Definition: $H(f)(x)_{i,j} = \frac{\partial^2}{\partial x_i \partial x_j} f(x)$
  - $$\begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{} & \frac{\partial^2 f}{} & \cdots & \frac{\partial^2 f}{} \end{bmatrix}$$

Matrix Calculus - Vector & Matrix Derivatives

- We will use most of the derivations "as is" without derivation.
- A good reference: **The Matrix Cookbook (http://www2.imm.dtu.dk/pubdb/views/edoc_download.php/3274/pdf/imm3274.pdf)**
- ⚠ **REMEMBER** - ALWAYS write the dimensions of each component and identify whether the expression is a **matrix, vector or scalar**!

Derivative of Vector Multiplication

- Let $x, a \in \mathbb{R}^N \rightarrow x, a$ are vectors
- $\frac{\partial x^T a}{\partial x} = \frac{\partial a^T x}{\partial x} = a$
  - $x^T a = a^T x$ are **scalars**
  - a is a **vector**
  - Derivation: $$ f = x^Ta = [x_1, x_2, ..., x_n] \begin{bmatrix}

```
a_{1} \\
a_{2} \\
\vdots \\
a_{n}
```

\end{bmatrix} = a_1 x_1 + a_2 x_2 + ... + a_n x*n* = *sum*{i=1}^n a_i x_i

\frac{\partial x^Ta}{\partial x} = \begin{bmatrix}

```
\frac{\partial f}{\partial x_1} \\
\frac{\partial f}{\partial x_2} \\
\vdots \\
\frac{\partial f}{\partial x_n}
```

\end{bmatrix} = \begin{bmatrix}

```
a_{1} \\
a_{2} \\
\vdots \\
a_{n}
```

\end{bmatrix} = a $$

Common Derivations

- $\nabla_x Ax = A^T$
- $\nabla_x x^T Ax = (A + A^T)x$
  - If W is **symetric**:
    - $\frac{\partial}{\partial s}(x - As)^T W(x - As) = -2A^T W(x - As)$
    - $\frac{\partial}{\partial x}(x - As)^T W(x - As) = 2W(x - As)$
- $\frac{\partial}{\partial A}\ln|A| = A^{-T}$
- $\frac{\partial}{\partial A}Tr[AB] = B^T$

The Chain Rule

- Let

$$f(x) = h(g(x))$$
$$x \in \mathbb{R}^n$$
$$f, g : \mathbb{R}^n \rightarrow \mathbb{R}$$
$$h : \mathbb{R} \rightarrow \mathbb{R}$$

- $\nabla f = h' \cdot \nabla g$ #### Exercise 1 - The Chain Rule Find the gradient of $f(x) = \sqrt{x^T Q x}$ (Q is positive definite) #### Solution 1
- $g(x) = x^T Q x \rightarrow \nabla g = (Q + Q^T)x = 2Qx$
- $h(z) = \sqrt{z} \rightarrow h'(z) = \frac{1}{2\sqrt{z}}$
- $\nabla f = \frac{1}{2\sqrt{x^T Q x}} 2Qx = \frac{Qx}{\sqrt{x^T Q x}}$

Multi-Dimensional Optimization

Optimality Conditions

- If f has *local* optimum at x_0 then $\nabla f(x_0) = 0$
- If the **Hessian** is:
  - **Positive Definite** (all eigenvalues *positive*) at $x_0 \rightarrow$ *local minimum*
  - **Negative Definite** (all eigenvalues *negative*) at $x_0 \rightarrow$ *local maximum*
  - Both **positive and negative** eigenvalues at $x_0 \rightarrow$ *saddle* point
  -

Example - (Multivariate) Linear Least Squares

- **Problem Formulation**
  - $y \in \mathbb{R}^N$ - vector of values
  - $X \in \mathbb{R}^{N \times L}$ - data matrix with N examples and L *features*
  - $w \in \mathbb{R}^L$ - the *parameters* to be learnt, a **weight for each feature**
- **Goal**: find w that best fits the measurement y, that is, find a *weighted linear combination* of the feature vector to best fit the measurment y
- Mathematiacally, the problem is:

$$\min_w f(w; x, y) = \min_w \sum_{i=1}^{N} ||x_i w - y_i||^2$$

- In vector form:

$$\min_w f(w; x, y) = \min_w ||Xw - Y||^2$$

(Multivariate) LLS - Analytical Solution

- Mathematiacally:

$$\min_w f(w; x, y) = \min_w ||Xw - Y||^2 = \min_w (Xw - Y)^T (Xw - Y) = \min_w (w^T X^T Xw - 2w^T X^T Y + Y^T Y)$$

- The derivative:

$$\nabla_w f(w; x, y) = (X^T X + X^T X)w - 2X^T Y = 0 \rightarrow w = (X^T X)^{-1} X^T Y$$
$$X^T X \in \mathbb{R}^{L \times L}$$

In [6]:
```
# let's load the cancer dataset
dataset = pd.read_csv('./datasets/cancer_dataset.csv')
# print the number of rows in the data set
number_of_rows = len(dataset)
# reminder, the data looks like this
dataset.sample(10)
```

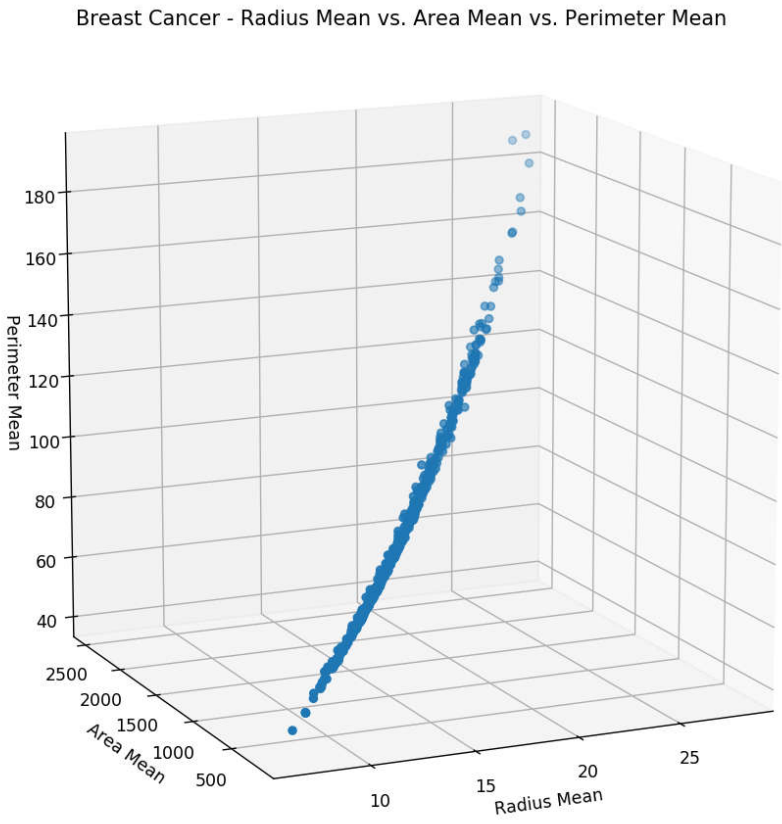
Out[6]:

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_m
107	863270	B	12.360	18.54	79.01	466.7	0.08477	0.06815
513	915940	B	14.580	13.66	94.29	658.8	0.09832	0.08918
524	917897	B	9.847	15.68	63.00	293.2	0.09492	0.08419
473	9113846	B	12.270	29.97	77.42	465.4	0.07699	0.03398
170	87139402	B	12.320	12.39	78.85	464.1	0.10280	0.06981
87	86135502	M	19.020	24.59	122.00	1076.0	0.09029	0.12060
491	91376702	B	17.850	13.23	114.60	992.1	0.07838	0.06217
69	859487	B	12.780	16.49	81.37	502.5	0.09831	0.05234
268	8910506	B	12.870	16.21	82.38	512.2	0.09425	0.06219
28	852973	M	15.300	25.27	102.40	732.4	0.10820	0.16970

10 rows × 33 columns

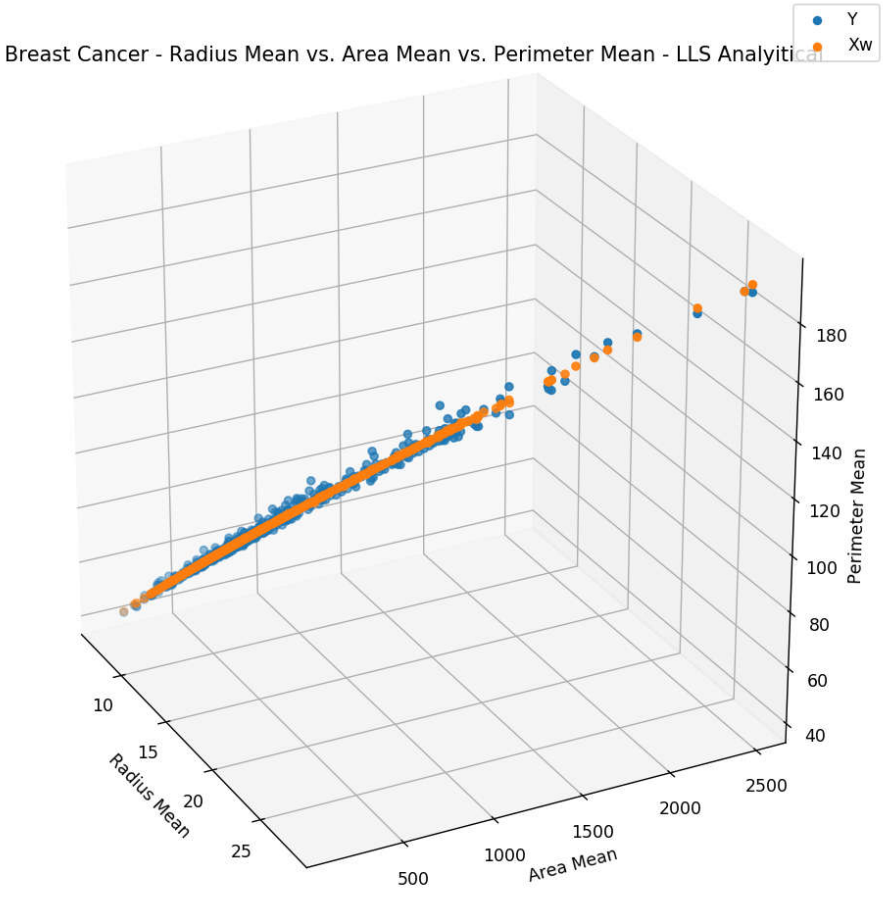
In [7]:
```
# let's plot X = [radius, area], y = perimeter
%matplotlib notebook
fig = plt.figure(figsize=(10, 10))
ax = fig.add_subplot(111, projection='3d')

xs = dataset[['radius_mean']].values
ys = dataset[['area_mean']].values
zs = dataset[['perimeter_mean']].values
ax.scatter(xs, ys, zs)
# ax.axis('equal')
ax.set_xlabel('Radius Mean')
ax.set_ylabel('Area Mean')
ax.set_zlabel('Perimeter Mean')
ax.set_title("Breast Cancer - Radius Mean vs. Area Mean vs. Perimeter Mean")
```



Out[7]: Text(0.5,0.92,'Breast Cancer - Radius Mean vs. Area Mean vs. Perimeter Mean')

In [8]:
```
# multivariate lls - analytical solution
X = dataset[['radius_mean', 'area_mean']].values
Y = dataset[['perimeter_mean']].values
w = np.linalg.inv(X.T @ X) @ X.T @ Y
lls_sol = X @ w
# plot
%matplotlib notebook
fig = plt.figure(figsize=(10, 10))
ax = fig.add_subplot(111, projection='3d')
xs = dataset[['radius_mean']].values
ys = dataset[['area_mean']].values
zs = dataset[['perimeter_mean']].values
ax.scatter(xs, ys, zs, label='Y')
ax.scatter(xs, ys, lls_sol, label='Xw')
ax.legend()
ax.set_xlabel('Radius Mean')
ax.set_ylabel('Area Mean')
ax.set_zlabel('Perimeter Mean')
ax.set_title("Breast Cancer - Radius Mean vs. Area Mean vs. Perimeter Mean - LLS Analyitical")
print("w:")
print(w)
```




```
w:
[[6.19721311]
 [0.00676913]]
```

What If L is Very Large???

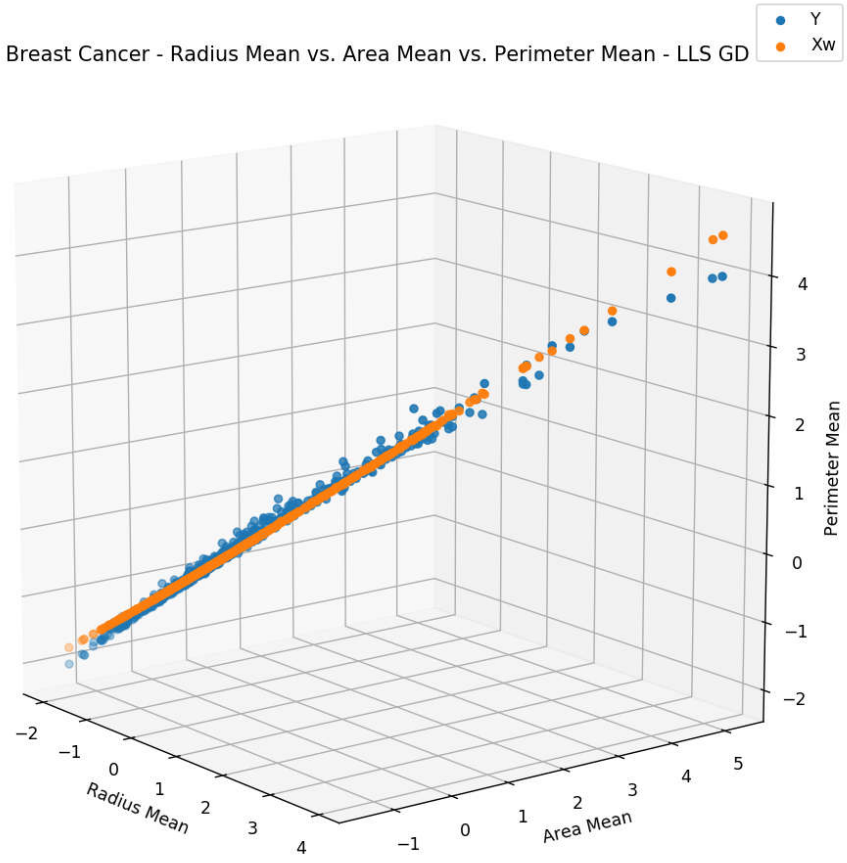
If $L = 1000$, we would need to invert a 1000×1000 matrix, which would take about 10^9 operations!

 (Batch) Gradient Descent

- **Pseudocode**:
  - **Require**: Learning rate α_k
  - **Require**: Initial parameter vector w
  - **While** stopping criterion not met **do**
    - Compute gradient: $g \leftarrow \nabla f(x, w)$
    - Apply update: $w \leftarrow w - \alpha_k g$
    - $k \leftarrow k + 1$
  - **end while**
- For **Linear Least Squares**:
  - **Require**: Learning rate α_k
  - **Require**: Initial parameter vector w
  - **While** stopping criterion not met **do**
    - Compute gradient: $g \leftarrow 2X^T Xw - 2X^T y$
    - Apply update: $w \leftarrow w - \alpha_k g$
    - $k \leftarrow k + 1$

In [9]:
```
# multivariate lls - gradient descent solution
X = dataset[['radius_mean', 'area_mean']].values
Y = dataset[['perimeter_mean']].values
# Scaling
X = (X - X.mean(axis=0, keepdims=True)) / X.std(axis=0, keepdims=True)
Y = (Y - Y.mean(axis=0, keepdims=True)) / Y.std(axis=0, keepdims=True)
num_iterations = 20
alpha_k = 0.0001
L = X.shape[1]
# initialize w
w = np.zeros((L, 1))
for i in range(num_iterations):
    print("iter:", i, " w = ")
    print(w)
    gradient = 2 * X.T @ X @ w - 2 * X.T @ Y
    w = w - alpha_k * gradient
lls_sol = X @ w
# plot
%matplotlib notebook
fig = plt.figure(figsize=(10, 10))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(X[:,0], X[:,1], Y, label='Y')
ax.scatter(X[:,0], X[:,1], lls_sol, label='Xw')
ax.legend()
ax.set_xlabel('Radius Mean')
ax.set_ylabel('Area Mean')
ax.set_zlabel('Perimeter Mean')
ax.set_title("Breast Cancer - Radius Mean vs. Area Mean vs. Perimeter Mean - LLS GD")
print("w:")
print(w)
```

```
iter: 0  w =
[[0.]
 [0.]]
iter: 1  w =
[[0.11355593]
 [0.11226447]]
iter: 2  w =
[[0.20157502]
 [0.19899397]]
iter: 3  w =
[[0.2698325 ]
 [0.26596371]]
iter: 4  w =
[[0.32279748]
 [0.31764279]]
iter: 5  w =
[[0.36392832]
 [0.35748959]]
iter: 6  w =
[[0.39590123]
 [0.38818031]]
iter: 7  w =
[[0.42078718]
 [0.41178591]]
iter: 8  w =
[[0.44018875]
 [0.42990898]]
iter: 9  w =
[[0.4553461 ]
 [0.44378965]]
iter: 10  w =
[[0.46721888]
 [0.45438761]]
iter: 11  w =
[[0.47654975]
 [0.46244548]]
iter: 12  w =
[[0.48391337]
 [0.46853793]]
iter: 13  w =
[[0.48975445]
 [0.47310968]]
iter: 14  w =
[[0.49441713]
 [0.47650485]]
iter: 15  w =
[[0.49816771]
 [0.47898975]]
iter: 16  w =
[[0.50121227]
 [0.48077045]]
iter: 17  w =
[[0.50371028]
 [0.48200641]]
iter: 18  w =
[[0.50578514]
 [0.48282104]]
iter: 19  w =
[[0.50753235]
 [0.48330983]]
```




```
w:
[[0.50902581]
 [0.48354668]]
```

Stochastic Gradient Descent (Mini-Batch Gradient Descent)

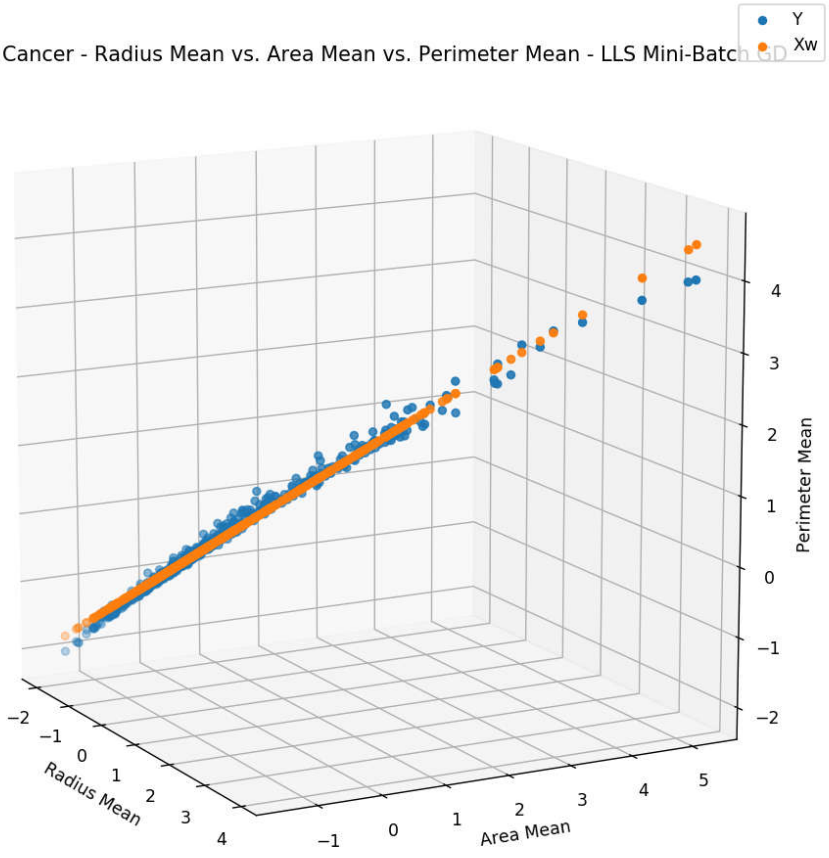
- **Pseudocode**:
  - **Require**: Learning rate α_k
  - **Require**: Initial parameter w
  - **While** stopping criterion not met **do**
    - Sample a minibatch of m examples from the training set ($m = 1$ for SGD)
    - Set $\tilde{X} = [x_1, \ldots, x_m]$ with corresponding targets $\tilde{Y} = [y_1, \ldots, y_m]$
    - Compute gradient: $g \leftarrow 2\tilde{X}^T\tilde{X} - 2\tilde{X}^T\tilde{Y}$
    - Apply update: $w \leftarrow w - \alpha_k g$
    - $k \leftarrow k + 1$
  - **end while**

In [10]:
```
def batch_generator(x, y, batch_size, shuffle=True):
    """
    This function generates batches for a given dataset x.
    """
    N, L = x.shape
    num_batches = N // batch_size
    batch_x = []
    batch_y = []
    if shuffle:
        # shuffle
        rand_gen = np.random.RandomState(0)
        shuffled_indices = rand_gen.permutation(np.arange(N))
        x = x[shuffled_indices, :]
        y = y[shuffled_indices, :]
    for i in range(N):
        batch_x.append(x[i, :])
        batch_y.append(y[i, :])
        if len(batch_x) == batch_size:
            yield np.array(batch_x).reshape(batch_size, L), np.array(batch_y).reshape(batch_size, 1)
            batch_x = []
            batch_y = []
    if batch_x:
        yield np.array(batch_x).reshape(-1, L), np.array(batch_y).reshape(-1, 1)


# multivaraite mini-batch gradient descent
X = dataset[['radius_mean', 'area_mean']].values
Y = dataset[['perimeter_mean']].values
# Scaling
X = (X - X.mean(axis=0, keepdims=True)) / X.std(axis=0, keepdims=True)
Y = (Y - Y.mean(axis=0, keepdims=True)) / Y.std(axis=0, keepdims=True)
N = X.shape[0]
batch_size = 10
num_batches = N // batch_size
print("total batches:", num_batches)
num_iterations = 10
alpha_k = 0.001
batch_gen = batch_generator(X, Y, batch_size, shuffle=True)
# initialize w
w = np.zeros((L, 1))
for i in range(num_iterations):
    for batch_i, batch in enumerate(batch_gen):
        batch_x, batch_y = batch
        if batch_i % 50 == 0:
            print("iter:", i, "batch:", batch_i, " w = ")
            print(w)
        gradient = 2 * batch_x.T @ batch_x @ w - 2 * batch_x.T @ batch_y
        w = w - alpha_k * gradient
    batch_gen = batch_generator(X, Y, batch_size, shuffle=True)

lls_sol = X @ w
# plot
%matplotlib notebook
fig = plt.figure(figsize=(10, 10))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(X[:,0], X[:,1], Y, label='Y')
ax.scatter(X[:,0], X[:,1], lls_sol, label='Xw')
ax.legend()
ax.set_xlabel('Radius Mean')
ax.set_ylabel('Area Mean')
ax.set_zlabel('Perimeter Mean')
ax.set_title("Breast Cancer - Radius Mean vs. Area Mean vs. Perimeter Mean - LLS Mini-Batch GD")
print("w:")
print(w)
```

```
total batches: 56
iter: 0 batch: 0  w =
[[0.]
 [0.]]
iter: 0 batch: 50  w =
[[0.4391737 ]
 [0.41735388]]
iter: 1 batch: 0  w =
[[0.45621431]
 [0.43296123]]
iter: 1 batch: 50  w =
[[0.50458588]
 [0.46953377]]
iter: 2 batch: 0  w =
[[0.50659246]
 [0.46976394]]
iter: 2 batch: 50  w =
[[0.51664408]
 [0.46913792]]
iter: 3 batch: 0  w =
[[0.51715596]
 [0.46786206]]
iter: 3 batch: 50  w =
[[0.52339741]
 [0.46367398]]
iter: 4 batch: 0  w =
[[0.52374047]
 [0.4622501 ]]
iter: 4 batch: 50  w =
[[0.5295512 ]
 [0.45779193]]
iter: 5 batch: 0  w =
[[0.52985556]
 [0.456353  ]]
iter: 5 batch: 50  w =
[[0.53556725]
 [0.45194588]]
iter: 6 batch: 0  w =
[[0.53584598]
 [0.45050492]]
iter: 6 batch: 50  w =
[[0.54149192]
 [0.44617918]]
iter: 7 batch: 0  w =
[[0.54174661]
 [0.44473749]]
iter: 7 batch: 50  w =
[[0.54733087]
 [0.44049501]]
iter: 8 batch: 0  w =
[[0.54756198]
 [0.43905271]]
iter: 8 batch: 50  w =
[[0.55308576]
 [0.43489257]]
iter: 9 batch: 0  w =
[[0.55329364]
 [0.43344969]]
iter: 9 batch: 50  w =
[[0.55875783]
 [0.42937075]]
```




```
w:
[[0.55894282]
 [0.42792729]]
```

Constrained Optimization

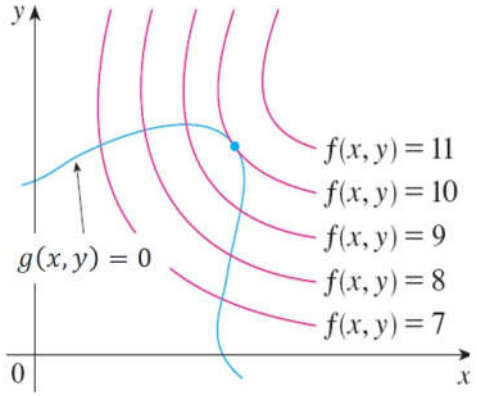
Largrange Multipliers

- A method for optimization with **equality constraints**
- The general case:

$$\min f(x,y)$$
$$s.t.\ (subject\ to) : g(x,y) = 0$$

- The *Lagrange* function (*Lagrangian*) is defined by:

$$\mathcal{L}(x,y,\lambda) = f(x,y) - \lambda \cdot g(x,y)$$

- Geometric Intuition: let's look at the following figure -
  - 

    - The blue line shows the constraint $g(x,y) = 0$
    - The red lines are contours of $f(x,y) = c$
    - The point where the blue line tangentially touches a red contour is the maximum of $f(x,y) = c$ that satisfy the constraint $g(x,y) = 0$
  - To maximize $f(x,y)$ subject to $g(x,y) = 0$ is to find the largest value $c \in \{7,8,9,10,11\}$ such that the level curve (contour) $f(x,y) = c$ intersects with $g(x,y) = 0$
  - It happens when the curves just touch each other
    - When they have a common tangent line
  - Otherwise, the value of c should be increased
  - Since the gradient of a function is **perperndicular** to the contour lines:
    - The *contour lines* of f and g are **parallel** iff the *gradients* of f and g are **parallel**
    - Thus, we want points (x,y) where $g(x,y) = 0$ and

$$\nabla_{x,y} f(x,y) = \lambda \nabla_{x,y} g(x,y)$$

      - λ - *"The Lagrange Multiplier"* is required to adjust the **magnitudes** of the (parallel) gradient vectors.

Multiple Constraints

- Extenstion of the above for problems with **multiple constraints** using a similar argument
- The general case: minimze $f(x)$ s.t. $g_i(x) = 0, i = 1,2,\ldots,m$
- The **Lagrangian** is a weighted sum of objective and constraint functions:

$$\mathcal{L}(x,\lambda_1,\ldots,\lambda_m) = f(x) - \sum_{i=1}^{m} \lambda_i g_i(x)$$

  - λ_i is the Lagrange multipler associated with $g_i(x) = 0$
- The **solution** is obtained by solving the (unconstrained) optimization problem:

$$\nabla_{x,\lambda_1,\ldots,\lambda_m} \mathcal{L}(x,\lambda_1,\ldots,\lambda_m) = 0 \iff \begin{cases} \nabla_x \left[f(x) - \sum_{i=1}^{m} \lambda_i g_i(x) = 0 \right] \\ g_1(x) = \ldots = g_m(x) = 0 \end{cases}$$

  - Amounts to solving $d + m$ equations in $d + m$ unkowns
    - $d = |x|$ is the dimension of x

Exercise 2 - Max Entropy Distribution

Maximize $H(P) = -\sum_{i=1}^{d} p_i \log p_i$ subject to $\sum_{i=1}^{d} p_i = 1$

Solution 2

- The Lagrangian is:

$$L(P, \lambda) = -\sum_{i=1}^{d} p_i \log p_i - \lambda\big(\sum_{i=1}^{d} p_i - 1\big)$$

- Find stationary point for L:
  - $\forall i,\ \frac{\partial L(P,\lambda)}{\partial p_i} = -\log p_i - 1 - \lambda = 0 \rightarrow p_i = e^{-\lambda - 1}$
  - $\frac{\partial L(P,\lambda)}{\partial \lambda} = -\sum_{i=1}^{d} p_i + 1 = 0 \rightarrow \sum_{i=1}^{d} e^{-\lambda-1} = 1 \rightarrow e^{-\lambda-1} = \frac{1}{d} = p_i$
  - The Max Entropy distribution is the **uniform distribution**

Credits

- Icons from Icon8.com (https://icons8.com/) - https://icons8.com (https://icons8.com)
- Datasets from Kaggle (https://www.kaggle.com/) - https://www.kaggle.com/ (https://www.kaggle.com/)