CSCI 4/5587
Machine Learning I
Chapter 2: Regression

Md Tamjidul Hoque

Example: Regression for given housing dataset.

Bed	Bath	Area (sqft)	Price
5	7.5	5823	695000
4	4	4314	749000
4	4	3800	320000
5	4	3500	38900
3	3	3193	500000
4	4	2962	300000
4	3	2860	439000
3	2	2576	225000
4	3	2480	225000
5	1	2440	204999
3	2	2400	75000
5	3	2367	139000
…	…	…	…

Figure: Snapshot: housing data-set (sorted by: area in descending order).

Terminologies

- **Input**:
  - X_i indicates the i^{th} element in vector **X** (or, X).
  - A set of N input (i.e., observations) p-vectors x_i = 1,…,N is a (N * p) matrix **X**.
  - Vectors are assumed to be column vectors:
    - x^T_i => the i^{th} row of **X** (i.e. the transpose of x_i)
- **Output**:
  - Quantitative *Y* [we use regression to predict]
  - Qualitative *G* (for group) [we use classification to predict]
- **Goal**:
  - Given the value of an input vector *X*, make a good prediction of the output *Y*, *denoted as* $\hat{Y}$ ("y-hat").
  - The prediction should be of the same kind as the searched output (categorical vs. quantitative)
  - Binary outputs can be approximated by values in [0,1], which can be interpreted as probabilities. This also generalizes to k-level outputs.

Linear Models and Least Squares: *Linear Regression*

- Here, Given a vector of inputs $X^T = (X_1, X_2, \ldots, X_p)$, we predict the output Y (i.e., $\hat{Y}$) via the model:

$$\hat{Y} = \hat{\beta}_0 + X_1 \hat{\beta}_1 + X_2 \hat{\beta}_2 + X_3 \hat{\beta}_3 + \ldots + X_p \hat{\beta}_p \qquad (1)$$

In the context of our housing data the X_i is the i^{th} feature of the available observed or training data. Here, $\hat{\beta}_i$ is the parameter or weight or coefficient of the linear equation. Our aim is to determine the best value for $\hat{\beta}_i$.

Equation (1) can be summarized as, $$\hat{Y} = \hat{\beta}_0 + \sum_{j=1}^{p} X_j \hat{\beta}_j \qquad (2)$$

The term $\hat{\beta}_0$ is the intercept, also known as the *bias* in machine learning.

We can also write (1) as: $$\hat{Y} = X_0 \hat{\beta}_0 + X_1 \hat{\beta}_1 + X_2 \hat{\beta}_2 + X_3 \hat{\beta}_3 + \ldots + X_p \hat{\beta}_p \qquad (3)$$

where $X_0 = 1$ is assumed.

Finally: $$\boxed{\hat{Y} = X^T \hat{\beta}} \qquad (4)$$

where, $\hat{\beta} \in \Re^{p+1}$

Fitting the linear model to a set of training data

- To apply *least squares* approach, we pick the coefficients β to minimize the residual sum of squares (RSS), also known as cost function in other context.

$$RSS(\beta) = \sum_{i=1}^{N} \left(y_i - x_i^T \beta\right)^2 \qquad (5)$$

- **Our target** for equation # 5 is to obtain a suitable value of β so that the *RSS*(β) is minimized, i.e.:

$$\min_{\beta} RSS(\beta)$$

Minimization Approaches

- We will discuss 4 different minimization approaches:
  - (1) Newton / Newton-Raphson method (iterative approach)
  - (2) Gradient Descent approach (iterative approach)
  - (3) Genetic Algorithms (iterative approach)
  - (4) Exact method/ Normal Equation (analytic approach).

1. Newton's Method

- Assume, we have an equation $f(x)=0$.

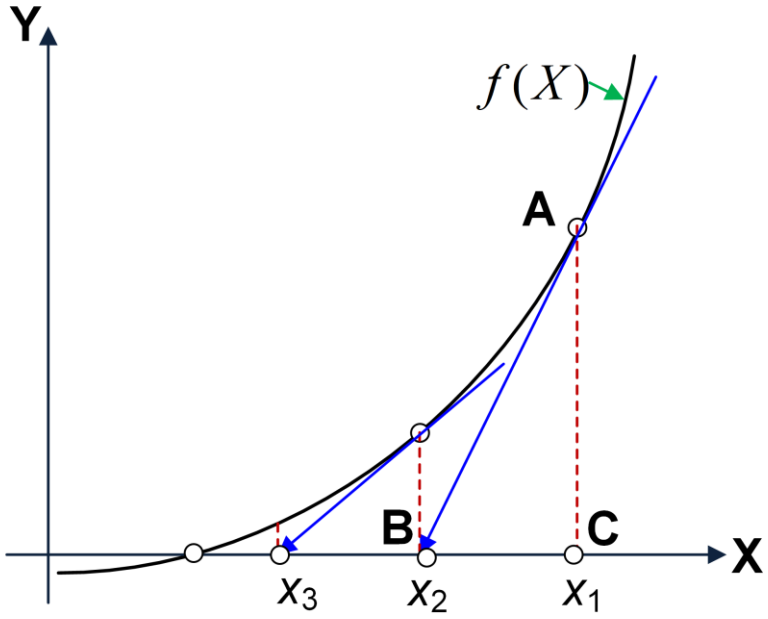
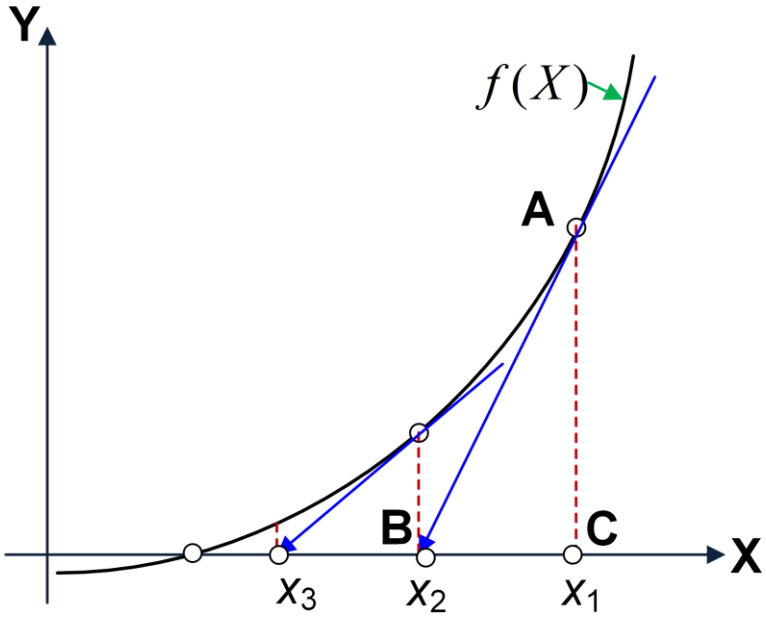


Figure (a): How does Newton's method work in finding the solution?

For the solution (i.e., to find for what value of x, $f(x) = 0$), assume our initial point is x_1 and, X = x_1 intersects x-axis at C and $f(x)$ at A (see Figure (a)). Also, assume that the tangent at A intersects x-axis at B, where the value of x is x_2. From ΔABC and the definition of the slope of an equation, we can write:

... Newton's Method



$$f'(x_1) = \frac{f(x_1) - 0}{x_1 - x_2} \qquad \text{(i)}$$

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} \qquad \text{(ii)}$$

In general, we can write: $$x_{t+1} = x_t - \frac{f(x_t)}{f'(x_t)} \qquad \text{(iii)}$$

Equation (iii) can be iteratively used to get the solution of the equation

Question: Are we after the solution of an equation or the **minimization**?

Answer: Minimization.

Now, if minimum exists then, we actually need to find, the value of *x* for which $f'(x) = 0$.

... Newton's Method

Minimization using newton's method:

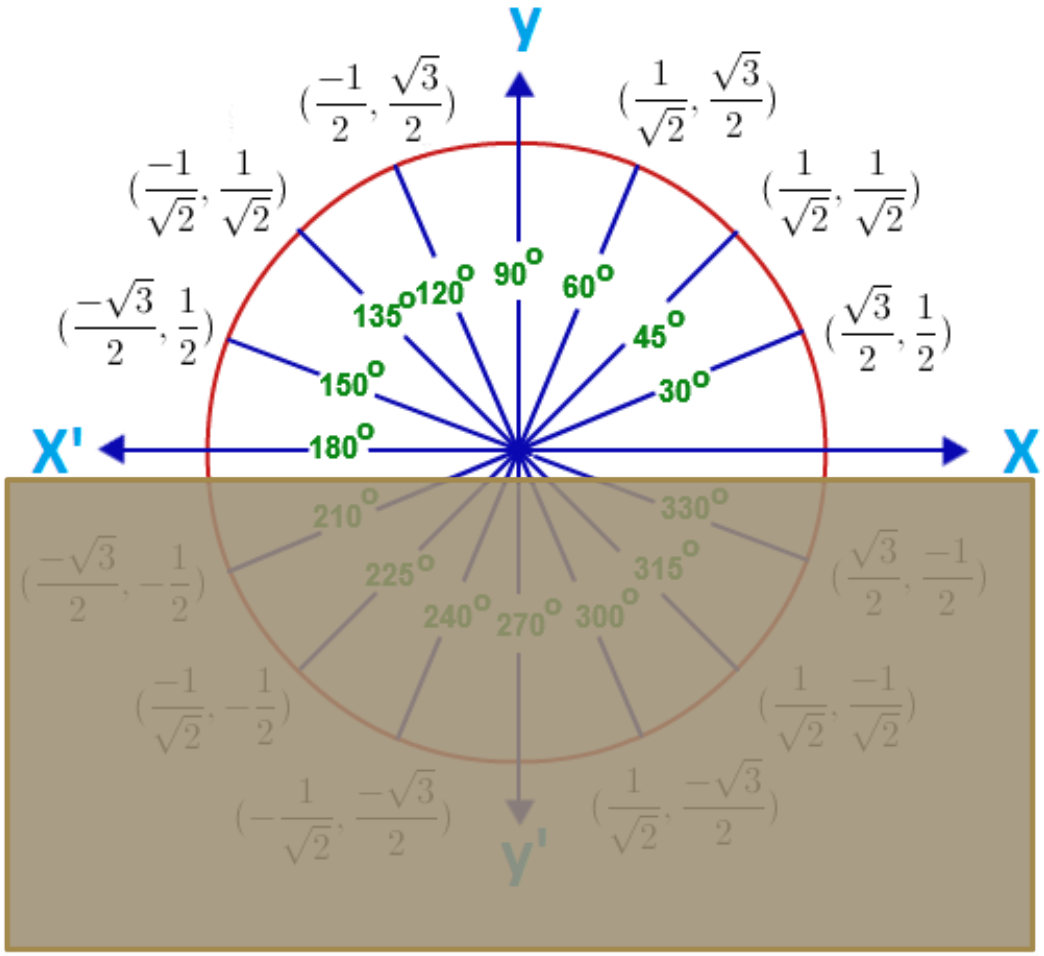
We can use newton's method for solving the equation, $f'(x) = 0$ using (iii).

Therefore, we can similarly write,

$$x_{t+1} = x_t - \frac{f'(x_t)}{f''(x_t)} \qquad \text{(iv)}$$

Caution: For a function of more than one variables (in such case the derivatives are called partial derivatives), say $f(x_1, x_2, ...) = 0$, in each iterations, the updating of each of the individual variables have to be done simultaneously, i.e., the assignment of $x_i(t+1)$ from x_i for $\forall i$ must be simultaneous.

Tan



Angle	Tangent
0	0
15°	.268
30°	.577
45°	1
60°	1.732
75°	3.732
90°	∞
105°	–3.732
120°	–1.732
135°	–1
150°	–.577
165°	–.268
180°	0
195°	.268
210°	.577
225°	1
240°	1.732
255°	3.732
270°	∞
285°	–3.732
300°	–1.732
315°	–1
330°	–.577
345°	–.268
360°	0

2. Gradient Descent

- Gradient descent is a simple equation to get the nearest local minima.

It starts at a point x_0 and moves from x_t (for the 1st point t = 0) to x_{t+1} by minimizing along the line extending from x_t in the direction of negative of the gradient at x_t, i.e., $-\nabla f(x_t)$. The equation is written as:

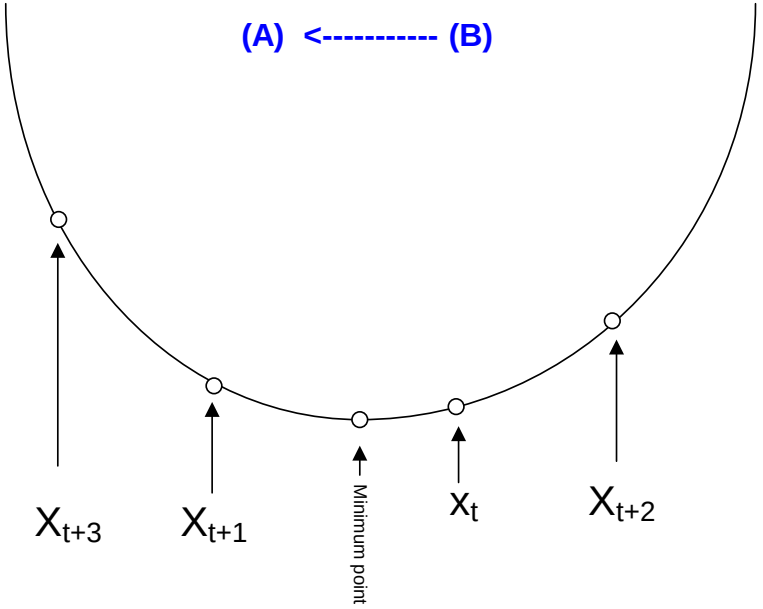
$$x_{t+1} = x_t - \alpha \ \nabla f(x_t) \qquad \text{(v)}$$

where, α is used to control the step size (also called the **learning rate**), and $\alpha > 0$. When it reaches at local minimum the term $\nabla f(x_t)$ becomes 0, therefore equation (v) becomes $x_{t+1} = x_t$.

These methods return the local minimum unless there is only one global minimum.

Overshooting Problem!

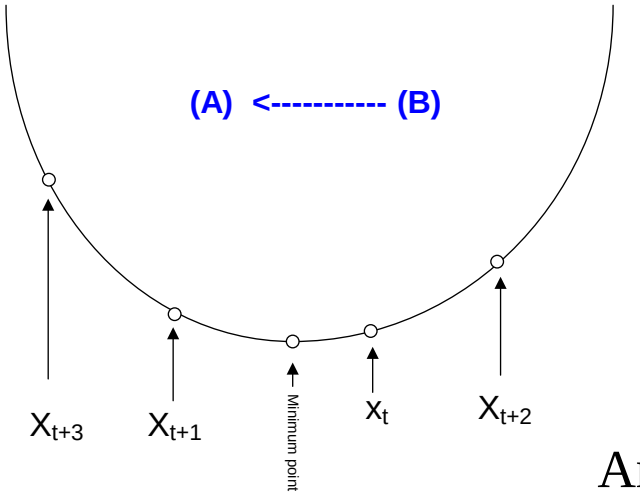
- If we set the value of α (alpha, the learning rate), to a higher value then overshooting might occur. How?



Assume, starting from x_t, with a higher value of α we compute, x_{t+1} as: $x_{t+1} = x_t - \alpha \ \nabla f(x_t)$

Now, x_{t+1} can surpass the minimum point when moving from **(B) to (A) direction** due to higher deduction (amount: $\alpha \ \nabla f(x_t)$) from x_t.

… Overshooting problem / Gradient Descent



From the figure, we see distance of x_{t+1} from the minimum point is higher than the distance of x_t from minimum point. So, it is obvious that:

$$|\nabla f(x_{t+1})| > |\nabla f(x_t)|$$

[in this case $\nabla f(x_t)$ is +ve and $\nabla f(x_{t+1})$ is -ve]

And, obviously $\alpha |\nabla f(x_{t+1})| > \alpha |\nabla f(x_t)|$ will be true

- Based on this information and the figure, we can say that the next point x_{t+2} would be behind x_t as we will apply:

$$x_{t+2} = x_{t+1} - \alpha \ \nabla f(x_{t+1})$$

Again, $\alpha |\nabla f(x_{t+2})| > \alpha |\nabla f(x_{t+1})| > \alpha |\nabla f(x_t)|$ will be true, so the next point x_{t+3} will be behind x_{t+1}, etc.

- Therefore, we see, instead of converging, it is diverging in each iterations which is called the overshooting problem (due to setting higher value of the step-size or α).

3. Genetic Algorithms

- Genetic Algorithm (GA) is a population based optimization algorithm. The formation was inspired by the natural evolution. A pseudo code for GA is given as:

1. Form the initial population (usually random)
2. Compute the fitness to evaluate each chromosomes (member of them population)
3. Select pairs to mate from best-ranked individuals and replenish population
    a. - Apply crossover operator
    b. - Apply mutation operator
4. Check for termination criteria, else go to step #2

Figure: Pseudo code for Genetic Algorithm

- GA can get global minima, however it is not always assured.

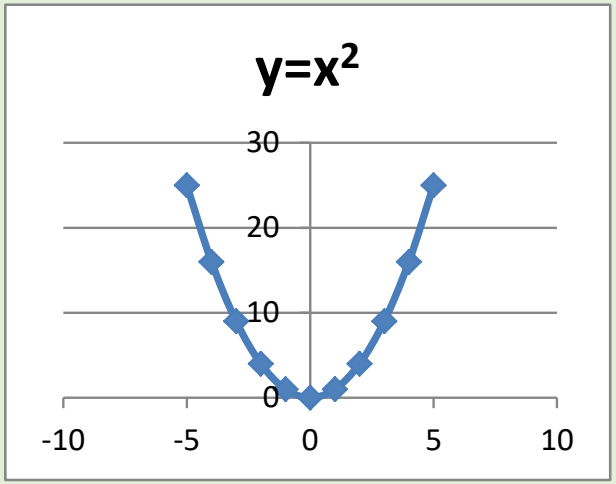
Does the minimum exist?

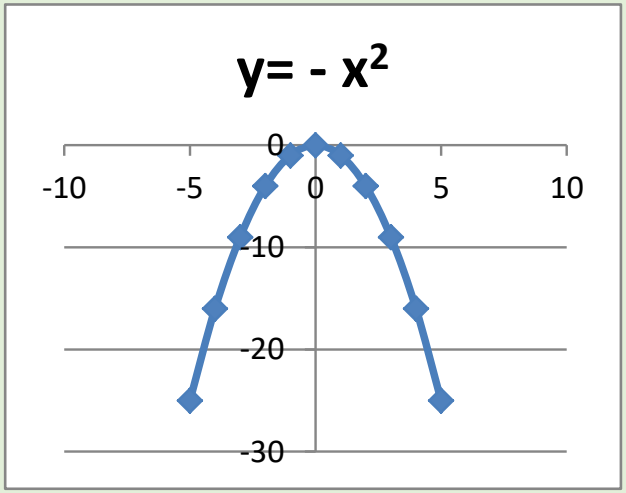
- It is important to ask, does the minimum exist for equation (5):

$$RSS(\beta) = \sum_{i=1}^{N} \left(y_i - x_i^T \beta\right)^2$$

- The answer is 'yes', but again How?

Say, $f(x) => ax^2 + bx + c = 0$ is a quadratic equation and $a \neq 0$,
Now if $a > 0$ then the equation has minimum point [compare $y = x^2 + ...$]
and if $a < 0$ then the equation has maximum point [compare $y = -x^2 + ...$]





Finally the RHS of equation (5) being in the whole square ensures that the coefficient of the (quadratic) variable is always positive and hence the minimum exists.

Next Application Overview

- To find $\min_{\beta} RSS(\beta)$ as our original target,

We will show the application of

(1) Newton's method and

(2) Gradient Descent approach

And then, we will go for method #4 (i.e., the Exact method, or the Normal Equation).

Apply Newton & GD

To find, $\min_{\beta} RSS(\beta)$, we can use Newton's methods as:

$$\beta_j(t+1) = \beta_j(t) - \frac{\frac{\partial}{\partial \beta_j} RSS(\beta)}{\frac{\partial}{\partial \beta_j}\left(\frac{\partial}{\partial \beta_j} RSS(\beta)\right)} \quad (6)$$

For gradient descent the corresponding equation should be:

$$\beta_j(t+1) = \beta_j(t) - \alpha \frac{\partial}{\partial \beta_j} RSS(\beta) \quad (7)$$

To accommodate and to differentiate index i from index j, where $j = \{0, 1, 2, \ldots, p\}$ to present each of the components of β, we rewrite target equation

$RSS(\beta) = \sum_{i=1}^{N} \left(y_i - x_i^T \beta\right)^2$ as $RSS(\beta) = \sum_{i=1}^{N} \left(y(i) - x^T(i)\, \beta\right)^2$.

… Apply Newton & GD

- Now starting common component of RHS of (6) and (7):

$$\frac{\partial}{\partial \beta_j} RSS(\beta) = \frac{\partial}{\partial \beta_j} \sum_{i=1}^{N} \left(y(i) - x^T(i)\,\beta\right)^2$$

$$= 2\sum_{i=i}^{N} \left(y(i) - x^T(i)\,\beta\right). \frac{\partial}{\partial \beta_j}\left(y(i) - x^T(i)\,\beta\right)$$

$$= 2\sum_{i=1}^{N} \left(y(i) - x^T(i)\,\beta\right). \frac{\partial}{\partial \beta_j}\left(y(i) \; - x(i)_0\beta_0 - x(i)_1\beta_1 - \ldots - x(i)_j\beta_j - \ldots\right)$$

$$= 2\sum_{i=1}^{N} \left(x^T(i)\,\beta - y(i)\right). x(i)_j$$

Thus, $\frac{\partial}{\partial \beta_j}\left(\frac{\partial}{\partial \beta_j} RSS(\beta)\right) = \frac{\partial}{\partial \beta_j}\left(2\sum_{i=1}^{N} \left(x^T(i)\,\beta - y(i)\right). x(i)_j\right)$

$$= 2\frac{\partial}{\partial \beta_j}\sum_{i=1}^{N} \left(\left(x(i)_0\beta_0 + x(i)_1\beta_1 + \ldots + x(i)_j\beta_j + \ldots - y(i)\right). x(i)_j\right)$$

$$= 2\sum_{i=1}^{N} \left(x(i)_j \,.\, x(i)_j\right)$$

… Apply Newton & GD

- Using the computed two terms from previous slides we can simplify Equation (6) derived from Newton's method as:

$$\beta_j(t+1) = \beta_j(t) - \frac{\sum_{i=1}^{N} \left(x^T(i)\,\beta - y(i)\right).\, x(i)_j}{\sum_{i=1}^{N} \left(x(i)_j \,.\, x(i)_j\right)} \qquad (8)$$

- Similarly from Equation (7) we can write for gradient descent:

$$\beta_j(t+1) = \beta_j(t) + \frac{2\alpha}{N} \sum_{i=1}^{N} \left(y(i) - x^T(i)\,\beta\right).\, x(i)_j \qquad (9)$$

Note: in Equation (8) and (9), $\frac{1}{N}$ is used to take the average error, but in (8)'s case the numerator and denumerator will have $\frac{1}{N}$ and will cancel each other.

… Apply Newton & GD

Implementation of the Gradient Descent:

Following Equation (9), in terms of MSE(β) we can write the partial derivatives of the cost function:

$$\frac{\partial}{\partial \beta_j} MSE(\beta) = \frac{2}{N} \sum_{i=1}^{N} \{x^T(i)\beta - y(i)\}.x(i)_j \quad \ldots\ldots\ldots\ldots \quad (9a)$$

Instead of computing these partial derivatives individually, we can use the following Equation to calculate them all in one go. The gradient vector noted $\nabla_\beta MSE(\beta)$, contains all the partial derivatives of the cost function (one for each model parameter).

$$\nabla_{\boldsymbol{\beta}} MSE(\boldsymbol{\beta}) = \begin{bmatrix} \frac{\partial}{\partial \beta_0} MSE(\beta) \\ \frac{\partial}{\partial \beta_1} MSE(\beta) \\ \vdots \\ \frac{\partial}{\partial \beta_p} MSE(\beta) \end{bmatrix} = \frac{2}{N} \mathbf{X}^{\mathrm{T}} (\mathbf{X}\boldsymbol{\beta} - \mathbf{Y}) \ldots\ldots\ldots\ldots (9b)$$

Thus, we can define the Gradient Descent step as following (including the learning rate α):

$$\boldsymbol{\beta}(t+1) = \boldsymbol{\beta}(t) - \alpha \nabla_{\boldsymbol{\beta}} MSE(\boldsymbol{\beta}) \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (9c)$$

$$\boldsymbol{\beta}(t+1) = \boldsymbol{\beta}(t) - \alpha \frac{2}{N} \mathbf{X}^{\mathrm{T}} (\mathbf{X}\boldsymbol{\beta} - \mathbf{Y}) \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (9d)$$

… Apply Newton & GD

Alternatively, we can use **Genetic Algorithm** (GA) where, we pick random value(s) for (vector) β and iterate for the values that minimize $RSS(\beta)$.

Note: For all of the three iterative methods, the value of $j = \{0, 1, 2, \ldots, p\}$ and $i = \{1, 2, \ldots, N\}$. Each of the individual values of β will require to be updated simultaneously (at least to be a correct approach theoretically).

We will formulate the exact approach next.

Exact Equation / Non-Iterative Algorithm

Continuing from Equation (5): $RSS(\beta)=\sum_{i=1}^{N}\left(y_i-x_i^T\beta\right)^2$, we can write

$$\mathrm{RSS}(\beta)=(\mathbf{y}-\mathbf{X}\beta)^T(\mathbf{y}-\mathbf{X}\beta) \qquad (10)$$

where **X** is an $N \times p$ matrix with each row an input vector, and **y** is an N-vector of the outputs in the training set. Differentiating w.r.t. β we get the ***normal equations***:

$$\mathbf{X}^T(\mathbf{y}-\mathbf{X}\beta)=0 \qquad (11)$$

Question: How do we get "$\mathbf{X}^T(\mathbf{y}-\mathbf{X}\beta)$" from "$(\mathbf{y}-\mathbf{X}\beta)^T(\mathbf{y}-\mathbf{X}\beta)$"?

Answer: we have, $RSS(\beta)=(\mathbf{y}-\mathbf{X}\beta)^T(\mathbf{y}-\mathbf{X}\beta)$

$$= [\mathbf{y}^T-(\mathbf{X}\beta)^T](\mathbf{y}-\mathbf{X}\beta) \qquad [\because (A\pm B)^T=A^T\pm B^T,\ (AB)^T=B^TA^T]$$

$$=(\mathbf{y}^T-\beta^T\mathbf{X}^T)(\mathbf{y}-\mathbf{X}\beta)$$

$$= \mathbf{y}^T\mathbf{y}-\beta^T\mathbf{X}^T\mathbf{y}-\mathbf{y}^T\mathbf{X}\beta+\beta^T\mathbf{X}^T\mathbf{X}\beta \qquad \left[\begin{array}{l}\because a^Tb=b^Ta\\ \therefore \mathbf{y}^T\mathbf{X}\beta=(\mathbf{X}\beta)^T\mathbf{y}=\beta^T\mathbf{X}^T\mathbf{y}\end{array}\right]$$

… Exact Equation / Non-Iterative Algorithm

$$= \mathbf{y}^T\mathbf{y} - \beta^T\mathbf{X}^T\mathbf{y} - \beta^T\mathbf{X}^T\mathbf{y} + \beta^T\mathbf{X}^T\mathbf{X}\beta$$

$$= \mathbf{y}^T\mathbf{y} - 2\beta^T\mathbf{X}^T\mathbf{y} + \beta^T\mathbf{X}^T\mathbf{X}\beta$$

Therefore, we have, $RSS(\beta) = \mathbf{y}^T\mathbf{y} - 2\beta^T\mathbf{X}^T\mathbf{y} + \beta^T\mathbf{X}^T\mathbf{X}\beta$

Now, differentiating w.r.t. β and equating it to zero, we get:

$$\frac{\partial}{\partial\beta}[\mathbf{y}^T\mathbf{y} - 2\beta^T\mathbf{X}^T\mathbf{y} + \beta^T\mathbf{X}^T\mathbf{X}\beta] = 0$$

$$\Rightarrow \frac{\partial}{\partial\beta}(\mathbf{y}^T\mathbf{y}) - 2\frac{\partial}{\partial\beta}(\beta^T)\mathbf{X}^T\mathbf{y} + \frac{\partial}{\partial\beta}(\beta^T\mathbf{X}^T\mathbf{X}\beta) = 0$$

$$\Rightarrow 0 - 2\mathbf{X}^T\mathbf{y} + \frac{\partial}{\partial\beta}(\beta^T)\mathbf{X}^T\mathbf{X}\beta + \beta^T\mathbf{X}^T\mathbf{X}\frac{\partial}{\partial\beta}(\beta) = 0$$

$$\Rightarrow 0 - 2\mathbf{X}^T\mathbf{y} + \mathbf{X}^T\mathbf{X}\beta + \beta^T\mathbf{X}^T\mathbf{X} = 0$$

$$\begin{bmatrix} \because \beta^T\mathbf{X}^T\mathbf{X} = \beta^T(\mathbf{X}^T\mathbf{X}) \\ = (\mathbf{X}^T\mathbf{X})^T\beta = (\mathbf{X}^T)(\mathbf{X}^T)^T\beta \\ = \mathbf{X}^T\mathbf{X}\beta \end{bmatrix}$$

… Exact Equation / Non-Iterative Algorithm

$$\Rightarrow -2\mathbf{X}^T\mathbf{y} + 2\mathbf{X}^T\mathbf{X}\beta = 0$$

$$\Rightarrow -2\mathbf{X}^T(\mathbf{y} - \mathbf{X}\beta) = 0$$

$$\therefore \mathbf{X}^T(\mathbf{y} - \mathbf{X}\beta) = 0$$

If $\mathbf{X}^T\mathbf{X}$ is nonsingular (or, invertible or, det $(\mathbf{X}^T\mathbf{X}) \neq 0$) then the unique solution is given by:

$$\hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \qquad (12)$$

Note: **Equation (12) is the final form of our "Exact Equation."**

Review

Newton's method:

$$\beta_j(t+1) = \beta_j(t) - \frac{\sum_{i=1}^{N} \left(x^T(i)\,\beta - y(i)\right).x(i)_j}{\sum_{i=1}^{N} \left(x(i)_j \,.\, x(i)_j\right)}$$

Gradient descent:

$$\beta_j(t+1) = \beta_j(t) + \frac{2\alpha}{N} \sum_{i=1}^{N} \left(y(i) - x^T(i)\,\beta\right).x(i)_j$$

Gradient descent in vector/matrix form:

$$\boldsymbol{\beta}(t+1) = \boldsymbol{\beta}(t) - \alpha \frac{2}{N} \mathbf{X}^{\mathrm{T}}(\mathbf{X}\boldsymbol{\beta} - \mathbf{Y})$$

Exact equation:

$$\hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\,\mathbf{X}^T\,\mathbf{y}$$

Exercise: Application of Exact Equation

- I have placed the data into X.txt and Y.txt. Note: X.txt contains a starting column with all ‘1’s.
- MATLAB/Octave code:
  - load X.txt
  - load Y.txt
  - B=inv(X'*X)*X'*Y
- Answers:

```
B =
  3.3868e+004
  -3.6762e+004
  1.0501e+004
  1.3291e+002
```

For a Query, Q=[1 5 3 2500] for example, we can predict the house price by:

Q*B; [which should return 2.1384e+005 or, 213,840]

… Exercise: Application of Exact Equation

- If $\mathbf{X}^T\mathbf{X}$ is **singular**:
  - then function ***inv()*** does not work.
  - In such a case, we can use:
    - ***pinv()*** function, which is a **pseudoinverse** function.
    - That is, we write B=**pinv**(X'*X)*X'*Y instead of B=inv(X'*X)*X'*Y.

- How is the *pseudoinverse* computed?

… Exercise: Application of Exact Equation

- Assume, the *pseudoinverse* of a matrix X is denoted as X^+.

- To compute X^+ :
  - Compute SVD (*Singular Value Decomposition*) of $X = \mathbf{U} \Sigma \mathbf{V}^T$
  - Then, compute Σ^+ from Σ as:
    - Take Σ and set to zero all values smaller than a tiny threshold value,
    - then replace all the nonzero values with their inverse, and
    - finally, transpose the resulting matrix and obtain Σ^+.

- Then, the *pseudoinverse* of X is computed as $X^+ = \mathbf{V} \Sigma^+ \mathbf{U}^T$

- Note: for SVD in python see numpy.linalg.svd().

Part2: Normal Equation versus Gradient Descent

Computational Complexity of the Normal Equation:

- The Normal Equation computes the inverse of $\mathbf{X}^T\mathbf{X}$, which is a $(p + 1) \times (p + 1)$ matrix (where p is the number of features).
- The computational complexity of inverting such a matrix is typically about $O(p^{2.4})$ to $O(p^3)$, depending on the implementation.
- In other words, if you double the number of features, you multiply the computation time by roughly $2^{2.4} = 5.3$ to $2^3 = 8$.
- The SVD approach, used by Scikit-Learn's **LinearRegression** class is about $O(p^2)$. Here, If you double the number of features, you multiply the computation time by roughly 4.

… Computational Complexity of the Normal Equation

- Both the Normal Equation and the SVD approach get very slow when the number of features grows large (e.g., 100,000).
- On the positive side, both are linear with regard to the number of instances in the training set (they are $O(N)$, where N is the number of instances or samples), so they handle large training sets efficiently, provided they can fit in memory.
- Also, once you have trained your Linear Regression model (using the Normal Equation or any other algorithm), predictions are very fast: the computational complexity is linear with regard to both the number of instances you want to make predictions on and the number of features.
  - In other words, making predictions on twice as many instances (or twice as many features) will take roughly twice as much time.

… Computational Complexity of the Normal Equation

- Next, we will look at a very different way to train a Linear Regression model, such as Gradient Descent.
- Gradient Descent is better suited for cases where there are:
  - a large number of features or
  - too many training instances to fit in memory.

Gradient Descent

- Cost functions (such as RSS(β) or MSE(β)) may not always look like nice, regular bowls.
- There may be holes, ridges, plateaus, and all sorts of irregular terrains, making convergence to the minimum difficult.
- Figure below shows the main challenges with Gradient Descent:

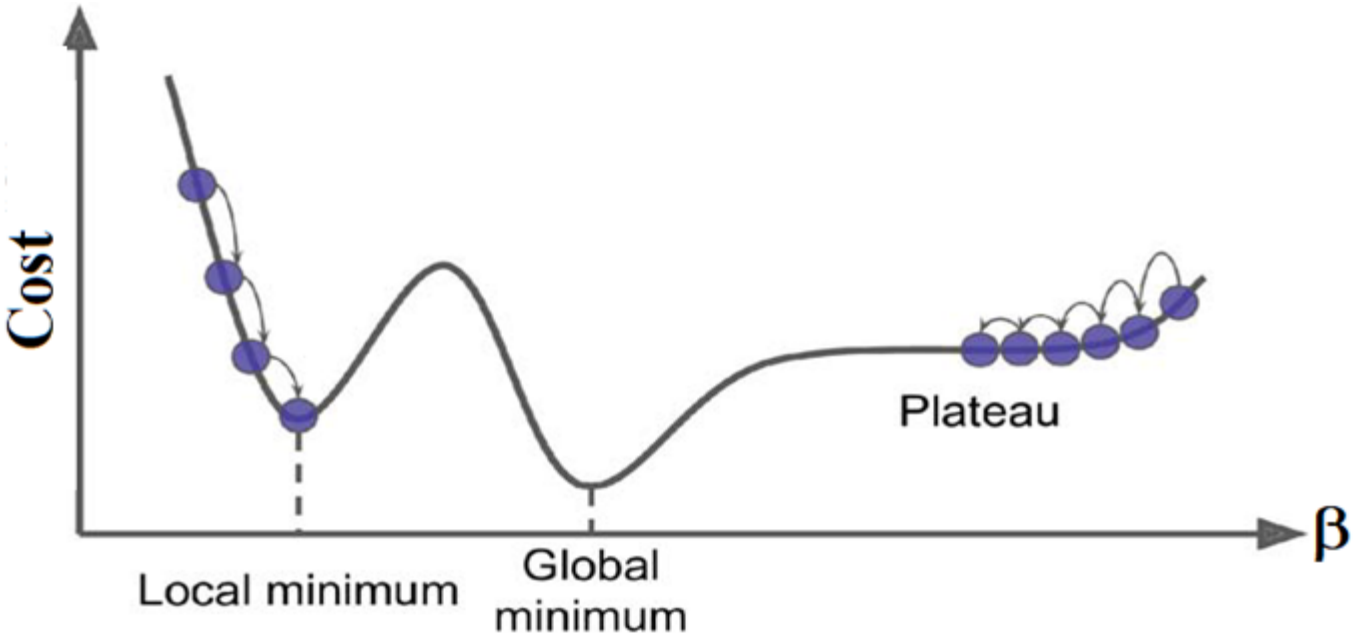


Figure: Gradient Descent pitfalls.

… Gradient Descent

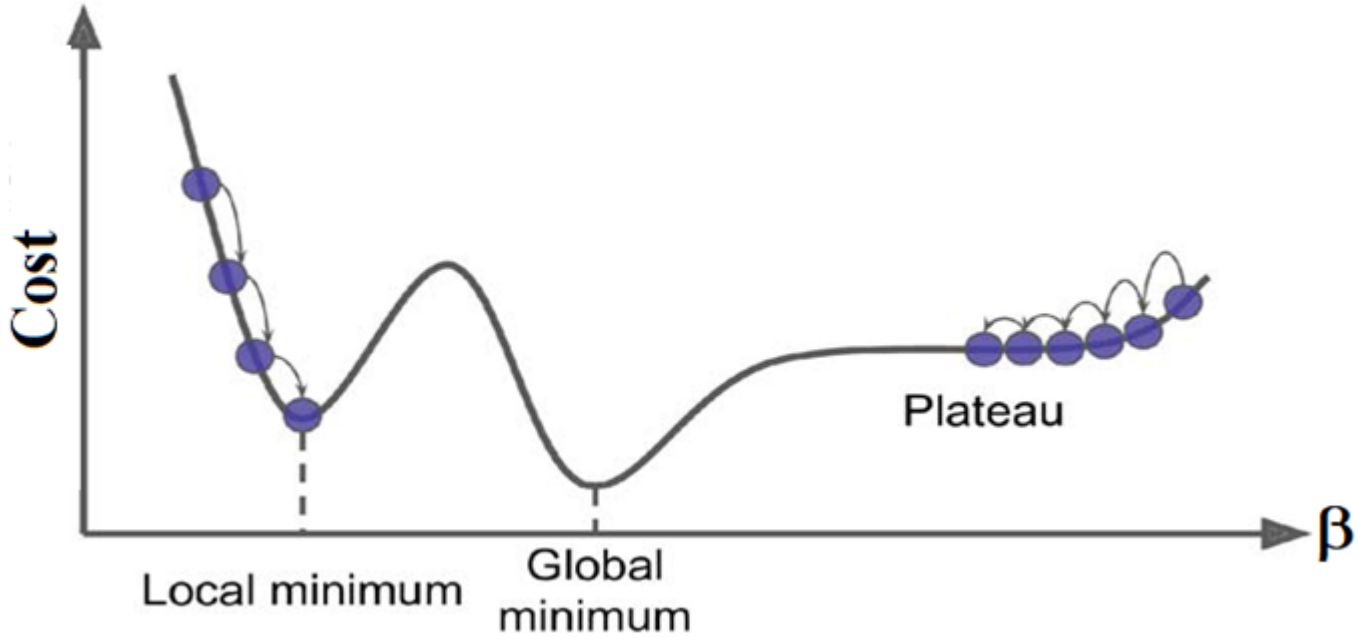
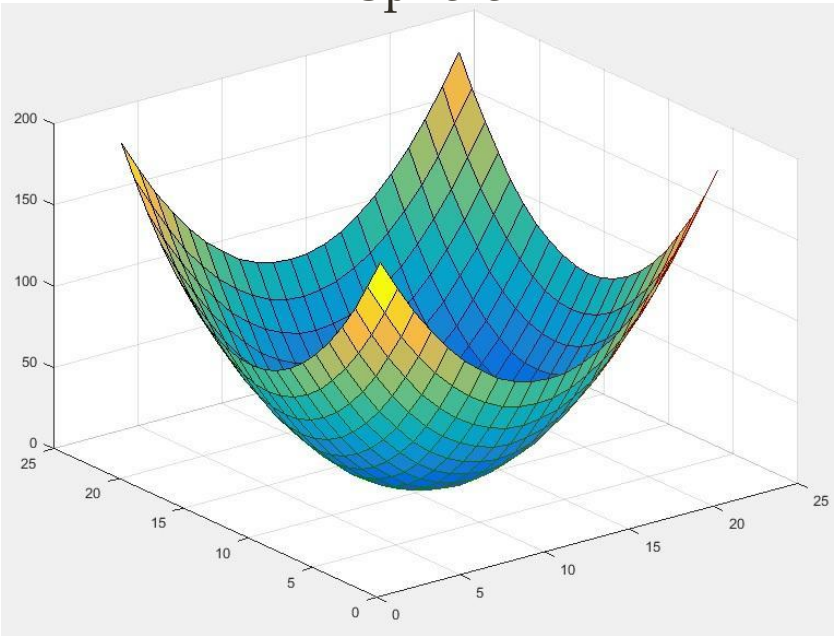


Figure: Gradient Descent pitfalls.

- If the random initialization starts the algorithm on the left, then it will converge to a *local minimum*, which is not as good as the *global minimum*.
- If it starts on the right, then it will take a very long time to cross the *plateau*.
- And if you stop too early, you will never reach the global minimum.

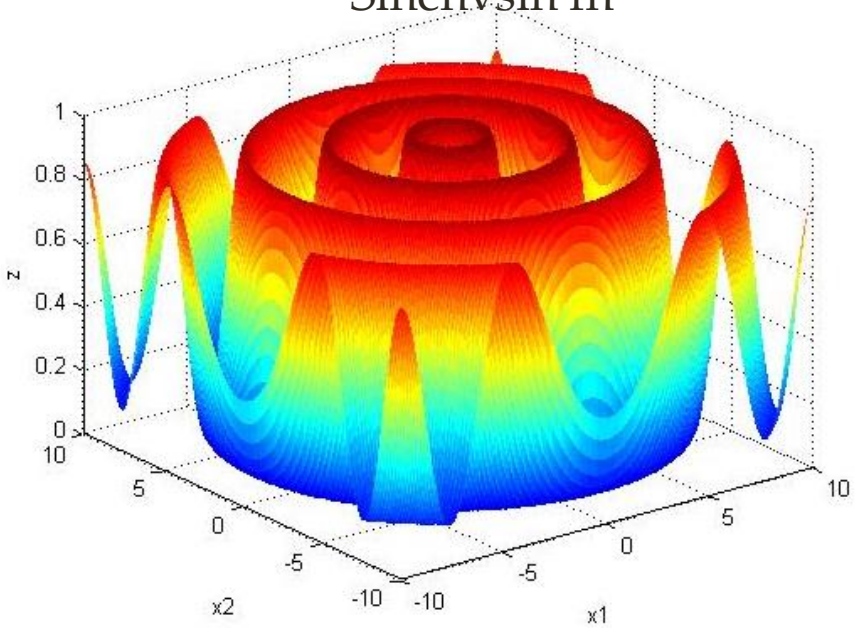
Some Math Functions and their 3D Plots

Sphere fn

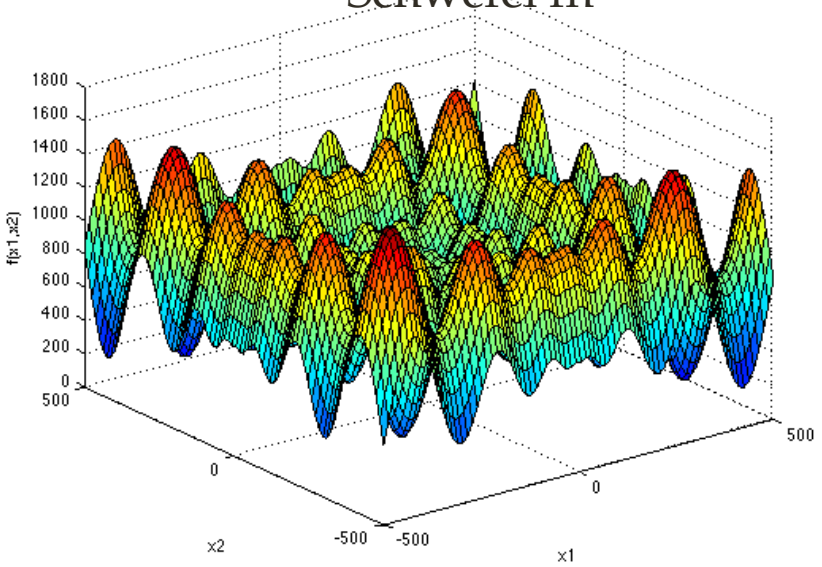


$$f(\mathbf{x}) = \sum_{i=1}^{d} (x_i)^2$$

Sinenvsin fn

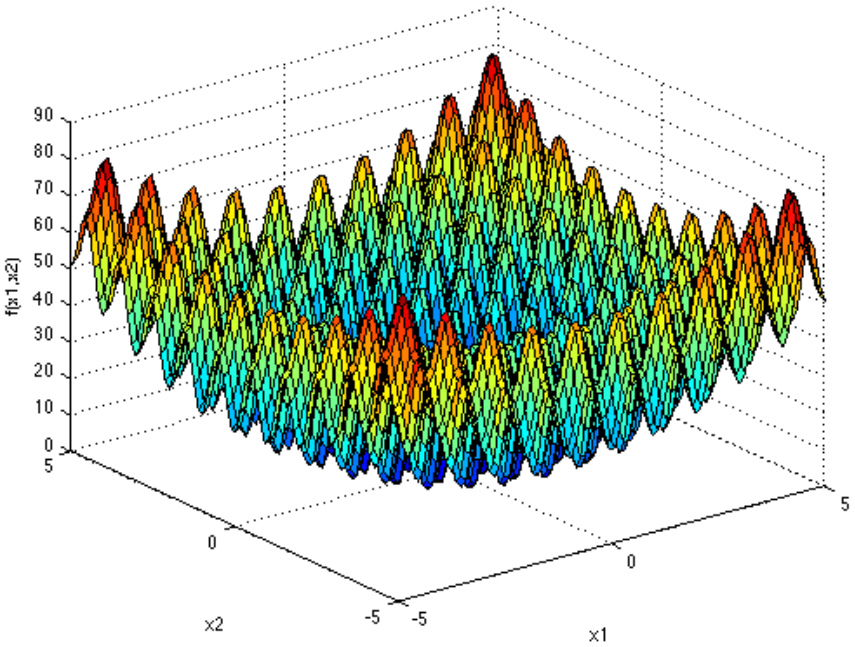


Schwefel fn

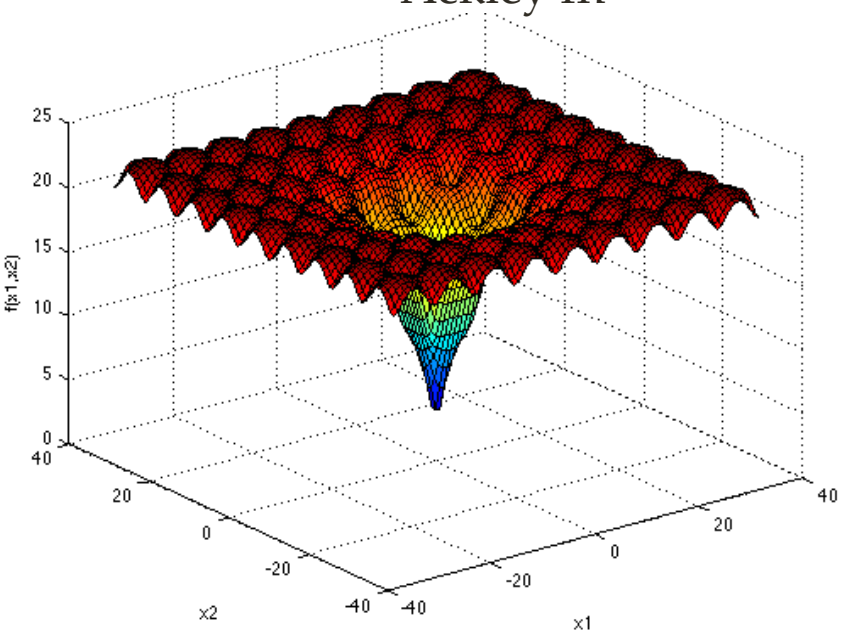


...Some Math Functions and their 3D Plots

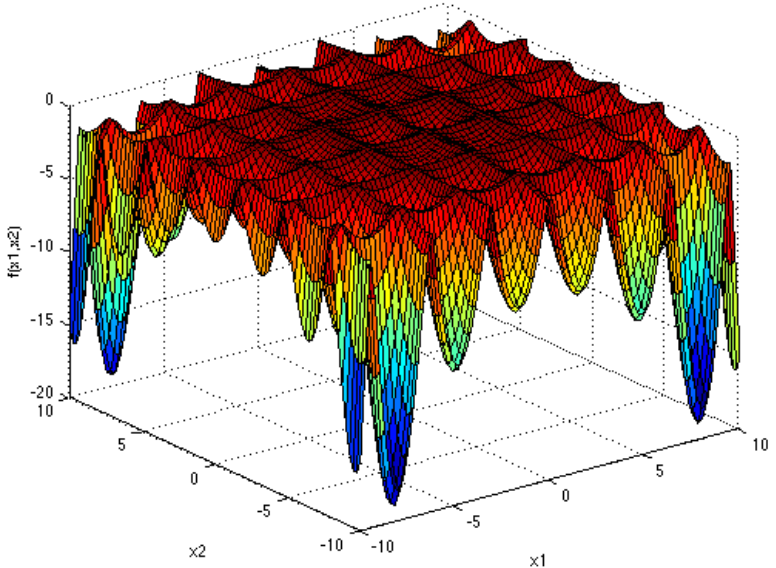
Rastrigin fn



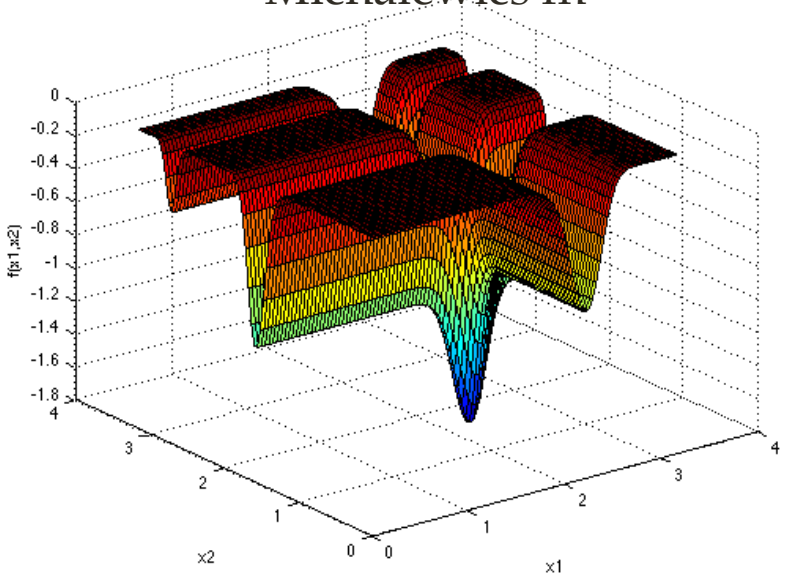
Ackley fn



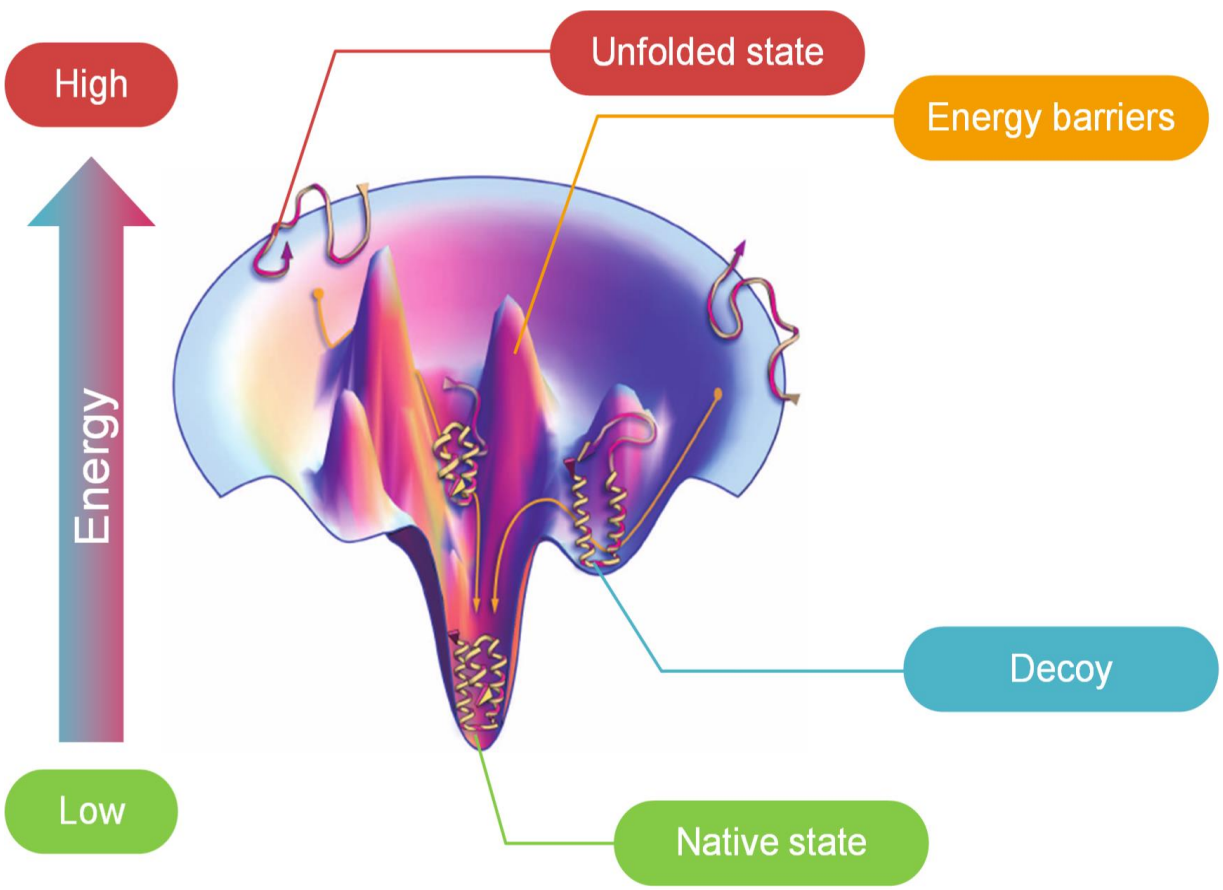
Holder Table fn



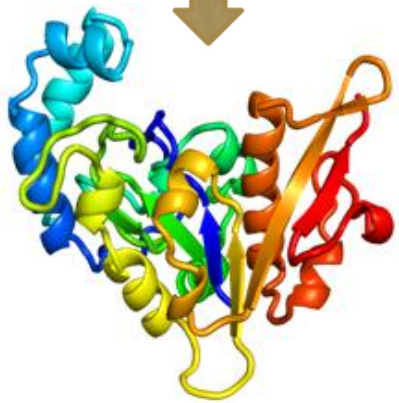
Michalewics fn



Schematic Representation of Protein Free-Energy Landscape



Primary Structure
(Sequence)



Folded 3D Protein
Structure

Figure: Proteins have a **funnel-shaped** energy landscape with many high-energy, unfolded structures and only a few low-energy, folded structures. Folding occurs via alternative microscopic trajectories.

… Gradient Descent

- Fortunately, the RSS(β) or MSE(β) cost function for a Linear Regression model happens to be a *convex* function.
  - Convex implies that if you pick any two points on the curve, the line segment joining them never crosses the curve.
  - This implies that there are no local minima, just one global minimum.
  - It is also a continuous function with a slope that never changes abruptly.
- These facts have a great consequence:
  - Gradient Descent is guaranteed to approach arbitrarily close the global minimum (if you wait long enough and if the learning rate is not too high).

… Gradient Descent

- In fact, the cost function has the shape of a bowl, but it can be an elongated bowl if the features have very different scales.

- Figure below shows Gradient Descent on a training set where features 1 and 2 have the same scale (on the left), and on a training set where feature 1 has much smaller values than feature 2 (on the right).

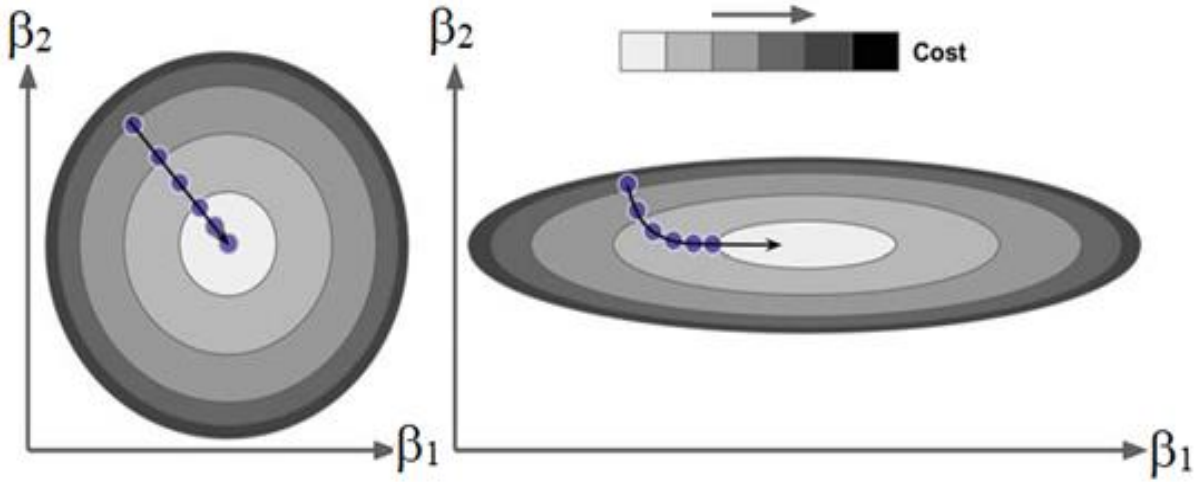


Figure: Gradient Descent with (left) and without (right) feature scaling.

- As you can see, on the left the Gradient Descent algorithm goes straight toward the minimum, thereby reaching it quickly,

- whereas on the right it first goes in a direction almost orthogonal to the direction of the global minimum, and it ends with a long march down an almost flat valley. It will eventually reach the minimum, but it will take a long time.

… Gradient Descent

- Thus, you should ensure that all features have a similar scale (e.g., using Scikit-Learn's StandardScaler class), or else it will take much longer to converge.

Batch Gradient Descent

- Batch Gradient Descent uses the whole batch of training data at every step.
- As a result, it is terribly slow on very large training sets.
- However, Gradient Descent scales well with the number of features; training a Linear Regression model when there are hundreds of thousands of features is much faster-using Gradient Descent (GD) than using the Normal Equation or SVD decomposition.
- The performance of GD dependents on learning rate:
  - If the learning rate is too low: the algorithm will eventually reach the solution, but it will take a long time.
  - If the learning rate fits well then in just a few iterations, GD converges to the solution.
  - If the learning rate is too high: the algorithm diverges, jumping all over the place and getting further and further away from the solution at every step.

… Batch Gradient Descent

- To find a good ***learning rate* (LR)**, you can use grid search.
- However, you may want to limit the number of iterations so that grid search can eliminate models that take too long to converge.
- If the LR is too low, you will still be far away from the optimal solution when the algorithm stops;
- but if it is too high, you will waste time while the model parameters do not change anymore.
- A simple solution is to set a very large number of iterations but to interrupt the algorithm when the gradient vector becomes tiny—
  - that is, when its norm becomes smaller than a tiny number ε (called the tolerance)—because this happens when Gradient Descent has (almost) reached the minimum.

Stochastic Gradient Descent (SGD)

- The main problem with Batch Gradient Descent is the fact that it uses the whole training set to compute the gradients at every step, which makes it very slow when the training set is large.
- At the opposite extreme, Stochastic Gradient Descent (SGD) picks a random instance in the training set at every step and computes the gradients based only on that single instance.
- Obviously, working on a single instance at a time makes the algorithm much faster because it has very little data to manipulate at every iteration.
- SGD also makes it possible to train on huge training sets, since only one instance needs to be in memory at each iteration.
- Thus, SGD can be implemented as an out-of-core algorithm.

… Stochastic Gradient Descent (SGD)

- But due to its stochastic (i.e., random) nature, SGD is much less regular than Batch Gradient Descent:
  - instead of gently decreasing until it reaches the minimum, the cost function will bounce up and down, decreasing only on average.
  - Over time it will end up very close to the minimum, but once it gets there it will continue to bounce around, never settling down (see **Figure below**).
  - So once the algorithm stops, the final parameter values are good, but not optimal.

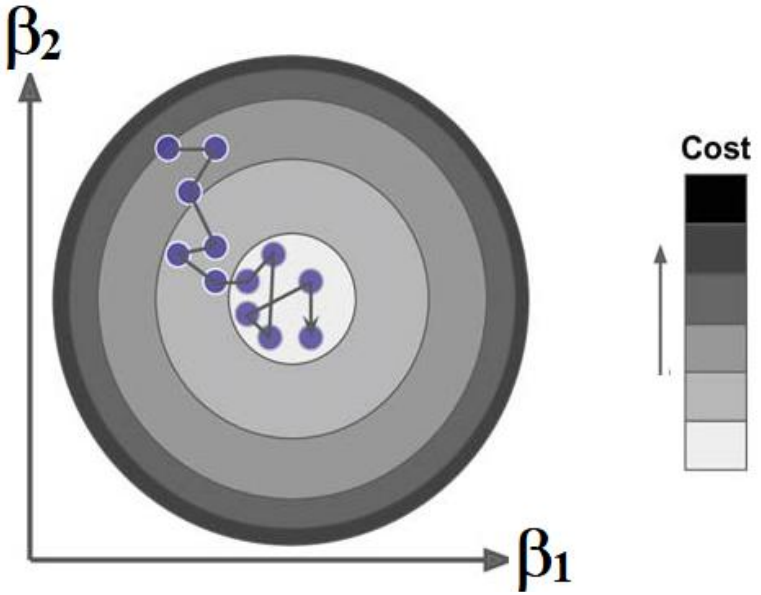


Figure: With Stochastic Gradient Descent, each training step is much faster but also much more stochastic than when using Batch Gradient Descent.

… Stochastic Gradient Descent (SGD)

- When the cost function is very irregular (has holes, ridges, plateaus, and all sorts of irregular terrains …), stochastic nature of SGD can help the algorithm jump out of local minima,
- Thus, SGD has a better chance of finding the global minimum than Batch Gradient Descent does for complex cases.
- Therefore, randomness is good to escape from local optima, but bad because it means that the algorithm can never settle at the minimum.
- One solution to this dilemma is to gradually reduce the learning rate.
  - The steps start out large (which helps make quick progress and escape local minima), then get smaller and smaller, allowing the algorithm to settle at the global minimum.
  - The function that determines the learning rate at each iteration is called the ***learning schedule***.

… Stochastic Gradient Descent (SGD)

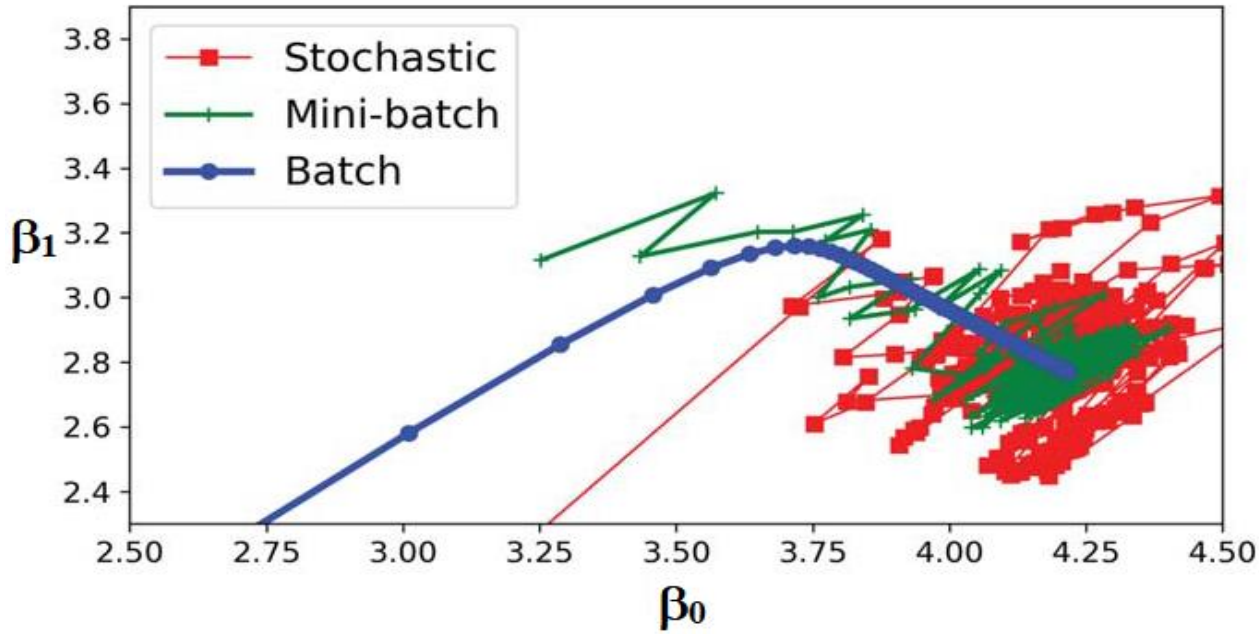
- In the *learning schedule*, if the *learning rate* is reduced too quickly, you may get stuck in a local minimum, or even end up frozen halfway to the minimum.
- If the learning rate is reduced too slowly, you may jump around the minimum for a long time and end up with a *suboptimal solution* if you halt training too early.
- Practically, while the Batch Gradient Descent code iterates 1,000 times through the whole training set (i.e., N samples per epoch), Stochastic Gradient Descent (SGD) code can go through the training set only 50 times (i.e., 50 epochs) and reaches a pretty good solution [see the hands-on exercise].

Mini-batch Gradient Descent

- At each step, Mini-batch GD computes the gradients on small *random sets* of instances called *mini-batches*.
- The main advantage of Mini-batch GD over Stochastic GD is that you can get a performance boost from hardware optimization of matrix operations, especially when using GPUs/TPUs.
- Mini-batch GD progress in parameter space is less erratic than with Stochastic GD, especially with fairly large mini-batches.
- As a result, Mini-batch GD will end up walking around a bit closer to the minimum than Stochastic GD.
- But it may be harder for Mini-batch GD to escape from local minima (in the case of problems that suffer from local minima, unlike Linear Regression).

BGD vs. SGD vs. Mini-batch GD

Figure: Gradient Descent paths in parameter space.



- Figure shows the paths taken by the three Gradient Descent algorithms in parameter space during training.
- They all end up near the minimum,
  - but Batch GD's path actually stops at the minimum,
  - while both Stochastic GD and Mini-batch GD continue to walk around.
  - However, don't forget that Batch GD takes a lot of time to take each step, and Stochastic GD and Mini-batch GD would also reach the minimum if you used a good learning schedule.

Comparison of All algorithms for Linear Regression

- We compare the algorithms we've discussed so far for Linear Regression (recall that N is the number of training instances and p is the number of features) in Table below:

Table: Comparison of algorithms for Linear Regression.

Algorithm	Large N	Out-of-core support	Large p	Hyperparams	Scaling required	Scikit-Learn
Normal Equation	Fast	No	Slow	0	No	N/A
SVD	Fast	No	Slow	0	No	`LinearRegression`
Batch GD	Slow	No	Fast	2	Yes	`SGDRegressor`
Stochastic GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`
Mini-batch GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`