

Machine Learning

June 12, 2025

1 Types of Losses

A loss function is a mathematical formula that measures how far a model's predictions are from the actual values; it tells the model how wrong it is. Lower loss means better performance

1.1 Regression Losses

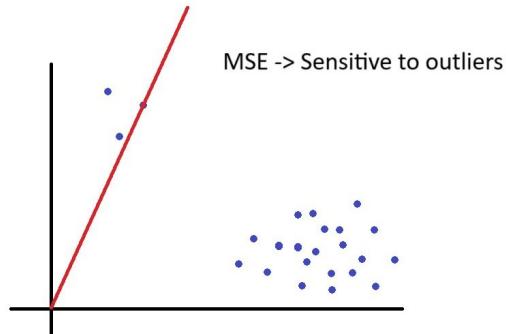
Regression losses are loss functions used in regression tasks (where the goal is to predict continuous values)

1.1.1 Mean Squared Error

$$L = (y_i - \hat{y}_i)^2$$

Disadvantages:

- **Sensitive to Outliers:** Squaring the errors gives more weight to large errors, which can distort the model if outliers are present.
- **Over-penalizes Large Errors:** A single big error can dominate the loss, possibly causing overfitting or poor model behavior.



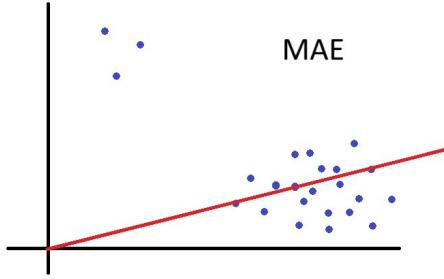
1.1.2 Mean Absolute Error

$$L = |y_i - \hat{y}_i|$$

It is robust to outliers.

Disadvantages:

- **Not Easily Differentiable** MAE's non-differentiability at zero can make optimization challenging for gradient-based methods, slowing down model training.



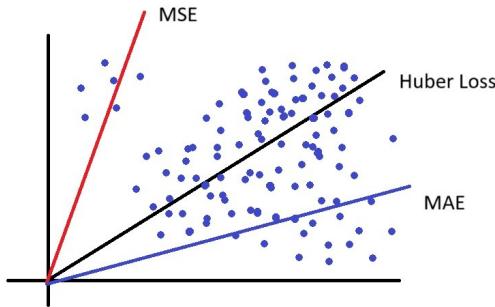
1.1.3 Huber Loss

$$L_\delta(a) = \begin{cases} \frac{1}{2}(y_i - \hat{y}_i)^2, & \text{for } |y_i - \hat{y}_i| \leq \delta \\ \delta(|y_i - \hat{y}_i| - \frac{1}{2}\delta), & \text{otherwise} \end{cases}$$

δ –> Hyperparameter

Disadvantages:

- A key disadvantage of Huber Loss is that it introduces a hyperparameter (delta) to control the transition between MSE and MAE, and choosing an optimal value for delta can be challenging and dataset-dependent.



1.2 Classification Losses

Classification losses are used in tasks where the goal is to predict categorical labels. They measure how well the model predictions match the true class labels.

1.2.1 Binary Cross Entropy

$$L = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$$

Disadvantages:

- **Multiple Local Minima** Multiple local minima in Binary Cross-Entropy Loss can cause optimization algorithms to get stuck in suboptimal solutions, slowing convergence.

1.2.2 Categorical Cross Entropy

$$L = \sum_{j=1}^k y_j \log(\hat{y}_j)$$

One hot encode the output classes.

Disadvantages:

- **Requirement for One-Hot Encoding** It increases memory usage and can be inefficient for tasks with many classes.

2 Linear Regression

Linear Regression is a statistical method used to model the relationship between a dependent variable and one or more independent variables by fitting a linear equation to observed data. It predicts outcomes by minimizing the difference between actual and predicted values using a best-fit line.

Distribution Model: We assume that the target variable y is generated from a linear model with Gaussian noise

$$y^i = \theta^T x^i + \epsilon^i, \quad \epsilon^i \sim N(0, \sigma^2)$$

This implies that

$$\begin{aligned} p(y^{(i)} | x^{(i)}; \theta) &= \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right). \\ L(\theta) &= \prod_{i=1}^n p(y^{(i)} | x^{(i)}; \theta) \\ &= \prod_{i=1}^n \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right). \\ \ell(\theta) &= \log L(\theta) \\ &= \log \prod_{i=1}^n \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right) \\ &= \sum_{i=1}^n \log \left(\frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right) \right) \\ &= n \log \frac{1}{\sqrt{2\pi}\sigma} - \frac{1}{\sigma^2} \cdot \frac{1}{2} \sum_{i=1}^n (y^{(i)} - \theta^T x^{(i)})^2. \end{aligned}$$

Hence, maximizing $\ell(\theta)$ gives the same answer as minimizing $\frac{1}{2} \sum_{i=1}^n (y^{(i)} - \theta^T x^{(i)})^2$.

$$\frac{\partial}{\partial \theta} \left(\frac{1}{2} \sum_{i=1}^n (y^{(i)} - \theta^T x^{(i)})^2 \right) = - \sum_{i=1}^n (y^{(i)} - \theta^T x^{(i)}) x^{(i)}.$$

Using gradient descent: $\theta := \theta - \alpha \nabla_{\theta} J(\theta)$,

$$\text{where } \nabla_{\theta} J(\theta) = - \sum_{i=1}^n (y^{(i)} - \theta^T x^{(i)}) x^{(i)}.$$

Therefore, for a single training example,

$$\theta_j := \theta_j + \alpha \left(y^{(i)} - h_{\theta}(x^{(i)}) \right) x_j^{(i)}.$$

Algorithm 1 Linear Regression

Require: Feature matrix $X \in R^{m \times n}$, target vector $y \in R^m$, learning rate α , number of iterations N

Ensure: Parameters $\theta \in R^n$

- 1: Initialize $\theta := \vec{0}$
- 2: **for** iter = 1 to N **do**
- 3: $h := X \cdot \theta$
- 4: $error := h - y$
- 5: $gradient := (y^{(i)} - h_{\theta}(x^{(i)})) x_j^{(i)}$
- 6: $\theta := \theta + \alpha \cdot gradient$
- 7: **end for**
- 8: **return** $\theta = 0$

Batch Gradient Descent - We scan through the entire training dataset before taking a single step

$$\theta_j := \theta_j + \sum_{i=1}^m \alpha \left(y^{(i)} - h_\theta(x^{(i)}) \right) x_j^{(i)}.$$

Can lead to global optimum

Stochastic Gradient Descent - Just look at the current datapoint and take a single step

$$\theta_j := \theta_j + \alpha \left(y^{(i)} - h_\theta(x^{(i)}) \right) x_j^{(i)}.$$

May end up oscillating

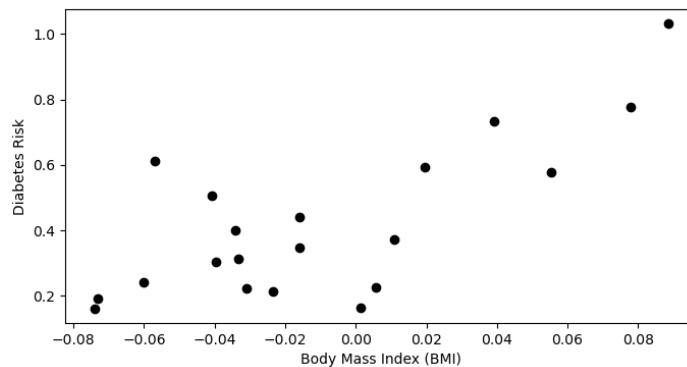
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn import datasets

# Load the diabetes dataset
X, y = datasets.load_diabetes(return_X_y=True, as_frame=True)

X['one'] = 1

# Collect 20 data points
X_train = X.iloc[-20:].loc[:, ['bmi', 'one']]
y_train = y.iloc[-20:] / 300

plt.scatter(X_train.loc[:, ['bmi']], y_train, color='black')
plt.xlabel('Body Mass Index')
plt.ylabel('Diabetes Risk')
```



Recall that a linear model has the form

$$y = \theta_0 + \theta_1 \cdot x_1 + \theta_2 \cdot x_2 + \dots + \theta_d \cdot x_d$$

where $\mathbf{x} \in R^d$ is a vector of features and y is the target. The θ_j are the *parameters* of the model.

By using the notation $x_0 = 1$, we can represent the model in a vectorized form

$$f_\theta(\mathbf{x}) = \sum_{j=0}^d \theta_j \cdot x_j = \theta^\top \mathbf{x}.$$

```
def f(X, theta):
    return X.dot(theta)

def mean_squared_error(theta, X, y):
    return 0.5 * np.mean((y - f(X, theta))**2)

def mse_gradient(theta, X, y):
```

```

    return np.mean((f(X, theta) - y) * X.T, axis=1)

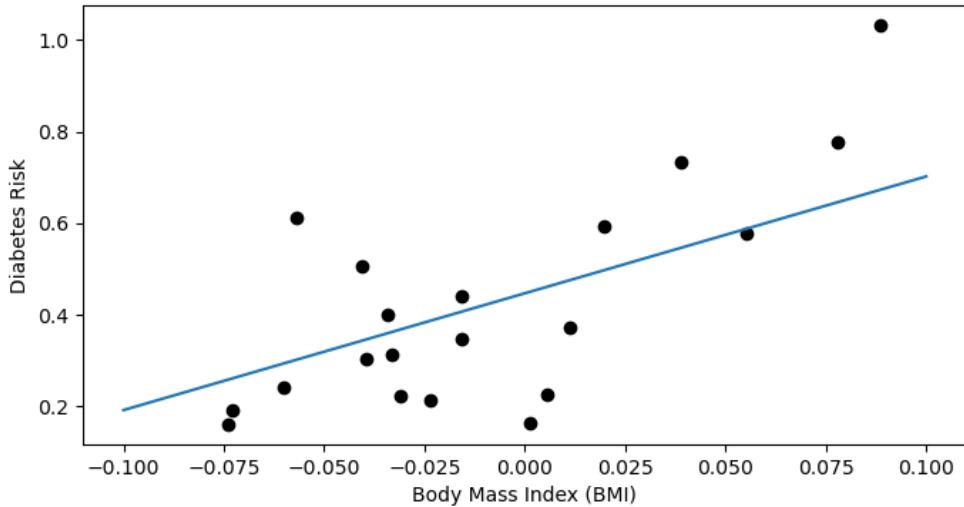
threshold = 1e-3
step_size = 4e-1
theta, theta_prev = np.array([2, 1]), np.ones(2,)
opt_pts = [theta]
opt_grads = []
iter = 0

while np.linalg.norm(theta - theta_prev) > threshold:
    if iter % 100 == 0:
        print('Iteration %d. MSE: %.6f' % (iter, mean_squared_error(theta, X_train, y_train)))
    theta_prev = theta
    gradient = mse_gradient(theta, X_train, y_train)
    theta = theta_prev - step_size * gradient
    opt_pts += [theta]
    opt_grads += [gradient]
    iter += 1

x_line = np.stack([np.linspace(-0.1, 0.1, 10), np.ones(10,)])
y_line = opt_pts[-1].dot(x_line)

plt.scatter(X_train.loc[:, ['bmi']], y_train, color='black')
plt.plot(x_line[0], y_line)
plt.xlabel('Body Mass Index (BMI)')
plt.ylabel('Diabetes Risk')

```



3 Logistic Regression

Logistic Regression is a statistical method used to model the probability that a given input belongs to a particular class (typically binary). It is used for classification tasks by modeling the probability of the output variable as a function of the input features using the logistic (sigmoid) function.

Distribution Model: We assume that the target variable y is generated from a Bernoulli distribution:

$$y^{(i)} \sim \text{Bernoulli}(p^{(i)}), \quad \text{where } p^{(i)} = \sigma(\theta^T x^{(i)}), \quad \sigma(z) = \frac{1}{1 + e^{-z}}$$

This implies that

$$\begin{aligned} p(y^{(i)}|x^{(i)}; \theta) &= [\sigma(\theta^T x^{(i)})]^{y^{(i)}} [1 - \sigma(\theta^T x^{(i)})]^{1-y^{(i)}} \\ L(\theta) &= \prod_{i=1}^n p(y^{(i)}|x^{(i)}; \theta) = \prod_{i=1}^n [\sigma(\theta^T x^{(i)})]^{y^{(i)}} [1 - \sigma(\theta^T x^{(i)})]^{1-y^{(i)}} \\ \ell(\theta) &= \log L(\theta) = \sum_{i=1}^n \left(y^{(i)} \log \sigma(\theta^T x^{(i)}) + (1 - y^{(i)}) \log(1 - \sigma(\theta^T x^{(i)})) \right) \end{aligned}$$

Hence, maximizing $\ell(\theta)$ is equivalent to minimizing the negative log-likelihood:

$$\begin{aligned} J(\theta) &= -\ell(\theta) = - \sum_{i=1}^n \left(y^{(i)} \log \sigma(\theta^T x^{(i)}) + (1 - y^{(i)}) \log(1 - \sigma(\theta^T x^{(i)})) \right) \\ \frac{\partial}{\partial \theta} J(\theta) &= - \sum_{i=1}^n \left(y^{(i)} - \sigma(\theta^T x^{(i)}) \right) x^{(i)} \end{aligned}$$

Using gradient descent: $\theta := \theta - \alpha \nabla_{\theta} J(\theta)$,

$$\text{where } \nabla_{\theta} J(\theta) = - \sum_{i=1}^n (y^{(i)} - \theta^T x^{(i)}) x^{(i)}.$$

Therefore, for a single training example,

$$\theta_j := \theta_j + \alpha \left(y^{(i)} - h_{\theta}(x^{(i)}) \right) x_j^{(i)}.$$

Algorithm 2 Logistic Regression

Require: Feature matrix $X \in R^{m \times n}$, target vector $y \in R^m$, learning rate α , number of iterations N

Ensure: Parameters $\theta \in R^n$

```

0: Initialize  $\theta := \vec{0}$ 
0: for iter = 1 to  $N$  do
0:    $z := X \cdot \theta$  {Linear combination}
0:    $h := \sigma(z)$  {Sigmoid hypothesis}
0:    $error := h - y$  {Error vector}
0:    $gradient := X^T \cdot error$ 
0:    $\theta := \theta - \alpha \cdot gradient$ 
0: end for
0: return  $\theta = 0$ 
```

```

import numpy as np
import pandas as pd
from sklearn import datasets
import warnings
warnings.filterwarnings('ignore')

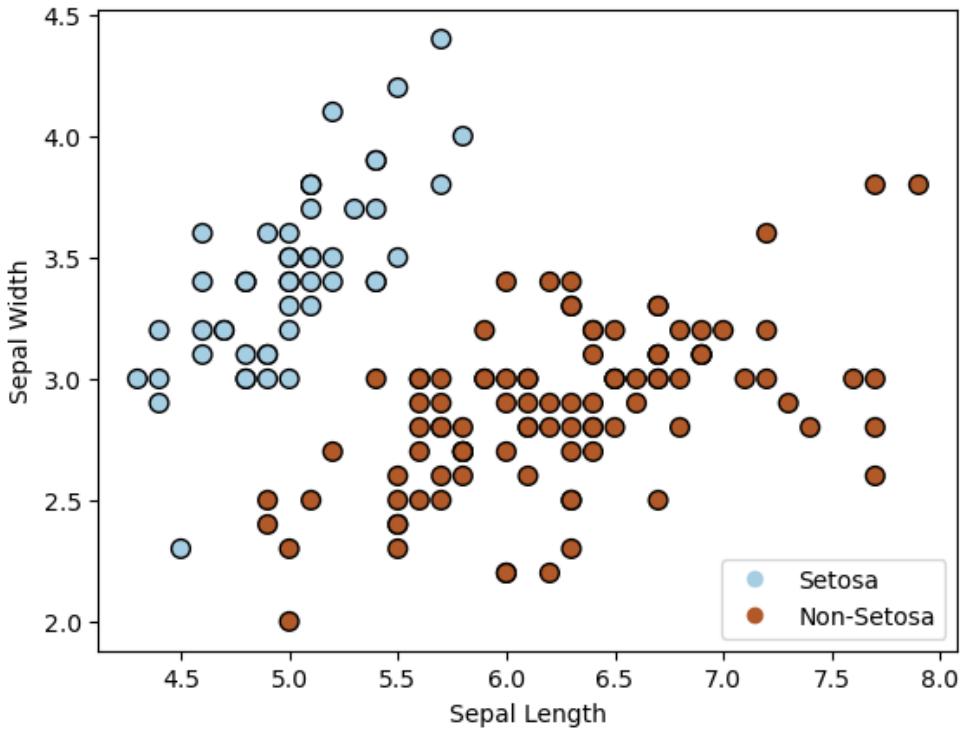
# Load the Iris dataset
iris = datasets.load_iris(as_frame=True)
iris_X, iris_y = iris.data, iris.target
```

```

iris_y2 = iris_y.copy()
iris_y2[iris_y2==2] = 1

# Plot also the training points
p1 = plt.scatter(iris_X.iloc[:, 0], iris_X.iloc[:, 1], c=iris_y2,
                 edgecolor='k', s=60, cmap=plt.cm.Paired)
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.legend(handles=p1.legend_elements()[0], labels=['Setosa', 'Non-Setosa'], loc='lower right
           ')

```



```

import numpy as np
from matplotlib import pyplot as plt

def sigmoid(z):
    return 1/(1+np.exp(-z))

def f(X, theta):
    return sigmoid(X.dot(theta))

def log_likelihood(theta, X, y):
    return (y*np.log(f(X, theta) + 1e-6) + (1-y)*np.log(1-f(X, theta) + 1e-6)).mean()

def loglik_gradient(theta, X, y):
    return np.mean((y - f(X, theta)) * X.T, axis=1)

threshold = 5e-5
step_size = 1e-1

theta, theta_prev = np.zeros((3,)), np.ones((3,))
opt_pts = [theta]
opt_grads = []
iter = 0
iris_X['one'] = 1
X_train = iris_X.iloc[:, [0, 1, -1]].to_numpy()

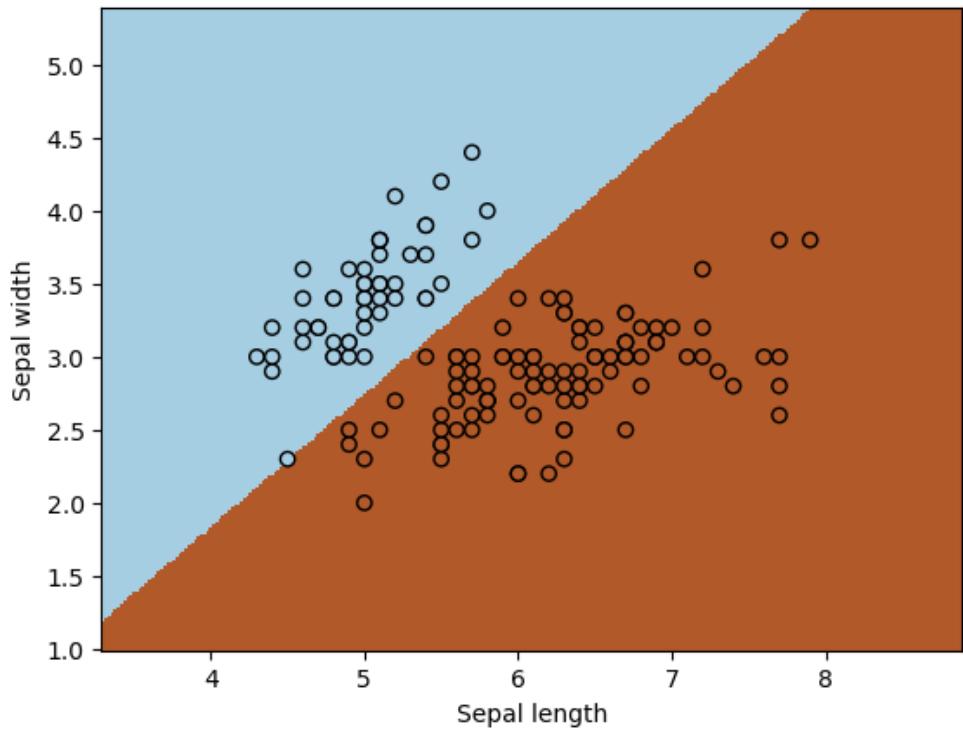
```

```

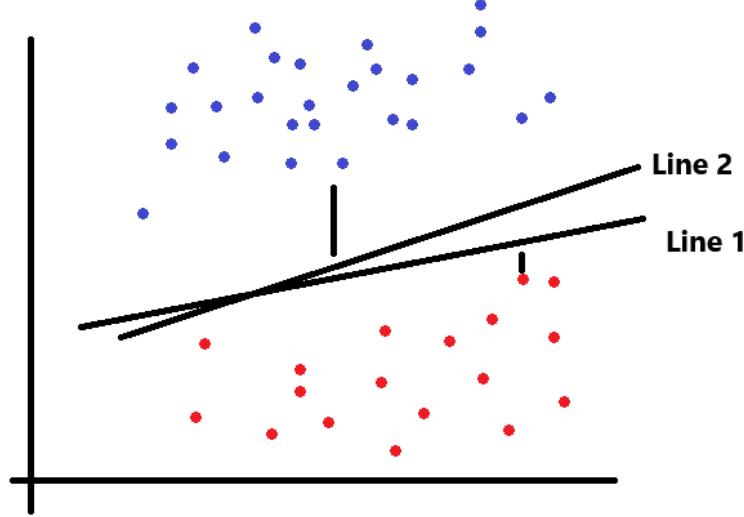
y_train = iris_y2.to_numpy()

while np.linalg.norm(theta - theta_prev) > threshold:
# while True:
    if iter % 50000 == 0:
        print('Iteration %d. Log-likelihood: %.6f' % (iter, log_likelihood(theta, X_train,
                           ↴ y_train)))
    theta_prev = theta
    gradient = loglik_gradient(theta, X_train, y_train)
    theta = theta_prev + step_size * gradient
    opt_pts += [theta]
    opt_grads += [gradient]
    iter += 1

```

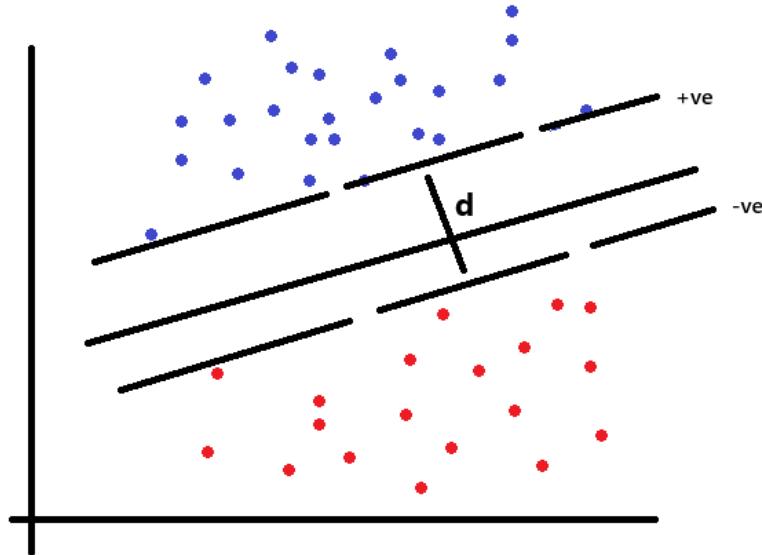


4 Support Vector Machines



The whole idea of SVM lies in the fact that it wants to select that hyperplane that classifies these points, as widely as possible. In this case it is Line 2.

4.1 Mathematics of SVM



Decision Rule:

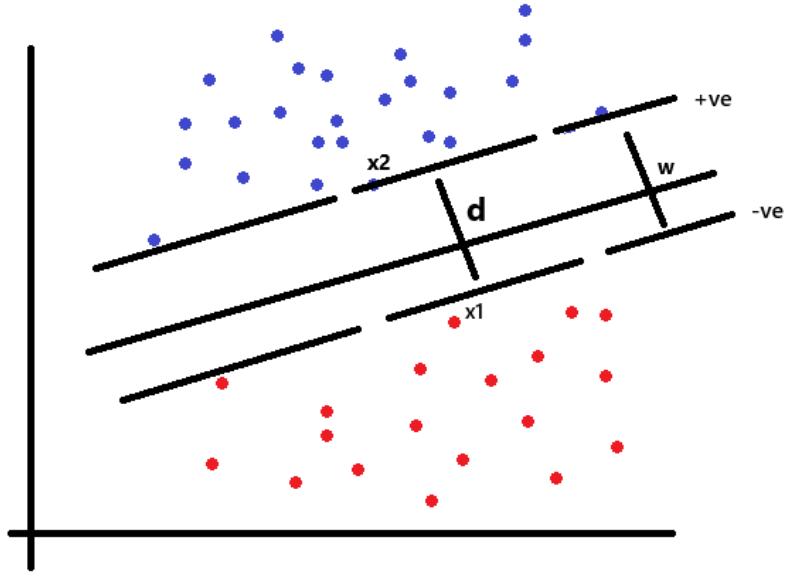
$$\hat{y} = \begin{cases} +1 & \text{if } \vec{w} \cdot \vec{x}_i + b \geq 0 \\ -1 & \text{if } \vec{w} \cdot \vec{x}_i + b < 0 \end{cases}$$

Therefore, by combining both the equations of the constraint, we get:

$$y_i \cdot (\vec{w} \cdot \vec{x}_i + b) \geq 1$$

For support vectors,

$$y_i \cdot (\vec{w} \cdot \vec{x}_i + b) = 1$$



$$d = \frac{(\vec{x}_2 - \vec{x}) \cdot \vec{\omega}}{\|\vec{\omega}\|} = \frac{\vec{x}_2 \cdot \vec{\omega} - \vec{x} \cdot \vec{\omega}}{\|\vec{\omega}\|}$$

$$y_i(\vec{\omega} \cdot \vec{x}_i + b) = 1$$

$$1 \cdot (\vec{\omega} \cdot \vec{x}_2 + b) = 1$$

$$d = \frac{(1 - b) - (-1 - b)}{\|\vec{\omega}\|}$$

$$d = \frac{2}{\|\vec{\omega}\|}$$

$$\arg \max_{(\vec{\omega}^*, b^*)} \frac{2}{\|\vec{\omega}\|} \quad \text{subject to} \quad y_i(\vec{\omega} \cdot \vec{x}_i + b) \geq 1$$

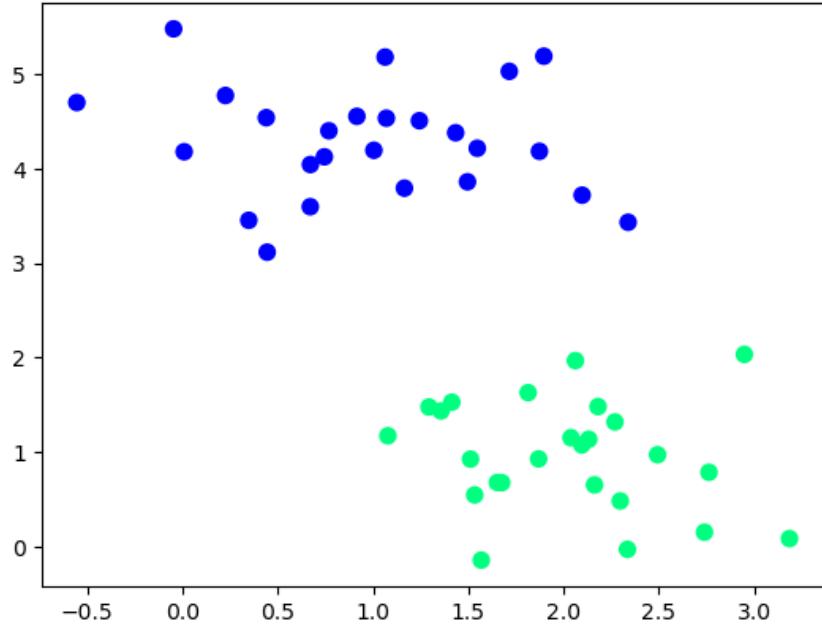
It is Hard Margin SVM for linearly separable data

```

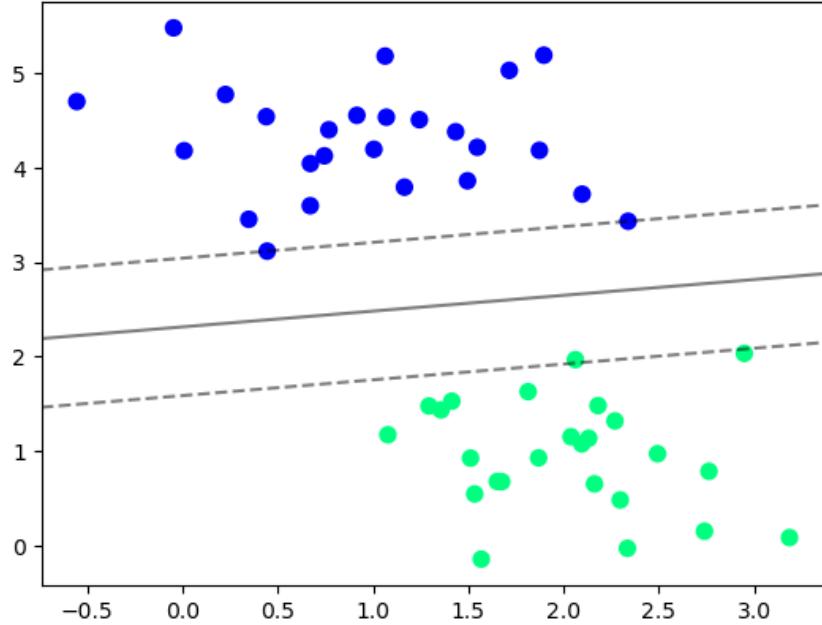
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
import seaborn as sns
from sklearn.datasets._samples_generator import make_blobs

X, y = make_blobs(n_samples = 50, centers = 2,
                   random_state= 0, cluster_std= 0.60)
plt.scatter(X[:,0], X[:, 1], c = y, s= 50, cmap='winter')

```



```
from sklearn.svm import SVC      # Support Vector Classifier
model = SVC(kernel='linear', C = 1)
model.fit(X,y)
```



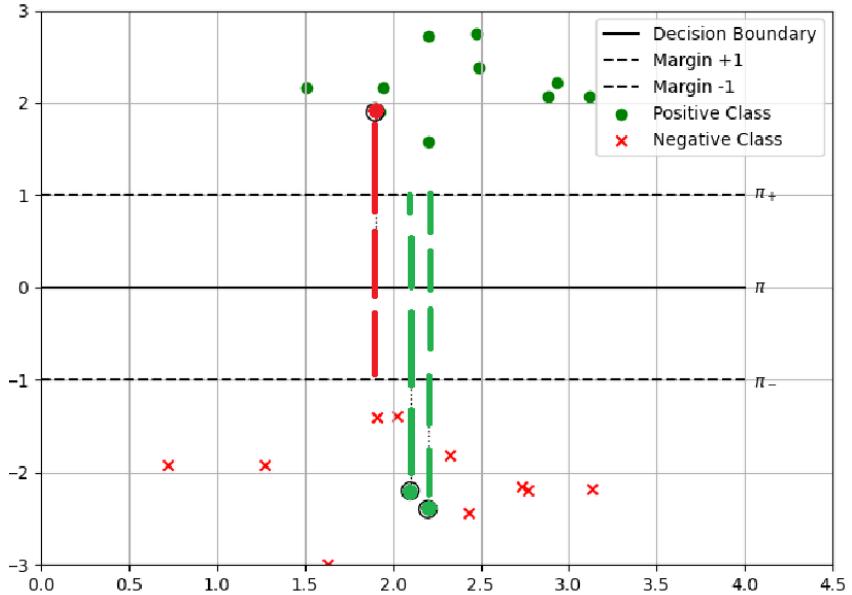
4.2 Non Linear Separable Data

We have

$$\arg \min_{\vec{\omega}, b} \left(\frac{1}{2} \|\vec{\omega}\|^2 + C \sum_{i=1}^n \zeta_i \right)$$

C is a hyperparameter.

If all the points are correctly classified then $\zeta = 0$. It shows how much a data point breaks the margin rule, helping the SVM allow some mistakes while still trying to keep the margin as wide as possible, basically it's the distance as indicated in the figure.



Thus, we have to minimize this distance.

If C value is increased then we are forcing the algorithm to **not** misclassify any point.

If we are making the C's value small, then we are insisting the algo to maximize the margin irrespective of the misclassifications that it will be making.

4.3 Kernel

A *kernel* is a function that helps Support Vector Machines (SVM) work in higher dimensions without actually converting the data. It calculates the similarity between data points in a way that makes complex patterns easier to separate.

Types of Kernels:

- **Linear Kernel:**

$$K(x, x') = x \cdot x'$$

Works well when the data is linearly separable.

- **Polynomial Kernel:**

$$K(x, x') = (x \cdot x' + c)^d$$

Can model curved boundaries. c is a constant and d is the degree of the polynomial.

- **RBF (Gaussian) Kernel:**

$$K(x, x') = \exp\left(-\frac{\|x - x'\|^2}{2\sigma^2}\right)$$

Good for non-linear data. It focuses more on nearby points.

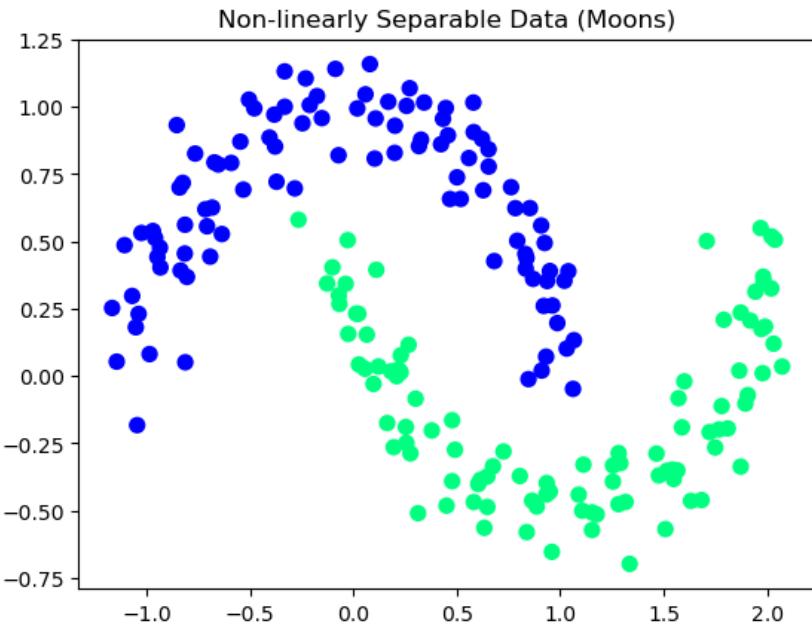
- **Sigmoid Kernel:**

$$K(x, x') = \tanh(a \cdot x \cdot x' + b)$$

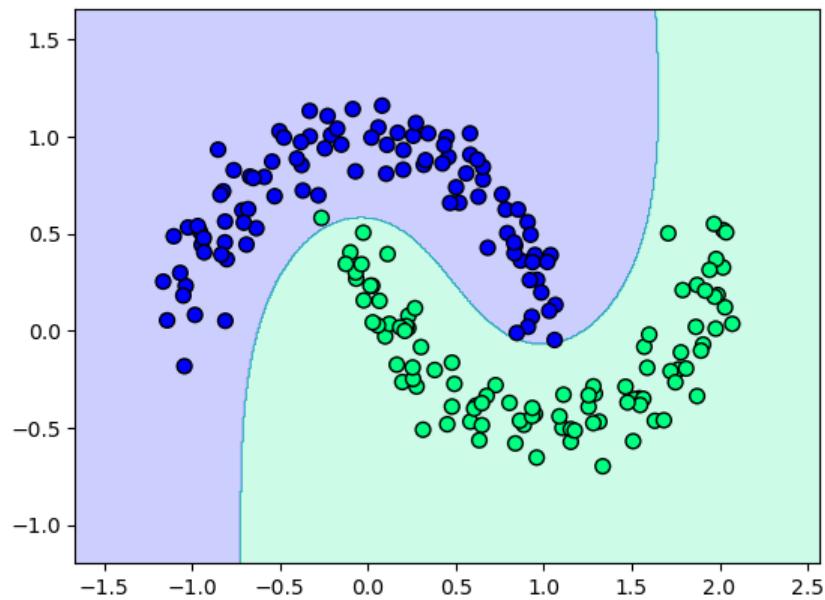
Similar to what neural networks do. Not used often in practice.

```
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=200, noise=0.1, random_state=0)
# Plot the data
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='winter')
plt.title("Non-linearly Separable Data (Moons)")
plt.show()
```



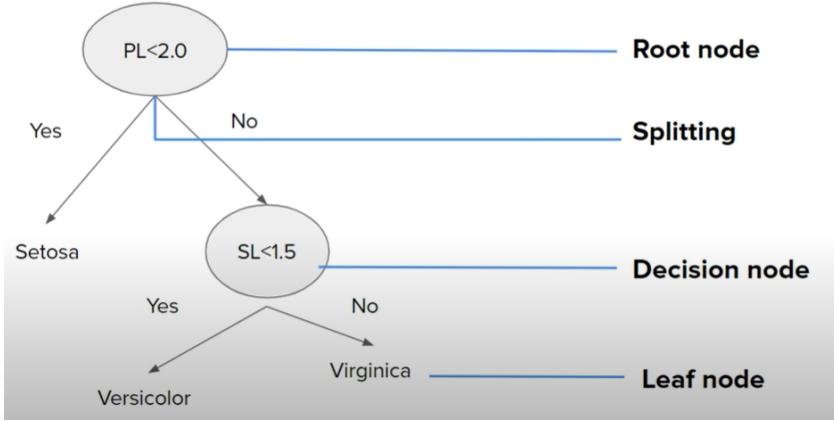
```
from sklearn.svm import SVC      # Support Vector Classifier
model = SVC(kernel='rbf', C = 1)
model.fit(X,y)
```



5 Decision Trees

A **Decision Tree** is a supervised machine learning algorithm used for both *classification* and *regression* tasks. It models decisions and their possible consequences using a tree-like structure.

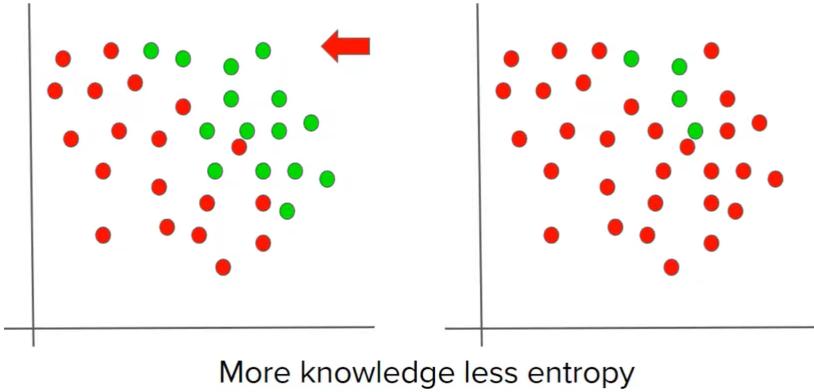
- Begin with your training dataset, which should have some feature variables and classification or regression output.
- Determine the “best feature” in the dataset to split the data on;
- Split the data into subsets that contain the correct values for this best feature. This splitting basically defines a node on the tree, i.e., each node is a splitting point based on a certain feature from our data.



5.1 Entropy

Entropy is a measure of disorder.

More Knowledge \Rightarrow Less Entropy



The mathematical formula for entropy is:

$$E(S) = \sum_{i=1}^c -p_i \log_2 p_i$$

Where p_i is simply the frequentist probability of an element/class i in our data.

For example, if our data has only 2 class labels **Yes** and **No**, then:

$$E(D) = -p_{\text{yes}} \log_2(p_{\text{yes}}) - p_{\text{no}} \log_2(p_{\text{no}})$$

Salary	Age	Purchase
20000	21	Yes
10000	45	No
60000	27	Yes
15000	31	No
12000	18	No

Let:

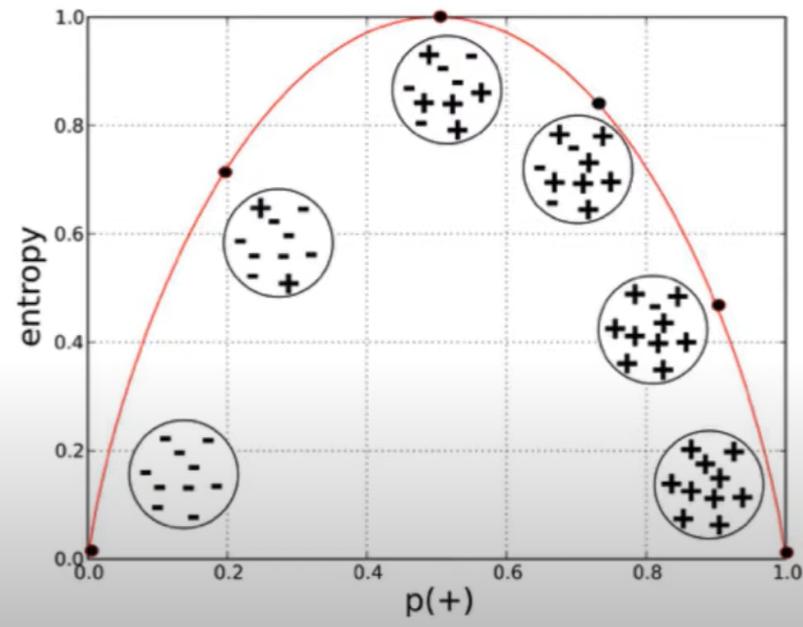
- $P_y = \text{probability of Yes} = \frac{2}{5}$

- $P_n = \text{probability of No} = \frac{3}{5}$

$$H(D) = -P_y \log_2(P_y) - P_n \log_2(P_n)$$

$$H(D) = -\frac{2}{5} \log_2 \left(\frac{2}{5} \right) - \frac{3}{5} \log_2 \left(\frac{3}{5} \right)$$

$$H(D) \approx 0.97$$



5.2 Information Gain

Information Gain is a metric used to train Decision Trees. Specifically, this metric measures the quality of a split in a Decision Tree

$$\text{Information Gain} = E(\text{Parent}) - \{\text{Weighted Average}\} \times E(\text{Children})$$

Outlook	Temperature	Humidity	Windy	PlayTennis
Sunny	Hot	High	False	No
Sunny	Hot	High	True	No
Overcast	Hot	High	False	Yes
Rainy	Mild	High	False	Yes
Rainy	Cool	Normal	False	Yes
Rainy	Cool	Normal	True	No
Overcast	Cool	Normal	True	Yes
Sunny	Mild	High	False	No
Sunny	Cool	Normal	False	Yes
Rainy	Mild	Normal	False	Yes
Sunny	Mild	Normal	True	Yes
Overcast	Mild	High	True	Yes
Overcast	Hot	Normal	False	Yes
Rainy	Mild	High	True	No

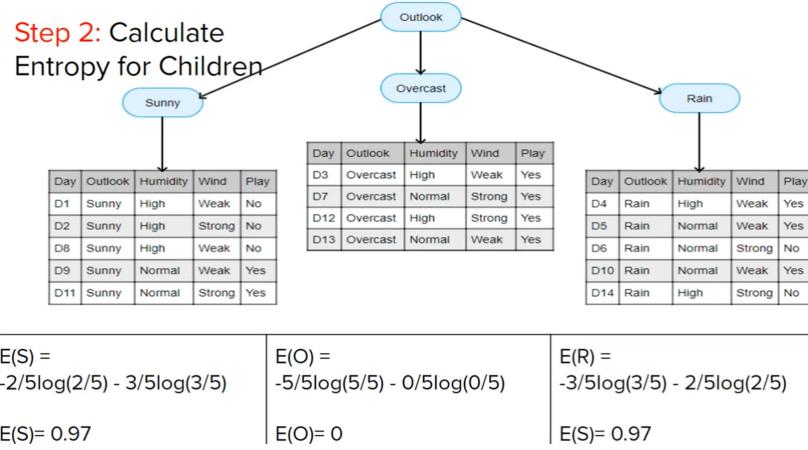
Entropy of Parent Dataset:

$$E(P) = -p_y \log_2(p_y) - p_n \log_2(p_n)$$

$$= -\frac{9}{14} \log_2 \left(\frac{9}{14} \right) - \frac{5}{14} \log_2 \left(\frac{5}{14} \right)$$

$$E(P) = 0.94$$

On the basis of 'Outlook' column, let's divide it. We will group by



$$\text{Weighted Entropy} = \frac{5}{14} \times 0.97 + \frac{4}{14} \times 0 + \frac{5}{14} \times 0.97$$

$$W.E(\text{Children}) = 0.69$$

$$\text{Information Gain} = E(\text{Parent}) - \{\text{Weighted Average}\} \times E(\text{Children})$$

$$IG = 0.97 - 0.69 = 0.28$$

So the information gain (or the decrease in entropy/impurity) when you split this data on the basis of **Outlook** condition/column is **0.28**.

To determine the best split, the algorithm calculates the **Information Gain** for all the columns. The column that has the **highest Information Gain** (i.e., the maximum decrease in entropy) is selected to split the data.

After selecting the best column, the decision tree **recursively** repeats this process. It uses a **greedy search** strategy from top to bottom, finding Information Gain at every level of the tree until the data is perfectly classified or a stopping condition is met.

5.3 Gini Impurity

Gini Impurity is a metric used to measure how often a randomly chosen element from the dataset would be incorrectly classified if it was randomly labeled according to the distribution of labels in the subset.

It is mainly used in the CART (Classification and Regression Tree) algorithm.

$$Gini = 1 - \sum_{i=1}^c p_i^2$$

Where p_i is the probability of an element being classified to class i .

A Gini value of 0 means perfect purity (all elements belong to a single class), and higher values indicate more impurity.

- **Gini Impurity** is **computationally faster**, making it a preferred choice when speed is important.

- Entropy tends to produce a **more balanced tree** in some cases, as it considers the distribution of class probabilities more precisely.

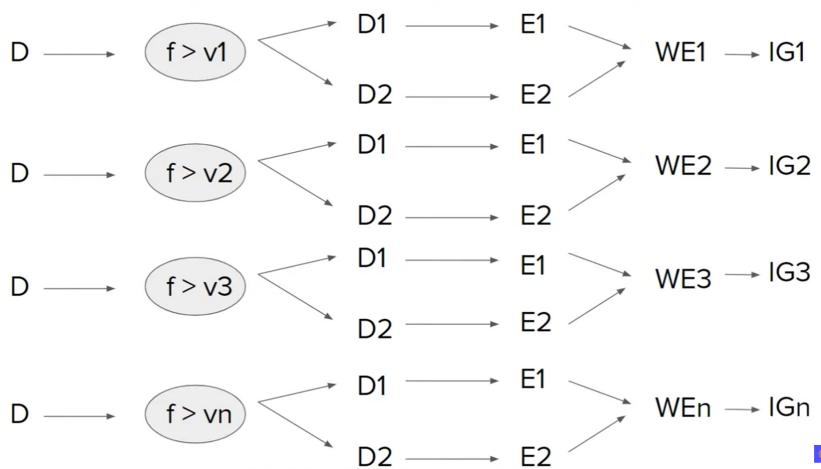
Solution: Perform **hyperparameter tuning** to choose between Gini and Entropy based on the performance of your specific dataset.

5.4 Handling Numerical values in Decision Trees

S No	User Rating	Downloaded
1	3.5	Yes
2	4.6	Yes
3	2.2	No
4	1.6	Yes
5	4.1	No
6	3.9	No
7	3.2	No
9	2.9	Yes
10	4.8	Yes
11	3.3	No
12	2.5	Yes
13	1.9	Yes

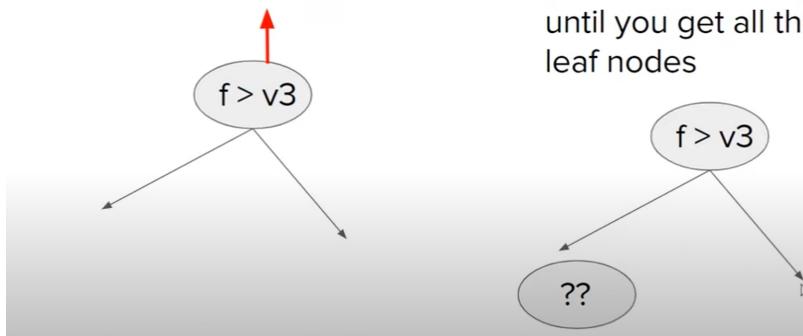
Sort the Data on the Basis of Numerical Column

S No	User Rating	Downloaded
1	1.6	Yes
2	1.9	Yes
3	2.2	No
4	2.5	Yes
5	2.9	Yes
6	3.2	No
7	3.3	No
9	3.5	Yes
10	3.9	No
11	4.1	No
12	4.6	Yes
13	4.8	Yes



Step 5:

$\text{Max}\{\text{IG1}, \text{IG2}, \text{IG3}, \dots, \text{IGn}\}$



Step 6:

Do this recursively until you get all the leaf nodes

5.5 Role of `max_depth` in Overfitting and Underfitting

In a Decision Tree, the parameter `max_depth` controls the maximum depth the tree is allowed to grow.

- **`max_depth=None`** means the tree will grow until all leaves are pure or contain fewer samples than the minimum required to split.
- This can lead to a **very deep tree**, capturing all the patterns, including noise in the training data.
- As a result, it often causes **overfitting**, where the model performs well on training data but poorly on unseen data.
- Conversely, setting `max_depth` to a very small value may result in a **shallow tree**, unable to capture sufficient structure in the data, leading to **underfitting**.

Conclusion: Proper tuning of `max_depth` is crucial to balance the bias-variance tradeoff and prevent overfitting or underfitting.

6 Key Hyperparameters of Decision Trees in scikit-learn

The following are the most important hyperparameters for tuning the performance of a `DecisionTreeClassifier`:

- **criterion (default = "gini")**
Function to measure the quality of a split:
 - "gini" — Gini Impurity
 - "entropy" or "log_loss" — Information Gain
- **max_depth = None**
The maximum depth of the tree. If set to `None`, the tree expands until all leaves are pure, which can lead to overfitting.
- **min_samples_split = 2**
The minimum number of samples required to split an internal node. If a node has fewer than this number of samples, it will not be split and becomes a leaf.

Setting a higher value helps control overfitting by preventing the model from learning overly specific or noisy patterns.

Example:

Consider the following dataset:

Feature (X)	Target (Y)
3.0	Yes
4.5	Yes
5.2	No
6.1	Yes
7.4	No

If `min_samples_split = 3`, a node must contain at least 3 samples to be split.

Suppose after the first split, one child node receives only 2 samples (e.g., $\{5.2, 7.4\}$). Even if it's impure, the split will be stopped because it doesn't meet the `min_samples_split` threshold. That node will become a leaf, helping prevent overfitting to tiny sample groups.

- `min_samples_leaf = 1`

The minimum number of samples required to be at a **leaf node**. This parameter ensures that no split will create a child node with fewer than the specified number of training samples.

Setting a higher value helps smooth the model and reduces overfitting by avoiding splits that result in very small, highly specific leaves.

Example:

Consider the dataset:

Feature (X)	Target (Y)
2.0	No
3.5	Yes
4.2	Yes
5.1	No
6.8	Yes

If `min_samples_leaf = 2`, then every **leaf** must have at least 2 data points.

Suppose a split divides the data such that one resulting leaf has only 1 sample (e.g., $\{6.8, \text{Yes}\}$). This split will be rejected, even if it improves purity, because it violates the minimum leaf size constraint. The algorithm will look for alternative splits that leave at least 2 samples in every leaf.

This helps reduce model complexity and improve generalization on unseen data.

- `max_features = None`

The number of features to consider when looking for the best split. Can be set to:

- "sqrt" — useful for classification tasks
- "log2" — for logarithmic feature scaling
- `None` — use all features

- `max_leaf_nodes = None`

Grows tree with the specified maximum number of leaf nodes using best-first search.

- `min_impurity_decrease = 0.0`

A node will split only if this condition is met: the decrease in impurity must be at least this value.

- `ccp_alpha = 0.0`

Complexity parameter used for Minimal Cost-Complexity Pruning. A higher value will prune more of the tree.

Interactive Decision Tree Visualizer:

<https://dt-visualise.herokuapp.com/>

6.1 Splitting in Regression Trees

In Regression Trees, unlike classification trees, the goal is to predict a continuous output variable. The tree splits the dataset into regions such that the **variance (or squared error)** within each region is minimized.

6.2 Splitting Criterion

At each node, the algorithm chooses a split that minimizes the following cost function:

$$\text{MSE}(t) = \frac{1}{|D_t|} \sum_{i \in D_t} (y_i - \bar{y}_t)^2$$

Where:

- D_t is the set of data points at node t
- y_i is the true value for sample i
- \bar{y}_t is the mean target value at node t

The best split is the one that minimizes the **weighted sum of mean squared errors (MSE)** for the two resulting child nodes.

6.3 Illustrative Example

Consider the following simple dataset:

Feature (X)	Target (Y)
2.0	10
3.0	12
4.0	14
5.0	16
6.0	18

Let us consider a possible split at $X = 4.5$. This results in two subsets:

- Left node: $\{(2.0, 10), (3.0, 12), (4.0, 14)\}$ with mean $\bar{y}_L = 12$
- Right node: $\{(5.0, 16), (6.0, 18)\}$ with mean $\bar{y}_R = 17$

Calculate MSE for both:

$$\text{MSE}_L = \frac{1}{3} [(10 - 12)^2 + (12 - 12)^2 + (14 - 12)^2] = \frac{1}{3}(4 + 0 + 4) = \frac{8}{3}$$

$$\text{MSE}_R = \frac{1}{2} [(16 - 17)^2 + (18 - 17)^2] = \frac{1}{2}(1 + 1) = 1$$

$$\text{Total Weighted MSE} = \frac{3}{5} \cdot \frac{8}{3} + \frac{2}{5} \cdot 1 = \frac{8}{5} + \frac{2}{5} = 2$$

This score is then compared with other possible splits (e.g., $X = 3.5$, $X = 5.5$), and the split with the **lowest total MSE** is chosen.

Summary

Regression Trees aim to create splits that minimize the variance within the resulting nodes. This is achieved using a greedy approach where, at each step, the split that leads to the lowest total MSE is selected. As more splits are added, the model becomes more complex and capable of capturing finer patterns — at the risk of overfitting.

7 Random Forest

Random Forest is a machine learning algorithm that combines **many decision trees** to make better predictions.

Each tree gives a result, and the forest **votes** for the best answer:

- For classification: it picks the class with the most votes.
- For regression: it takes the average of all trees.

It is more accurate and stable than using a single decision tree, and it helps avoid overfitting.

7.1 Difference Between Bagging and Random Forest

Bagging (Bootstrap Aggregating) involves training multiple base models (typically Decision Trees) on different bootstrapped subsets of the dataset. However, each base model receives all the features (columns), and the randomness comes only from **row sampling**.

Random Forest, on the other hand, extends Bagging by introducing an additional layer of randomness: **feature (column) sampling**. In Random Forest:

- Each Decision Tree is trained not just on a random subset of the data rows, but also on a random subset of the features.
- Moreover, **column sampling happens at every node split**, not just at the start of tree training.

Key Differences:

- In **Bagging**, all trees see all features — randomness is introduced only via row sampling.
- In **Random Forest**, each split in every tree considers a randomly selected subset of features, leading to more diversity among trees.

This additional randomness in Random Forest often leads to **better generalization performance** compared to standard Bagging, because it reduces correlation among the trees in the ensemble.

7.2 Hyperparameters in Random Forest

RandomForestClassifier has several hyperparameters that affect model performance. Below are the key ones with simple explanations and examples.

- **n_estimators** — Number of decision trees in the forest. More trees usually improve performance but increase computation time.

Example: `RandomForestClassifier(n_estimators=100)` creates a forest with 100 trees.

- **max_depth** — Controls the maximum depth of each tree. Prevents overfitting if set properly.

Example: `RandomForestClassifier(max_depth=3)` limits tree height to 3 levels.

- **min_samples_split** — Minimum number of samples needed to split a node.

Example: `RandomForestClassifier(min_samples_split=5)` means a node must have at least 5 samples to be split further.

- **min_samples_leaf** — Minimum number of samples required in a leaf node.

Example: `RandomForestClassifier(min_samples_leaf=2)` ensures each leaf has at least 2 samples.

- **max_features** — Number of features to consider when looking for the best split.

Example: `RandomForestClassifier(max_features='sqrt')` considers only $\sqrt{\text{total features}}$ at each node.

- **bootstrap** — Whether to use bootstrap samples (sampling with replacement).

Example: `RandomForestClassifier(bootstrap=True)` trains each tree on a random subset of the data.

- **oob_score** — Whether to use out-of-bag samples to estimate accuracy.

Example: `RandomForestClassifier(oob_score=True)` evaluates performance using unused data from bootstrap sampling.

- **random_state** — Controls randomness for reproducibility.

Example: `RandomForestClassifier(random_state=42)` ensures consistent results across runs.

- **ccp_alpha** — Complexity parameter for pruning. Higher value prunes more aggressively.

Example: `RandomForestClassifier(ccp_alpha=0.01)` simplifies the tree by removing less useful splits.

7.3 Out of Bag Evaluation (OOB)

OOB (Out-of-Bag) is a validation technique used in Random Forests. When training each tree, a random subset of the data is selected using **bootstrap sampling** (sampling with replacement). The remaining data points that are not selected for training a particular tree are called **out-of-bag samples**.

These out-of-bag samples can be used as a test set to evaluate the tree's performance. The Random Forest aggregates these evaluations across all trees to estimate the model's overall accuracy — this is called the **OOB Score**.

Key Point: OOB provides an efficient, internal cross-validation mechanism without needing to hold out a dedicated test set.

7.4 Feature Importance in Random Forest

Feature importance is a score that tells us how useful or valuable each feature was in building the decision trees within a random forest model.

It helps us understand which features have the most influence on the prediction.

Why is it Useful?

- Helps in understanding the model's decision-making.
- Allows us to reduce dimensionality by removing less important features.
- Useful in feature selection for faster and better-performing models.

Example Analogy: Think of solving a puzzle. Some pieces (features) are more central to solving the image than others. Similarly, in a Random Forest, some features contribute more to accurate predictions.

How is Feature Importance Calculated?

- Every time a feature is used to split a node, the impurity (e.g., Gini or Entropy) decreases.
- This decrease in impurity is recorded and summed over all trees in the forest.
- The importance of a feature is the total reduction of the criterion (like Gini or Entropy) it brings.
- Finally, the values are normalized so that they add up to 1.

Next Steps After Identifying Important Features:

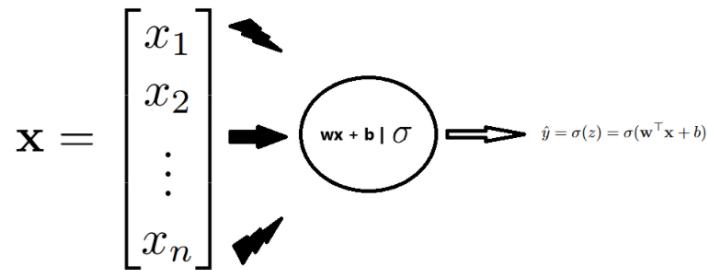
- Drop features with very low importance to reduce complexity.
- Retrain the model with only the top important features.
- Compare the new model's accuracy to check if performance improves or remains the same.
- Use the refined feature set for model interpretation or deployment.

Use in Practice: After training, you can access feature importances using the `feature_importances_` attribute of a `RandomForestClassifier` object.

8 Neural Network

A Neural Network is a computational model inspired by the human brain. It consists of layers of nodes (neurons) where each node applies a weighted sum and activation function

8.1 Logistic Regression



1. Initialize weights w and bias b
2. Find the optimal w and b (e.g., using gradient descent)
3. Use the model to predict:

$$\hat{y} = \sigma(w^\top x + b)$$

Key Concepts

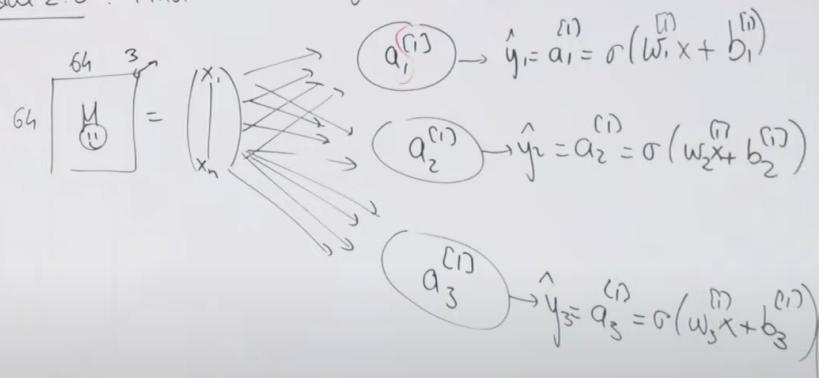
- **Neuron** = Linear transformation + Activation function

$$\text{Neuron output: } \hat{y} = \sigma(w^\top x + b)$$

- **Model** = Architecture + Parameters

Architecture: structure of layers and connections
Parameters: weights w and biases b

goal 2.0: Find cat/lion/iguana in image.

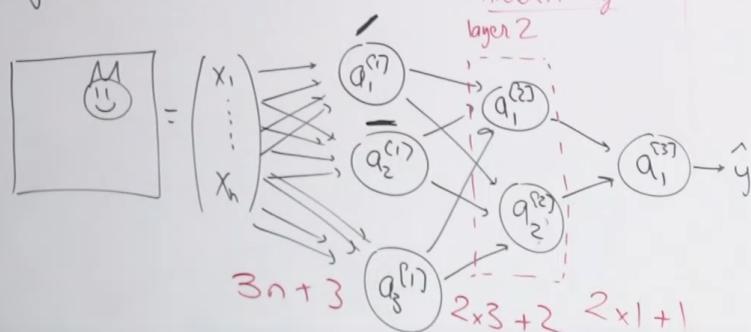


goal 3.0: + constraint
unique animal on an image

The diagram shows the softmax activation function for three categories. The softmax function is defined as $e^{z_i} / \sum_{k=1}^K e^{z_k}$ for each category i . For three categories, it is shown as $e^{z_1} / \sum_{k=1}^3 e^{z_k}$, $e^{z_2} / \sum_{k=1}^3 e^{z_k}$, and $e^{z_3} / \sum_{k=1}^3 e^{z_k}$.

(II) Neural Networks

goal image \Rightarrow cat vs. no cat
(1) (0) hidden layer



Propagation equations

$$\begin{aligned}
 z^{(0)} &= w^{(0)} x + b^{(0)} & (3,1) \\
 a^{(1)} &= \sigma(z^{(0)}) & (3,1) \\
 z^{(1)} &= w^{(1)} \cdot a^{(0)} + b^{(1)} & (2,3) \quad (2,1) \quad (2,1) \\
 a^{(2)} &= \sigma(z^{(1)}) & (2,1) \\
 z^{(2)} &= w^{(2)} \cdot a^{(1)} + b^{(2)} & (1,3) \quad (1,1) \\
 a^{(3)} &= \sigma(z^{(2)}) & (1,1)
 \end{aligned}$$

8.2 Gradient Descent Variants

Batch Gradient Descent: Uses the entire training dataset to compute the gradient of the loss function.

- Stable convergence
- Slow for large datasets
- Update: $\theta := \theta - \eta \nabla J(\theta)$

Batch Gradient Descent: Use all data at once

```
model.fit(X_train, y_train, batch_size = len(X_train), epochs=100)
```

Mini-Batch Gradient Descent: Divides the training set into small batches and updates parameters using each mini-batch.

- Faster and more memory-efficient
- Introduces noise that can help escape local minima
- Update: $\theta := \theta - \eta \nabla J_{\text{mini-batch}}(\theta)$

Mini-Batch Gradient Descent:

Mini-Batch Gradient Descent: Use small batch (e.g., 32)

```
model.fit(X_train, y_train, batch_size = 32, epochs=100)
```

8.3 Vanishing Gradient Problem

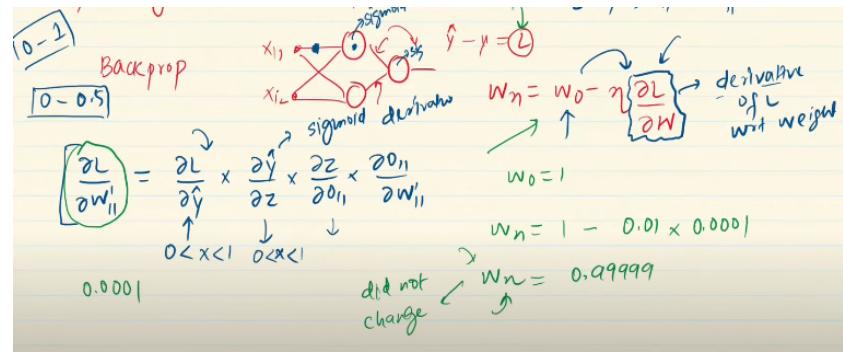
Vanishing Gradient Problem is encountered when training artificial neural networks with gradient based learning methods and backpropagation. In such methods, during each iteration of training each of the neural network's weights receives an update proportional to the partial derivative of the error function with respect to the current weight. The problem is that in some cases, the gradient will be vanishingly small, effectively preventing the weight from changing its value. In the worst case, this may completely stop the neural network from further training

$$0.1 \times 0.1 \times 0.1 \times \dots = 0.00000..1$$

Occurs in deep neural networks

Particularly with sigmoid / tanh. These functions map a large range of input values to a small output range. The gradients

of these activation functions become very small and if multiplied by each other across many layers, cause them to shrink exponentially.



How to recognize ?

- Loss focus \Rightarrow no changes
- Weights graph \Rightarrow if constant

How to handle Vanishing Gradient Problem ?

- Reduce model complexity
- Use different activation function
- Proper weight initialization
- Batch normalization
- Residual Network

8.4 Improving the Performance

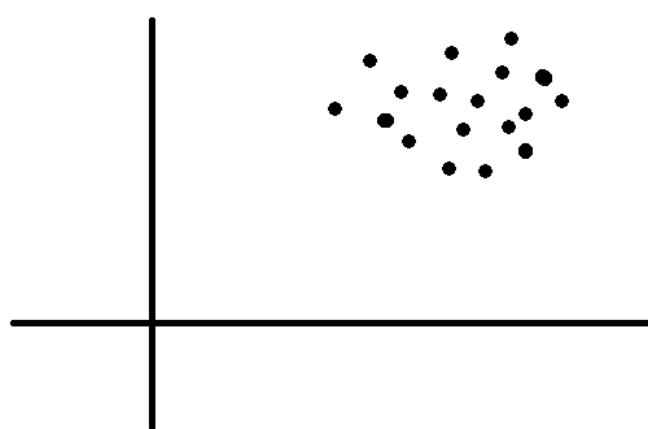
8.4.1 Standardization

Formula:

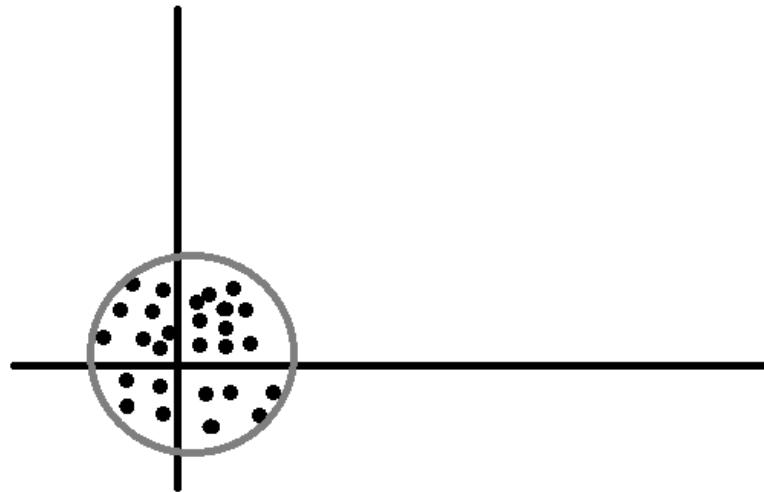
$$z = \frac{x - \mu}{\sigma}$$

where x is the data point, μ is the mean, and σ is the standard deviation.

When to Use: Use standardization when features have different scales and the data follows (or approximately follows) a Gaussian distribution. It is commonly used in algorithms like SVM, logistic regression, and k-means.



After Standardization



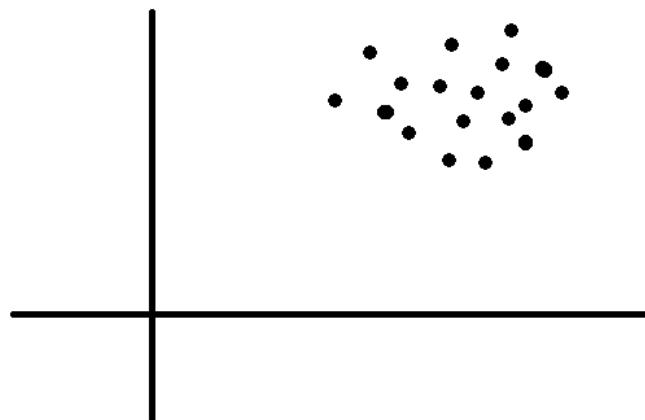
8.4.2 Normalization

Formula:

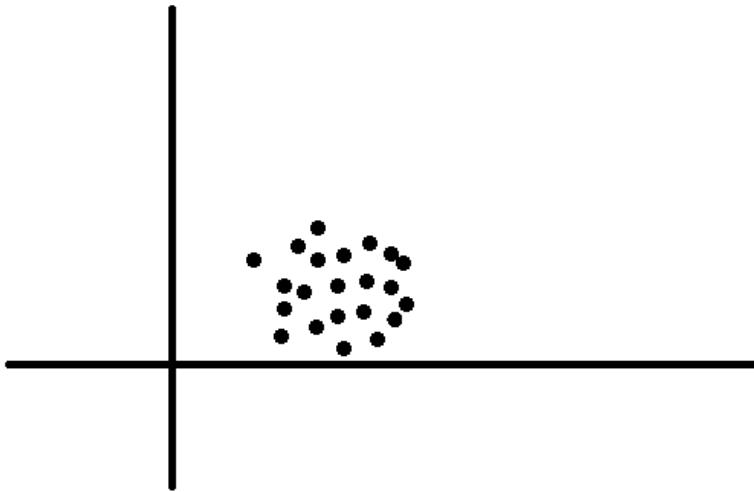
$$x_{\text{norm}} = \frac{x - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}}$$

where x is the data point, x_{min} and x_{max} are the minimum and maximum values of the feature.

When to Use: Use normalization when the data does not follow a Gaussian distribution and needs to be scaled to a specific range, typically $[0, 1]$. It is useful for distance-based algorithms like k-NN and neural networks.



After Normalization



8.4.3 Difference

Standardization

- Transforms data to have **zero mean and unit variance**.
- Use when the data is (approximately) **normally distributed**.
- Suitable for **linear models, SVM, logistic regression**, and PCA.

Normalization

- Scales data to a fixed range, usually [0, 1].
- Use when the distribution is **not Gaussian**.
- Preferred for **distance-based algorithms** like k-NN, K-Means, or Neural Networks.

8.5 Dropout

Dropout is a regularization technique used in neural networks to reduce overfitting. It works by randomly deactivating a fraction of neurons during training, which prevents the model from relying too heavily on specific neurons and encourages redundancy and robustness in learning.

- Randomly sets a fraction of input units to zero at each update during training.
- Helps improve generalization by reducing reliance on any specific neurons.
- Typically used in fully connected (dense) layers.
- Inactive during inference (testing).

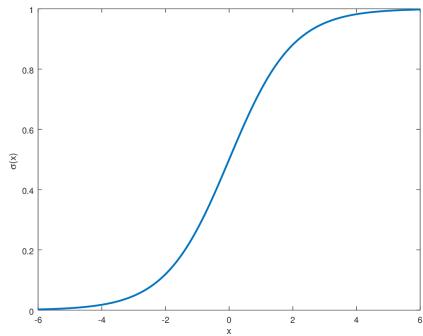
```
model.add(tf.keras.layers.Dropout(0.5))
```

8.6 Activation Functions

8.6.1 Sigmoid

Advantages

- $[0, 1] \Rightarrow$ Probability \Rightarrow Output Layer \Rightarrow Binary Classification
- Non-Linear \Rightarrow It can capture non linear data pattern
- Differentiable



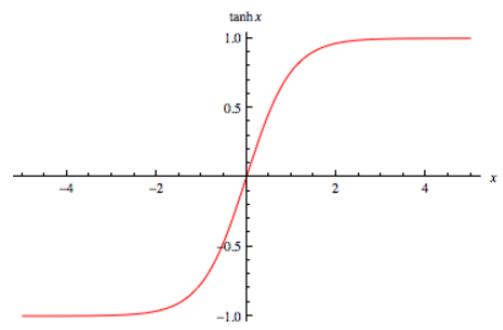
Disadvantages

- Saturating Function \Rightarrow Vanishing Gradient Problem
- Non zero centered \Rightarrow Training gets slow

8.6.2 Tanh

Advantages

- Non-Linear \Rightarrow It can capture non linear data pattern
- Zero centered \Rightarrow Training faster
- Differentiable



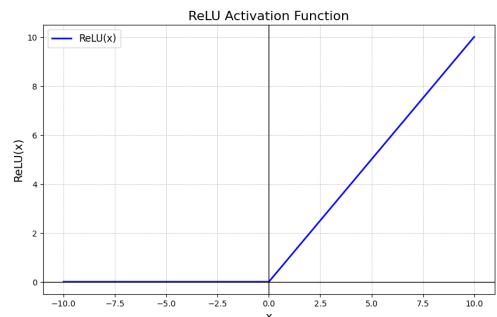
Disadvantages

- Saturating Function \Rightarrow Vanishing Gradient Problem
- Computationally expensive

8.6.3 ReLU

Advantages

- Non-Linear \Rightarrow It can capture non linear data pattern
- Not Saturated in the positive region
- Computationally inexpensive
- Converge faster



Disadvantages

- Not purely differentiable
- Not zero centered \Rightarrow Use Batch Normalization

8.6.4 Dying ReLU

Dying ReLU refers to a situation where a large number of neurons output zero for all inputs and stop learning during training.

Causes:

- If the weighted input to the ReLU is negative, the output becomes zero.
- During backpropagation, gradients also become zero, so weights stop updating.
- Large negative bias initialization or high learning rate can push neurons into this inactive state permanently.

Solution:

- Use variants like Leaky ReLU, Parametric ReLU (PReLU), or ELU.
- Proper weight initialization (e.g., He initialization).
- Lower learning rate to avoid large negative updates.

8.6.5 Leaky ReLU

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases}$$

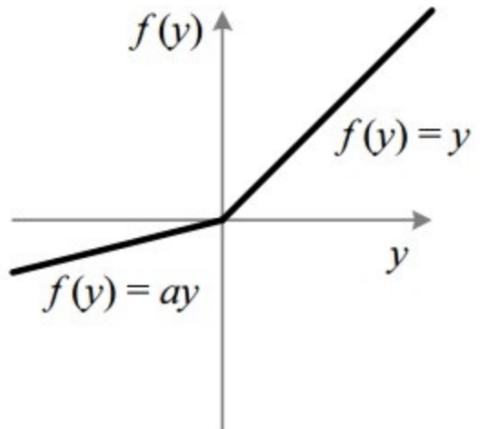
where α is a small constant (e.g., 0.01).

Advantages

- Non saturated
- Easily computed
- No dying relu
- Close to zero centre

Disadvantages

- The value of α (slope for negative inputs) is fixed and manually set. If not chosen properly, it may not completely solve the Dying ReLU problem.
- It introduces a small gradient for negative values, which might still lead to slow learning for those neurons.



8.6.6 Parametric ReLU (PReLU)

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases} \quad \text{where } \alpha \text{ is a learnable parameter.}$$

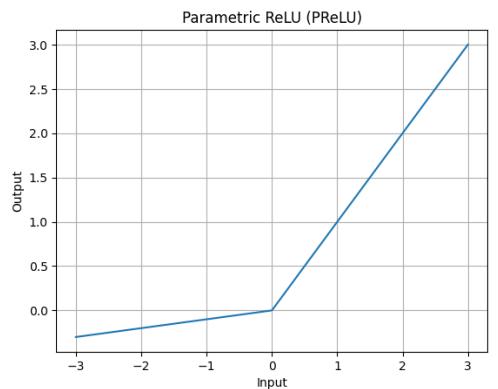
where α is a small constant (e.g., 0.01).

Advantages:

- Learns the optimal slope for negative inputs during training.
- Can adapt better to different types of data compared to fixed-slope functions like Leaky ReLU.
- Helps mitigate the Dying ReLU problem more effectively.

Disadvantages:

- Introduces additional parameters, increasing model complexity.
- Risk of overfitting if not properly regularized.



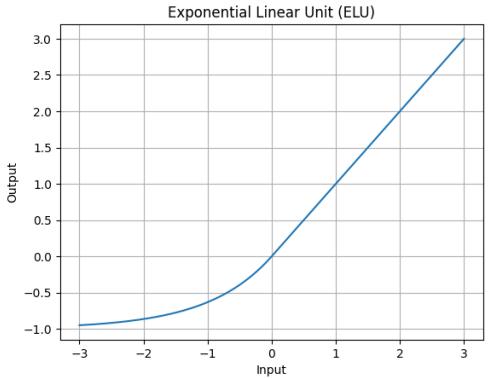
8.6.7 Exponential Linear Unit (ELU)

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases} \quad \text{where } \alpha > 0 \text{ is a hyperparameter (commonly } \alpha = 1\text{).}$$

where α is a small constant (e.g., 0.01).

Advantages:

- Smooth and differentiable at all points, including at 0.
- Avoids dying ReLU problem with non-zero gradients for negative inputs.
- Helps bring the mean activation closer to zero, which speeds up learning.



Disadvantages:

- Computationally more expensive due to the exponential function.
- May cause exploding gradients if α is not properly chosen.

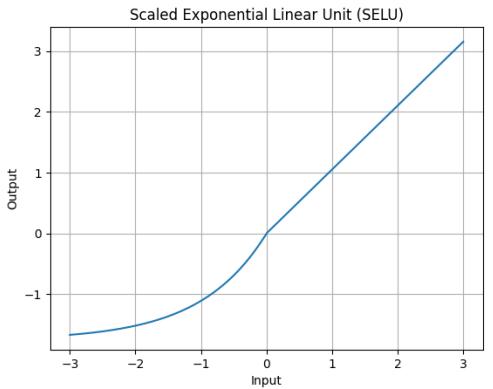
8.6.8 Scaled Exponential Linear Unit (SELU)

$$f(x) = \lambda \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases}$$

where $\lambda \approx 1.0507$ and $\alpha \approx 1.6733$ are fixed constants.

Advantages:

- Self-normalizing: maintains zero mean and unit variance across layers.
- Speeds up training and improves convergence in deep networks.
- Reduces need for explicit normalization techniques like Batch Normalization.



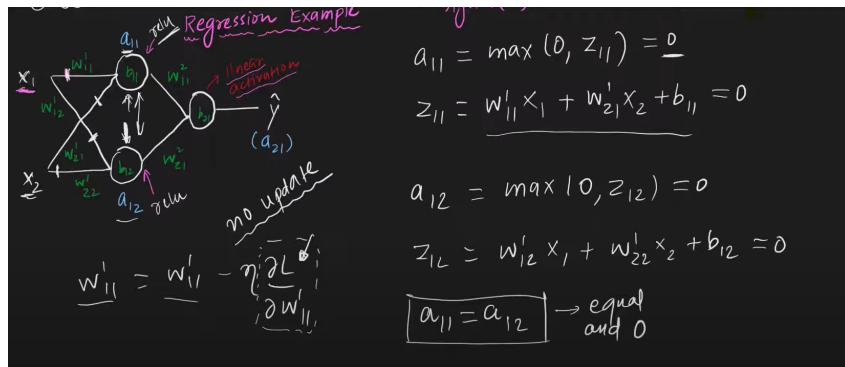
Disadvantages:

- Requires careful initialization (e.g., LeCun normal) and use of specific architectures (e.g., no dropout).
- Only works well with specific types of networks (e.g., fully connected feedforward networks).

8.7 Weight Initialization

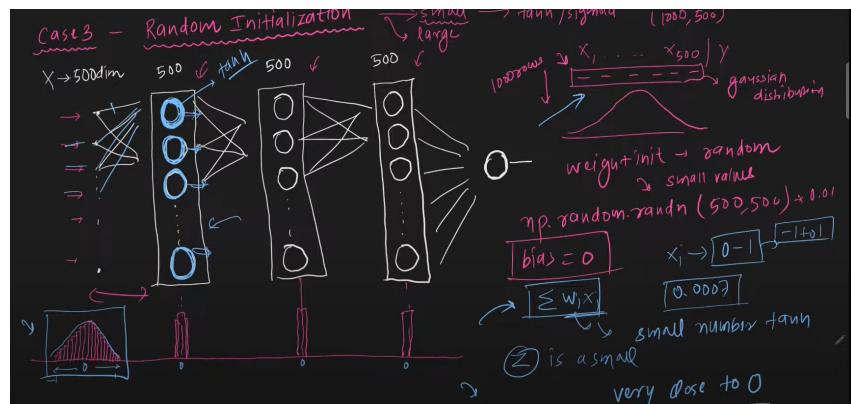
8.7.1 What not to do ?

- Zero Initialization



- Non 0 Constant Value \Rightarrow Model acts like a linear model

- Randomly initialized: Small values



- Randomly initialized: Large values

