BASICS OF AI ML AND DL

Artificial Intelligence (AI) is a technology that enables computers and machines to simulate human intelligence. This includes learning from data, understanding information, solving problems, making decisions, and sometimes even exhibiting creativity and autonomy.

Weak AI: Also known as “narrow AI,” defines AI systems designed to perform a specific task or a set of tasks. Examples might include “smart” voice assistant apps, such as Amazon’s Alexa, Apple’s Siri, a social media chatbot or the autonomous vehicles promised by Tesla.

Strong AI Also known as “artificial general intelligence” (AGI) or “general AI,” possess the ability to understand, learn and apply knowledge across a wide range of tasks at a level equal to or surpassing human intelligence.

Machine Learning (ML) is a subset of Artificial Intelligence that enables computers to learn patterns from data and make predictions or decisions without being explicitly programmed. In simple terms, ML allows machines to improve their performance automatically as they are exposed to more data.

Deep Learning is a subset of Machine Learning that uses multilayered neural networks, called deep neural networks, to model complex patterns and simulate human-like decision-making.

Generative AI, or Gen AI, refers to deep learning models that can create original content—such as text, images, audio, or video—based on a user’s input. These models learn patterns from existing data to generate realistic and creative outputs.

Types of Machine Learning

1. Supervised Learning

- Definition: The model learns from labeled data, where both inputs and desired outputs are provided, to make predictions on new data.
- Example: Predicting house prices based on features like size and location.

2. Unsupervised Learning

- Definition: The model analyzes unlabeled data to identify hidden patterns, structures, or groupings without predefined outputs.
- Example: Customer segmentation based on purchasing behavior.

3. Semi-Supervised Learning

- Definition: The model uses a combination of labeled and unlabeled data to improve learning efficiency and accuracy.
- Example: Image recognition where only some images are labeled.

4. Reinforcement Learning

- Definition: The model learns by interacting with an environment, receiving rewards or penalties, and optimizing actions to achieve long-term goals.
- Example: Training a robot to walk or a system to play games like chess or Go.

Difference between Regression Problem and Classification Problem

1. Regression predicts a number, classification predicts a category
2. Regression uses Mean Squared Error (MSE), Root Mean Squared Error (RMSE), R^2 as its evaluation metrics and Classification uses Accuracy, Precision, Recall, F1-score as its evaluation metrics

NEURAL NETWORK

A neural network (also called an artificial neural network) is an adaptive system that learns by using interconnected nodes or neurons in a layered structure that resembles a human brain.

It consists of an input layer, one or more hidden layers, and an output layer. In each layer there are several nodes, or neurons, with each layer using the output of the previous layer as its input, so neurons interconnect the different layers.

Neural network's behaviour is defined by the way its individual elements are connected and by the strength, or weights of those connections. These weights are automatically adjusted during training according to a specified learning rule until the artificial neural network performs the desired task correctly.

A shallow neural network has only one hidden layer between the input and output. A deep neural network has two or more hidden layers between input and output.

NEURON'S LAYERS

Neurons are organized into multiple layers — an input layer, hidden layers, and an output layer.

1. Input Layer: This is where the network receives input from your dataset. It's often transformed into a suitable form for the network.

2. Hidden Layer(s): These are the layers in between the input and output layers. They perform transformations on the input data with the goal of learning features that are useful for the task at hand. The term "deep" in deep learning refers to having multiple hidden layers in the network.

3. Output Layer: This is the final layer. It provides the predictions or classifications that the network has been trained to produce.

PERCEPTRON

A perceptron is a type of artificial neuron or linear binary classifier that takes multiple input values, applies corresponding weights, adds a bias, and passes the result through an activation function to produce an output and is primarily used for binary classification tasks.

It was introduced by Frank Rosenblatt in 1958 and serves as the building block of more complex neural networks.

WEIGHTS AND BIAS

Weights determine how strongly each input feature influences the output of a neuron and its importance is influenced by its scaling.

Bias allows the activation function to be shifted left or right, helping the model better fit the data.

ACTIVATION FUNCTIONS

An Activation Function decides whether a neuron should be activated or not. This means that it will decide whether the neuron's input to the network is important or not in the process of prediction using simpler mathematical operations.

The primary role of the Activation Function is to transform the summed weighted input from the node into an output value to be fed to the next hidden layer or as output.

Activation functions introduces non-linearity in neural structure and without non-linearity, no matter how many layers the network has, it would behave just like a single layer network because summing these layers would give another linear function.

Non-linear activation functions allow the network to learn from the error, and adjust its weights and bias accordingly, so as to improve the model during training.

LOSS FUNCTION

Loss function is a mathematical function that measures the difference between the model's predicted output and the actual target value. It tells the model how wrong its predictions are. And it is also used to quantify errors.

VANISHING GRADIENT

The vanishing gradient problem in deep learning occurs during the training of deep neural networks when the gradients (used to update the network's weights via backpropagation) become extremely small—almost vanishing—as they propagate backward from the output layer to earlier layers. This causes the early layers of the network to update their weights very slowly or not at all, which significantly slows down learning or can even halt it completely.

This issue is especially prominent with activation functions like sigmoid and hyperbolic tangent (tanh), whose derivatives fall between 0 and 1 and approach zero when inputs saturate. As backpropagation multiplies these small derivatives layer by layer, the gradients shrink exponentially, making weight updates in earlier layers negligible. Consequently, the network struggles to learn from data in those layers, impeding effective training of deep networks.

How to recognise Vanishing Gradient Problem -

1. Calculate loss using Keras and if its consistent during epochs that means it is Vanishing Gradient Problem.

2. Draw the graphs between weights and epochs and if it is constant that means weight has not changed and hence vanishing gradient problem.

Several solutions help mitigate this problem -

1. Using activation functions like ReLU (Rectified Linear Unit) or its variants (leaky ReLU), which have gradients that do not vanish for positive inputs.

2. Implementing batch normalization to stabilize gradient flow.

3. Employing architectures with skip connections or residual networks (ResNets), allowing gradients to bypass certain layers.

4. Using gated recurrent units like LSTMs in sequence models.

5. Proper weight initialization and gradient clipping to maintain gradient magnitudes.

HOW IS DEEP LEARNING DIFFERENT AND BETTER FROM MACHINE LEARNING APPROACHES

Deep Learning differs from traditional Machine Learning by automatically learning hierarchical features, handling complex unstructured data, scaling with massive datasets, and achieving higher accuracy on tasks like computer vision, NLP, and speech recognition, whereas ML relies on manual feature engineering and works best for simpler, structured problems.

HISTORY OF AI

1950: Alan Turing asks "Can machines think?" and introduces the Turing Test to evaluate machine intelligence.

1956: John McCarthy coins “Artificial Intelligence,” and the first AI program, Logic Theorist, is created.

1967: Frank Rosenblatt builds the first neural network, the Mark 1 Perceptron, sparking early neural network research.

1980: Neural networks using backpropagation gain widespread use in AI applications.

1995: Russell and Norvig publish *Artificial Intelligence: A Modern Approach*, defining AI through rationality and thinking vs. acting.

1997: IBM’s Deep Blue defeats world chess champion Garry Kasparov.

2004: John McCarthy proposes a formal AI definition amid the rise of big data and cloud computing.

2011: IBM Watson wins *Jeopardy!* and data science emerges as a popular field.

2015: Baidu’s Minwa uses convolutional neural networks to outperform humans in image recognition.

2016: DeepMind’s AlphaGo defeats Go champion Lee Sedol, highlighting AI’s strategic capabilities.

2022: Large language models like ChatGPT revolutionize AI performance and enterprise applications.

2024: Multimodal AI models combine vision and language processing, and smaller efficient models advance alongside massive ones.

Internal Covariate Shift and Batch Normalization in Deep Learning

1. Internal Covariate Shift

Internal Covariate Shift (ICS) refers to the change in the distribution of inputs to a layer during training, caused by updates in the parameters of the previous layers. As each layer's parameters are updated, the output distribution changes, and the next layer must constantly adapt to new input distributions. This phenomenon can slow convergence and make optimization more difficult.

Key Characteristics:

- • Analogous to covariate shift in statistics, but occurs within the neural network.
- • Caused by weight updates during training that shift activation distributions.
- • Can lead to vanishing/exploding gradients in deep architectures.
- • Accumulates in deeper layers, increasing instability.

Impact on Training:

- • Slower convergence due to constantly changing distributions.
- • Requires smaller learning rates for stability.
- • Can lead to sub-optimal local minima.

2. Batch Normalization

Batch Normalization (BatchNorm) is a technique introduced by Sergey Ioffe and Christian Szegedy in 2015 to address ICS and improve the speed, performance, and stability of deep networks. It normalizes the input to each layer so that they have a mean close to 0 and variance close to 1, followed by a learnable scaling and shifting.

Step-by-Step Process:

1. 1. For each feature, calculate the mini-batch mean μ_B and variance σ^2_B.
2. 2. Normalize: subtract μ_B and divide by $\sqrt{(\sigma^2\_B + \varepsilon)}$, where ε is a small constant for stability.
3. 3. Apply learnable scale (γ) and shift (β) parameters to allow the network to recover original representations if needed.

Mathematical Formulation:

- • $\mu\_B = (1/m)\ \Sigma\ x_i$
- • $\sigma^2\_B = (1/m)\ \Sigma\ (x_i - \mu\_B)^2$
- • $\hat{x}_i = (x_i - \mu\_B) / \sqrt{(\sigma^2\_B + \varepsilon)}$
- • $y_i = \gamma\ \hat{x}_i + \beta$

Advantages of BatchNorm:

- • Reduces ICS and stabilizes learning.
- • Enables larger learning rates.
- • Provides regularization effect, reducing overfitting.
- • Helps maintain gradient flow, mitigating vanishing/exploding gradients.
- • Can make the network less sensitive to weight initialization.

Limitations:

- • Performance depends on batch size; small batches can cause noise in mean/variance estimates.
- • Adds computation and memory overhead.
- • Different behavior during training and inference (uses running estimates during inference).
- • May not be ideal for certain architectures like RNNs without modifications.

During Inference:

- • The mean and variance used are the moving averages computed during training.
- • Ensures deterministic outputs and stable predictions.

3. Relationship Between ICS and BatchNorm

BatchNorm was initially proposed as a direct solution to Internal Covariate Shift by keeping intermediate feature distributions stable. However, later research suggested its benefits also stem from smoothing the optimization landscape and improving gradient conditioning. While BatchNorm reduces ICS, its primary advantage may be enabling better training dynamics overall.

4. Variants and Alternatives

- • Layer Normalization – Normalizes across features for each sample, useful in RNNs.
- • Instance Normalization – Normalizes each channel separately, common in style transfer.
- • Group Normalization – Normalizes over groups of channels, works well for small batches.
- • Weight Normalization – Normalizes weights rather than activations.

5. Practical Tips

- • Place BatchNorm before the activation function in most cases (e.g., before ReLU).
- • Adjust momentum parameter in frameworks for better moving average estimates.
- • For very small batches, consider GroupNorm or LayerNorm instead.
- • Remember that BatchNorm parameters γ and β are trainable and can adapt.

Layer Normalization in Deep Learning

Layer Normalization (LayerNorm) is a normalization technique used in deep learning to stabilize and accelerate training by normalizing the activations across the features for each training example. Unlike Batch Normalization, which normalizes across the batch, LayerNorm operates independently for each sample, making it effective in settings with small batch sizes or sequential data.

1. How Layer Normalization Works

Given an input vector for a single training example, LayerNorm:

1. 1. Computes the mean and variance across all features of that example.
2. 2. Normalizes each feature by subtracting the mean and dividing by the standard deviation.
3. 3. Applies learnable scale (γ) and shift (β) parameters to allow the network to adapt.

2. Mathematical Formulation

Given an input vector $x = [x_1, x_2, \ldots, x\_H]$ with H features:

- • Mean: $\mu = (1/H)\ \Sigma\ x_j$
- • Variance: $\sigma^2 = (1/H)\ \Sigma\ (x_j - \mu)^2$
- • Normalization: $\hat{x}_j = (x_j - \mu) / \sqrt{}(\sigma^2 + \varepsilon)$
- • Scaling & Shifting: $y_j = \gamma\ \hat{x}_j + \beta$

3. Advantages of LayerNorm

- • Works well with small batch sizes, even batch size = 1.
- • Effective in recurrent neural networks (RNNs) where batch statistics vary across time steps.
- • Invariant to batch size, providing consistent performance.
- • Reduces training instability and accelerates convergence.

4. Limitations of LayerNorm

- • Slightly more computation per sample compared to BatchNorm.
- • May be less effective than BatchNorm for large convolutional networks with large batch sizes.

5. Applications of LayerNorm

- • Transformers (used in attention layers for NLP and vision models).
- • RNNs, LSTMs, and GRUs to stabilize training across time steps.
- • Scenarios with small batch sizes where BatchNorm fails.

6. Comparison with Batch Normalization

- • BatchNorm normalizes across the batch dimension, LayerNorm normalizes across features for each sample.
- • BatchNorm requires consistent batch statistics, LayerNorm does not.
- • BatchNorm’s performance can degrade for small batches; LayerNorm is unaffected.

Gradient Descent Optimizers in Deep Learning - Detailed Explanation

Batch Gradient Descent

Explanation: Batch Gradient Descent computes the gradient of the cost function with respect to the parameters for the entire dataset at each iteration. It guarantees convergence to the global minimum for convex functions and to a local minimum for non-convex functions. However, it can be slow for large datasets, and each update requires going through all training samples. It is best suited for small datasets where computational resources are not a constraint.

Formula: $\theta = \theta - \eta \nabla J(\theta)$

Explained Formula: Here, θ represents the model parameters (weights), η is the learning rate controlling the step size, and $\nabla J(\theta)$ is the gradient of the cost function with respect to θ. The update subtracts a fraction of the gradient from the current parameters to move towards the minimum.

When to Use: Use when datasets are small and you want stable, precise convergence.

Pros: Stable convergence; theoretically exact for convex problems.

Cons: Slow for large datasets; high memory requirements.

Stochastic Gradient Descent (SGD)

Explanation: SGD updates the parameters for each training example individually, making updates noisy but fast. The noise can help escape local minima, making it useful for non-convex optimization. However, due to the noise, the path to convergence can be irregular, requiring learning rate schedules or decay to stabilize the final solution.

Formula: $\theta = \theta - \eta \nabla J(\theta; x_i, y_i)$

Explained Formula: θ is updated using the gradient calculated from a single data point (x_i, y_i). This results in faster updates per iteration compared to batch gradient descent, but with more variance in the updates.

When to Use: Use when datasets are too large for batch processing.

Pros: Fast iterations; can escape local minima; low memory usage.

Cons: Noisy convergence; requires careful learning rate tuning.

Mini-Batch Gradient Descent

Explanation: Mini-Batch Gradient Descent splits the training dataset into small batches and updates the model parameters for each batch. This combines the advantages of both batch and stochastic gradient descent by reducing variance in updates while allowing vectorized computation for efficiency. It is the most commonly used optimizer in deep learning.

Formula: $\theta = \theta - \eta \nabla J(\theta; B)$

Explained Formula: Here, B is a batch of data points. The gradient is computed over this small subset of the dataset, making the update faster than batch gradient descent and less noisy than SGD.

When to Use: Default choice for deep learning on large datasets.

Pros: Efficient computation; more stable than SGD.

Cons: Requires tuning of batch size; still some variance in updates.

Momentum

Explanation: Momentum helps accelerate gradient descent by accumulating a moving average of past gradients in the direction of consistent improvement. It reduces oscillations, especially in scenarios where the gradient direction changes frequently (e.g., narrow valleys).

Formula: $v_t = \beta v_{t-1} + \eta \nabla J(\theta); \theta = \theta - v_t$

Explained Formula: v_t is the velocity term, β is the momentum coefficient ($0 < \beta < 1$), η is the learning rate, and $\nabla J(\theta)$ is the gradient. The velocity smooths updates, allowing faster movement in consistent directions.

When to Use: When gradients oscillate; to accelerate convergence.

Pros: Speeds up convergence; reduces oscillations.

Cons: May overshoot minima; requires tuning β.

Nesterov Accelerated Gradient (NAG)

Explanation: NAG improves on momentum by first making a lookahead step using the current velocity, then computing the gradient at that approximate future position. This provides more accurate gradient estimates, leading to better convergence.

Formula: $v_t = \beta v_{t-1} + \eta \nabla J(\theta - \beta v_{t-1}); \theta = \theta - v_t$

Explained Formula: First, a lookahead position $\theta - \beta v_{t-1}$ is computed, and the gradient is calculated at this point. Then, velocity and parameters are updated accordingly.

When to Use: When momentum is helpful but needs more precise updates.

Pros: More accurate updates; faster convergence.

Cons: Slightly more computationally expensive per iteration.

Adagrad

Explanation: Adagrad adapts the learning rate for each parameter individually based on the history of gradients. It performs larger updates for parameters with infrequent updates and smaller updates for parameters with frequent updates, making it ideal for sparse data.

Formula: $\theta = \theta - \eta / (\sqrt{G_t} + \varepsilon) \nabla J(\theta)$

Explained Formula: G_t is the sum of the squares of past gradients for each parameter. The learning rate η is divided by the root of G_t plus a small constant ε to avoid division by zero.

When to Use: When working with sparse data like text features.

Pros: No need to tune learning rate manually; works well with sparse features.

Cons: Learning rate decreases over time, possibly becoming too small.

RMSprop

Explanation: RMSprop maintains an exponentially decaying average of squared gradients and uses it to normalize updates. It is designed to fix Adagrad's diminishing learning rate issue, making it effective for non-stationary objectives.

Formula: $E[g^2]_t = \beta E[g^2]_{t-1} + (1-\beta) g^2_t$; $\theta = \theta - \eta / (\sqrt{E[g^2]_t} + \varepsilon) g_t$

Explained Formula: The denominator contains an exponentially weighted moving average of squared gradients, which helps maintain a consistent learning rate.

When to Use: When training RNNs or dealing with non-stationary problems.

Pros: Prevents diminishing learning rate; effective in practice.

Cons: Requires tuning β and η carefully.

Adam

Explanation: Adam combines the ideas of momentum and RMSprop. It maintains exponentially decaying averages of past gradients and squared gradients, applies bias correction, and updates parameters adaptively. It is widely used in deep learning.

Formula: $m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$; $v_t = \beta_2 v_{t-1} + (1-\beta_2) g^2_t$; $\theta = \theta - \eta (\hat{m}_t / (\sqrt{\hat{v}_t} + \varepsilon))$

Explained Formula: m_t is the first moment (mean of gradients), v_t is the second moment (mean of squared gradients), and $\hat{m}_t$, $\hat{v}_t$ are bias-corrected estimates. The parameters are updated using these adjusted moments.

When to Use: Default choice for most deep learning problems.

Pros: Works well without much tuning; fast convergence.

Cons: May overfit; can be sensitive to hyperparameters.

AdamW

Explanation: AdamW is a variant of Adam that decouples weight decay from the gradient update step. This improves generalization and has become the default choice in many transformer models.

Formula: $\theta = \theta - \eta\ (\hat{m}_t\ \ / (\sqrt{\hat{v}_t}\ \ + \varepsilon) + \lambda\theta)$

Explained Formula: Adam updates are computed normally, and a separate weight decay term $\lambda\theta$ is applied to the parameters to regularize them.

When to Use: When using Adam but needing better regularization.

Pros: Improved generalization; better than L2 regularization with Adam.

Cons: Requires tuning λ carefully.

What is generalization

In supervised learning, the main goal is to use training data to build a model that will be able to make accurate predictions based on new, unseen data, which has the same characteristics as the initial training set. This is known as generalization.

Generalization error

To train a machine learning model, you split the dataset into 3 sets: training, validation, and testing. we train your models using the training data, then we compare and tune them using the evaluation results on the validation set, and in the end, evaluate the performance of your best model on the testing set. The error rate on new cases is called the generalization error (or out-of-sample error)

A model's generalization error (also known as a prediction error) can be expressed as the sum of three very different errors: Bias error, variance error, and irreducible error.

Note: The irreducible error occurs due to the noisiness of the data itself. The only way to reduce this part of the error is to clean up the data (e.g., fix the data sources, such as broken sensors, or detect and remove outliers).

Bias error

Characteristics of a high-bias model include:

- Failure to capture proper data trends
- Potential towards underfitting
- More generalized/overly simplified
- High error rate

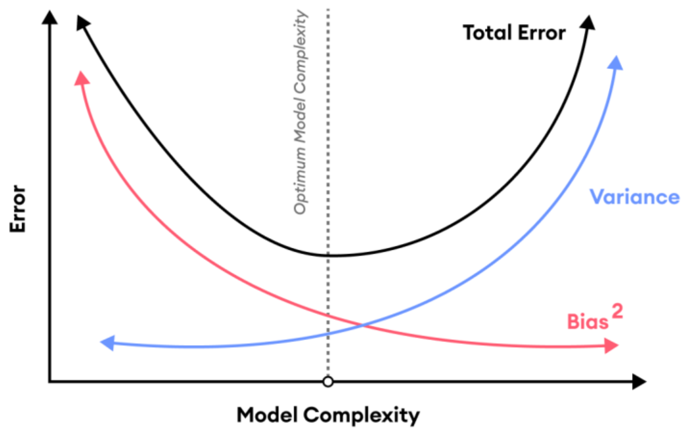
Variance error

A high-variance model typically has the following qualities:

- Noise in the data set
- Potential towards overfitting
- Complex models

Bias-variance tradeoff

Bias/variance in machine learning relates to the problem of simultaneously minimizing two error sources (bias error and variance error).



In order to find a balance between underfitting and overfitting (the best model possible), you need to find a model which minimizes the total error.

Total Error = Bias²+ Variance + Irreducible Error

What is underfitting

Underfitting occurs when a model is not able to make accurate predictions based on training data and hence, doesn't have the capacity to generalize well on new data.

Machine learning models with underfitting tend to have poor performance both in training and testing sets (like the child who learned only addition and was not able to solve problems related to other basic arithmetic operations both from his math problem book and during the math exam).

Underfitting models usually have high bias and low variance.

What is overfitting

A model is considered overfitting when it does extremely well on training data but fails to perform on the same level on the validation data (like the child who memorized every math problem in the problem book and would struggle when facing problems from anywhere else).

Models that are overfitting usually have low bias and high variance.

How to detect underfitting

1) **Training and test loss:** If the model is underfitting, the loss for both training and validation will be considerably high. In other words, for an underfitting dataset, the training and the validation error will be high.

2) **Oversimplistic prediction graph:** If a graph with the data points and the fitted curve is plotted, and the classifier curve is oversimplistic, then, most probably, your model is underfitting. In those cases, a more complex model should be tried out.

How to avoid underfitting

There are several things you can do to prevent underfitting in AI and machine learning models:

1) **Train a more complex model** – Lack of model complexity in terms of data characteristics is the main reason behind underfitting models. For example, you may have data with upwards of 100000 rows and more than 30 parameters. If you train data with the Random Forest model and set max depth (max depth determines the maximum depth of the tree) to a small number (for example, 2), your model will definitely be underfitting. Training a more complex model (in this respect, a model with a higher value of max depth) will help us solve the problem of underfitting.

2) **More time for training** - Early training termination may cause underfitting. As a machine learning engineer, you can increase the number of epochs or increase the duration of training to get better results.

3) **Eliminate noise from data** – Another cause of underfitting is the existence of outliers and incorrect values in the dataset. Data cleaning techniques can help deal with this problem.

4) **Adjust regularization parameters** - the regularization coefficient can cause both overfitting and underfitting models.

How to detect overfitting

Some of the techniques you can use to detect overfitting are as follows:

1) Use a resampling technique to estimate model accuracy. The most popular resampling technique is k-fold cross-validation. It allows you to train and test your model k-times on different subsets of training data and build up an estimate of the performance of a machine learning model on unseen data. The drawback here is that it is time-consuming and cannot be applied to complex models, such as deep neural networks.

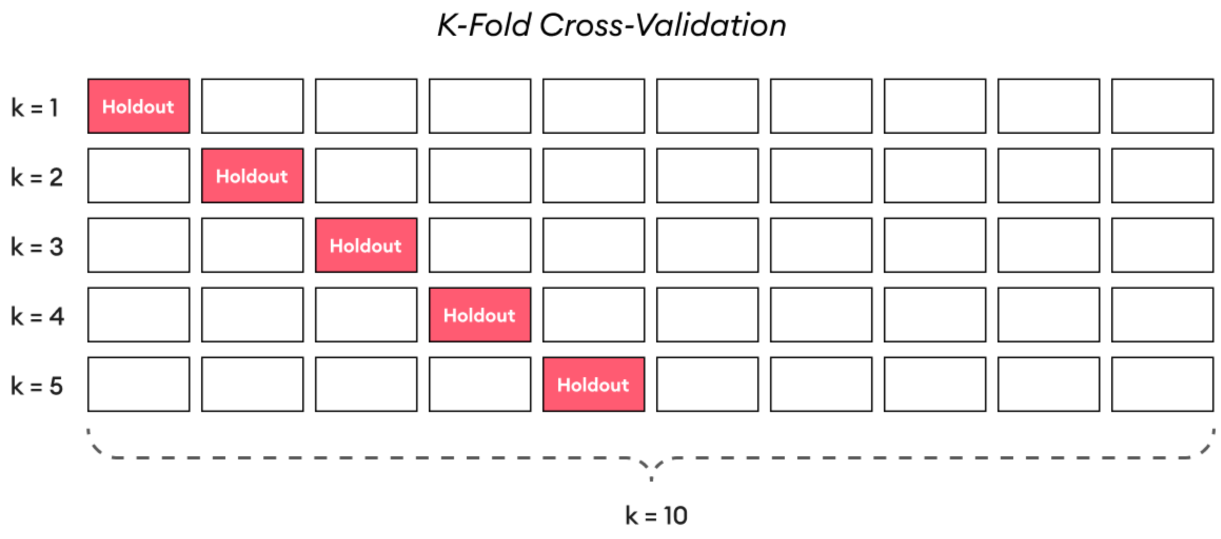


Figure 6. K-fold cross-validation: [Image source]

2) Hold back a validation set. Once a model is trained on the training set, you can evaluate it on the validation dataset, then compare the accuracy of the model in the training dataset and the validation dataset. A significant variance in these two results allows assuming that you have an overfitted model.

3) Another way to detect overfitting is by starting with a simplistic model that will serve as a benchmark. With this approach, if you try more complex algorithms, you will have a general understanding of whether the additional complexity for the model is worthwhile, if at all. This principle is known as *Occam's razor* test. This principle suggests that with all else being equal, simpler solutions to problems are preferred over more complex ones (if your model is not getting significantly better after using a much more complex model, it is preferable to use a simpler model).

How to prevent overfitting

Some of those methods are listed below.

1) **Adding more data** – Most of the time, adding more data can help machine learning models detect the "true" pattern of the model, generalize better, and prevent overfitting. However, this is not always the case, as adding more data that is inaccurate or has many missing values can lead to even worse results.

2) **Early stopping** – In iterative algorithms, it is possible to measure how the model iteration performance. Up until a certain number of iterations, new iterations improve the model. After that point, however, the model's ability to generalize can deteriorate as it begins to overfit the training data. Early stopping refers to stopping the training process before the learner passes that point.

3) **Data augmentation** – In machine learning, data augmentation techniques increase the amount of data by slightly changing previously existing data and adding new data points or by producing synthetic data from a previously existing dataset.

4) **Remove features** – You can remove irrelevant aspects from data to improve the model. Many characteristics in a dataset may not contribute much to prediction. Removing non-essential characteristics can enhance accuracy and decrease overfitting.

5) **Regularization** – Regularization refers to a variety of techniques to push your model to be simpler. The approach you choose will be determined by the model you are training. For example, you can add a penalty parameter for a regression (L1 and L2 regularization), prune a decision tree or use dropout on a neural network.

6) **Ensembling** – Ensembling methods merge predictions from numerous different models. These methods not only deal with overfitting but also assist in solving complex machine learning problems (like combining pictures taken from different angles into the overall view of the surroundings). The most popular ensembling methods are boosting and bagging.

- **Boosting** – In boosting method, you train a large number of weak learners (constrained models) in sequence, and each sequence learns from the mistakes of the previous sequence. Then you combine all weak learners into a single strong learner.

- **Bagging** is another technique to reduce overfitting. It trains a large number of strong learners (unconstrained models) and then combines them all in order to optimize their predictions.

* LINEAR REGRESSION

aim:- to predict a dependent variable by establishing a linear relationship between one or more independent variables.

·) Simple Linear Regression

(one independent feature and one dependent variable)

model :- $f_{\omega,b}(x) = \omega x + b$

training :- ① aim — to find the best fit line ($f_{\omega,b}(x)$)
Part

Ⓘ known — $f_{\omega,b}(x)$ and x

unknown — ω and b

ⓘⓘⓘ Goal — ·) to minimize errors (residuals).

·) find ω and b

ⓘⓥ

$$\begin{tikzcd}
\text{Cost Function (represent residuals)} \arrow[r] & \text{Minimize Cost Function (minimize errors)} \arrow[dr] & \\
\text{found the value of 'ω' and 'b'} \arrow[dr] & \text{Convergence Algorithm (to find the value of ω and b)} \arrow[l] & \text{Global Minima} \arrow[l] \\
& \text{will get the equation of best fit line} &
\end{tikzcd}$$

(v) Cost function

$$J(\omega, b) = \frac{1}{2m} \cdot \sum \left(f_{\omega,b}(x^{(i)}) - y^{(i)} \right)^2 = \frac{1}{2m} \cdot \sum_{i=1}^{m} \left(\hat{y}^{(i)} - y^{(i)} \right)^2$$

Convergence Algorithm

$$\{$$

$$\omega = \omega - \alpha \cdot \frac{\partial J(\omega, b)}{\partial \omega}$$

$$b = b - \alpha \cdot \frac{\partial J(\omega, b)}{\partial b}$$

$$\}$$

where, $\dfrac{\partial J(\omega, b)}{\partial \omega} = \dfrac{1}{m} \cdot \sum_{i=1}^{m} \left(f_{\omega,b}(x^{(i)}) - y^{(i)} \right) \cdot x^{(i)}$

$$\frac{\partial J(\omega, b)}{\partial b} = \frac{1}{m} \cdot \sum_{i=1}^{m} \left(f_{\omega,b}(x^{(i)}) - y^{(i)} \right)$$

* Types of Cost Function :-

(i) MSE ≡ Mean Squared Error ≡ $\frac{1}{m} \cdot \sum_{i=1}^{m} (\hat{y} - y)^2$

-> easily differentiable } advantages
-> one global minima }

->, not robust to outliers. } disadvantages

(ii) MAE ≡ Mean Absolute Error ≡ $\frac{1}{m} \cdot \sum_{i=1}^{m} |\hat{y} - y|$

-> robust to outliers } advant.

-> convergence takes more time. } disadv.

(iii) RMSE ≡ Root Mean Squared Error ≡ $\sqrt{\frac{1}{m} \cdot \sum_{i=1}^{m} (\hat{y} - y)^2}$

-> easily differentiable } advan.

-> not robust to outliers. } disadv.

* Performance Metrics

(i) R-Squared :- indicates how well the independent variables in a linear regression model explain the variation in the dependent variable.

(≤ 1)

$$R^2 \equiv 1 - \frac{\sum_{i=1}^{m} (y_i - \hat{y}_i)^2}{\sum_{i=1}^{m} (y_i - \bar{y})^2} \equiv 1 - \frac{TSS}{RSS}$$

where :- y_i ≡ actual value of dependent variable

$\hat{y}_i$ ≡ predicted value from regression model.

$\bar{y}$ ≡ mean of actual values of dependent variable.

-> if its value is (-)ve, then the model is not working good.

-> Value of R-Squared increases, if more features are added, even if they don't increase model's ability.

$$R \equiv 1 - \frac{\sum_{i=1} (y_i\ \ y_i)}{\sum_{i=1}^{m} (y_i - \bar{y})^2} = \frac{}{RSS}$$

where :- $y_i \equiv$ actual value of dependent variable

$\hat{y}_i \equiv$ predicted value from regression model.

$\bar{y} \equiv$ mean of actual values of dependent variable.

-> if its value is (−)ve , then the model is not working good.

-> Value of R-Squared increases , if more features are added, even if they don't increase model's ability.

(ii) Adjusted R-Squared :-

$$\text{adjusted R-Squared} \equiv 1 - \left(\frac{(1 - R^2)(m-1)}{m - p - 1} \right)$$

where, $m \equiv$ number of data-points (observations)

$p \equiv$ number of predictors (independent variables) in the model.

-> Provides better evaluation to model.

* REGULARIZATION (to avoid overfitting , which then lead to more generalized model with little less accurate)

* RIDGE REGRESSION

(L2 Regularization

(i) used to reduce overfitting by introducing bias in the data , which led to decrease the variance in data , by which test data will be near to best - fit - line and then performance of model will be better.

Cost Function $\equiv J(\omega, b) \equiv \frac{1}{2m} \cdot \sum_{i=1}^{m} \left(\hat{y}^{(i)} - y^{(i)} \right)^2 + \underbrace{\frac{\lambda}{2m} \cdot \sum_{j=1}^{n} \omega_j^2}_{\text{penalty}}$

when, $\lambda = 0$, then cost fcn. will be same as of linear regression.

(ii) Global minima shift towards left with increase in value of λ.

(iii) $\lambda \propto \frac{1}{\text{Slope}}$

(ii) Global minima shift towards left with increase in value of λ.

(iii) $\lambda \propto \frac{1}{\text{Slope}}$

* LASSO REGRESSION

(i) Stands for Least Absolute Shrinkage and Selection operator Regression.

(ii) It tends to eliminate the weights of the least important features by setting their weights to 0.

(iii) Cost Function $\equiv J(\omega, b) \equiv \frac{1}{2m} \cdot \sum_{i=1}^{m} \left(\hat{y}^{(i)} - y^{(i)} \right)^2 + \underbrace{\frac{\lambda}{2m} \cdot \sum_{j=1}^{m} |\omega_j|}_{\text{penalty}}$

* LOGISIC REGRESSION

aim :- is to predict the probability of a specific outcome (for a binary classification problem) by determining the relationship input features and target outcome using Sigmoid fun.

Sigmoid fun is $f(x) = \frac{1}{1+e^{-x}}$.

Simple Logistic Regression

model :- $f_{\omega,b}(x) = \frac{1}{1+e^{-(\vec{\omega}\cdot\vec{x}+b)}} \equiv \frac{1}{1+e^{-(\omega^T x+b)}}$

:- (i) aim – We have to separate or classify data-points into two-classes using decision boundary.

(ii) Eqn. of Decision boundary – $\boxed{\vec{\omega}\cdot\vec{x}+b=0} \equiv \boxed{\omega^T x+b=0}$

(iii) unknown – ω and b (iv) goal – to find eqn. of decision boundary

(iv) working –

$$\begin{tikzcd}
\begin{array}{c}\text{Cost Function}\\ \text{(represent residuals)}\end{array} \arrow[r] & \begin{array}{c}\text{Minimize Cost Function}\\ \text{(minimize errors)}\end{array} \arrow[rd] & \\
\begin{array}{c}\text{found the value}\\ \text{of '}\omega\text{' and 'b'}\end{array} \arrow[rd] & \begin{array}{c}\text{Convergence Algorithm}\\ \left(\begin{array}{c}\text{to find the value}\\ \text{of } \omega \text{ and } b\end{array}\right)\end{array} \arrow[l] & \begin{array}{c}\text{Global}\\ \text{Minima}\end{array} \arrow[l] \\
& \begin{array}{c}\text{will get the}\\ \text{equation of}\\ \text{decision boundary}\end{array} &
\end{tikzcd}$$

[Decision Boundary or Hyperplane doesn't make predictions as, it is used to divide datasets into classes]

(v) Formulas :-

Cost Function $\equiv J(\omega, b) = \frac{1}{m} \cdot \sum_{i=1}^{m} L(f_{\omega,b}(x^{(i)}), y^{(i)})$

where;

$$L(f_{\omega,b}(x^{(i)}), y^{(i)}) = \begin{cases} -\log(f_{\omega,b}(x^{(i)})) & \text{if } y^{(i)} = 1 \\ -\log(1 - f_{\omega,b}(x^{(i)})) & \text{if } y^{(i)} = 0 \end{cases}$$

$$\equiv J(\omega, b) = \frac{-1}{m} \cdot \sum_{i=1}^{m} \left[y^{(i)} \cdot \log(f_{\omega,b}(x^{(i)})) + (1 - y^{(i)}) \cdot \log(1 - f_{\omega,b}(x^{(i)})) \right]$$

Convergence Algorithm $\equiv$

$$\left\{ \begin{aligned} \omega_j &= \omega_j - \alpha \cdot \frac{\partial J(\omega, b)}{\partial \omega_j} \\ b &= b - \alpha \cdot \frac{\partial J(\omega, b)}{\partial b} \end{aligned} \right\}$$

(vi) For using Logistic Regression, dataset should be free of missing values and all independent variables must be independent and not strongly correlated to each other.

(vii) Logistic Regression normally doesn't work so well with large datasets and especially messy data containing outliers, complex relationships and missing values.

LOGISTIC REGRESSION

Types of Logistic Regression:

1. Binary Logistic Regression:

- **Description**: The standard form of logistic regression where the dependent variable Y is binary (e.g., 0 or 1, yes or no).
- **Application**: Used when the outcome of interest is binary and there are one or more predictor variables X influencing the probability of the outcome.

2. Multinomial Logistic Regression:

- **Description**: Extends logistic regression to handle scenarios where the dependent variable Y has more than two categories (i.e., it's multinomial rather than binary).
- **Application**: Useful when the outcome variable has multiple discrete categories that are not ordered, and the goal is to predict the probability of each category relative to a base category.

3. Ordinal Logistic Regression:

- **Description**: Used when the dependent variable Y represents ordered categories (ordinal data), such as low, medium, high.
- **Application**: Suitable for predicting the probability of the dependent variable falling into each ordered category based on predictor variables X

Performance Metrics

True Positive - An outcome which is actually positive and classified as positive, then it is called **TRUE POSITIVE (*TP*).**

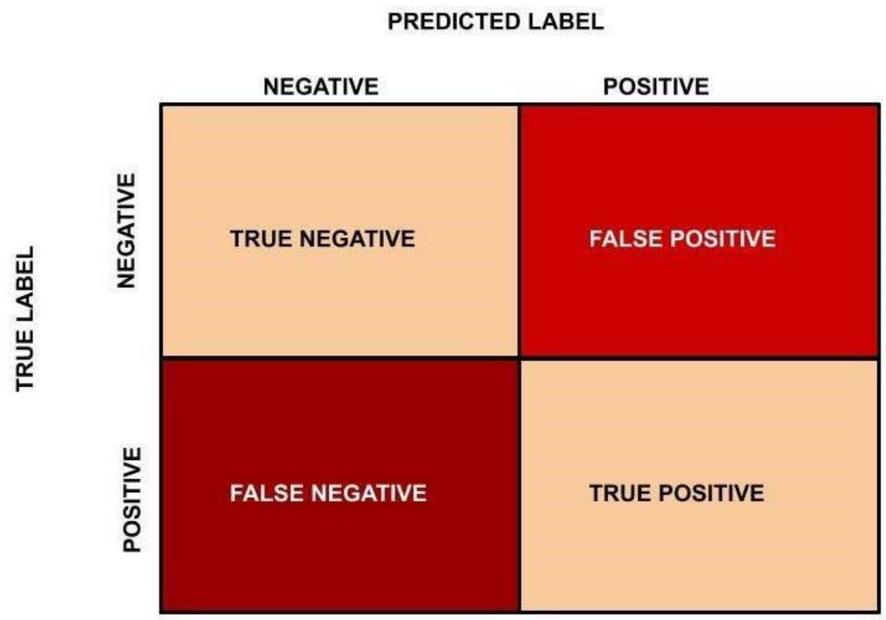
True Negative - An outcome which is actually negative and classified as negative, then it is called **TRUE NEGATIVE (*TN*).**

False Positive - An outcome which is actually negative and classified as positive, then it is called **FALSE POSITIVE (*FP*).**

False Negative - An outcome which is actually positive and classified as negative, then it is called **FALSE NEGATIVE (*FN*).**

Confusion matrix

A confusion matrix is a performance evaluation tool used in classification problems. It shows how well a classification model is performing by comparing the predicted labels with the actual labels.



Accuracy

It measures how many observations, both positive and negative, were correctly classified.

$$ACC = \frac{tp + tn}{tp + fp + tn + fn}$$

You shouldn't use accuracy on imbalanced problems. Then, it is easy to get a high accuracy score by simply classifying all observations as the majority class.

Precision

$$Precision = \frac{TP}{TP + FP}$$

Precision should ideally be 1 (high) for a good classifier. ***Precision*** becomes 1 only when the numerator and denominator are equal i.e ***TP = TP +FP***, this also means ***FP*** is zero. As ***FP*** increases the value of denominator becomes greater than the numerator and ***precision*** value decreases (which we don't want).

Recall

$$Recall = \frac{TP}{TP + FN}$$

Recall should ideally be 1 (high) for a good classifier. ***Recall*** becomes 1 only when the numerator and denominator are equal i.e ***TP = TP +FN***, this also means ***FN*** is zero. As ***FN*** increases the value of denominator becomes greater than the numerator and ***recall*** value decreases (which we don't want).

F1 score

Simply put, it combines precision and recall into one metric by calculating the harmonic mean between those two. It is actually a special case of the more general function F beta:

$$F_{beta} = (1+\beta^2)\frac{precision * recall}{\beta^2 * precision + recall}$$

When choosing beta in your F-beta score, the more you care about recall over precision, the higher beta you should choose. For example, with the F1 score, we care equally about recall and precision; with the F2 score, recall is twice as important to us

With 0<beta<1, we care more about precision, and so the higher the threshold, the higher the F beta score. When beta > 1, our optimal threshold moves toward lower thresholds, and when beta = 1, it is somewhere in the middle.

When should you use it?

- Pretty much in every binary classification problem where you care more about the positive class. It is my go-to metric when working on those problems.
- It can be easily explained to business stakeholders, which in many cases can be a deciding factor. Always remember that machine learning is just a tool to solve a business problem.

ROC AUC

ROC AUC stands for Receiver Operating Characteristic - Area Under the Curve.

ROC Curve is a graph that shows the performance of a classification model at all classification thresholds.

It plots:

- True Positive Rate (TPR) on the Y-axis

  TPR = TP / (TP + FN)

- False Positive Rate (FPR) on the X-axis

  FPR = FP / (FP + TN)

AUC Score Interpretation

0.9–1.0	Excellent model
0.8–0.9	Good model
0.7–0.8	Fair model
0.6–0.7	Poor model
0.5–0.6	Fail / Random guessing

You should use it when you ultimately care about ranking predictions and not necessarily about outputting well-calibrated probabilities.

You should not use it when your data is heavily imbalanced. The intuition is the following: the false positive rate for highly imbalanced datasets is pulled down due to a large number of true negatives.

You should use it when you care equally about positive and negative classes. It naturally extends the imbalanced data discussion from the last section. If we care about true negatives as much as we care about true positives, then it totally makes sense to use ROC AUC

Logistic Regression: Important Interview/Exam Questions and Answers

KNN

K-nearest neighbours (KNN) is a type of supervised learning algorithm used for both regression and classification.

KNN is called a non-parametric algorithm because it makes no assumptions about the underlying data distribution. Instead of learning fixed parameters, it uses the entire training dataset to make predictions based on similarity.

It is also called a **lazy learner algorithm** because it does not learn from the training set immediately instead it stores the dataset and at the time of classification, it performs an action on the dataset.

K-NN algorithm assumes the similarity between the new case/data and available cases and puts the new case into the category that is most similar to the available categories.

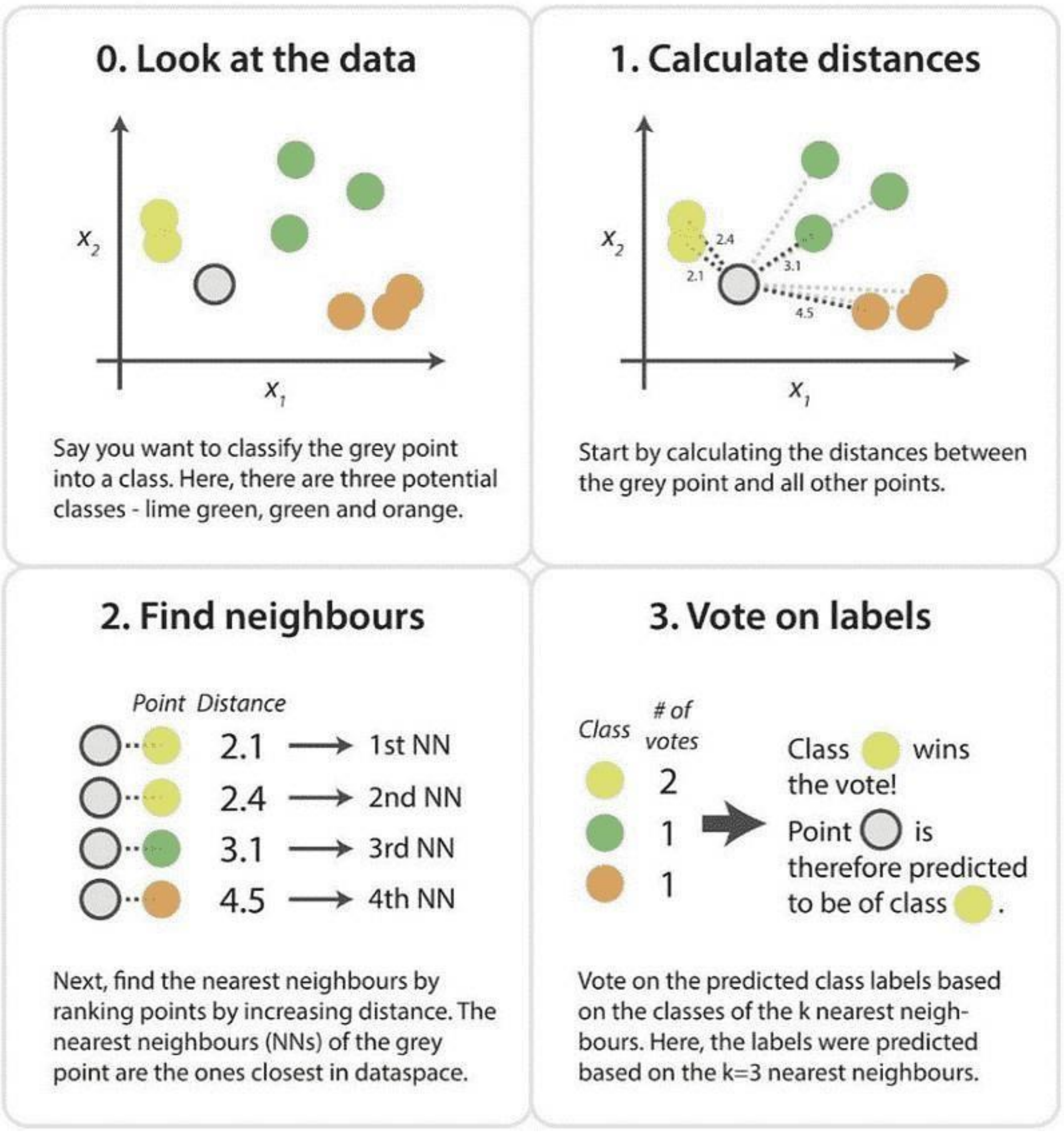
How does K-NN work in Classification Problem

The K-NN working can be explained on the basis of the below algorithm:

- Step-1: Select the value of K.
- Step-2: Calculate the distance between new data point and all data points in training data.
- Step-3: Take the K nearest neighbours as per the calculated distance.

  Or

  Take K data points (neighbours) whose calculated distance from new data point is minimum.
- Step-4: Among these k neighbours, count the number of the data points belong to each category.
- Step-5: Assign the new data points to that category for which the number of the neighbour is maximum.
- Step-6: Our model is ready.


0. Look at the data
x_2
x_1
Say you want to classify the grey point into a class. Here, there are three potential classes - lime green, green and orange.
1. Calculate distances
2.4
2.1
3.1
4.5
Start by calculating the distances between the grey point and all other points.
2. Find neighbours
Point Distance
2.1 → 1st NN
2.4 → 2nd NN
3.1 → 3rd NN
4.5 → 4th NN
Next, find the nearest neighbours by ranking points by increasing distance. The nearest neighbours (NNs) of the grey point are the ones closest in dataspace.
3. Vote on labels
Class
# of votes
2
1
1
Class wins the vote!
Point is therefore predicted to be of class .
Vote on the predicted class labels based on the classes of the k nearest neighbours. Here, the labels were predicted based on the k=3 nearest neighbours.

How does K-NN work in Regression Problem

The K-NN working can be explained on the basis of the below algorithm:

- Step-1: Select the value of K.
- Step-2: Calculate the distance between new data point and all data points in training data.
- Step-3: Take the K nearest neighbours as per the calculated distance.

  Or

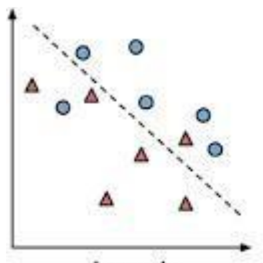
  Take K data points (neighbours) whose calculated distance from new data point is minimum.

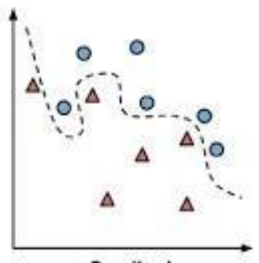
- Step-4: Then, take average of those K selected minimum distances.
- Step-5: And that will be the predicted value.
- Step-6: Our model is ready.

How to select the value of K in the K-NN Algorithm?

Below are some points to remember while selecting the value of K in the K-NN algorithm:

- There is no particular way to determine the best value for "K", so we need to try some values to find the best out of them. The most preferred value for K is 5.
- A very low value for K such as K=1 or K=2, can be noisy and lead to the effects of outliers in the model.
- Large values for K are good, but it may find some difficulties.
- The impact of selecting a smaller or larger K value on the model
- Larger K value: The case of underfitting occurs when the value of k is increased. In this case, the model would be unable to correctly learn on the training data.
- Smaller k value: The condition of overfitting occurs when the value of k is smaller. The model will capture all of the training data, including noise. The model will perform poorly for the test data in this scenario.


Larger k


Smaller k

Calculating distance:

There are various methods for calculating the distance are — Euclidean (for continuous), Manhattan (for continuous) and Hamming distance (for categorical).

Euclidean Distance: Euclidean distance is calculated as the square root of the sum of the squared differences between a new point (x) and an existing point (y).

Manhattan Distance: This is the distance between real vectors using the sum of their absolute difference.

Distance functions

Euclidean
$$\sqrt{\sum_{i=1}^{k}(x_i - y_i)^2}$$

Manhattan
$$\sum_{i=1}^{k}|x_i - y_i|$$

Hamming Distance: It is used for categorical variables. If the value (x) and the value (y) are the same, the distance D will be equal to 0 , Otherwise D=1.

$$D_H = \sum_{i=1}^{k}|x_i - y_i|$$

$$x = y \Rightarrow D = 0$$
$$x \neq y \Rightarrow D = 1$$

Advantages of KNN Algorithm:

- It is simple to implement.
- It is robust to the noisy training data
- It can be more effective if the training data is large.

Disadvantages of KNN Algorithm:

- Always needs to determine the value of K which may be complex sometimes.
- The computation cost is high because of calculating the distance between the data points for all the training samples.

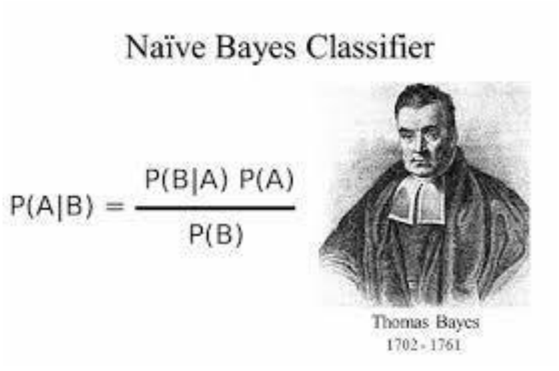
KNN: Important Interview/Exam Questions and Answers

Naïve Bayes Algorithm

The naïve Bayes algorithm is a family of probabilistic classification algorithms used for tasks like text classification, such as spam filtering and sentiment analysis.

It assumes that features are independent of each other, meaning the presence or absence of one feature doesn't impact the probability of another feature.

This algorithm is based on bayes theorem in Probability.



Intuition of Naïve bayes algorithm:

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

(see its solution in - https://youtu.be/Vlj6xS937E4?si=XnaPFtRaNuLKfW2C)

What happens inside naïve bayes during training phase?

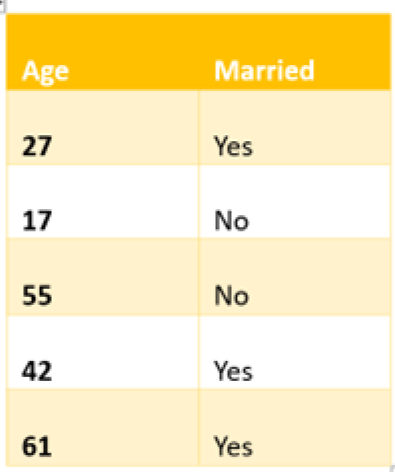
During the training phase of the Naïve Bayes algorithm, probabilities for all possible combinations of feature values and classes are calculated and stored in a hashing format.

In the testing phase, the algorithm retrieves the corresponding probabilities based on the observed feature values, multiplies them together, and provides the final output, indicating the predicted class.

By precomputing and storing the probabilities during training, the testing phase becomes more efficient.

How naïve bayes handles numerical data ?

Suppose I have a dataset in which age and whether the person is married or not given.



Age	Married
27	Yes
17	No
55	No
42	Yes
61	Yes

I have a age 46 and I want to predict whether the person is married or not using naïve bayes. So I need to calculate P(Y | 46) and P(N | 46). But in given table may be the given age not present then that time these probabilities becomes zero. To resolve this issue we assume that our Age columns follows a gaussian distribution and based on this distribution we calculate μ and σ and x= 46 and we put all the values in gaussian distribution function:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

$f(x)$ = probability density function

σ = standard deviation

μ = mean

Now we calculate f(x) which gives the probability of corresponding age. So in this way using Probability density function we can use naïve bayes algorithm for numerical data.

What if data distribution is not Gaussian Distribution?

1. **Data Transformation:** Depending on the nature of your data, you could apply a transformation to make it more normally distributed. Common transformations include the logarithm, square root, and reciprocal transformations.

2. **Alternative Distributions:** If you know or suspect that your data follow a specific non-normal distribution (e.g., exponential, Poisson, etc.), you can modify the Naïve Bayes algorithm to assume that specific distribution when calculating the likelihoods.

3. **Discretization:** You can turn your continuous data into categorical data by binning the values. There are various ways to decide on the bins, including equal width bins, equal frequency bins, or using a more sophisticated method like k-means clustering. Once your
data is binned, you can use the standard Multinomial or Bernoulli Naïve Bayes methods.

4. **Kernel Density Estimation:** A non-parametric way to estimate the probability density function of a random variable. Kernel density estimation can be used when the distribution is unknown.

5. **Use other models:** If none of the above options work well, it may be best to consider a different classification algorithm that doesn't make strong assumptions about the distributions of the features, such as Decision Trees, Random Forests, or Support Vector Machines

Zero Probability Problem in naïve bayes: (Laplace Additive Smoothing)

Laplace additive smoothing is a technique commonly used in Naïve Bayes algorithms to handle the issue of zero probabilities.

It is applied to avoid the problem of encountering unseen combinations of features and classes during classification.

In Naïve Bayes, when calculating probabilities, there is a possibility of encountering feature values in the test data that were not observed in the training data.

As a result, the conditional probability for such combinations would be zero, which can lead to inaccurate predictions.

Laplace additive smoothing addresses this problem by adding a small constant (usually 1) to both the numerator and the denominator when estimating probabilities. This way, even if a specific feature value has not been observed for a particular class in the training data, it will still have a non-zero probability.

P(feature value | class) = (count of feature value given the class + α) / (count of class + n * α)

Here α is generally taken 1 and value of n depends on which type of naïve bayes you used.

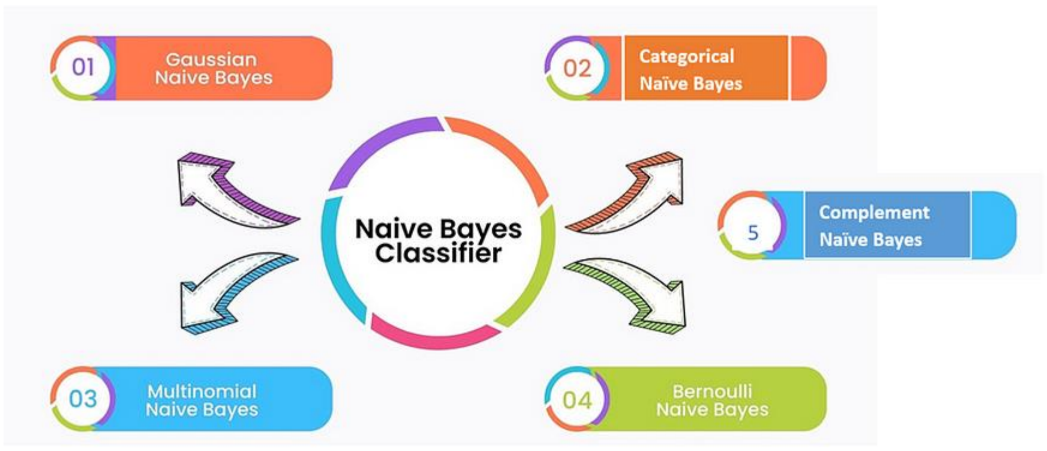
(study detailed info about this later)

Bias-Variance Trade-Off in naïve bayes classifier using Laplacian additive smoothing coefficient ‘α’:

An extremely large alpha value in Laplace smoothing results in over smoothing, where all feature occurrences are assigned similar probabilities. This leads to an underfitting scenario, where the model becomes less sensitive to the observed data and performs poorly in generalizing to new, unseen instances.

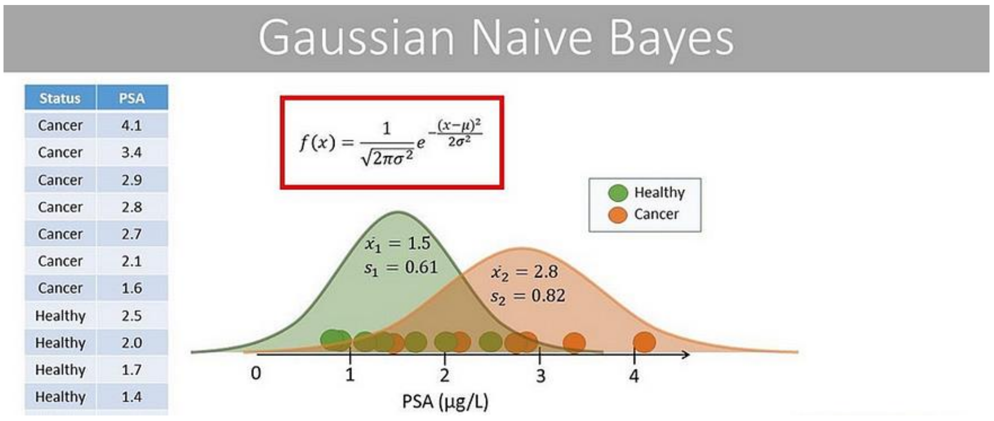
(related reading - https://medium.com/@sachinsoni600517/na%C3%AFve-bayes-machine-learning-algorithm-from-basic-to-advanced-91a8fb749ee3)

Types of naïve Bayes:



Gaussian Naïve Bayes:

When all the input features are numerical, then that time use Gaussian Naïve Bayes. Laplace Additive smoothing is not applicable on Gaussian Naïve Bayes because here probability is never zero for any input feature.



$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Status	PSA
Cancer	4.1
Cancer	3.4
Cancer	2.9
Cancer	2.8
Cancer	2.7
Cancer	2.1
Cancer	1.6
Healthy	2.5
Healthy	2.0
Healthy	1.7
Healthy	1.4

Categorical Naïve Bayes:

When all the input features are categorical, then that time use Categorical Naïve Bayes. When applying Laplace additive smoothing in this case **’n’** indicates number of unique feature values or categories for a specific input column.

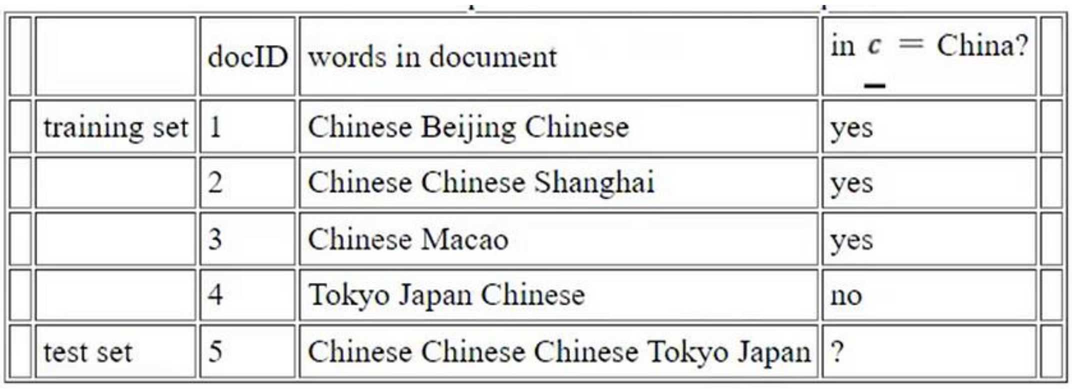
Multinomial Naïve Bayes:

Multinomial Naïve Bayes is a variant of the Naïve Bayes algorithm that is specifically designed for text classification problems where the features represent the frequency or occurrence of words in a document.

It is commonly used in natural language processing tasks, such as sentiment analysis, spam filtering, topic classification, and document categorization.

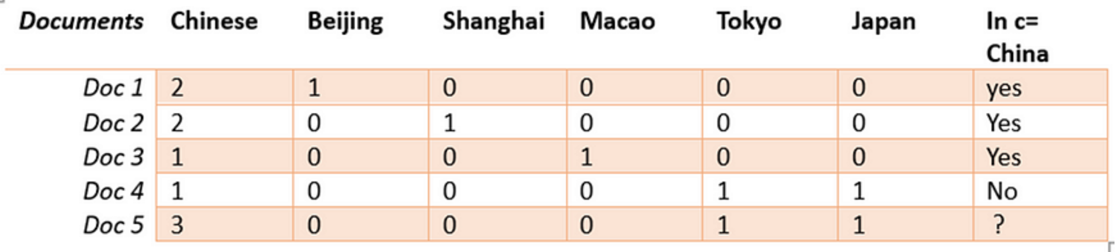
In Multinomial Naïve Bayes, the features are typically represented as term frequencies, such as the number of times a word appears in a document, or as TF-IDF (Term Frequency-Inverse Document Frequency) values, which take into account both the frequency of a word in a document and its rarity across the entire dataset.

Let’s take an example to understand Multinomial Naïve bayes.



	docID	words in document	in c = China?
training set	1	Chinese Beijing Chinese	yes
	2	Chinese Chinese Shanghai	yes
	3	Chinese Macao	yes
	4	Tokyo Japan Chinese	no
test set	5	Chinese Chinese Chinese Tokyo Japan	?

The above table represents docID with different words are coming in that document. Training set contains four different documents and on the basis of words, we need to predict whether docID 5 is from China or not. Now applying count based bag-of-words I am converting each document into word frequency table, where the columns represent different words and the rows represent each document.



Documents	Chinese	Beijing	Shanghai	Macao	Tokyo	Japan	In c= China
Doc 1	2	1	0	0	0	0	yes
Doc 2	2	0	1	0	0	0	Yes
Doc 3	1	0	0	1	0	0	Yes
Doc 4	1	0	0	0	1	1	No
Doc 5	3	0	0	0	1	1	?

Now for Document 5, we need to predict whether it is c= China or not.

So, for predicting above we need to calculate,
P(Yes | Chinese=3, Beijing= 0, Shanghai=0, Macao= 0, Tokyo=1, Japan=1) and P(No | Chinese=3, Beijing= 0,Shanghai=0,Macao= 0, Tokyo=1, Japan=1)

Let say, B=(Chinese=3, Beijing= 0, Shanghai=0, Macao= 0, Tokyo=1,Japan=1)

P(Yes | B) = P(Yes) * P(B | Yes)
P(Yes)= 3/4,

P(Chinese | Yes) = 5 + 1/8+6 = 6/14, Here adding 1 in numerator and 6 in denominator indicates Laplace additive smoothing and alpha=1 and n=6 because n indicates total number of different words.

P(Beijing | Yes)= 1+1/8+6 = 2/14
P(Shanghai | Yes) = 1+1/8+6 = 2/14
P(Macao | Yes) = 1+1/8+6 = 2/14

P(Tokyo | Yes) = 0+1/8+6= 1/14
P(Japan | Yes) = 0+1/8+6= 1/14

Similarly all these Probabilities is calculated for No option also,

P(No) = 1/4
P(Beijing | No)= 0+1/3+6 = 1/9
P(Shanghai | No) = 1+1/3+6 = 1/9
P(Macao | No) = 0+1/3+6 = 1/9
P(Tokyo | No) = 1+1/3+6= 2/9
P(Japan | No) = 1+1/3+6= 2/9
P(Chinese | No) = 1+1/3+6 = 2/9

P(Yes | B) = 3/4 * (6/14)^3 *1/14 * 1/14 = 0.0003
P(No | B) = 1/4 * (2/9)^3 * 2/9 * 2/9 = 0.0001

So for document 5 is from China because P(Yes | B) > P(No | B).

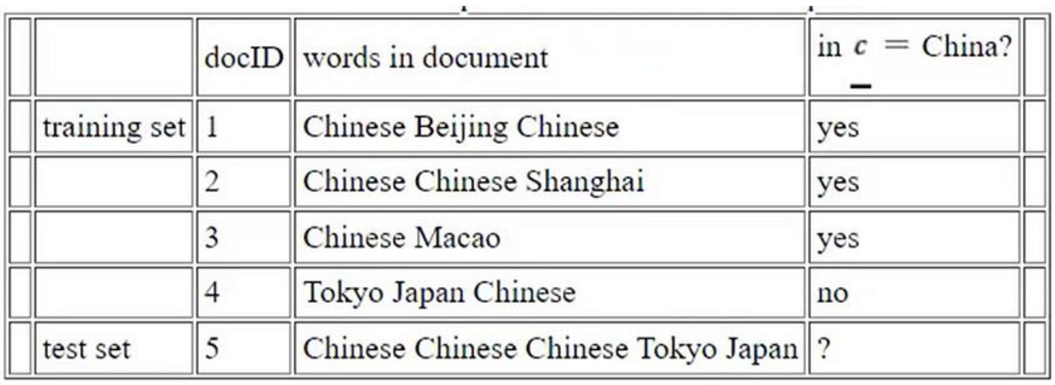
Bernoulli Naïve Bayes:

Bernoulli Naïve Bayes is commonly used in scenarios where the features are binary or Boolean variables.

In Bernoulli Naïve Bayes, the input data is represented as a binary feature vector, where each feature represents the presence or absence of a particular attribute.

For example, in text classification, each feature could correspond to the presence or absence of a specific word in a document.

Let’s take an example to understand Bernoulli Naïve bayes.



	docID	words in document	in c = China?
training set	1	Chinese Beijing Chinese	yes
	2	Chinese Chinese Shanghai	yes
	3	Chinese Macao	yes
	4	Tokyo Japan Chinese	no
test set	5	Chinese Chinese Chinese Tokyo Japan	?

The above table represents docID with different words are coming in that document. Training set contains four different documents and on the basis of words, we need to predict whether docID 5 is from China or not. Now applying binary bag-of-words I am converting each document into binary word frequency table, where the columns represent different words and the rows represent each document.

Documents	Chinese	Beijing	Shanghai	Macao	Tokyo	Japan	In c= China
Doc 1	1	1	0	0	0	0	yes
Doc 2	1	0	1	0	0	0	Yes
Doc 3	1	0	0	1	0	0	Yes
Doc 4	1	0	0	0	1	1	No
Doc 5	1	0	0	0	1	1	?

Now for Document 5, we need to predict whether it is c= China or not.

So, for predicting above we need to calculate,
P(Yes | Chinese=1, Beijing= 0, Shanghai=0, Macao= 0, Tokyo=1, Japan=1) and P(No | Chinese=1, Beijing= 0,Shanghai=0,Macao= 0, Tokyo=1, Japan=1)

Let say, B=(Chinese=1, Beijing= 0, Shanghai=0, Macao= 0, Tokyo=1,Japan=1)

P(Yes | B) = P(Yes) * P(B | Yes)
P(Yes)= 3/4,

Probability of Bernoulli random variable is given by,
P(X=K) = PK + (1-P)(1-K)

P(Chinese=1 | Yes) = 3 +1/3+2 =4/5 , because here K=1
and P(X=1) = P*1 +0 = P ,and P indicates probability of 1 and also I apply Laplace Additive Smoothing on probabilities.

P(Beijing =0 | Yes)= 2+1/3+2 = 3/5
P(Shanghai=0 | Yes) = 2+1/3+2 = 3/5
P(Macao=0 | Yes) = 2+1/3+2 = 3/5
P(Tokyo=1 | Yes) = 0+1/3+2 = 1/5
P(Japan=1 | Yes) = 0+1/3+2 = 1/5

Similarly all these Probabilities is calculated for No option also,

P(No) = 1/4
P(Beijing =0| No)= 1+1/1+2 = 2/3
P(Shanghai=0 | No) = 1+1/1+2=2/3
P(Macao=0 | No) = 1+1/1+2=2/3
P(Tokyo=1 | No) = 1+1/1+2=2/3
P(Japan=1 | No) = 1+1/1+2=2/3
P(Chinese=1 | No) = 1+1/1+2=2/3

P(Yes | B) =3/4 *4/5 * 3/5 * 3/5 * 3/5 *1/5 *1/5 = 0.005
P(No | B) = 1/4 * 2/3 * 2/3 * 2/3 * 2/3 * 2/3 * 2/3= 0.022

So for document 5 is not from China because P(No| B) > P(Yes | B).