

Data Science

Interview Preparation Guide



SADUM YESHWANTH

Table of Contents

ML Concepts	3
Linear Regression.....	3
Logistic Regression	6
Decision Trees	8
GDA & Naive Bayes	8
Ensembles: Gradient boosting, random forests, bagging, voting, stacking XGBoost	10
General ML Questions	13
Clustering.....	17
K-means Clustering.....	17
Hierarchical Clustering	18
DBSCAN	19
Gaussian Mixture Models (GMM)	19
Other Methods.....	19
Evaluation metrics	20
Neural Networks.....	21
Sequence Models	24
RNN	24
Transformers	26
Tf- IDF.....	29
Word Piece tokenization.....	31
Word2Vec.....	31
Training objective of BERT	32
Sentence transformers	33
Evaluating LLM response.....	35
Statistics.....	35
Linear Algebra.....	36
Coding	38
Kubernetes.....	38

ML Concepts

Linear Regression

References:

1. [Linear regression](#)
2. <https://mlu-explain.github.io/linear-regression/>

Linear regression equation

In algebraic terms, the model would be defined as $y = mx + b$, where

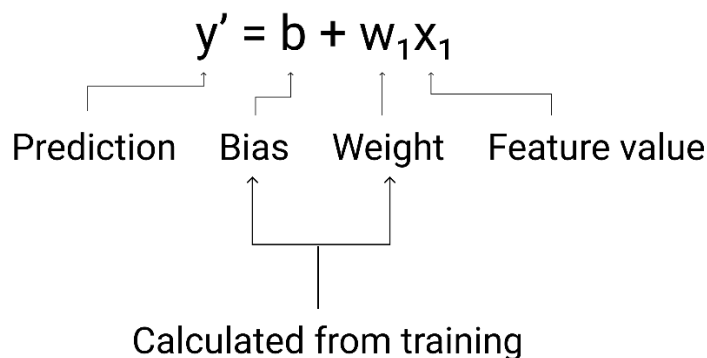
- y is miles per gallon—the value we want to predict.
- m is the slope of the line.
- x is pounds—our input value.
- b is the y-intercept.

In ML, we write the equation for a linear regression model as follows:

$$y' = b + w_1 x_1$$

where:

- y' is the predicted label—the output.
- b is the **bias** of the model. Bias is the same concept as the y-intercept in the algebraic equation for a line. In ML, bias is sometimes referred to as w_0 . Bias is a **parameter** of the model and is calculated during training.
- w_1 is the **weight** of the feature. Weight is the same concept as the slope m in the algebraic equation for a line. Weight is a **parameter** of the model and is calculated during training.
- x_1 is a **feature**—the input.



Models with multiple features

Although the example in this section uses only one feature—the heaviness of the car—a more sophisticated model might rely on multiple features, each having a separate weight (w_1, w_2 , etc.). For example, a model that relies on five features would be written as follows:

$$y' = b + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5$$

For example, a model that predicts gas mileage could additionally use features such as the following:

- Engine displacement
- Acceleration
- Number of cylinders
- Horsepower

This model would be written as follows:

The diagram shows the equation $y' = b + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5$. Below the equation, five feature names are listed with arrows pointing to their corresponding variables in the equation:

- Pounds Displacement** points to x_1
- Acceleration** points to x_2
- Number of cylinders** points to x_3
- Number of cylinders** points to x_4
- Horsepower** points to x_5

Types of loss

In linear regression, there are four main types of loss, which are outlined in the following table.

Loss type	Definition	Equation
L_1 loss	The sum of the absolute values of the difference between the predicted values and the actual values.	$\sum actual\ value - predicted\ value $
Mean absolute error (MAE)	The average of L_1 losses across a set of *N* examples.	$\frac{1}{N} \sum actual\ value - predicted\ value $
L_2 loss	The sum of the squared difference between the predicted values and the actual values.	$\sum (actual\ value - predicted\ value)^2$
Mean squared error (MSE)	The average of L_2 losses across a set of *N* examples.	$\frac{1}{N} \sum (actual\ value - predicted\ value)^2$

The functional difference between L_1 loss and L_2 loss (or between MAE and MSE) is squaring. When the difference between the prediction and label is large, squaring makes the loss even larger. When the difference is small (less than 1), squaring makes the loss even smaller.

When processing multiple examples at once, we recommend averaging the losses across all the examples, whether using MAE or MSE.

Evaluation Metrics

- MSE
- RMSE

- MAE
- R-Squared

R-Squared

Regression models may also be evaluated with the so-called *goodness of fit* measures, which summarize how well a model fits a set of data. The most popular goodness of fit measure for linear regression is r-squared, a metric that represents the percentage of the variance in y explained by our features x . [1] More specifically, r-squared measures the percentage of variance explained normalized against the baseline variance of our model (which is just the variance of the mean):

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

The highest possible value for r-squared is 1, representing a model that captures 100% of the variance. A negative r-squared means that our model is doing worse (capturing less variance) than a flat line through mean of our data would.

Model Assumptions**1. Linearity**

- The relationship between the dependent variable and the independent variables is linear.

2. Independence of Errors

- The residuals (errors) are independent of each other.
- This is especially important in time series data to avoid autocorrelation.

3. Homoscedasticity

- The residuals have constant variance at every level of the independent variables.
- If this assumption is violated, it results in heteroscedasticity, which can affect the reliability of confidence intervals.

4. Normality of Errors

- The residuals should be approximately normally distributed.
- This is crucial for conducting hypothesis tests and constructing confidence intervals.

5. No Multicollinearity

- The independent variables should not be highly correlated with each other.
- High multicollinearity makes it difficult to isolate the effect of each predictor.

6. No or Little Measurement Error

- The independent variables are assumed to be measured accurately.
- Measurement error in predictors can lead to biased coefficient estimates.

Loss formulation

1.5.8. Mathematical formulation

We describe here the mathematical details of the SGD procedure. A good overview with convergence rates can be found in [\[12\]](#).

Given a set of training examples $(x_1, y_1), \dots, (x_n, y_n)$ where $x_i \in \mathbf{R}^m$ and $y_i \in \mathbf{R}$ ($y_i \in \{-1, 1\}$ for classification), our goal is to learn a linear scoring function $f(x) = w^T x + b$ with model parameters $w \in \mathbf{R}^m$ and intercept $b \in \mathbf{R}$. In order to make predictions for binary classification, we simply look at the sign of $f(x)$. To find the model parameters, we minimize the regularized training error given by

$$E(w, b) = \frac{1}{n} \sum_{i=1}^n L(y_i, f(x_i)) + \alpha R(w)$$

where L is a loss function that measures model (mis)fit and R is a regularization term (aka penalty) that penalizes model complexity; $\alpha > 0$ is a non-negative hyperparameter that controls the regularization strength.

Regularization

Popular choices for the regularization term R (the **penalty** parameter) include:

- L2 norm: $R(w) := \frac{1}{2} \sum_{j=1}^m w_j^2 = \|w\|_2^2$,
- L1 norm: $R(w) := \sum_{j=1}^m |w_j|$, which leads to sparse solutions.
- Elastic Net: $R(w) := \frac{\rho}{2} \sum_{j=1}^m w_j^2 + (1 - \rho) \sum_{j=1}^m |w_j|$, a convex combination of L2 and L1, where ρ is given by **1 - l1_ratio**.

Gradient Descent

[Linear regression: Gradient descent](#)

Logistic Regression

References:

1. <https://developers.google.com/machine-learning/crash-course/logistic-regression>

Transforming linear output using the sigmoid function

The following equation represents the linear component of a logistic regression model:

$$z = b + w_1 x_1 + w_2 x_2 + \dots + w_N x_N$$

To obtain the logistic regression prediction, the z value is then passed to the sigmoid function, yielding a value (a probability) between 0 and 1:

$$y' = \frac{1}{1 + e^{-z}}$$

Instead, the loss function for logistic regression is **Log Loss**. The Log Loss equation returns the logarithm of the magnitude of the change, rather than just the distance from data to prediction. Log Loss is calculated as follows:

$$\text{Log Loss} = \sum_{(x,y) \in D} -y \log(y') - (1 - y) \log(1 - y')$$

Evaluation

References:

1. <https://developers.google.com/machine-learning/crash-course/classification/thresholding>
2. <https://developers.google.com/machine-learning/crash-course/classification/accuracy-precision-recall>¹
3. <https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc>

Metric	Guidance
Accuracy	Use as a rough indicator of model training progress/convergence for balanced datasets. For model performance, use only in combination with other metrics. Avoid for imbalanced datasets. Consider using another metric.
Recall (True positive rate)	Use when false negatives are more expensive than false positives.
False positive rate	Use when false positives are more expensive than false negatives.
Precision	Use when it's very important for positive predictions to be accurate.

¹ Note: For imbalance datasets, recall is preferred as the Positives are very less in number.
Recall = TP/(TP+FN)

This will penalise false negatives.¹

Decision Trees

References

1. <https://scikit-learn.org/stable/modules/tree.html>
2. [Statquest \(Decision and Classification Trees, Clearly Explained!!!\)](#)
3. [StatQuest: Decision Trees, Part 2 – Feature Selection and Missing Data](#)
4. [Statquest \(Regression Trees, Clearly Explained!!!\)](#)

DT Classifier

1. Explain how decision tree classifier works.
2. How to avoid overfitting in decision trees?
3. Is DT scalable for large amounts of data?
4. What are the pros and cons of using DT?

DT Regressor

1. How can we use DT for regression?
2. Is DT parametric or non-parametric? What do you mean by parametric algorithm?
3. How to perform feature selection using decision tree?
4. What do the nodes represent in a decision tree?

GDA & Naive Bayes

References:

1. [Lecture 5 – GDA & Naive Bayes | Stanford CS229: Machine Learning Andrew Ng \(Autumn 2018\)](#)
2. [Deriving Naive Bayes Classifier \(Ardian Umam blog\)](#)

Questions

1. What is Gaussian Discriminant Analysis (GDA)?

GDA is a generative classification algorithm that assumes the class-conditional distribution of features follows a multivariate Gaussian distribution. It uses Bayes' theorem to compute the posterior probability of classes and classify accordingly.

2. What assumptions does GDA make about the data?

- The features for each class are normally distributed.
- For LDA: All classes share the same covariance matrix.
- For QDA: Each class can have a different covariance matrix.

3. What is the Naive Bayes assumption?

It assumes that all features are conditionally independent given the class label.

4. How does Naive Bayes use Bayes' theorem for classification?

It computes:

$$P(y \mid x_1, x_2, \dots, x_n) \propto P(y) \prod_{i=1}^n P(x_i \mid y)$$

Then it selects the class y with the highest posterior probability.

5. Why is it called "Naive" Bayes?

Because it naively assumes feature independence, which rarely holds true in real-world data.

6. What are the differences between GDA and Naive Bayes?

- GDA models the joint feature distribution as multivariate Gaussian; Naive Bayes assumes independent features.
- GDA can model feature correlation; Naive Bayes cannot.
- Naive Bayes is more robust with smaller datasets.

7. What happens to Naive Bayes performance if features are highly correlated?

It degrades because the independence assumption is violated, leading to inaccurate probability estimates.

8. How would you estimate the parameters in GDA and Naive Bayes from training data?

- GDA: Use MLE to estimate class priors, means μ_k , and covariances Σ_k .
- Naive Bayes: Estimate class priors and each feature's distribution parameters independently.

4. Can Naive Bayes be used for continuous features? How?

Yes, by assuming a distribution like Gaussian for each feature:

$$P(x_i | y) = \mathcal{N}(x_i | \mu_{iy}, \sigma_{iy}^2)$$

2. Why does GDA sometimes outperform logistic regression, and vice versa?

GDA is more efficient when its Gaussian assumptions hold; logistic regression is more robust when assumptions are violated since it's discriminative.

3. Explain the MLE steps in training GDA.

- Estimate class priors: $\phi = \frac{1}{m} \sum y^{(i)}$
- Estimate means: $\mu_k = \text{mean of } x^{(i)} \text{ where } y^{(i)} = k$
- Estimate covariances: $\Sigma_k = \text{covariance of samples from class } k$

4. How does GDA handle multiclass classification?

Train a Gaussian for each class k : $P(x | y = k) = \mathcal{N}(\mu_k, \Sigma_k)$, then compute posterior probabilities and pick the class with the highest value.

5. If a Naive Bayes model performs poorly, how would you diagnose and fix the issue?

- Check feature correlations.
- Use feature selection or transformation (e.g., PCA).
- Try a non-naive generative model (e.g., GDA).
- Use discretization or binning for numeric features.

Ensembles: Gradient boosting, random forests, bagging, voting, stacking XGBoost

Ensemble methods combine the predictions of several base estimators built with a given learning algorithm in order to improve generalizability / robustness over a single estimator.

Two very famous examples of ensemble methods are [gradient-boosted trees](#) and [random forests](#).

More generally, ensemble models can be applied to any base learner beyond trees, in averaging methods such as [Bagging methods](#), [model stacking](#), or [Voting](#), or in boosting, as [AdaBoost](#).

Ensemble Learning Techniques

Ensemble learning is a method where multiple models (weak learners) are combined to improve overall performance. Below are key ensemble techniques:

1. Gradient Boosting

Reference:

1. [Gradient Boosting In Depth Intuition- Part 1 Machine Learning](#)

Gradient Boosting is an iterative boosting technique that builds an ensemble of weak learners (usually decision trees), where each new tree corrects the errors of the previous one. It minimizes loss by sequentially fitting new models to the residual errors of the prior models.

- Process:
 - Train a weak model.
 - Compute residual errors.
 - Train the next model to predict these residuals.
 - Update predictions by adding the weighted residuals.
 - Repeat until convergence or a set number of iterations.
- Popular Implementations:
 - XGBoost (Extreme Gradient Boosting)
 - LightGBM (Light Gradient Boosting Machine)
 - CatBoost (Categorical Boosting)
- Pros:
 - Highly accurate.
 - Handles non-linearity well.
 - Works well with structured data.
- Cons:
 - Computationally expensive.
 - Prone to overfitting if not tuned properly.

2. Random Forests

Reference:

1. [StatQuest: Random Forests Part 1 - Building, Using and Evaluating](#)
2. [StatQuest: Random Forests Part 2: Missing data and clustering](#)

Random Forests is an ensemble of decision trees where each tree is trained on a random subset of the data and features. The final prediction is made by averaging (for regression) or majority voting (for classification).

- Process:
 - Randomly sample training data (bootstrap sampling).
 - Train a decision tree on each sample.
 - For each split in the tree, consider a random subset of features.
 - Aggregate predictions from all trees.
- Pros:
 - Reduces overfitting compared to individual trees.
 - Works well with high-dimensional data.
 - Handles missing data well.
- Cons:
 - Slower than a single decision tree.
 - Less interpretable than individual trees.

3. Bagging (Bootstrap Aggregating)

Bagging is a general ensemble technique that reduces variance by training multiple models on different bootstrap samples (randomly drawn with replacement) and aggregating their outputs.

- Process:
 - Generate multiple datasets by bootstrapping.
 - Train independent models (e.g., decision trees) on each dataset.
 - Average the predictions for regression or take majority voting for classification.
- Example:
 - Random Forest is a special case of bagging with decision trees.
- Pros:
 - Reduces overfitting.
 - Works well with high variance models (e.g., decision trees).
- Cons:
 - Does not significantly reduce bias.

4. Voting

Voting is a simple ensemble method where multiple models are trained independently, and their outputs are combined to make the final prediction.

- Types:
 - Hard Voting: Takes the majority class (classification).
 - Soft Voting: Averages the probabilities and selects the class with the highest probability.
- Pros:
 - Simple to implement.
 - Works well when combining diverse models.
- Cons:
 - May not always improve performance if models are highly correlated.

5. Stacking (Stacked Generalization)

Stacking is a meta-ensemble method where multiple base models are trained, and a meta-model learns how to best combine their predictions.

- Process:
 - Train multiple base models.
 - Collect predictions from base models.
 - Train a meta-model using these predictions as input.
 - The meta-model learns the optimal combination of base models' outputs.
- Pros:
 - Can capture complex relationships between models.
 - Often leads to better performance than individual models.
- Cons:
 - Computationally expensive.
 - Requires careful tuning to avoid overfitting.

6. XGBoost (Extreme Gradient Boosting)

XGBoost is an optimized version of gradient boosting that uses regularization and parallelization to enhance efficiency and prevent overfitting.

- **Key Features:**
 - Regularization: L1 and L2 regularization to prevent overfitting.
 - Tree Pruning: Uses a depth-first approach to remove unnecessary branches.
 - Parallel Processing: Speeds up training.
 - Handling Missing Values: Automatically handles missing data.
- **Pros:**
 - Fast and efficient.
 - Handles large datasets well.
 - Provides state-of-the-art accuracy.
- **Cons:**
 - Can be difficult to tune.
 - Computationally demanding for very large datasets.

General ML Questions

1. Tell about the different preprocessing steps in ML.

Standardization and Normalization

1. Standardization (Z-score Normalization)

Standardization transforms data to have a **mean of 0** and a **standard deviation of 1**. This ensures that all features contribute equally to the model.

Formula:

$$X' = \frac{X - \mu}{\sigma}$$

Where:

- X is the original value,
- μ is the mean of the feature,
- σ is the standard deviation.

2. Normalization (Min-Max Scaling)

Normalization scales data within a **fixed range, usually [0,1] or [-1,1]**. It's sensitive to outliers since it depends on the min/max values.

Formula:

$$X' = \frac{X - X_{\min}}{X_{\max} - X_{\min}}$$

Where:

- X is the original value,
- X_{\min} is the minimum value in the feature,
- X_{\max} is the maximum value in the feature.

Which One Should You Use?

- **Use Standardization when:**
 - Your data follows a **normal (Gaussian) distribution**.
 - You're using **distance-based algorithms** (SVM, K-Means, PCA, KNN).
 - Your data has **outliers** (because standardization is more robust to them).
- **Use Normalization when:**
 - Your data has **different scales and a fixed range is required**.
 - You're using **neural networks**, where input ranges impact gradient descent.
 - Your dataset has **no strong normal distribution assumption**.

Handling missing values

Method	Use Case
Drop rows	When missing data is minimal
Drop columns	When a feature has >50% missing values
Mean/Median imputation	For numerical data (mean for normal, median for skewed)
Mode imputation	For categorical features
Forward/Backward fill	When data has a time-series pattern
KNN Imputation	When data has relationships with other features
Iterative Imputation	When missing data is complex and structured
Indicator Variable	When missing values might hold information

How Does Iterative Imputation Work?

1. **Initialize Missing Values** – First, missing values are replaced with an initial guess (e.g., mean or median).
2. **Iterative Predictions:**
 - Pick a feature with missing values.
 - Use all other features to train a regression model.
 - Predict the missing values for that feature.
 - Move to the next feature and repeat the process.
3. **Repeat Multiple Rounds** – This process is repeated several times to refine the estimates.
4. **Convergence Check** – Stops when imputed values no longer change significantly.

2. How Can You Choose a Classifier Based on a Training Set Data Size?

1. Small Training Data (Few Samples, <1,000)

- Prefer simple models to avoid overfitting.

- Use Naïve Bayes, Logistic Regression, or k-Nearest Neighbors (k-NN).
- Consider Support Vector Machines (SVM) with a linear kernel.
- Avoid deep learning models as they require large datasets.

2. Medium Training Data (1,000 – 100,000 samples)

- Use Decision Trees, Random Forests, Gradient Boosting (XGBoost, LightGBM, CatBoost).
- SVM with non-linear kernels can work, but might be computationally expensive.
- Neural networks can be used but may require tuning.

3. Large Training Data (>100,000 samples)

- Consider Deep Learning (Neural Networks, CNNs, RNNs, Transformers).
- Gradient Boosting models still work well for structured data.
- SVMs become computationally expensive for large datasets.

Other Factors to Consider:

- Feature Dimensionality: If the number of features is high, dimensionality reduction (PCA, feature selection) may be needed.
- Computational Power: Deep learning requires GPUs, while simpler models like logistic regression run efficiently on CPUs.
- Data Complexity: Non-linear models (e.g., deep learning, ensemble methods) handle complex relationships better.

3. Train, Test, and Validation Sets (Short Explanation)

1. Training Set – Used to train the model (largest portion, ~60-80% of data).
2. Validation Set – Used to fine-tune the model and prevent overfitting (~10-20%).
3. Test Set – Used for final evaluation to check model performance on unseen data (~10-20%).

NOTE: Learn about cross validation and grid search.

4. What is Semi-Supervised Learning?

Semi-supervised learning is a machine learning approach that falls between supervised learning (where all data is labelled) and unsupervised learning (where no labels are provided). In SSL, a small amount of labelled data is combined with a large amount of unlabelled data to improve learning efficiency.

5. Why Use Semi-Supervised Learning?

- Labelling data is expensive and time-consuming (e.g., medical images, speech recognition).
- Unlabelled data is abundant and easier to obtain.
- Helps improve model accuracy with minimal labelled data.

How It Works

1. A small labelled dataset is used to train an initial model.
 2. The model makes predictions on the unlabelled data.
 3. High-confidence predictions are added to the labelled dataset (self-training) or used with other SSL techniques.
 4. The model is retrained iteratively, improving performance.
6. K means vs KNN?
7. Bias vs Variance

Bias and variance are two fundamental sources of error in a machine learning model that affect its performance and generalization.

1. Bias

Bias refers to the error introduced by approximating a real-world problem with a simplified model. It measures how much the predicted values deviate from the actual values.

- High Bias (Underfitting): The model is too simple and fails to capture the underlying patterns in the data.
- Low Bias: The model closely follows the training data.

Example:

- A linear regression model trying to fit a highly complex dataset will have high bias because it oversimplifies the relationships.

2. Variance

Variance refers to the model's sensitivity to fluctuations in the training data. It measures how much predictions vary when the model is trained on different subsets of data.

- High Variance (Overfitting): The model is too complex and captures noise in the training data, making it perform poorly on new data.
- Low Variance: The model generalizes well to unseen data.

Example:

- A deep neural network trained on a small dataset may have high variance because it memorizes the training data instead of learning general patterns.

Bias-Variance Tradeoff

- High Bias, Low Variance → Simple models (e.g., linear regression) → Underfitting
- Low Bias, High Variance → Complex models (e.g., deep neural networks) → Overfitting
- Optimal Model: Balances bias and variance to minimize total error.

Total Error = Bias² + Variance + Irreducible Error

The goal is to find a model that generalizes well to new data by managing this tradeoff effectively.

8. Correlation vs Covariance

Correlation and **covariance** are both statistical concepts that describe the relationship between two variables, but they differ in several important ways.

Covariance:

Covariance measures the degree to which two random variables change together. In other words, it indicates whether an increase in one variable tends to be associated with an increase (or decrease) in another variable. It is defined mathematically as:

$$\text{Cov}(X, Y) = \frac{1}{n} \sum_{i=1}^n (X_i - \bar{X})(Y_i - \bar{Y})$$

Where:

- X_i and Y_i are individual data points of variables X and Y ,
- \bar{X} and \bar{Y} are the means of X and Y ,
- n is the number of data points.

Correlation:

Correlation is a standardized version of covariance that quantifies the strength and direction of the linear relationship between two variables. It is computed by dividing the covariance of the variables by the product of their standard deviations. The most commonly used correlation is the **Pearson correlation coefficient**, denoted as r , and it is calculated as:

$$r = \frac{\text{Cov}(X, Y)}{\sigma_X \sigma_Y}$$

Where:

- $\text{Cov}(X, Y)$ is the covariance of variables X and Y ,
- σ_X and σ_Y are the standard deviations of X and Y .

Key Points about Correlation:

- **Range:** The value of correlation always lies between -1 and $+1$.
 - A correlation of **+1** indicates a perfect positive linear relationship.

Clustering

K-means Clustering

- Randomly initialize centroids
- Assign points to clusters
- Update the centroid
- Repeat until clusters don't change

Elbow method to find k – the point at which the variation in clusters reduces drastically.

Hierarchical Clustering

- Assign points to the same cluster depending on how close they are to each other.

[K not needed]

Agglomerative vs. Hierarchical Clustering: What's the Difference?

Short Answer: **Agglomerative clustering** is a specific type of **hierarchical clustering**. Hierarchical clustering is a general category that includes both **agglomerative** and **divisive** methods.

1. Definitions

Method	Description
Hierarchical Clustering	A general clustering approach that builds a hierarchy of clusters. It includes both agglomerative and divisive methods.
Agglomerative Clustering	A bottom-up hierarchical clustering method where each data point starts as its own cluster, and clusters are merged iteratively.
Divisive Clustering	A top-down hierarchical clustering method where all data points start in one large cluster, and clusters are split recursively.

So, **agglomerative clustering is a type of hierarchical clustering, but hierarchical clustering also includes divisive methods.**

3. How Agglomerative Clustering Works

1. Start with each point as its **own cluster**.
2. Compute a **distance matrix** (e.g., Euclidean distance).
3. Merge the **two closest clusters** (based on linkage criteria: single, complete, average, or Ward's method).
4. Repeat until **one large cluster remains** or a stopping criterion is met.

Common Linkage Methods in Agglomerative Clustering		
Linkage Type	Description	
Single Linkage	Merge clusters with the closest points.	
Complete Linkage	Merge clusters based on the farthest points.	
Average Linkage	Merge clusters based on the average distance between points.	
Ward's Method	Minimizes variance within clusters (preferred for balanced clustering).	

4. Pros & Cons		
Feature	Agglomerative Clustering	Divisive Clustering
Accuracy	Often produces better results due to bottom-up merging.	Can be better for certain datasets but is computationally costly.
Computation Time	$O(n^2)$ (quadratic), can be slow for large datasets.	Even more expensive than agglomerative.
Flexibility	Can use different linkage methods.	Usually implemented using basic distance measures.
Common Usage	Most widely used hierarchical clustering method.	Less common due to computational complexity.

DBSCAN

(Density-Based Spatial Clustering of Applications with Noise)

- Eps: Distance beyond which the points won't be considered as the same cluster.
- Min Pts: Minimum number of points that must be present within Eps to be considered a core point.

[K not needed]

Gaussian Mixture Models (GMM)

This is a soft-clustering algorithm

- Assign multiple distributions to the data randomly.
- Update the mean and standard deviation iteratively until the clusters don't change further.

Other Methods

Mean Shift Clustering (*optional*)

Repeat the following steps until convergence or a maximum number of iterations is reached:

- For each data point, calculate the mean of all points within a certain radius (i.e., the "kernel") centered at the data point.
- Shift the data point to the mean.
- Identify the cluster centroids as the points that have not moved after convergence.
- Return the final cluster centroids and the assignments of data points to clusters.

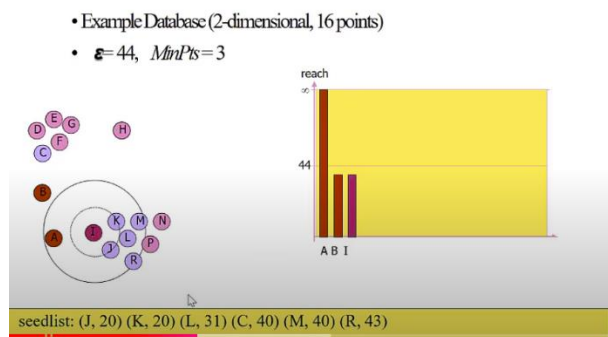
[K not needed]

Affinity Propagation is another method which does not need the number of clusters in advance [K not needed]

OPTICS (optional)

(Ordering Points to Identify the Clustering Structure)

- **Core Distance:** It is the minimum value of radius required to classify a given point as a core point. If the given point is not a Core point, then its Core Distance is undefined.
- **Reachability Distance:** It is defined with respect to another data point q (Let). The Reachability distance between a point p and q is the maximum of the Core Distance of p and the Euclidean Distance (or some other distance metric) between p and q . Note that The Reachability Distance is not defined if q is not a Core point.



Source: [Ch 10 OPTICS](#)

[k not needed]

Other methods:

Birch (Balanced Iterative Reducing and Clustering using Hierarchies)

Evaluation metrics

- Rand index
- Silhouette score
- DBCV

2. Silhouette Score (s)

Definition:

Measures how well a point fits within its own cluster compared to others.

Formula for a single point:

$$s = \frac{b - a}{\max(a, b)}$$

Where:

- **a** = average intra-cluster distance (mean distance to all other points in the same cluster)
- **b** = lowest average inter-cluster distance (mean distance to the nearest other cluster)

Silhouette score for the whole dataset is the mean of all individual scores.

The score is between -1 and 1.

DBC

The DBCV metric measures the average ratio of the distances between the data points and their cluster centroids, to the distances between the data points and the nearest data points in other clusters. The idea is that a good clustering solution should have compact and well-separated clusters, so the ratio of these distances should be high.

The DBCV metric is calculated using the following formula:

$$DBC = (1 / n) * \sum_{i=1}^n (\sum_{j=1}^n (d(i, j) / \max\{d(i, k), k \neq j\}))$$

where n is the number of data points, d(i,j) is the Euclidean distance between data points i and j, and max{d(i,k), k!=j} is the maximum distance between data point i and any other data point in a different cluster.

The DBCV metric ranges from 0 to 1, with lower values indicating better clustering solutions.

Neural Networks

Manually crafting features or choosing multiple ML models to extract different aspects of the data is tedious.

Neural Networks (NNs) automatically learn hierarchical features from raw data — simple patterns at early layers, complex concepts at deeper layers.

Activation Functions

Purpose: Introduce non-linearity so that the network can learn complex patterns.

Common Activation Functions

Activation	Formula / Curve	Use-case	Pros	Cons
ReLU	$\max(0, x)$	Default for most layers	Simple, fast	Dying ReLU problem

Sigmoid	$1 / (1 + e^{-x})$	Binary classification (output)	Smooth	Vanishing gradients
Tanh	$(e^x - e^{-x}) / (e^x + e^{-x})$	Better than sigmoid	Zero-centered	Vanishing gradients
Leaky ReLU	x if $x > 0$ else αx	Avoids dying ReLU	Gradient flows for $x < 0$	Extra parameter
Softmax	Probabilistic output	Multi-class output layer	Probabilistic interpretation	Only at output layer

Common Hyperparameters in Neural Networks

- Learning rate
- Batch size
- Number of epochs
- Number of layers
- Number of neurons per layer
- Activation functions
- Dropout rate
- Weight initialization method
- Optimizer

How to Avoid Overfitting?

- Dropout: Randomly disables neurons during training
- Regularization (L1/L2): Penalize large weights
- Early stopping: Stop training when validation loss increases
- Data augmentation: Expand dataset artificially
- Cross-validation: Tune hyperparameters
- Smaller model: Reduce model capacity

Exploding and Vanishing Gradients

Problem	Description	Fixes
Vanishing gradients	Gradients shrink → weights stop updating → network stops learning	<ul style="list-style-type: none"> - ReLU over sigmoid/tanh - Use batch normalization - Proper weight init (He/Xavier) - Skip connections (ResNet)
Exploding gradients	Gradients grow large → unstable updates → NaNs	<ul style="list-style-type: none"> - Gradient clipping - Better init - Normalization

Optimizers

Optimizer	Description	Notes
SGD	Basic gradient descent	Can be slow; sensitive to learning rate
SGD with Momentum	Adds velocity	Faster convergence
Adam	Momentum + adaptive learning rate	Popular default
RMSprop	Adapts learning rate	Good for RNNs
Adagrad	Parameter-specific LR	Can shrink LR too much

Normalization vs Batch Normalization

Feature	Normalization	Batch Normalization
What it is	Scaling input data to a standard range or distribution	A layer in the network that normalizes activations
When it's applied	Before training	During training
Where it's used	On the input data	On hidden layer activations
Goal	Help training converge with cleaner input	Stabilize and speed up training
How it works	Standardization: subtract mean, divide std dev	Batch-wise normalization + learnable scale/shift (γ , β)
Helps with	Consistent input scale	Faster, more stable training; acts as regularizer
Examples	StandardScaler, MinMaxScaler	BatchNorm1d, BatchNorm2d

Sequence Models

RNN

Recurrent Neural Networks (RNNs) are a type of neural network architecture designed to process sequential data by maintaining a hidden state that captures information from previous time steps. Below is an explanation of the architecture and steps involved in RNNs, including the formulas at each stage, along with the shapes of data and weights.

1. Input Data:

RNNs take in a sequence of tokens (such as words in a sentence or frames in a video) and process them one step at a time. The input at each time step is a vector representation of the token.

- **Input Data:** A sequence of tokens x_1, x_2, \dots, x_T , where T is the sequence length (number of tokens in the input).
- **Shape of Input:** (batch_size, sequence_length, input_dim), where:
 - batch_size is the number of sequences processed at once (mini-batch size),
 - sequence_length is the number of tokens in the sequence,
 - input_dim is the dimensionality of the input vector (e.g., the word embedding dimension).

2. Hidden State:

The RNN processes the input one token at a time. At each time step t , it updates its hidden state based on the current input x_t and the previous hidden state h_{t-1} .

- **Hidden State:** The hidden state at time step t is a vector that summarizes the information from the input sequence up to time step t .
- **Shape of Hidden State:** (batch_size, hidden_dim), where hidden_dim is the number of units (neurons) in the hidden layer of the RNN.

3. RNN Update Equations:

At each time step t , the hidden state is updated using the following formula:

$$h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

Where:

- h_t is the hidden state at time step t ,
- x_t is the input vector at time step t ,
- $W_{xh} \in \mathbb{R}^{\text{input_dim} \times \text{hidden_dim}}$ is the weight matrix for the input-to-hidden connection,
- $W_{hh} \in \mathbb{R}^{\text{hidden_dim} \times \text{hidden_dim}}$ is the weight matrix for the hidden-to-hidden connection,
- $b_h \in \mathbb{R}^{\text{hidden_dim}}$ is the bias term for the hidden state,
- \tanh is the activation function, typically used \downarrow introduce non-linearity.

- $b_h \in \mathbb{R}$ is the bias term for the hidden state,
- \tanh is the activation function, typically used to introduce non-linearity.

- **Shape of Weights:**

- W_{xh} : (input_dim, hidden_dim)
- W_{hh} : (hidden_dim, hidden_dim)
- b_h : (hidden_dim)

- **Shape after Hidden State Update:**

$$h_t \in \mathbb{R}^{\text{batch_size} \times \text{hidden_dim}}$$

At time $t = 1$, the initial hidden state h_0 is typically initialized as a vector of zeros:

$$h_0 = \mathbf{0} \in \mathbb{R}^{\text{batch_size} \times \text{hidden_dim}}$$

4. Output Layer:

After processing the input sequence, an output is typically generated at each time step. This can be a sequence of outputs (in the case of sequence-to-sequence tasks) or just a final output (in the case of sequence classification tasks). The output at each time step is computed as:

- **Output at time step t :**

$$y_t = W_{hy}h_t + b_y$$

Where:

- y_t is the output vector at time step t ,
- $W_{hy} \in \mathbb{R}^{\text{hidden_dim} \times \text{output_dim}}$ is the weight matrix for the hidden-to-output connection,
- $b_y \in \mathbb{R}^{\text{output_dim}}$ is the bias term for the output.

- **Shape of Output Weights:**

- W_{hy} : (hidden_dim, output_dim)
- b_y : (output_dim)

- **Shape after Output Computation:**

$$y_t \in \mathbb{R}^{\text{batch_size} \times \text{output_dim}}$$

- **Softmax (Optional):** If the task involves classification (such as predicting the next word in a sequence), a softmax function is applied to the output:

$$P(y_t) = \text{softmax}(y_t)$$

This converts the logits into probabilities.

5. Sequence Output or Final Output:

- **Sequence Output:** If we are dealing with a sequence-to-sequence task (e.g., language modeling or machine translation), the output is a sequence of vectors over time:

$$Y = [y_1, y_2, \dots, y_T] \in \mathbb{R}^{\text{batch_size} \times \text{sequence_length} \times \text{output_dim}}$$

- **Final Output:** If we are dealing with sequence classification (e.g., sentiment analysis), the output could be the final hidden state h_T :

$$y_{\text{final}} = W_{hy} h_T + b_y \in \mathbb{R}^{\text{batch_size} \times \text{output_dim}}$$

Transformers

1. Input Embedding Layer:

- **Input Data:** A sequence of tokens x_1, x_2, \dots, x_n , which are tokenized integers.
- **Shape of Input:** $(\text{batch_size}, \text{sequence_length})$, where the sequence length is the number of tokens in the sequence.
- **Embedding Layer:** The token IDs are mapped to vectors using an embedding matrix E .

$$E \in \mathbb{R}^{\text{vocab_size} \times \text{embedding_dim}}$$

The token embeddings are:

$$\text{embeddings} = \text{Embedding}(x) \in \mathbb{R}^{\text{batch_size} \times \text{sequence_length} \times \text{embedding_dim}}$$

- **Shape of Embedding Matrix:** $(\text{vocab_size}, \text{embedding_dim})$

2. Positional Encoding:

Since transformers don't inherently model sequentiality like RNNs, we add **Positional Encodings** to incorporate information about the position of tokens.

The positional encoding for a given position i and dimension d is defined as:

$$PE(i, 2d) = \sin\left(\frac{i}{10000^{2d/d_{\text{model}}}}\right), \quad PE(i, 2d + 1) = \cos\left(\frac{i}{10000^{2d/d_{\text{model}}}}\right)$$

Where:

- i is the position (index of the token in the sequence),
- d is the dimension of the embedding (each position is mapped to an embedding of size d_{model}).
- d_{model} is the dimensionality of the model (i.e., the size of the embedding vectors).

This positional encoding is added to the embeddings element-wise:

$$\text{embeddings_with_positional_encoding} = \text{embeddings} + \text{PE}$$

Where $\text{PE} \in \mathbb{R}^{\text{batch_size} \times \text{sequence_length} \times \text{embedding_dim}}$.

3. Encoder Layer:

The encoder consists of multiple layers, each containing the following components:

- **Multi-Head Self-Attention:** Self-attention allows each token to attend to every other token in the sequence. For a given token, we compute the attention with respect to all other tokens in the sequence. Here's how it's done:

1. **Query, Key, and Value:** The input embeddings are linearly projected into three spaces:

$$Q = \text{Input} \times W_Q \in \mathbb{R}^{\text{batch_size} \times \text{sequence_length} \times \text{attention_dim}}$$

$$K = \text{Input} \times W_K \in \mathbb{R}^{\text{batch_size} \times \text{sequence_length} \times \text{attention_dim}}$$

$$V = \text{Input} \times W_V \in \mathbb{R}^{\text{batch_size} \times \text{sequence_length} \times \text{attention_dim}}$$

Where:

- $W_Q, W_K, W_V \in \mathbb{R}^{\text{embedding_dim} \times \text{attention_dim}}$

2. **Scaled Dot-Product Attention:** Attention is computed using the scaled dot-product formula:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V$$

Where d_k is the dimension of the key vectors (i.e., the size of K).

- This gives us attention scores that indicate how much focus a particular token should give to other tokens in the sequence.
- **Shape of Attention Output:** $(\text{batch_size}, \text{sequence_length}, \text{attention_dim})$

3. **Multi-Head Attention:** We use multiple heads (parallel attention computations) to capture different kinds of relationships in the data. Each head has its own Q , K , and V weight matrices.

$$\text{MultiHeadAttention}(Q, K, V) = \text{Concat}(\text{head}_1, \text{head}_2, \dots, \text{head}_h) \times W_O$$

Where each head is computed as a separate attention operation.

- **Shape of Multi-Head Attention Output:**
 $(\text{batch_size}, \text{sequence_length}, \text{embedding_dim})$

4. **Add & Normalize:** After the attention computation, we add a residual connection and normalize the result:

$$\text{Output} = \text{LayerNorm}(\text{Input} + \text{MultiHeadAttention_Output})$$

- **Shape after Add & Normalize:** $(\text{batch_size}, \text{sequence_length}, \text{embedding_dim})$

- **Feed-Forward Network (FFN):** The output of the multi-head attention is passed through a position-wise feed-forward network:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

Where:

- $W_1 \in \mathbb{R}^{\text{embedding_dim} \times d_{\text{ff}}}$,
- $W_2 \in \mathbb{R}^{d_{\text{ff}} \times \text{embedding_dim}}$,
- $b_1 \in \mathbb{R}^{d_{\text{ff}}}$,
- $b_2 \in \mathbb{R}^{\text{embedding_dim}}$,
- d_{ff} is the dimensionality of the hidden layer in the feed-forward network.

The output of the feed-forward layer is:

$$\text{Output} = \text{LayerNorm}(\text{Input} + \text{FFN_Output})$$

- **Shape after Feed-Forward Network:** (batch_size, sequence_length, embedding_dim)


4. Decoder Layer:

The decoder is similar to the encoder but with an additional cross-attention mechanism (attention to encoder's output).

- **Masked Multi-Head Self-Attention:** Same as in the encoder but with masking to prevent the model from attending to future tokens during training.
- **Encoder-Decoder Attention:** The decoder also performs attention over the encoder's output:

$$\text{Encoder-Decoder Attention} = \text{softmax}\left(\frac{QK^{\text{encoder}}}{\sqrt{d_k}}\right)V^{\text{encoder}}$$

Where:

- $Q \in \mathbb{R}^{\text{batch_size} \times \text{sequence_length} \times \text{embedding_dim}}$
- $K^{\text{encoder}}, V^{\text{encoder}} \in \mathbb{R}^{\text{batch_size} \times \text{sequence_length} \times \text{embedding_dim}}$
- Output of Encoder-Decoder Attention: 

- Output of Encoder-Decoder Attention:
(batch_size, sequence_length, embedding_dim)

- **Add & Normalize:**

$$\text{Output} = \text{LayerNorm}(\text{Input} + \text{Encoder-Decoder Attention Output})$$

- **Feed-Forward Network:** Same as in the encoder.

5. Final Linear Layer and Softmax:

Finally, the decoder output is projected into a space of size vocab_size using a linear transformation:

$$\text{Logits} = \text{Output} \times W_{\text{linear}} + b_{\text{linear}}$$

Where:

- $W_{\text{linear}} \in \mathbb{R}^{\text{embedding_dim} \times \text{vocab_size}}$,
- $b_{\text{linear}} \in \mathbb{R}^{\text{vocab_size}}$.

Softmax is applied to convert the logits into probabilities:

Softmax is applied to convert the logits into probabilities:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

Where z_i are the logits for the i -th token.

Tf- IDF

Term frequency

Tells how often the term is occurring in the document

$$\text{TF}(t, d) = \frac{\text{Number of times term } t \text{ appears in document } d}{\text{Total number of terms in document } d}$$

Inverse document frequency

Tell how rare is the term among the documents

$$\text{IDF}(t, D) = \log \frac{\text{Total number of documents in corpus } D}{\text{Number of documents containing term } t}$$

$TFIDF = TF * IDF$

Document 1: "The cat sat on the mat."

Document 2: "The dog played in the park."

Document 3: "Cats and dogs are great pets."

The TF-IDF score is the product of TF and IDF:

$$TF-IDF(t, d, D) = TF(t, d) \times IDF(t, D)$$

For Document 1:

$$TF-IDF(\text{cat}, \text{Document 1}, D) = 0.167 \times 0.176 \approx 0.029$$

For Document 2:

$$TF-IDF(\text{cat}, \text{Document 2}, D) = 0 \times 0.176 = 0$$

For Document 3:

$$TF-IDF(\text{cat}, \text{Document 3}, D) = 0.167 \times 0.176 \approx 0.029$$

TF-IDF as Vectors — Quick Summary:

1. **TF-IDF** stands for **Term Frequency-Inverse Document Frequency**.
2. It gives a score to each word based on:
 - How often it appears in a document (**TF**)
 - How rare it is across all documents (**IDF**)
3. For each document, we calculate a **TF-IDF score** for every word in the **vocabulary**.
4. These scores form a **vector** — one value per word.
5. The result: each document becomes a **numeric vector** representing its content.

Word Piece tokenization

The WordPiece algorithm is a subword tokenization method used in NLP models like BERT. It works by breaking words into smaller, more frequent subword units, helping handle rare and out-of-vocabulary words.

Steps:

1. Start with characters: Each word is initially split into individual characters.
2. Merge frequent pairs: The algorithm iteratively merges the most common adjacent subword pairs based on frequency in the training corpus.
3. Stop when vocabulary size is reached: This continues until a predefined vocabulary size is built.

For example, "unhappiness" may be split into "un ##happiness", where "##" indicates a subword. This allows better generalization for rare words.

WordPiece improves efficiency and handling of unseen words, making it widely used in Transformer-based NLP models.

Word2Vec

Word2Vec converts words into dense vector representations that capture semantic meaning and relationships. It has two main architectures:

1. CBOW (Continuous Bag of Words)

- Goal: Predict the target word using its surrounding context words.
- Approach: Combine multiple context words into a single input vector to predict the central word.
- Training Samples: Multiple context words are used as input to predict one target word.

2. Skip-Gram

- Goal: Predict the surrounding context words using a single target word.
- Approach: For each target word, generate multiple training examples, one for each context word.
- Training Samples: One input word is used to predict multiple output context words.

Model Architecture

Let n = vocabulary size

dim = embedding dimension

$W1$: weight matrix of shape (n, dim) — maps one-hot input to embedding

$W2$: weight matrix of shape (dim, n) — maps embedding to output scores

Forward Pass

1. Input Word \rightarrow One-hot vector of shape $(n,)$
2. Embedding:

- Multiply with $W1$: $\text{one-hot} \times W1 \rightarrow \text{hidden layer output (dim,)}$

3. Output:

- Multiply with $W2$: $\text{hidden} \times W2 \rightarrow \text{output scores (n,)}$

4. Softmax: Converts scores into a probability distribution over the vocabulary

Training Process

- A large corpus of text is used to generate training samples.
- For each sample:
 - CBOW: Multiple context words are combined to predict a single target word.
 - Skip-Gram: One target word produces multiple training examples, each targeting one of the context words.
- The model computes a probability distribution using softmax and compares it with the actual word using cross-entropy loss.
- Gradients are computed and weights ($W1$ and $W2$) are updated using backpropagation and gradient descent.

Intuition: Why This Works?

- Words that occur in similar contexts tend to have similar meanings.
- The model learns embeddings such that words appearing in similar contexts are close together in vector space.
- This builds a semantic space where:
 - Similar words cluster together
 - Analogies can be captured through vector arithmetic (e.g., $\text{king} - \text{man} + \text{woman} \approx \text{queen}$)
- It reflects the distributional hypothesis: "You shall know a word by the company it keeps."

Inference

After training, the word embedding for any word is obtained by looking up the corresponding row in matrix $W1$.

Training objective of BERT

The training objective of BERT (Bidirectional Encoder Representations from Transformers) consists of two tasks:

1. Masked Language Model (MLM)

- A percentage (typically 15%) of tokens in the input text is randomly masked ([MASK] token).
- The model is trained to predict these masked words based on the surrounding context.
- This forces BERT to learn deep bidirectional representations.

Example:

Input: "The cat sat on the [MASK]."

Target: "mat"

2. Next Sentence Prediction (NSP)

- The model is given two sentences and learns to predict whether the second sentence follows the first in the original text (label: IsNext) or if it's a random sentence (label: NotNext).
- Helps in understanding relationships between sentences.

Example:

Sentence A: "I went to the store."

Sentence B: "I bought some milk." → Label: IsNext

Sentence A: "I went to the store."

Sentence B: "The sun is shining." → Label: NotNext

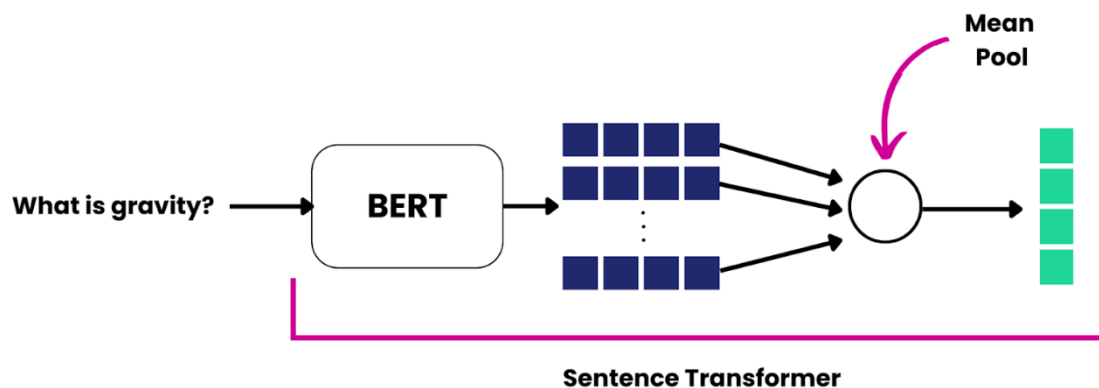
These objectives allow BERT to learn rich contextual representations, making it highly effective for various NLP tasks.

Sentence transformers

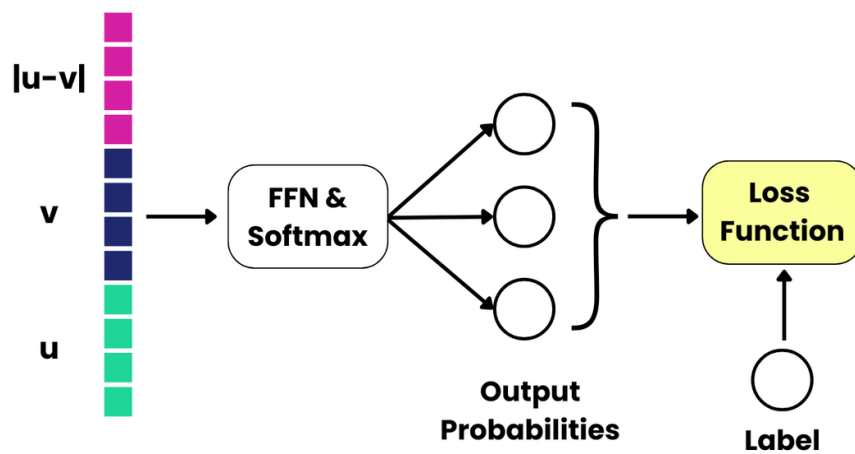
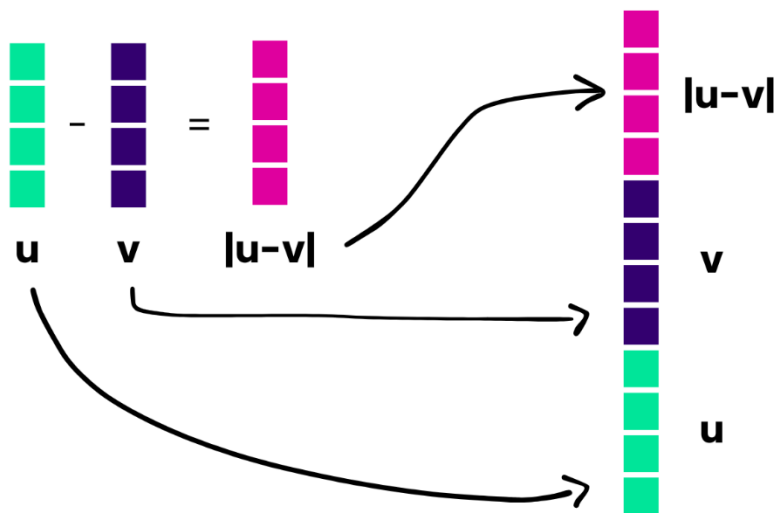
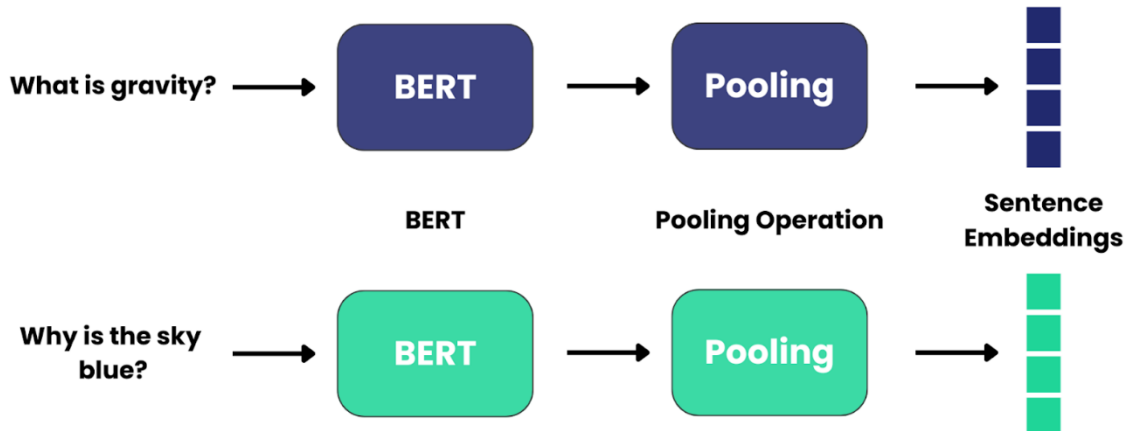
References:

<https://www.marqo.ai/course/introduction-to-sentence-transformers>

Inference



Training



Evaluating LLM response

When there is ground truth:

1. BLEU/ROUGE
2. Reranking models

When there is no ground truth:

1. Human evaluation
2. Reward models
3. LLM eval

Statistics

Links to questions:

- <https://www.datacamp.com/blog/statistics-interview-questions>
- <https://www.analyticsvidhya.com/blog/2022/08/top-40-data-science-statistics-interview-questions/>

Central Limit Theorem (CLT)

The **Central Limit Theorem** (CLT) is a fundamental concept in statistics. It states that if you take a large enough sample size from a population with any shape of distribution (whether normal or not), the sampling distribution of the sample mean will be approximately normal. This holds true regardless of the population's original distribution, as long as the sample size is sufficiently large.

Formally:

- If X_1, X_2, \dots, X_n are random variables drawn from any population with mean μ and standard deviation σ , then the distribution of the sample mean \bar{X} approaches a normal distribution with mean μ and standard deviation $\frac{\sigma}{\sqrt{n}}$, as n , the sample size, gets larger.

$$\text{As } n \rightarrow \infty, \quad \frac{\bar{X} - \mu}{\sigma/\sqrt{n}} \sim \mathcal{N}(0, 1)$$

where $\mathcal{N}(0, 1)$ is the standard normal distribution.

Why Is Hypothesis Testing Used?

Hypothesis testing is a fundamental statistical method used to make decisions or inferences about a population based on sample data. It helps determine whether a certain assumption (hypothesis) about a parameter is likely to be true.

Bootstrap distribution

Creating a bootstrap distribution involves resampling your data with replacement to create new datasets and then calculating the statistic of interest (such as the mean, median, standard

deviation, etc.) for each of these resampled datasets. This is repeated many times to form a distribution of the statistic.

Explain Z-test

Linear Algebra

Link to Questions:

<https://aiquest.org/datascience-ml-interview-questions-answers-on-linear-algebra/>

Define vector normalization.

Sample Answer: Vector normalization is the process of scaling a vector to have a unit norm (length). It is often used to ensure all vectors are on the same scale for distance-based algorithms like k-nearest neighbors.

How do you find the angle between two vectors?

Sample Answer: The angle (θ) between two vectors A and B can be calculated using the dot product formula: $\cos(\theta) = (A \cdot B) / (|A| * |B|)$.

How t-SNE Works?

The t-SNE algorithm finds the similarity measure between pairs of instances in higher and lower dimensional space. After that, it tries to optimize two similarity measures. It does all of that in three steps.

- t-SNE models a point being selected as a neighbor of another point in both higher and lower dimensions. It starts by calculating a pairwise similarity between all data points in the high-dimensional space using a Gaussian kernel. The points far apart have a lower probability of being picked than the points close together.
- The algorithm then tries to map higher-dimensional data points onto lower-dimensional space while preserving the pairwise similarities.
- It is achieved by minimizing the divergence between the original high-dimensional and lower-dimensional probability distribution. The algorithm uses gradient descent to minimize the divergence. The lower-dimensional embedding is optimized to a stable state.

How PCA works?

Principal Component Analysis (PCA) is a dimensionality reduction technique used to reduce the complexity of data while retaining as much variability as possible. It works by identifying the directions (or principal components) in which the data varies the most. Here's how it works:

1. Standardize the Data:
 - PCA starts by standardizing the data, especially if the features are on different scales. This is done by subtracting the mean of each feature and dividing by its standard deviation. Standardization ensures that each feature contributes equally to the analysis.
2. Compute the Covariance Matrix:

- The covariance matrix is calculated to understand how the features relate to each other. If two features vary together (positively or negatively), their covariance will be large. If they vary independently, their covariance will be close to zero.
3. Compute the Eigenvalues and Eigenvectors:
 - The next step is to compute the eigenvalues and eigenvectors of the covariance matrix. The eigenvectors represent the directions of maximum variance, and the eigenvalues represent the amount of variance along those directions. In simple terms, eigenvectors determine the new axes (principal components), and eigenvalues tell you how much information (variance) each principal component holds.
 4. Sort the Eigenvectors:
 - The eigenvectors are sorted by their corresponding eigenvalues in descending order. The eigenvectors with the largest eigenvalues capture the most variance in the data, so they are considered the most important principal components.
 5. Choose the Top k Eigenvectors:
 - Depending on how many dimensions you want to reduce the data to, you select the top k eigenvectors. These eigenvectors form a new set of axes for the data, and the number of eigenvectors corresponds to the desired number of dimensions.
 6. Transform the Data:
 - Finally, the data is projected onto the new axes (formed by the top k eigenvectors), resulting in a lower-dimensional representation of the data. This transformation preserves as much of the original variance as possible.

In summary, PCA finds the directions (principal components) that maximize the variance in the data and projects the data into these directions, reducing its dimensionality while keeping the most significant features.

How would you use matrix decomposition techniques to detect collinearity in a dataset?

Sample Answer: Matrix decomposition methods like SVD can help detect collinearity by revealing small singular values, indicating that some features are nearly linearly dependent, which can lead to issues in machine learning models.

Projection

Projection formula:

The projection of vector \mathbf{a} onto vector \mathbf{b} is given by:

$$\text{proj}_{\mathbf{b}}(\mathbf{a}) = \frac{\mathbf{a} \cdot \mathbf{b}}{\mathbf{b} \cdot \mathbf{b}} \cdot \mathbf{b}$$

$$\text{proj}_b(a) = \frac{a \cdot b}{|b|^2} \cdot b$$

What happens when features are correlated while doing linear regression

Collinearity in linear regression occurs when features are highly correlated, which leads to several issues:

1. **Unstable Coefficients:** The model's coefficient estimates become sensitive to small data changes.
2. **Inflated Standard Errors:** Makes it difficult to determine the statistical significance of features.
3. **Poor Model Interpretation:** It's hard to assess the individual impact of each feature since they contribute similarly to the outcome.
4. **Overfitting:** The model may fit noise, leading to poor generalization.

To handle collinearity:

- **Remove collinear features** (based on correlation).
- **Use PCA** to transform features into uncorrelated components.
- **Apply regularization** (Ridge or Lasso) to penalize large coefficients and reduce the effect of collinearity.

Coding

- [TimeSeries.ipynb](#)
- [Pandas Practice Set -1.ipynb](#)
- [Joining and merging.ipynb](#)
- [Aggregation.ipynb](#)
- [regex.ipynb](#)
- [Mobiles Dataset \(2025\).ipynb](#)
- [Python practice.ipynb](#)

Kubernetes

1. What is Kubernetes?

Kubernetes is an open-source platform for automating the deployment, scaling, and management of containerized applications. It helps manage clusters of hosts running containers, and provides features like load balancing, service discovery, scaling, and resource monitoring. It was originally developed by Google and is now maintained by the Cloud Native Computing Foundation (CNCF).

2. What are the key components of Kubernetes?

Kubernetes has several key components:

- **Cluster:** A set of nodes (physical or virtual machines) that run containerized applications.
 - **Node:** A single machine in the cluster, either a master or worker node.
 - **Pod:** The smallest deployable unit in Kubernetes, which can contain one or more containers.
 - **Deployment:** A higher-level construct that manages Pods and ensures a specified number of replicas are running.
 - **Service:** Exposes a set of Pods as a network service.
 - **ReplicaSet:** Ensures a specified number of Pod replicas are running.
 - **Namespace:** Provides a mechanism to scope resources in a cluster.
-

3. What is a Pod in Kubernetes?

A Pod is the smallest and simplest unit in Kubernetes. It represents a single instance of a running process in a cluster and can contain one or more containers. All containers in a Pod share the same network namespace, which means they can communicate with each other using localhost. Pods also share storage volumes, making it easier for the containers within them to share data.

4. What is the difference between a Pod and a container in Kubernetes?

- A Pod is a higher-level abstraction that can contain one or more containers.
 - A Container is a lightweight, standalone executable package that includes everything needed to run a piece of software, including code, libraries, and dependencies. Containers within a Pod share networking resources (e.g., IP address, ports) and storage. Pods are used to manage containerized applications.
-

5. What is a ReplicaSet in Kubernetes?

A ReplicaSet ensures that a specified number of replicas of a Pod are running at all times. If a Pod fails or is deleted, the ReplicaSet automatically creates new Pods to maintain the desired state. It is often used indirectly through Deployments, which manage ReplicaSets and provide features like rolling updates and rollbacks.

6. What is a Deployment in Kubernetes?

A Deployment is a higher-level abstraction in Kubernetes that manages a set of Pods using a ReplicaSet. Deployments provide declarative updates to Pods, which means you can define the desired state (e.g., number of replicas, container image) and Kubernetes will ensure the current state matches the desired state. Deployments also handle rolling updates, rollbacks, and scaling.

7. What is a Kubernetes Service?

A Service is a Kubernetes resource that exposes a set of Pods as a network service. It allows

communication between Pods and clients outside the Kubernetes cluster. Services provide stable DNS names and IP addresses to Pods, even if the Pods are rescheduled or replaced.

Types of Services include:

- ClusterIP (default): Exposes the Service on a cluster-internal IP.
- NodePort: Exposes the Service on a static port on each node's IP.
- LoadBalancer: Exposes the Service to external traffic through a cloud provider's load balancer.

8. What is the role of the Kubernetes Scheduler?

The Scheduler in Kubernetes is responsible for selecting a node for a newly created Pod to run on. It takes into account various factors such as resource availability, affinity/anti-affinity rules, and taints/tolerations. The Scheduler ensures that Pods are placed on nodes in a way that maximizes resource utilization while adhering to the specified policies.

9. What is the difference between StatefulSet and Deployment in Kubernetes?

- StatefulSet is used to manage stateful applications that require persistent storage and stable network identities. Each Pod in a StatefulSet gets a unique, persistent identifier (such as a DNS name) and retains its data across rescheduling.
- Deployment is used for stateless applications where Pods can be replaced at any time and don't need persistent storage.

10. What is a Namespace in Kubernetes?

A Namespace is a logical partition of a Kubernetes cluster that allows you to divide resources and organize workloads. It helps in isolating resources, such as Pods and Services, within a cluster. Namespaces are useful for multi-tenant environments or for organizing applications based on teams or projects.

11. What is a ConfigMap in Kubernetes?

A ConfigMap is a Kubernetes resource that allows you to store non-sensitive configuration data in key-value pairs. You can mount ConfigMaps as files or environment variables into Pods to provide configuration information for your applications without modifying the application code.

12. What are Kubernetes Secrets?

Secrets are similar to ConfigMaps, but they are specifically used for storing sensitive data like passwords, API keys, and tokens. Kubernetes provides encryption at rest and limited access to Secrets, making them more secure for storing sensitive information.

13. What is Kubernetes Ingress?

Ingress is an API object in Kubernetes that manages external access to services in the cluster,

typically HTTP or HTTPS. It provides routing based on hostnames and paths, and can be used with a load balancer or reverse proxy. Ingress controllers implement the actual routing mechanism.

14. What is a Helm chart?

Helm is a package manager for Kubernetes that helps in deploying applications. A Helm chart is a collection of files that describes a Kubernetes application, including deployment, configuration, and service details. Charts simplify the process of deploying and managing complex applications in Kubernetes.

15. What is a Node in Kubernetes?

A Node is a physical or virtual machine that is part of a Kubernetes cluster. There are two types of nodes:

- **Master Node:** Manages the Kubernetes cluster and makes global decisions about the cluster (e.g., scheduling, monitoring, etc.).
 - **Worker Node:** Runs the actual application workloads in the form of Pods.
-

16. How does Kubernetes handle scaling?

Kubernetes supports horizontal scaling through:

- **Horizontal Pod Autoscaler (HPA):** Automatically adjusts the number of Pods based on CPU utilization or custom metrics.
- **Manual Scaling:** Allows you to scale the number of Pods manually by changing the replica count in a Deployment or ReplicaSet.
Kubernetes also supports vertical scaling, which involves increasing the resources (CPU/Memory) allocated to individual Pods.