Machine Learning Conceptual Review

20 Deep-Understanding Questions and Answers

Based on the comprehensive notes by Riccardo Toniolo

Instructions

This document contains 20 open-ended questions designed to test a deep understanding of fundamental and advanced machine learning concepts as covered in the provided notes. Each question is followed by a detailed, comprehensive answer that explains the reasoning, derivations, and connections between ideas.

Contents

Foundations of Learning Theory

The No Free Lunch Theorem and Inductive Bias

Question 1: Define the **"No Free Lunch Theorem"** in the context of machine learning. Explain how it motivates the necessity of the *inductive bias* and describe its two main types with a practical example for each.

Answer: The **No Free Lunch Theorem** states that there is no single learning algorithm that is universally better than all others across all possible problems. If an algorithm performs well on a certain class of problems, it must perform worse on another class. This theorem implies that for an ML algorithm to be successful, it must incorporate prior assumptions about the nature of the problem it is trying to solve. These assumptions are collectively known as the **inductive bias**. It comes in two types:

- **Restriction Bias:** This limits the *hypothesis space* $\mathcal{H}$. For example, assuming the target function is linear restricts $\mathcal{H}$ to the set of all linear functions, excluding all non-linear possibilities.
- **Preference Bias:** This imposes an ordering or preference *within* the restricted hypothesis space. For example, even among linear functions, we might prefer those with smaller weights (via regularization), or in decision trees, we prefer the smallest tree that fits the data (Occam's Razor).

Bias-Variance Decomposition

Question 2: Derive and explain the **Bias-Variance decomposition** of the true risk for a model h. Define each term (noise, bias^2, variance) and explain what they represent about the model's learning process. How does the choice of hypothesis space $\mathcal{H}$ influence this trade-off?

Answer: The true risk (expected prediction error) for a model can be decomposed as follows:

$$\text{RISK}(h, P) = \sigma^2 + \text{BIAS}(h, P)^2 + \text{VARIANCE}(h, P)$$

- **Noise (σ^2):** The irreducible error inherent in the data generation process (e.g., measurement error). It sets a fundamental lower bound on the prediction error.
- **Bias^2:** Measures how much the average prediction of models (trained on different datasets from the distribution P) deviates from the true target value. BIAS $= E[h(x)] - y$. **High bias** indicates **underfitting**; the model's assumptions are too strong/simple to capture the underlying pattern (e.g., fitting a line to cubic data).
- **Variance:** Measures how much the predictions of individual models fluctuate around their average. VARIANCE $= E[(E[h(x)] - h(x))^2]$. **High variance** indicates **overfitting**; the model is too complex and sensitive to the specific training set.

The hypothesis space $\mathcal{H}$ directly controls this trade-off. A **simple** $\mathcal{H}$ (e.g., low-degree polynomials) leads to **high bias, low variance**. A **complex** $\mathcal{H}$ (e.g., high-degree poly-

nomials or large neural networks) leads to **low bias, high variance**. The goal is to find a $\mathcal{H}$ of "right" complexity that minimizes the sum.

VC Dimension and Generalization

Question 3: Explain the concept of **VC Dimension** for a hypothesis class $\mathcal{H}$. Describe the algorithm to compute it and compute the VC dimension for a linear perceptron in $\mathbb{R}^2$. How is the VC dimension used to bound the generalization error?

Answer: The Vapnik-Chervonenkis (VC) Dimension of a hypothesis class $\mathcal{H}$ is a measure of its capacity or complexity. Formally, $VC(\mathcal{H})$ is the largest number d of points that can be **shattered** by $\mathcal{H}$. A set of points is shattered if $\mathcal{H}$ can realize **all possible** 2^d dichotomies (labelings) of those points.

- **Computation Algorithm:** Start with $|S| = 1$. Try to find a placement of $|S|$ points that can be shattered by $\mathcal{H}$. If successful, increment $|S|$ and repeat. If, for a given $|S|$, **no** placement of $|S|$ points can be shattered, then $VC(\mathcal{H}) = |S| - 1$.
- **Example - Linear Perceptron in $\mathbb{R}^2$:** It can shatter any 3 points that are not collinear (all 8 labelings are achievable). However, for 4 points (e.g., in an XOR configuration), it cannot shatter them. Therefore, $VC(\text{Perceptron}_{\mathbb{R}^2}) = 3$. In general, $VC(\text{Perceptron}_{\mathbb{R}^d}) = d + 1$.
- **Generalization Bound:** Statistical learning theory provides a bound on the true risk: $\text{Risk}(g) \leq \text{Risk}_{\text{Tr}}(g) + F(VC(\mathcal{H})/m, \delta)$. The term F increases with $VC(\mathcal{H})$ and decreases with the training set size m. This formalizes the trade-off: a higher VC dimension allows fitting the training data better (lower Risk_{Tr}) but increases the penalty term F, risking overfitting.

Core Algorithms

The Perceptron

Question 4: Describe the **perceptron algorithm** for a linearly separable dataset. State its update rule and provide a step-by-step **convergence proof**, explaining the roles of the margin γ and the maximum norm R. Why does the perceptron fail for non-linearly separable data?

Answer: The perceptron is an online algorithm for binary classification ($y \in \{-1, 1\}$) that assumes linear separability. It starts with $\theta = 0$. For each training example (x, y), if $\theta^T x \cdot y \leq 0$ (misclassification), it updates: $\theta \leftarrow \theta + xy$.

- **Convergence Proof Sketch:** Let θ^* be the optimal separating hyperplane with $||\theta^*|| = 1$ and margin γ (min distance of any point to the boundary). Let R be the maximum norm of any data point $||x^i|| \leq R$. The proof shows two things after k updates:

  1. **Lower bound on alignment:** $\theta^k \cdot \theta^* \geq k\gamma$. The angle between the current θ and the optimal one improves with each mistake.
  2. **Upper bound on norm growth:** $||\theta^k||^2 \leq kR^2$. The length of θ cannot grow too fast.

  Combining these: $k^2\gamma^2 \leq (\theta^k \cdot \theta^*)^2 \leq ||\theta^k||^2||\theta^*||^2 \leq kR^2$. Therefore, $k \leq R^2/\gamma^2$. The number of mistakes (and thus updates) is bounded, guaranteeing convergence.

- **Failure on Non-Separable Data:** If the data is not linearly separable, the perceptron update condition will be triggered indefinitely. The algorithm will never achieve zero mistakes on the training set and will not converge to a stable solution; it will cycle through updates forever unless stopped by an external limit.

Logistic Regression vs. Perceptron

Question 5: Contrast **Logistic Regression** with the **Perceptron**. Derive the cost function for Logistic Regression (the cross-entropy loss) from the principle of **Maximum Likelihood Estimation (MLE)**. Show why Mean Squared Error (MSE) is not suitable for this task.

Answer:

- **Contrast:** Both are linear classifiers. The Perceptron uses a **step function** as activation, outputs a hard class label (-1/1), and only updates on mistakes. It converges only for linearly separable data. Logistic Regression uses the **sigmoid function**, outputs a **probability** $P(y = 1|x; \theta)$, and updates on every example via gradient descent on a convex loss. It always converges to a minimum (global for convex loss) even for non-separable data.
- **MLE Derivation:** We model $P(y = 1|x; \theta) = h_\theta(x) = \sigma(\theta^T x)$. Assuming i.i.d. samples, the likelihood is $L(\theta) = \prod_i P(y^i|x^i; \theta) = \prod_i h_\theta(x^i)^{y^i}(1 - h_\theta(x^i))^{1-y^i}$. Max-

imizing $L(\theta)$ is equivalent to minimizing the negative log-likelihood:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^i \log(h_\theta(x^i)) + (1-y^i)\log(1-h_\theta(x^i))\right]$$

This is the cross-entropy loss.

- **Why not MSE?** The MSE loss $J(\theta) = \frac{1}{m}\sum(h_\theta(x^i) - y^i)^2$, when combined with the sigmoid activation, becomes **non-convex** for logistic regression. This results in multiple local minima, making optimization via gradient descent unreliable. The cross-entropy loss, however, yields a convex optimization landscape for this problem.

Regularization

Question 6: How does **regularization** (e.g., in Linear or Logistic Regression) address the **bias-variance trade-off**? Derive the update rule for gradient descent with L2 regularization (weight decay) for a linear regression model. Explain the effect of the regularization parameter λ on the learned weights and the model's complexity.

Answer:

- **Role in Bias-Variance Trade-off:** Regularization explicitly controls model complexity. It penalizes large parameter values, effectively restricting the hypothesis space $\mathcal{H}$. Increasing regularization strength reduces variance (prevents overfitting to noise) but increases bias (limits the model's capacity to fit complex patterns).
- **Update Rule Derivation (Linear Regression with L2):** Cost: $J(\theta) = \frac{1}{2m}[\sum_{i=1}^{m}(h_\theta(x^i) - y^i)^2 + \lambda\sum_{j=1}^{n}\theta_j^2]$. Gradient for θ_j: $\frac{\partial J}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^i) - y^i)x_j^i + \frac{\lambda}{m}\theta_j$. Gradient Descent Update: $\theta_j \leftarrow \theta_j - \eta\frac{\partial J}{\partial \theta_j} = \theta_j(1 - \eta\frac{\lambda}{m}) - \eta\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^i) - y^i)x_j^i$. The term $(1 - \eta\frac{\lambda}{m})$ is the **weight decay** factor, shrinking θ_j towards zero at each step before the error-based update.
- **Effect of λ:** A **large** λ strongly penalizes large weights, pushing them toward zero. This results in a **simpler** model (e.g., a flatter regression line) with **high bias, low variance**. A **small** λ (or $\lambda = 0$) allows weights to grow large to fit the training data, leading to a **complex** model with **low bias, high variance**.

Kernel Methods and SVMs

The Kernel Trick

Question 7: Detail the **kernel trick**. Provide a toy example (e.g., with a quadratic kernel) to show how computing a kernel function $K(x, z)$ is computationally more efficient than explicitly computing the dot product $\phi(x)^T\phi(z)$ in the high-dimensional feature space. Name two common kernel functions and their properties.

Answer: The kernel trick allows learning non-linear models by implicitly computing dot products in a very high (even infinite) dimensional feature space $\phi(x)$ without ever explicitly calculating $\phi(x)$. It replaces $\phi(x)^T\phi(z)$ with a kernel function $K(x, z)$ computable in the original input space.

- **Toy Example:** Consider $\phi : \mathbb{R}^2 \to \mathbb{R}^3$ with $\phi(x) = (x_1^2, x_2^2, \sqrt{2}x_1x_2)$. The explicit dot product is $\phi(x)^T\phi(z) = x_1^2z_1^2 + x_2^2z_2^2 + 2x_1z_1x_2z_2$. The kernel function $K(x, z) = (x^Tz)^2 = (x_1z_1 + x_2z_2)^2$ yields the **same result** with $O(d)$ operations instead of $O(d')$ where $d' > d$.
- **Common Kernels:**
  1. **Polynomial Kernel:** $K(x, z) = (x^Tz + c)^d$. Corresponds to a feature space of all monomials up to degree d.
  2. **Radial Basis Function (RBF) / Gaussian Kernel:** $K(x, z) = \exp(-\gamma||x - z||^2)$. Corresponds to an **infinite-dimensional** feature space. It measures similarity that decays with distance, controlled by the bandwidth γ.

Support Vector Machines

Question 8: Describe the **Support Vector Machine (SVM)** optimization problem for both the **separable** and **non-separable** (soft-margin) cases. Explain the roles of the parameters C and ξ_i. How does the SVM objective function relate to the **Hinge Loss**?

Answer:

- **Separable Case:** The goal is to find the hyperplane with the maximum margin.

$$\min_{w,b} \frac{1}{2}||w||^2 \quad \text{subject to} \quad y_i(w^Tx_i + b) \geq 1, \forall i$$

  The constraint enforces correct classification with a margin of at least 1.

- **Non-Separable Case (Soft-Margin):** Slack variables $\xi_i \geq 0$ are introduced to allow constraint violations (misclassifications/margin violations).

$$\min_{w,b,\xi} \frac{1}{2}||w||^2 + C\sum_i \xi_i \quad \text{subject to} \quad y_i(w^Tx_i + b) \geq 1 - \xi_i, \forall i$$

- **Roles:** $C > 0$ is a **regularization hyperparameter** that controls the trade-off between maximizing the margin ($\frac{1}{||w||}$) and minimizing the classification error ($\sum \xi_i$). A large C heavily penalizes errors (narrower margin, less tolerance for violations).

A small C prioritizes a large margin, accepting more violations. ξ_i is the amount by which the point i violates the margin.

- **Connection to Hinge Loss:** The constraint $\xi_i \geq 1 - y_i(w^T x_i + b)$ and the minimization of $\sum \xi_i$ mean that optimally, $\xi_i = \max(0, 1 - y_i(w^T x_i + b))$. This is exactly the **Hinge Loss**. Therefore, the SVM objective is equivalent to minimizing $||w||^2/2 + C \sum_i \text{HingeLoss}(f(x_i), y_i)$.

The XOR Problem

Question 9: Analyze the **XOR problem** in the context of linear classifiers. Why is it impossible for a linear model (like a perceptron or logistic regression) to solve it? Propose and explain **two distinct solutions** from the notes that enable a model to solve the XOR problem.

Answer:

- **Impossibility for Linear Models:** The XOR problem has four points: (0,0)-¿0, (1,1)-¿0, (0,1)-¿1, (1,0)-¿1. There is **no single straight line (hyperplane in 2D)** that can separate the class 0 points from the class 1 points. The problem is not linearly separable.
- **Solution 1: Feature Engineering / Non-linear Transformation:** Map the original features (x_1, x_2) into a new, higher-dimensional space where the data becomes linearly separable. Example: use $\phi(x) = (x_1, x_2, x_1 \cdot x_2)$. In this 3D space, the classes can be separated by a plane. A linear classifier can then be applied in this new feature space.
- **Solution 2: Kernel Methods:** Instead of explicitly computing the transformation $\phi(x)$, use a **kernel function** that computes dot products in the high-dimensional space directly. For example, a polynomial kernel $K(x, z) = (x^T z + 1)^2$ corresponds to a specific feature expansion that can make XOR linearly separable. The kernel trick allows the linear classifier to operate in this implicit high-dimensional space.
- **Solution 3: Neural Networks:** Use a neural network with at least one hidden layer and non-linear activations. The hidden neurons can learn to construct the necessary non-linear features (like an AND gate, OR gate, etc.) whose combination at the output layer can solve XOR, as shown in the notes with a 2-layer network.

Trees, Ensembles, and Unsupervised Learning

Decision Trees

Question 10: Explain how **Decision Trees** learn from data. Define **Information Gain** and **Gini Index** as impurity measures. Write the pseudocode for the ID3 algorithm and discuss the fundamental challenge of tree learning that leads to **overfitting**, along with two strategies to mitigate it (pre-pruning and post-pruning).

Answer: Decision trees learn by recursively partitioning the feature space based on the feature that best splits the data, aiming to create "pure" leaf nodes (containing samples from primarily one class).

- **Information Gain (based on Entropy):** $IG(S, a) = H(S) - \sum_{t \in T} p(t)H(t)$. $H(S) = -\sum_c p(c) \log_2 p(c)$ is the entropy (uncertainty) of set S. IG measures the reduction in entropy after splitting on attribute a. The attribute with the highest IG is chosen.
- **Gini Index:** $Gini(S) = 1 - \sum_c p(c)^2$. It measures the probability of misclassifying a randomly chosen element if labeled according to the class distribution. A split minimizing the weighted sum of child Gini indices is preferred.
- **ID3 Pseudocode (recursive):**

```
function ID3(examples, attributes):
    if all examples same class: return leaf_node(class)
    if attributes empty: return leaf_node(majority_class)
    A = attribute in attributes with highest Information Gain
    node = internal_node(test on A)
    for each value v of A:
        examples_v = subset(examples where A=v)
        if examples_v empty: child = leaf_node(majority_class)
        else: child = ID3(examples_v, attributes - {A})
        node.add_child(v, child)
    return node
```

- **Overfitting Challenge:** A tree can grow until every training example is perfectly classified (one leaf per example), which is a complex model that memorizes noise and fails to generalize.
- **Mitigation Strategies:**
  1. **Pre-pruning (Early Stopping):** Stop growing the tree before it perfectly fits the data. Criteria: max depth, minimum samples per leaf, minimum impurity decrease.
  2. **Post-pruning:** Grow the full tree, then remove subtrees that do not provide a significant gain in generalization accuracy (e.g., using a validation set to evaluate the impact of pruning a node).

Ensemble Methods: Bagging vs. Boosting

Question 11: Compare and contrast **Bagging** (e.g., Random Forests) and **Boosting** (e.g., AdaBoost) as ensemble methods. Explain the underlying justification for why en-

sembles work, focusing on the bias-variance decomposition. What are the key differences in their training procedures and their typical effects on bias and variance?

Answer:

- **Justification:** Ensembles work by combining multiple weak learners. According to the bias-variance decomposition, averaging predictions from multiple models trained on different data reduces **variance** (as individual model errors cancel out), while sequentially focusing on errors can reduce **bias**.
- **Bagging (Bootstrap Aggregating - Parallel):**
  - **Procedure:** Train many base learners (e.g., decision trees) **independently** on different bootstrap samples (random samples with replacement) of the training data. For prediction, aggregate results via **voting** (classification) or **averaging** (regression).
  - **Effect:** Primarily reduces **variance**. By averaging, it smooths out the high variance of unstable learners like deep decision trees. It leaves **bias** relatively unchanged. Random Forests add further variance reduction by also randomizing the feature choice at each split.
- **Boosting (Sequential):**
  - **Procedure:** Train base learners **sequentially**. Each new learner is trained to correct the errors of the combined ensemble so far. Training examples are re-weighted, giving more weight to previously misclassified examples. Learners are combined via a **weighted sum**.
  - **Effect:** Can reduce both **bias** and **variance**. Early learners capture broad patterns, reducing bias. Combining many learners smooths the decision boundary, reducing variance. It is more prone to overfitting on noisy data.

K-Means Clustering

Question 12: Describe the **K-Means clustering** algorithm. Write down its objective function J and derive the update rule for the centroids μ_k by taking the partial derivative of J with respect to μ_k and setting it to zero. What are the major weaknesses of K-Means, and how can the initialization issue be addressed?

Answer: K-Means is an unsupervised algorithm that partitions data into K clusters.

1. **Algorithm:** Initialize K centroids randomly. Repeat: (a) **Assignment:** Assign each point to the nearest centroid. (b) **Update:** Recompute each centroid as the mean of all points assigned to it. Until centroids stabilize.
2. **Objective Function:** $J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} ||x_n - \mu_k||^2$, where $r_{nk} = 1$ if point n is assigned to cluster k, else 0.
3. **Update Rule Derivation:**

$$\frac{\partial J}{\partial \mu_k} = \sum_{n=1}^{N} r_{nk} \cdot 2(x_n - \mu_k) \cdot (-1) = -2 \sum_{n=1}^{N} r_{nk}(x_n - \mu_k)$$

Set derivative to zero: $-2\sum_n r_{nk}(x_n - \mu_k) = 0 \implies \sum_n r_{nk}x_n = \mu_k \sum_n r_{nk}$. Therefore,

$$\mu_k = \frac{\sum_{n=1}^{N} r_{nk}x_n}{\sum_{n=1}^{N} r_{nk}}$$

The new centroid is the mean of points in cluster k.

4. **Weaknesses & Initialization:**
   - **Weaknesses:** Sensitive to initial centroid placement (converges to local optimum), assumes spherical clusters of similar size, struggles with non-globular clusters and different densities.
   - **Addressing Initialization:** Use **K-Means++** initialization: choose first centroid randomly, then choose subsequent centroids with probability proportional to the squared distance from the closest existing centroid. This spreads out initial centroids and leads to better, more consistent results.

Neural Networks

Backpropagation

Question 13: Explain the **backpropagation** algorithm for training neural networks. Use a simple 3-layer network as an example to illustrate the **chain rule** flow for calculating the gradient of the loss with respect to a weight in the first layer. Why is the vanishing/exploding gradient problem particularly relevant in deep networks, and how can ReLU activations help mitigate it?

Answer: Backpropagation is an algorithm to efficiently compute gradients of the loss function with respect to all network parameters by applying the chain rule from the output layer backwards.

- **Illustration (3-layer net):** Let layers be $L1 \to L2 \to L3$ (output). Let $z^l = W^l a^{l-1} + b^l$ and $a^l = f(z^l)$. Loss J. We compute gradients backward:
  1. Output layer error: $\delta^3 = \frac{\partial J}{\partial a^3} \odot f'(z^3)$.
  2. Propagate to L2: $\delta^2 = ((W^3)^T \delta^3) \odot f'(z^2)$.
  3. Gradient for W^2: $\frac{\partial J}{\partial W^2} = \delta^2 (a^1)^T$.

  This shows the chain rule: the error δ^2 depends on δ^3 multiplied by the transposed weights and the activation derivative.
- **Vanishing/Exploding Gradients:** In deep networks, gradients are computed via a long chain of multiplications. If the derivatives $f'(z)$ or the weight matrices have norms consistently ¡1, the gradient can **vanish** (become extremely small) as it propagates backward, preventing early layers from learning. If they are ¿1, it can **explode**.
- **Role of ReLU:** The ReLU activation $f(z) = \max(0, z)$ has a derivative of 1 for positive inputs. Unlike sigmoid/tanh whose derivatives are ¡1, this constant gradient of 1 helps alleviate the vanishing gradient problem for active neurons, allowing gradients to flow more easily through deep networks.

Inductive Biases in Neural Networks

Question 14: Elaborate on the **inductive biases of Neural Networks**. While a one-hidden-layer network is a universal approximator, what practical constraints during training act as an implicit bias? Discuss the importance of **weight initialization**, the choice of **activation functions**, and techniques like **dropout** and **batch normalization** in shaping what the network learns.

Answer: The **Universal Approximation Theorem** states that a neural network with a single hidden layer can approximate any continuous function, suggesting minimal *representational* bias. However, the *learning* process introduces strong inductive biases:

- **Training Constraints as Bias:** The optimization algorithm (e.g., SGD), the network architecture (depth, width), the regularization, and the limited training data all constrain which of the many possible function approximations the network

will actually find. SGD favors solutions that are reachable via a path of gradual loss decrease from the initialization point.

- **Weight Initialization:** Crucial to break symmetry. If weights are initialized to zero, all neurons in a layer compute the same function, preventing learning. Proper initialization (e.g., He or Xavier) ensures activations and gradients have appropriate scales at the start of training, facilitating stable gradient flow.
- **Activation Functions:** ReLU introduces **sparsity** and helps with vanishing gradients but can cause "dying neurons." Choices like Leaky ReLU or ELU mitigate this. The non-linearity allows the network to learn complex representations.
- **Dropout:** Acts as a powerful **regularizer** that prevents co-adaptation of neurons. During training, it randomly "drops" units, effectively training an ensemble of thinned subnetworks. This biases the network towards learning robust, distributed representations that don't rely too heavily on any single neuron.
- **Batch Normalization:** Reduces **internal covariate shift** by normalizing layer inputs. It acts as a regularizer, often allows for higher learning rates, and makes the optimization landscape smoother, biasing the network towards solutions that are more stable and easier to optimize.

Recommender Systems

Collaborative Filtering

Question 15: For **Collaborative Filtering** in recommender systems, compare **user-based** and **item-based** approaches using the **adjusted cosine similarity**. Provide the formulas and explain why adjusted cosine is preferred over standard cosine or Pearson correlation for item similarity. How do these similarities feed into a **k-Nearest Neighbors** prediction for an explicit rating?

Answer:

- **Adjusted Cosine Similarity (Item-Based):**

$$s_{ij} = \frac{\sum_{u \in U_{ij}} (r_{ui} - \mu_u)(r_{uj} - \mu_u)}{\sqrt{\sum_{u \in U_{ij}} (r_{ui} - \mu_u)^2} \sqrt{\sum_{u \in U_{ij}} (r_{uj} - \mu_u)^2}}$$

  where μ_u is user u's average rating. The key is subtracting the **user's mean** μ_u, not the item's mean.

- **Why Adjusted Cosine?** Users have different rating baselines (some are strict, some are generous). Standard cosine on raw ratings or Pearson correlation (which centers by item mean) fails to account for this user bias. Adjusted cosine corrects for this by centering each rating relative to the user who gave it, providing a better measure of item-to-item similarity based on user preference deviation.

- **User-Based vs. Item-Based:** User-based finds users similar to the target user and aggregates their ratings. Item-based finds items similar to the target item and aggregates the target user's ratings for those similar items. Item-based is often more stable as user preferences change more rapidly than item characteristics.

- **k-NN Prediction:** For item-based predicting user u's rating on item i:

$$\hat{r}_{ui} = \frac{\sum_{j \in N_u^k(i)} s_{ij} \cdot r_{uj}}{\sum_{j \in N_u^k(i)} |s_{ij}|}$$

  where $N_u^k(i)$ are the k items most similar to i that user u has rated. It's a weighted average of u's ratings on similar items.

Matrix Factorization

Question 16: Explain the concept of **Matrix Factorization** for collaborative filtering. Derive the **Stochastic Gradient Descent (SGD)** update rules for the user latent vector p_u and item latent vector q_i starting from the regularized objective function $\mathcal{L}$. What is the interpretation of the dot product $p_u^T q_i$?

Answer: Matrix Factorization models the user-item rating matrix R as the product of two lower-dimensional matrices: $R \approx PQ^T$, where P (user factors) and Q (item factors) are learned.

- **Objective:** $\mathcal{L} = \sum_{(u,i) \in \text{Train}} (r_{ui} - p_u^T q_i)^2 + \lambda(\|p_u\|^2 + \|q_i\|^2)$.

- **SGD Update Derivation:** For a single training example (u, i), define error $e_{ui} = r_{ui} - p_u^T q_i$.

$$\frac{\partial \mathcal{L}}{\partial p_u} = -2e_{ui}q_i + 2\lambda p_u, \quad \frac{\partial \mathcal{L}}{\partial q_i} = -2e_{ui}p_u + 2\lambda q_i$$

  Using the SGD update rule $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}$:

$$p_u \leftarrow p_u + \eta(e_{ui}q_i - \lambda p_u)$$

$$q_i \leftarrow q_i + \eta(e_{ui}p_u - \lambda q_i)$$

- **Interpretation:** The dot product $p_u^T q_i$ represents the **interaction** between user u and item i in a shared **latent factor space**. Each dimension can be thought of as a latent feature (e.g., "action-level," "comedy-level," "thought-provoking"). A high value in p_u for a feature indicates user preference, a high value in q_i indicates item possession of that feature. Their dot product estimates how well the item matches the user's preferences.

Alternating Least Squares

Question 17: Describe the **Alternating Least Squares (ALS)** optimization method for Matrix Factorization. Contrast it with SGD. What is the key advantage of ALS that makes it attractive for large-scale, distributed implementations?

Answer:

- **ALS Method:** The MF objective $\min_{P,Q} \sum (r_{ui} - p_u^T q_i)^2 + \lambda(\|p_u\|^2 + \|q_i\|^2)$ is biconvex. ALS fixes one set of variables and solves for the other.
  1. **Fix** Q, solve for p_u. The problem becomes a ridge regression for each user: $p_u = (Q_I^T Q_I + \lambda I)^{-1} Q_I^T r_u$, where Q_I contains rows for items rated by user u.
  2. **Fix** P, solve for q_i similarly: $q_i = (P_U^T P_U + \lambda I)^{-1} P_U^T r_i$.
  3. Alternate steps 1 and 2 until convergence.
- **Contrast with SGD:** SGD updates parameters for each training example immediately. ALS works in "batch" mode for each user/item, using all relevant data at once in a closed-form solution.
- **Key Advantage (Parallelization/Distribution):** When Q is fixed, the updates for all users p_u are **independent** of each other. They can be computed in parallel across many machines/cores. Similarly, when P is fixed, all q_i updates are independent. This makes ALS exceptionally well-suited for distributed computation frameworks (like Spark), allowing it to scale to massive datasets far more easily than the inherently sequential nature of standard SGD.

Model Evaluation and Practical Challenges

Validation and Cross-Validation

Question 18: In the context of model evaluation and selection, explain why a **validation set** is necessary and how **K-Fold Cross-Validation** works. Derive the probabilistic bound on the error estimate based on **Hoeffding's inequality** and discuss its implications for the size of the validation set and the number of models/hyperparameters tried (M).

Answer:

- **Need for Validation Set:** The test set must remain untouched for a final, unbiased estimate of generalization error. If we use the test set to tune hyperparameters, we indirectly "train" on it, causing **overfitting to the test set** and an optimistic error estimate. The validation set provides an independent dataset for model selection and hyperparameter tuning.
- **K-Fold Cross-Validation:** The training data is split into K equal-sized folds. For $k = 1...K$: train on $K-1$ folds, validate on the k-th fold. The final performance estimate is the average of the K validation errors. This provides a more reliable and less variable estimate than a single train/validation split, making better use of limited data.
- **Hoeffding's Bound:** For a single model, Hoeffding's inequality gives: $P(|\hat{E}_{val} - E| \geq \epsilon) \leq 2e^{-2v\epsilon^2}$, where v is validation set size. This bounds the probability that the empirical error deviates from the true error by more than ϵ.
- **Implications for Model Search:** When trying M models, the **union bound** modifies this to $P(...) \leq 2Me^{-2v\epsilon^2}$. To keep the probability of a large deviation ($\geq \epsilon$) below a threshold δ, we need $v \geq \frac{1}{2\epsilon^2}\log(\frac{2M}{\delta})$. This shows that the required validation set size grows **logarithmically** with the number of models M tried. Trying many hyperparameters requires a larger validation set for reliable comparisons.

The Curse of Dimensionality

Question 19: Discuss the **curse of dimensionality** in the context of the **k-Nearest Neighbors (KNN)** algorithm. How does it affect distance metrics, and what assumption do we typically make about real-world data to justify using KNN despite this curse? Mention one technique to alleviate the problem.

Answer: The curse of dimensionality refers to the phenomenon where, as the number of features d increases, the volume of the space increases so rapidly that the available data becomes sparse. This has severe consequences for KNN:

- **Effect on Distance:** In high dimensions, the distance between any two points becomes increasingly similar. The ratio of the nearest distance to the farthest distance approaches 1, making the concept of "nearest neighbors" less meaningful and discriminant.

- **Assumption for Justification:** We assume that real-world data does **not** uniformly fill the high-dimensional space. Instead, it lies on or near a much lower-dimensional **manifold** (a lower-dimensional subspace with a complex geometric structure). For example, meaningful images of faces occupy a tiny fraction of the space of all possible pixel arrays.
- **Mitigation Technique: Dimensionality Reduction** via techniques like PCA (Principal Component Analysis) or autoencoders can be applied before using KNN. This projects the data onto a lower-dimensional subspace that (hopefully) preserves the meaningful structure and distances, mitigating the curse.

Recommender System Evaluation

Question 20: In recommender systems, distinguish between the tasks of **rating prediction** and **top-N recommendation**. List and define at least three evaluation metrics appropriate for each task. For top-N recommendation, explain the **Discounted Cumulative Gain (DCG)** metric and its normalized version (nDCG).

Answer:

- **Rating Prediction:** Predicts the exact rating a user would give to an unseen item. It's a **regression** task, often using explicit feedback (e.g., 1-5 stars).
  - **Metrics:** Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE). These measure the average deviation between predicted and true ratings.
- **Top-N Recommendation:** Generates a ranked list of N items the user is most likely to prefer/interact with. It's a **ranking/retrieval** task, often using implicit feedback.
  - **Metrics:**
    1. **Precision:** Fraction of recommended items (in the top N) that are relevant. $= \frac{\#\text{relevant in top N}}{N}$.
    2. **Recall:** Fraction of all relevant items that are found in the top N. $= \frac{\#\text{relevant in top N}}{\text{total \# relevant}}$.
    3. **Mean Average Precision (MAP):** Summarizes precision-recall trade-off across all ranks, emphasizing ranking quality.
- **Discounted Cumulative Gain (DCG):** Measures the quality of a ranked list by summing the relevance of each item, discounted by its rank (logarithmically). High-quality items at the top contribute more.

$$DCG = \sum_{i=1}^{P} \frac{rel_i}{\log_2(i+1)}$$

  where rel_i is the graded relevance of the item at position i.
- **nDCG (Normalized DCG):** Normalizes DCG by the **Ideal DCG (IDCG)**, which is the DCG of the perfect ranking (all relevant items sorted by relevance at

the top).

$$nDCG@P = \frac{DCG}{IDCG}$$

This results in a score between 0 and 1, making it comparable across different queries/users.