Machine Learning Exam Notes

Tomáš Jelínek

January 12, 2026

Contents

Disclaimer

These notes are based on the lecture slides from NPFL129 course: Introduction to Machine Learning with Python in winter semester 2025/26 which are under CC-BY-SA-4.0 license. The notes are not guaranteed to be correct and are not a substitute for the lecture. They are intended to be used as a study aid and should not be used as the only source of information for the exam.

Artificial Intelligence such as GPT-4, GitHub Copilot and possibly others were used in the proccess of writing this document.

License

Part I
Lecture 1

Q1.1 Explain how reinforcement learning differs from supervised and unsupervised learning in terms of the type of input the learning algorithms use to improve model performance. [5]

Reinforcement learning differs from supervised and unsupervised learning based on the type of input and feedback used to improve model performance. In supervised learning, the algorithm learns from labeled data where each input is paired with a known output, aiming to minimize prediction errors. Unsupervised learning uses unlabeled data to uncover hidden patterns or structures without explicit feedback. In contrast, reinforcement learning involves an agent interacting with an environment, receiving rewards or penalties as feedback for actions taken, and learning through trial-and-error to maximize cumulative rewards over time. Unlike supervised learning's direct guidance and unsupervised learning's pattern discovery, reinforcement learning focuses on sequential decision-making.

Q1.2 Explain why we need separate train and test data? What is generalization and how the concept relates to underfitting and overfitting? [10]

Why Separate Train and Test Data:

- To evaluate the performance of a machine learning model reliably.
- Training data is used to fit the model, while test data assesses its performance on unseen data.
- Prevents overfitting, ensuring the model generalizes well to new, unseen data.

Generalization:

- The ability of a model to perform well on new, unseen data.
- Indicates how well the model learns the underlying patterns, not just memorizing the training data.

Relation to Underfitting and Overfitting:

- **Underfitting:** Model is too simple, fails to capture underlying patterns in data, leading to poor performance on both training and test data.
- **Overfitting:** Model is too complex, captures noise along with patterns in the training data, leading to poor generalization on test data.

Q1.3 Define the three key components of Mitchell's definition of machine learning (Task T, Performance measure P, and Experience E). Give a concrete example for each component in the context of email spam classification. [10]

Mitchell (1997) states that a program *learns* from **experience** E with respect to a class of **tasks** T and **performance measure** P if its performance at tasks in T, as measured by P, improves with experience E.

Task T (what the system must do)

Definition: The task specifies the mapping the model should learn (e.g., classification or regression, structured prediction, denoising, density estimation).

Spam example: *Binary classification* of emails: given an email (subject + body), predict one of two classes

$$T: \; x \mapsto y, \quad y \in \{\text{spam}, \; \text{ham}\}.$$

Here x can be represented by features such as word counts/TF–IDF, presence of URLs, sender domain, etc.

Performance measure P (how success is evaluated)

Definition: A quantitative metric used to evaluate how well the program performs the task.

Spam example: Measure performance on a held-out test set using, e.g., accuracy, error rate or F_1 score:

$$P = \text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \quad \text{or} \quad P = F_1 = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}.$$

(Using F_1 is common when the classes are imbalanced.)

Experience E (what data/feedback the system learns from)

Definition: The source of experience used to improve performance (e.g., labeled data in supervised learning).

Supervised: usually a dataset with desired outcomes (labels or targets)

Unsupervised: usually data without any annotation (raw text, raw images, ...

Reinforcement learning, semi-supervised learning, ...

Spam example: A supervised training dataset of emails with ground-truth labels:

$$E = \{(x^{(i)}, y^{(i)})\}_{i=1}^{N}, \quad y^{(i)} \in \{\text{spam}, \text{ham}\}.$$

The model "learns" by adjusting its parameters based on these labeled examples so that P improves over time.

Q1.4 Explain the difference between classification and regression tasks. For each task type, provide: (a) the mathematical representation of the target variable, (b) a real-world example, and (c) one appropriate evaluation metric. [10]

In supervised learning, each example has an input $\boldsymbol{x} \in \mathbb{R}^D$ and a target t. The key difference is the *type* of the target: classification predicts a discrete label, while regression predicts a real value.

Classification

(a) **Target variable:** a class/label from a fixed set of K categories, e.g.

$$t \in \{0, 1, \ldots, K-1\}.$$

(Equivalently, one can use a one-hot vector $\boldsymbol{t} \in \{0,1\}^K$ with $\sum_k t_k = 1$.)

(b) **Real-world example:** email spam detection, where the model assigns the label

$$t \in \{\text{spam}, \text{ham}\}.$$

(c) **Evaluation metric:** *accuracy* (or error rate / F-score), e.g.

$$\text{Accuracy} = \frac{\#\text{correct predictions}}{\#\text{all predictions}}.$$

Regression

(a) **Target variable:** a real-valued number,

$$t \in \mathbb{R}.$$

(b) **Real-world example:** predicting the price of a house (a continuous value) from features such as area, location, and number of rooms.

(c) **Evaluation metric:** *mean squared error (MSE)* (often reported as RMSE),

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(\hat{t}_i - t_i)^2, \qquad \text{RMSE} = \sqrt{\text{MSE}}.$$

Q1.5 Define prediction function of a linear regression model and write down L^2-regularized mean squared error loss. [10]

Prediction Function: Given an input vector $x \in \mathbb{R}^D$, the prediction function f for linear regression is defined as:

$$f(x; w, b) = w^T x + b$$

where w is the weight vector, b is the bias term, and T denotes the transpose of w.

L_2-Regularized Mean Squared Error Loss: The L_2-regularized mean squared error loss (also known as Ridge Regression) for a dataset with N samples is defined as:

$$L(w, b) = \frac{1}{2N}\sum_{i=1}^{N}(f(x_i; w) - t_i)^2 + \lambda\|w\|^2$$

where t_i is the true target value for the i-th sample, λ is the regularization parameter, and $\|w\|^2$ denotes the L_2 norm of the weight vector, which is the sum of the squares of its components.

Q1.6 Starting from unregularized sum of squares error of a linear regression model, show how the explicit solution can be obtained, assuming X^TX is regular. [10]

In order to find a minimum of $\frac{1}{2}\sum_{i=1}^{N}(x_i^Tw - t_i)^2$, we can inspect values where the derivative of the error function is zero, with respect to all weights w_j.

$$\frac{\partial}{\partial w_j}\frac{1}{2}\sum_{i=1}^{N}(x_i^Tw - t_i)^2 = \frac{1}{2}\sum_{i=1}^{N}2(x_i^Tw - t_i)x_{ij} = \sum_{i=1}^{N}x_{ij}(x_i^Tw - t_i)$$

Therefore, we want for all j that $\sum_{i=1}^{N}x_{ij}(x_i^Tw - t_i) = 0$. We can rewrite the explicit sum into $X_{*,j}^T(Xw - t) = 0$, then write the equations for all j together using matrix notation as $X^T(Xw - t) = 0$, and finally, rewrite to

$$X^TXw = X^Tt.$$

The matrix X^TX is of size $D \times D$. If it is regular, we can compute its inverse and therefore

$$w = (X^TX)^{-1}X^Tt.$$

Part II
Lecture 2

Q2.1 Describe standard gradient descent and compare it to stochastic (i.e., online) gradient descent and minibatch stochastic gradient descent. Explain what it is used for in machine learning. [10]

We use it to search for the best model weights in an iterative/incremental/sequential fashion. Either because there is too much data, or the direct optimization is not feasible.

Standard gradient descent, also known as batch gradient descent, computes the gradient of the cost function with respect to the parameters (w) for the entire training dataset:

$$w \leftarrow w - \alpha\nabla_w E(w)$$

where α is the learning rate.

Stochastic Gradient Descent (SGD), or online gradient descent, on the other hand, updates the parameters for each training example:

$$\nabla_w E(w) \approx \nabla_w L(y(x_i; w), t_i)$$

This method is noisier but can converge faster for large datasets.

Minibatch SGD is a compromise between the two, updating the parameters for a small subset of the training data:

$$\nabla_w E(w) \approx \frac{1}{B}\sum_{i=1}^{B}\nabla_w L(y(x_i; w), t_i)$$

This approach aims to balance the computational efficiency of standard gradient descent with the faster convergence of SGD.

Q2.2 Explain the relationship between model capacity and overfitting/underfitting. How does increasing polynomial degree in linear regression affect model capacity, and what are the consequences? [10]

Model capacity vs. underfitting/overfitting

Model capacity describes how rich a set of functions a model can represent (representational capacity) and, in practice, how rich it effectively behaves under training/regularization (effective capacity). If capacity is *too small*, the model cannot capture the underlying pattern in the data, leading to **underfitting**: both training and test errors are high. If capacity is *too large* relative to the amount/noise of data, the model can fit idiosyncrasies of the training set, leading to **overfitting**: training error becomes very low, but test/validation error increases (a growing generalization gap). Typically, as capacity increases, training error decreases monotonically, while test error follows a U-shape with an optimal intermediate capacity.

Polynomial degree as capacity control in linear regression

Although linear regression is linear in parameters, using **polynomial features** increases its capacity. For scalar input $x \in \mathbb{R}$, define the feature vector

$$\boldsymbol{x} = (x^0, x^1, \ldots, x^M),$$

then the prediction becomes

$$y(x) = \sum_{j=0}^{M} w_j x^j,$$

i.e., a polynomial of degree M. Increasing the polynomial degree M therefore increases the model's **representational capacity** (the hypothesis space becomes larger and can represent more complex functions).

Consequences of increasing M

- **From underfitting to good fit:** small M may be too rigid (e.g. almost linear), so the model underfits and cannot match the curvature of the data.
- **Risk of overfitting:** large M can fit the training points extremely closely (very low training error), but may produce highly oscillatory functions and become sensitive to noise/outliers, worsening validation/test error.
- **Need for control:** M is a *hyperparameter* typically chosen using a validation set. Regularization (e.g. L_2 weight decay) can reduce *effective capacity* by discouraging large weights, often mitigating the overfitting observed at high degrees.

Q2.3 Explain possible intuitions behind L^2 regularization. [5]

L^2 regularization helps prevent overfitting by adding a penalty for large weights in a model. This penalty is the sum of the squares of the model's parameters, making it costly to have very large values. By keeping the weights smaller, the model becomes simpler and less likely to fit noise in the data. It also balances bias and variance, reducing the model's sensitivity to specific features and improving its ability to generalize to new data.

Generally, we want to reduce overfitting of the model. In short, the regularization controls complexity to create smoother, more reliable models.

Q2.4 Explain the difference between hyperparameters and parameters. [5]

Parameters are internal values that the model learns from the training data. Examples include weights in linear regression or neural networks and the split points in decision trees. Parameters change automatically during training to optimize the model's performance by minimizing the loss function.

Hyperparameters are external configurations set before training begins and control how the learning process operates. Examples include the learning rate, regularization strength, and the number of hidden layers in a neural network. Unlike parameters, hyperparameters are not learned from the data but must be tuned manually or using automated search techniques to find the best model performance.

In summary, parameters are learned by the model, while hyperparameters are predefined settings that influence how the model learns.

Q2.5 Write an L^2-regularized minibatch SGD algorithm for training a linear regression model, including the explicit formulas (i.e, formulas you would need to code it with `numpy`c) of the loss function and its gradient. [10]

The loss function for L^2-regularized linear regression is given by:

$$E(w) = \frac{1}{2}\mathbb{E}_{(x,t)\sim p_{\text{data}}}[(x^T w - t)^2] + \frac{\lambda}{2}\|w\|^2$$

where w are the weights, x is the input, t is the target, and λ is the regularization parameter.

The gradient of the loss function with respect to the weights is:

$$\nabla_w E(w) \approx \frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}}((x_i^T w - t_i)x_i) + \lambda w$$

where $\mathcal{B}$ is a minibatch of examples.

Pseudocode of the minibatch SGD algorithm

Require: Dataset $\{X \in \mathbb{R}^{N\times D}, t \in \mathbb{R}^N\}$, learning rate $\alpha \in \mathbb{R}_+$, L_2 strength $\lambda \in \mathbb{R}$
Ensure: Weights $w \in \mathbb{R}^D$ minimizing the regularized MSE of a linear regression model.

1: Initialize w randomly
2: **repeat**
3: Sample a minibatch $\mathcal{B}$ of examples with indices $\mathcal{B}$
4: Compute gradient g according to $\nabla_w E(w)$ using $\mathcal{B}$
5: Update w: $w \leftarrow w - \alpha \cdot g$
6: **until** convergence or maximum number of iterations is reached

---

Q2.6 Does the SGD algorithm for linear regression always find the best solution on the training data? If yes, explain under what conditions it happens, if not explain why it is not guaranteed to converge. What properties of the error function does this depend on? [10]

Stochastic Gradient Descent (SGD) for linear regression does not always guarantee finding the best solution on the training data. It converges to the global optimum if the following conditions are met:

- The loss function is convex and continuous.
- The sequence of learning rates α_i meets the Robbins-Monro conditions, which are:
  - $\alpha_i > 0$
  - $\sum_{i=1}^{\infty} \alpha_i = \infty$
  - $\sum_{i=1}^{\infty} \alpha_i^2 < \infty$
- The third condition ensures that $\alpha_i \to 0$ as $i \to \infty$.

When these conditions are satisfied, SGD converges to the unique optimum of convex problems. However, for non-convex loss functions, SGD is not guaranteed to find the global minimum; it may converge to a local minimum instead. The noise in the gradient estimation due to the stochastic nature of the algorithm can also affect convergence. Thus, while SGD can perform well in practice, especially for large datasets, it doesn't always find the best solution due to these factors.

Q2.7 After training a model with SGD, you ended up with a low training error and a high test error. Using the learning curves, explain what might have happened and what steps you might take to prevent this from happening. [10]

The learning curves might indicate that while the training loss decreases over time, the test loss decreases initially but then starts to increase. This scenario suggests that the **model is overfitting** to the training data. Overfitting occurs when a model learns the training data too well, including noise and details that do not generalize to unseen data. Consequently, the model performs well on the training data but poorly on the test data.

To prevent overfitting, you can take the following steps:

1. Use regularization techniques such as L_1 (LASSO) or L_2 (Ridge) to penalize large weights in the model.
2. Implement early stopping based on validation performance to halt training before overfitting occurs.
3. Increase the size of the training set if possible, to provide the model with more generalizable examples.
4. Simplify the model by reducing its complexity to prevent it from capturing noise in the data. (Make less features, use less layers, etc.)

These methods can help in guiding the model to generalize better to unseen data and thus improve its test performance.

Another reason might be that the model **failed to converge**. In this case, you can try to increase the number of iterations or decrease the learning rate to improve convergence.

Q2.8 You were given a fixed training set and a fixed test set, and you are supposed to report model performance on that test set. You need to decide what hyperparameters to use. How will you proceed and why? [10]

To determine the best hyperparameters for a model given a fixed training and test set, the following procedure should be employed:

1. **Split the training set:** Divide the training set into a smaller training set and a validation set.
2. **Hyperparameter tuning:** Use the smaller training set to train different models with various hyperparameter configurations. (Grid Search, Random Search, Hyperband, SMAC, etc.)
3. **Validation:** Evaluate the performance of each model on the validation set.
4. **Selection:** Choose the hyperparameters that yield the best performance on the validation set.
5. **Final Model:** Train a new model on the full training set using the selected hyperparameters.
6. **Testing:** Report the model's performance on the fixed test set.

This procedure is crucial because it helps to estimate the model's performance on unseen data and prevents overfitting to the training set. The validation set acts as a proxy for the test set, allowing for an unbiased evaluation of hyperparameter choices.

Q2.9 What method can be used for normalizing feature values? Explain why it is useful. [5]

Feature normalization can be achieved through methods such as Min-Max normalization and Z-score standardization. These methods are useful for several reasons:

- **Min-Max Normalization:** Scales the features to a fixed range, typically [0, 1]. It is given by the formula:
$$x'_{i,j} = \frac{x_{i,j} - \min_k x_{k,j}}{\max_k x_{k,j} - \min_k x_{k,j}}$$
This method is beneficial when we need to bound our features within a specific scale without distorting differences in the ranges of values.
- **Z-score Standardization:** Transforms the features to have a mean of zero and a standard deviation of one. The formula is:
$$x'_{i,j} = \frac{x_{i,j} - \bar{x}_j}{\sigma_j}$$
This is particularly useful in optimization algorithms that require features on a comparable scale for efficient learning.

Additionally, techniques similar to PCA, such as Principal Component Analysis itself, can be used for feature scaling and reduction. PCA transforms the data into a new coordinate system, reducing dimensionality and potentially improving model performance by removing noise and redundancy in the data.

Q2.10 You have a dataset with a categorical feature "color" with values "red", "green", "blue". Explain why using integer encoding ('red=0', 'green=1', 'blue=2') is problematic for linear regression. How would encode such feature instead? [10]

Using integer encoding (red= 0, green= 1, blue= 2) is problematic in linear regression because it imposes an *artificial ordering and distance* between categories. The model treats "color" as a numeric variable, so its contribution is

$$w_{\text{color}} \cdot \text{color},$$

which implies (i) a ranking red < green < blue, and (ii) that the step from red→green is the same as from green→blue. This is not meaningful for nominal categories and forces the effect to be linear in these arbitrary integers (e.g. blue would be assumed to have twice the effect of green relative to red).

A suitable encoding is **one-hot (dummy) encoding**. Represent the three categories by binary indicator variables:

$$\text{red} = (1, 0, 0), \quad \text{green} = (0, 1, 0), \quad \text{blue} = (0, 0, 1).$$

Then the linear model can learn an independent weight for each category. In practice, to avoid redundancy with the bias term (perfect multicollinearity), one typically uses $K - 1$ dummy variables (reference coding), e.g.

$$x_1 = \mathbb{I}[\text{green}], \qquad x_2 = \mathbb{I}[\text{blue}],$$

and treat "red" as the reference category. The prediction becomes

$$\hat{y} = b + w_1 \, \mathbb{I}[\text{green}] + w_2 \, \mathbb{I}[\text{blue}],$$

so w_1 and w_2 represent the effects of green/blue relative to red.

Part III
Lecture 3

Q3.1 Define binary classification, write down the perceptron algorithm, and show how a prediction is made for a given data instance x. [10]

Binary classification is the task of classifying the elements of a given set into two groups based on a classification rule. In binary classification, the output variable can take only two values, typically denoted as 0 and 1, or -1 and 1 in some contexts.

The perceptron algorithm is a binary classifier that linearly separates these two classes. The algorithm iteratively adjusts the weights based on the training data. Given a set of features x and a target t, the perceptron rule updates the weights w as follows:

Algorithm 1 Perceptron Learning Algorithm

Require: Linearly separable dataset $X \in \mathbb{R}^{N \times D}$, labels $\boldsymbol{t} \in \{-1, +1\}^N$
Ensure: Weights $\boldsymbol{w} \in \mathbb{R}^D$ such that $t_i \, \boldsymbol{x}_i^\top \boldsymbol{w} > 0$ for all i

$\boldsymbol{w} \leftarrow \mathbf{0}$
while not all examples are classified correctly **do**
  for $i = 1$ **to** N **do**
    $y \leftarrow \boldsymbol{x}_i^\top \boldsymbol{w}$
    if $t_i \, y \leq 0$ **then** ▷ misclassified example
      $\boldsymbol{w} \leftarrow \boldsymbol{w} + t_i \, \boldsymbol{x}_i$
    end if
  end for
end while

To make a prediction $\hat{y}$ for a new example with feature vector x, the perceptron uses the sign of the dot product between the features and weights:

$$\hat{y} = \text{sign}(x^T w)$$

where sign is an activation function that maps positive values to +1 and non-positive values to -1.

Prediction Example

Given a new input x and trained weights w, the perceptron prediction is computed as:

$$\hat{y} = \text{sign}(x^T w + b)$$

where b is the bias term of the perceptron. If $\hat{y}$ is positive, the input is classified into one class, and if it is negative, it is classified into the other class.

Q3.2 Explain what it means for a dataset to be linearly separable. Give an example of a simple 2D dataset that is *not* linearly separable and explain why the perceptron algorithm would fail on it. [10]

A binary-labeled dataset $\{(\boldsymbol{x}_i, t_i)\}_{i=1}^N$ with $t_i \in \{-1, +1\}$ is called **linearly separable** if there exist parameters $(\boldsymbol{w}, b)$ such that a single linear decision boundary separates the two classes:

$$\exists\,(\boldsymbol{w}, b) \;\text{ s.t. }\; t_i\big(\boldsymbol{w}^\top \boldsymbol{x}_i + b\big) > 0 \quad \forall i.$$

Equivalently, all positive examples lie in one open half-space and all negative examples lie in the other half-space induced by the hyperplane (line in 2D) $\boldsymbol{w}^\top \boldsymbol{x} + b = 0$.

Example of a simple 2D dataset that is *not* linearly separable

A standard example is the XOR pattern with four points in $\mathbb{R}^2$:

$$\text{Positive } (+1): \; (0,0),\; (1,1), \qquad \text{Negative } (-1): \; (0,1),\; (1,0).$$

This set is *not* linearly separable because any line divides the plane into two half-planes, but in XOR each class occupies two opposite corners of a square; whichever side of a line contains $(0,0)$ will necessarily contain at least one negative point (or vice versa). Therefore, no single linear boundary can make all signs correct simultaneously.

Why the perceptron fails on such data

The perceptron algorithm iteratively updates $\boldsymbol{w}$ whenever it finds a misclassified example, and its convergence guarantee relies on the existence of a separating hyperplane (linear separability). If the dataset is not linearly separable, there is *no* $\boldsymbol{w}$ (and b) satisfying $t_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b) > 0$ for all i, so the algorithm keeps encountering misclassified points and continues updating indefinitely (it does not terminate/converge).

Q3.3 For discrete random variables, define entropy, cross-entropy, and Kullback-Leibler divergence, and prove the Gibbs inequality (i.e., that KL divergence is non-negative). [20]

Entropy $H(P)$ for a discrete random variable with probability distribution P is defined as:

$$H(P) = -\sum_x P(x) \log P(x)$$

It measures the expected level of 'surprise' or uncertainty inherent in the variable's possible outcomes.

Cross-entropy $H(P, Q)$ between two discrete probability distributions P and Q is defined as:

$$H(P, Q) = -\sum_x P(x) \log Q(x)$$

It measures the expected number of bits (if the log is in base 2) required to identify an event from a set of possibilities if a wrong distribution Q is used instead of the true distribution P.

Kullback-Leibler divergence $D_{KL}(P||Q)$ from Q to P is defined as:

$$D_{KL}(P||Q) = H(P, Q) - H(P) = \sum_x P(x) \log \frac{P(x)}{Q(x)}$$

It measures how one probability distribution diverges from a second, expected probability distribution.

Proof of Gibbs Inequality: We want to prove that $D_{KL}(P||Q) \geq 0$, with equality if and only if $P = Q$.

Using the log sum inequality $\log \frac{a}{b} \leq \frac{a}{b} - 1$ with equality only if $a = b$, we have:

$$\begin{aligned} H(P) - H(P, Q) &= \sum_x P(x) \log \frac{Q(x)}{P(x)} \\ &\leq \sum_x P(x) \left(\frac{Q(x)}{P(x)} - 1 \right) \\ &= \sum_x Q(x) - \sum_x P(x) \\ &= 0 \end{aligned}$$

since $\sum_x P(x) = 1$ and $\sum_x Q(x) = 1$. The inequality is strict unless $P(x) = Q(x)$ for all x, which proves Gibbs Inequality.

Q3.4 Explain the notion of likelihood in machine learning. What likelihood are we estimating, and why do we do it? [10]

Likelihood in the context of maximum likelihood estimation (MLE) is a function that measures the probability of observing the given data under different parameter values of a statistical model. For a set of independent and identically distributed (i.i.d) data points $X = \{x_1, x_2, \ldots, x_N\}$, the likelihood of a parameter w is defined as:

$$L(w) = \prod_{i=1}^{N} P_{\text{model}}(x_i; w)$$

where $P_{\text{model}}(x_i; w)$ is the probability of observing the specific data point x_i under the model parameterized by w.

In MLE, we seek the parameter w that maximizes this likelihood function, which is equivalent to maximizing the probability of observing the given data. While the likelihood itself is not a probability distribution, it serves as a scoring function that indicates how well the model with a particular set of parameters explains the observed data. Maximizing the likelihood function leads to finding the parameter values that make the observed data most probable under the assumed model.

Q3.5 Describe maximum likelihood estimation as minimizing NLL, cross-entropy, and KL divergence and explain whether they differ or are the same and why. [20]

Let $x_1, \ldots, x_N$ be i.i.d. samples from an (unknown) data distribution $p_{\text{data}}(x)$. We fit a probabilistic model $p_{\text{model}}(x; w)$.

1) Maximum likelihood $\Leftrightarrow$ minimizing negative log-likelihood (NLL)

Maximum likelihood estimation chooses parameters that maximize the likelihood of the observed dataset:

$$w_{\text{MLE}} = \arg\max_{w} \prod_{i=1}^{N} p_{\text{model}}(x_i; w).$$

Taking the logarithm (monotone) and negating turns this into minimization of the **negative log-likelihood**:

$$w_{\text{MLE}} = \arg\min_{w} \left[-\sum_{i=1}^{N} \log p_{\text{model}}(x_i; w) \right].$$

Often we use the average NLL (same minimizer, just scaled by $1/N$):

$$\mathcal{L}_{\text{NLL}}(w) = -\frac{1}{N} \sum_{i=1}^{N} \log p_{\text{model}}(x_i; w).$$

2) NLL $\Leftrightarrow$ cross-entropy with the empirical distribution

Define the empirical distribution $\hat{p}(x)$ that assigns probability $1/N$ to each observed sample (and 0 elsewhere). Then the average NLL can be written as an expectation:

$$\mathcal{L}_{\text{NLL}}(w) = \mathbb{E}_{x \sim \hat{p}} \big[-\log p_{\text{model}}(x; w) \big].$$

By definition, the **cross-entropy** between distributions p and q is

$$H(p, q) = \mathbb{E}_{x\sim p}\big[-\log q(x)\big].$$

Therefore,

$$\boxed{\mathcal{L}_{\mathrm{NLL}}(w) = H(\hat{p},\ p_{\mathrm{model}}(x; w))}.$$

So, for a fixed dataset, **minimizing average NLL is exactly the same objective as minimizing cross-entropy with $\hat{p}$.**

3) Cross-entropy $\Leftrightarrow$ KL divergence (up to an additive constant)

The KL divergence is

$$D_{\mathrm{KL}}(p\|q) = \mathbb{E}_{x\sim p}\left[\log \frac{p(x)}{q(x)}\right] = \mathbb{E}_{x\sim p}[\log p(x)] - \mathbb{E}_{x\sim p}[\log q(x)].$$

Using $H(p) = \mathbb{E}_{x\sim p}[-\log p(x)]$ and $H(p, q) = \mathbb{E}_{x\sim p}[-\log q(x)]$, we get the identity

$$\boxed{H(p, q) = H(p) + D_{\mathrm{KL}}(p\|q)}.$$

Applying this with $p = \hat{p}$ and $q = p_{\mathrm{model}}(x; w)$ yields

$$H(\hat{p},\ p_{\mathrm{model}}(x; w)) = H(\hat{p}) + D_{\mathrm{KL}}(\hat{p}\,\|\,p_{\mathrm{model}}(x; w))\,.$$

4) Are NLL, cross-entropy, and KL the same or different?

- **NLL vs. cross-entropy:** for a fixed dataset, *they are the same objective* (NLL is cross-entropy with the empirical distribution $\hat{p}$), possibly differing only by a constant scale factor (1 vs. $1/N$), which does not change the minimizer.
- **Cross-entropy vs. KL:** they differ by an *additive constant* $H(\hat{p})$ that does *not* depend on w. Hence,

$$\arg\min_w H(\hat{p}, p_{\mathrm{model}}(\cdot; w)) \;=\; \arg\min_w D_{\mathrm{KL}}(\hat{p}\,\|\,p_{\mathrm{model}}(\cdot; w)).$$

Therefore, **MLE can be viewed equivalently as minimizing NLL, minimizing cross-entropy, or minimizing KL divergence**, because these objectives are identical up to scaling and/or adding constants independent of w.

5) Conditional (supervised) case

For supervised learning with pairs (x_i, t_i) and model $p_{\mathrm{model}}(t \mid x; w)$:

$$w_{\mathrm{MLE}} = \arg\min_w \left[-\sum_{i=1}^{N} \log p_{\mathrm{model}}(t_i \mid x_i; w)\right] = \arg\min_w\ \mathbb{E}_{(x,t)\sim\hat{p}}\big[-\log p_{\mathrm{model}}(t \mid x; w)\big].$$

This is the **conditional cross-entropy** between the empirical conditional distribution and the model, and it equals a conditional KL divergence plus an additive constant independent of w.

Let $X = \{x_1, x_2, \ldots, x_N\}$ be training data drawn independently from the data-generating distribution p_{data}. We denote the empirical data distribution as $\hat{p}_{\mathrm{data}}$, where $\hat{p}_{\mathrm{data}}(x) = \frac{1}{N}\sum_{i=1}^{N} \mathbb{I}[x_i = x]$. Let $p_{\mathrm{model}}(x; w)$ be a family of distributions.

The maximum likelihood estimation of w is:

$$w_{\mathrm{MLE}} = \arg\max_w p_{\mathrm{model}}(X; w) = \arg\max_w \prod_{i=1}^{N} p_{\mathrm{model}}(x_i; w)$$

$$= \arg\min_w -\sum_{i=1}^{N} \log p_{\text{model}}(x_i; w)$$

$$= \arg\min_w \mathbb{E}_{x\sim\hat{p}_{\text{data}}}[-\log p_{\text{model}}(x; w)]$$

$$= \arg\min_w H(\hat{p}_{\text{data}}(x), p_{\text{model}}(x; w))$$

$$= \arg\min_w D_{KL}(\hat{p}_{\text{data}}(x)||p_{\text{model}}(x; w)) + H(\hat{p}_{\text{data}}(x))$$

For MLE generalized to the conditional case, where the goal is to predict t given x:

$$w_{\text{MLE}} = \arg\max_w p_{\text{model}}(t|x; w) = \arg\max_w \prod_{i=1}^{N} p_{\text{model}}(t_i|x_i; w)$$

$$= \arg\min_w -\sum_{i=1}^{N} \log p_{\text{model}}(t_i|x_i; w)$$

$$= \arg\min_w \mathbb{E}_{(x,t)\sim\hat{p}_{\text{data}}}[-\log p_{\text{model}}(t|x; w)]$$

$$= \arg\min_w H(\hat{p}_{\text{data}}(t|x), p_{\text{model}}(t|x; w))$$

$$= \arg\min_w D_{KL}(\hat{p}_{\text{data}}(t|x)||p_{\text{model}}(t|x; w)) + H(\hat{p}_{\text{data}}(t|x))$$

Where $H(\hat{p}_{\text{data}})$ is the entropy of the empirical data distribution and D_{KL} is the Kullback-Leibler divergence. The terms are defined such that the conditional entropy is $H(\hat{p}_{\text{data}}) = \mathbb{E}_{(x,t)\sim\hat{p}_{\text{data}}}[-\log(\hat{p}_{\text{data}}(t|x))]$ and the conditional cross-entropy is $H(\hat{p}_{\text{data}}, p_{\text{model}}) = \mathbb{E}_{(x,t)\sim\hat{p}_{\text{data}}}[-\log(p_{\text{model}}(t|x;$ The negative log-likelihood (NLL) is equivalent to cross-entropy or Kullback-Leibler divergence in the context of MLE.

Q3.6 Provide an intuitive justification for why cross-entropy is a good optimization objective in machine learning. What distributions do we compare in cross-entropy? Why is it good when the cross-entropy is low? [5]

Cross-entropy is a good optimization objective in machine learning because it measures how well the predicted probability distribution from a model matches the true distribution of the target labels. Intuitively, it quantifies the difference between what the model believes (its predicted probabilities) and the actual outcomes.

In classification tasks, cross-entropy compares:

- True distribution: Represented as a one-hot encoded vector for each class, where the correct class has a probability of 1, and all others are 0.
- Predicted distribution: The model's output probabilities for each class (softmax or sigmoid outputs).

When the cross-entropy is low, it means the predicted probabilities are close to the true labels — the model is confident and correct. For instance, if a model correctly predicts a probability near 1 for the correct class and near 0 for others, the cross-entropy loss is minimal. Conversely, high cross-entropy means the predictions are far from the true labels, indicating poor performance.

In essence, minimizing cross-entropy encourages the model to assign high probabilities to correct labels, leading to better classification accuracy and more confident predictions aligned with the true data distribution.

Q3.7 Considering binary logistic regression model, write down its parameters (including their size) and explain how prediction is performed (including the formula for the sigmoid function). [10]

In binary logistic regression we model the conditional class probabilities for two classes C_0 and C_1 using a linear score followed by a sigmoid.

Parameters (including sizes)

Assume an input feature vector $\boldsymbol{x} \in \mathbb{R}^D$. The model parameters are:

$$\boldsymbol{w} \in \mathbb{R}^D \quad \text{(weight vector)}, \qquad b \in \mathbb{R} \quad \text{(bias / intercept)}.$$

(Equivalently, one may absorb b into $\boldsymbol{w}$ by padding $\boldsymbol{x}$ with a constant 1.)

Prediction (including sigmoid)

First compute the *linear part* (score / logit)

$$\bar{y}(\boldsymbol{x}; \boldsymbol{w}, b) = \boldsymbol{x}^\top \boldsymbol{w} + b.$$

Then apply the sigmoid function σ to obtain a probability:

$$y(\boldsymbol{x}; \boldsymbol{w}, b) = \sigma(\bar{y}(\boldsymbol{x}; \boldsymbol{w}, b)) = \sigma(\boldsymbol{x}^\top \boldsymbol{w} + b), \qquad \sigma(z) = \frac{1}{1 + e^{-z}}.$$

Thus the model outputs

$$p(C_1 \mid \boldsymbol{x}) = \sigma(\boldsymbol{x}^\top \boldsymbol{w} + b), \qquad p(C_0 \mid \boldsymbol{x}) = 1 - p(C_1 \mid \boldsymbol{x}).$$

A hard class prediction is typically obtained by thresholding the probability, e.g.

$$\hat{c} = \begin{cases} C_1, & p(C_1 \mid \boldsymbol{x}) \geq 0.5, \\ C_0, & \text{otherwise.} \end{cases}$$

In a binary logistic regression model, the prediction $\hat{y}$ is based on the probability that a given input x belongs to a particular class C_1, which is modeled using the logistic function σ. The parameters of the model include:

- Weight vector $w \in \mathbb{R}^D$, where D is the number of features.
- Bias $b \in \mathbb{R}$.

The logistic regression model makes predictions using the sigmoid function σ applied to the linear combination of the input features and the model weights:

$$\sigma(z) = \frac{1}{1+e^{-z}} = \frac{e^z}{e^z+1}$$

Given an input x, the linear part of the logistic regression model computes z as:

$$z = x^T w + b$$

The final prediction $\hat{y}$ is given by:

$$\hat{y}(x; w) = \sigma(z) = \sigma(x^T w + b)$$

The output of the linear part $x^T w + b$ can be interpreted as logits, which are the log odds of the probability that x belongs to class C_1 before the sigmoid transformation. Logits can take any real value, and transforming them through the sigmoid function maps them to the $(0, 1)$ interval, representing probabilities.

Q3.8 Write down an L^2-regularized minibatch SGD algorithm for training a binary logistic regression model, including the explicit formulas (i.e., formulas you would need to code it in `numpy`) of the loss function and its gradient. [20]

To train the logistic regression, we use MLE (maximum likelihood estimation). The loss for a minibatch $\mathcal{X} = \{(\vec{x}_1, t_1), (\vec{x}_2, t_2), \ldots, (\vec{x}_N, t_N)\}$ is given by

$$E(w) = \frac{1}{N} \sum_i -\log(p(C_{t_i}|x_i; w)) + \frac{\lambda}{2}\|w\|^2$$

Model and sigmoid

$$p(C_1 \mid \boldsymbol{x}; \boldsymbol{w}) = \sigma(\boldsymbol{x}^\top \boldsymbol{w}), \qquad \sigma(z) = \frac{1}{1+e^{-z}}.$$

Minibatch SGD algorithm

Algorithm 2 L^2-regularized minibatch SGD for binary logistic regression

Require: Input dataset ($X \in \mathbb{R}^{N \times D}$, $\boldsymbol{t} \in \{0,1\}^N$), learning rate $\alpha > 0$, regularization $\lambda \geq 0$
Ensure: Parameters $\boldsymbol{w} \in \mathbb{R}^D$

1: $\boldsymbol{w} \leftarrow \mathbf{0}$ **or** initialize $\boldsymbol{w}$ randomly
2: **while** until convergence (or patience runs out) **do**
3: Choose a minibatch of indices B
4: $\boldsymbol{g} \leftarrow \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla_w \left(-\log(p(C_{t_i}|x_i; w))\right) + \lambda w$ ▷ $\boldsymbol{g} = \frac{1}{|B|} X_B^\top (\sigma(X_B \boldsymbol{w}) - \boldsymbol{t}_B) + \lambda \boldsymbol{w}$
5: $\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha\, \boldsymbol{g}$
6: **end while**

The parameters are updated using the SGD algorithm as follows:

1. Initialize the weight vector $\vec{w} \leftarrow \vec{0}$ or randomly.
2. Repeat until convergence:

- Compute the gradient for a minibatch $\mathcal{B}$:

$$\vec{g} \leftarrow \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla_{\vec{w}} \left(-\log(p(C_{t_i} | \vec{x}_i; \vec{w})) \right) + \lambda \vec{w}$$

- Update the weights:

$$\vec{w} \leftarrow \vec{w} - \alpha \vec{g}$$

Logistic Regression Gradient

Consider the log-likelihood of logistic regression $\log p(t|\vec{x}; \vec{w})$. For brevity, we denote $\bar{y}(\vec{x}; \vec{w}) = \vec{x}^T \vec{w}$ simply as $\bar{y}$ in the following computation.

Given that for $t \sim \text{Ber}(\phi)$ we have $p(t) = \phi^t (1-\phi)^{1-t}$, we can rewrite the log-likelihood as:

$$\log p(t|\vec{x}; \vec{w}) = \log \sigma(\bar{y})^t (1 - \sigma(\bar{y}))^{1-t}$$

This simplifies to:

$$t \cdot \log(\sigma(\bar{y})) + (1-t) \cdot \log(1 - \sigma(\bar{y}))$$

The gradient of the logistic regression's likelihood with respect to the weights $\vec{w}$ is derived as follows:

$$\begin{aligned}
\nabla_{\vec{w}} - \log p(t|\vec{x}; \vec{w}) &= \nabla_{\vec{w}} \left(-t \log(\sigma(\hat{y})) - (1-t) \log(1 - \sigma(\hat{y})) \right) \\
&= \nabla_{\vec{w}} \left(-t \log(\sigma(\vec{x}^T \vec{w})) - (1-t) \log(1 - \sigma(\vec{x}^T \vec{w})) \right) \\
&= -t \cdot \frac{1}{\sigma(\hat{y})} \cdot \nabla_{\vec{w}} \sigma(\hat{y}) + (1-t) \cdot \frac{1}{1 - \sigma(\hat{y})} \cdot \nabla_{\vec{w}} (-\sigma(\hat{y})) \\
&= \left(-t + t\sigma(\hat{y}) + \sigma(\hat{y}) - t\sigma(\hat{y}) \right) \vec{x} \\
&= \left(\sigma(\vec{x}^T \vec{w}) - t \right) \vec{x}
\end{aligned}$$

where $\hat{y} = \vec{x}^T \vec{w}$ and the gradient of the sigmoid function $\sigma(x)$ is $\sigma(x)(1 - \sigma(x))$. The resulting gradient is used to update the weights in the SGD algorithm.

Therefor the gradient of the loss function is:

$$\nabla_{\vec{w}} E(\vec{w}) = \frac{1}{N} \sum_i \left(\sigma(\vec{x}_i^T \vec{w}) - t_i \right) \vec{x}_i + \lambda \vec{w}$$

Q3.9 Compare and contrast perceptron and logistic regression by discussing: (a) what each algorithm optimizes, (b) whether each provides probability estimates, (c) whether each is guaranteed to converge, and (d) the quality of solutions each finds. [10]

Consider binary classification with inputs $\boldsymbol{x} \in \mathbb{R}^D$ and labels $t \in \{-1, +1\}$ (perceptron) or $t \in \{0, 1\}$ (logistic regression).

(a) What each algorithm optimizes

Perceptron: The perceptron update is mistake-driven: when an example is misclassified ($t_i \, \boldsymbol{w}^\top \boldsymbol{x}_i \leq 0$), it updates $\boldsymbol{w} \leftarrow \boldsymbol{w} + t_i \boldsymbol{x}_i$. It can be viewed as minimizing the *perceptron loss*

$$\ell_{\text{perc}}(\boldsymbol{w}; \boldsymbol{x}_i, t_i) = \max\{0, -t_i \, \boldsymbol{w}^\top \boldsymbol{x}_i\},$$

using an online/stochastic update (it does not directly maximize a likelihood).

Logistic regression: Logistic regression explicitly optimizes a smooth convex objective: it minimizes the negative log-likelihood (cross-entropy) of a Bernoulli model (often with optional L^2 regularization), e.g.

$$\min_{\boldsymbol{w},b} -\sum_{i=1}^{N}\Big[t_i \log \sigma(\boldsymbol{w}^\top \boldsymbol{x}_i + b) + (1-t_i)\log\big(1-\sigma(\boldsymbol{w}^\top \boldsymbol{x}_i + b)\big)\Big] + \frac{\lambda}{2}\|\boldsymbol{w}\|^2.$$

(b) Probability estimates

Perceptron: Produces a hard decision $\text{sign}(\boldsymbol{w}^\top \boldsymbol{x}+b)$; it does *not* define calibrated probabilities.

Logistic regression: Produces probability estimates

$$p(C_1 \mid \boldsymbol{x}) = \sigma(\boldsymbol{w}^\top \boldsymbol{x} + b), \qquad \sigma(z) = \frac{1}{1+e^{-z}},$$

so it naturally outputs class probabilities.

(c) Convergence guarantee

Perceptron: Guaranteed to converge in a finite number of updates *only if* the data are linearly separable (perceptron convergence theorem). If the data are not separable, it may cycle and never converge.

Logistic regression: The (unregularized) logistic loss is convex, so gradient-based optimization converges to a global minimum in the optimization sense (up to numerical tolerance) under standard conditions (e.g. appropriate step sizes). However, if the data are perfectly separable, the MLE has no finite optimum (weights can grow without bound); L^2 regularization fixes this by making the objective strongly convex and ensuring a unique finite solution.

(d) Quality of solutions

Perceptron: When separable, it finds *some* separating hyperplane, but not necessarily the best one by margin or likelihood; the final solution depends on the order of examples and updates, and can have poor generalization.

Logistic regression: Finds the *global optimum* of the (regularized) log-likelihood / cross-entropy objective. This typically yields more stable solutions, better-calibrated outputs, and often better generalization than perceptron, especially on noisy or non-separable data.c

Q3.10 Explain in intuitive terms why we use the logarithm when working with likelihoods in machine learning. What are the computational and optimization advantages of using negative log-likelihood instead of directly maximizing the likelihood? [5]

Given i.i.d. data, the likelihood is a product of per-example probabilities,

$$p(X; w) = \prod_{i=1}^{N} p(x_i; w).$$

We use the logarithm because it turns this product into a sum,

$$\log p(X; w) = \sum_{i=1}^{N} \log p(x_i; w),$$

which has several advantages:

- **Numerical stability:** Products of many probabilities (each ≤ 1) can underflow to zero in floating-point arithmetic, especially for large N. Summing log-probabilities avoids underflow and is stable.
- **Computational simplicity:** Sums are cheaper and easier to manipulate than products; gradients also become sums over examples, which naturally supports minibatch SGD.
- **Optimization convenience:** $\log(\cdot)$ is strictly increasing, so maximizing likelihood and maximizing log-likelihood have the same maximizer. Minimizing the *negative* log-likelihood (NLL) is equivalent but fits the standard "minimize a loss" framework.
- **Better-behaved objectives:** For many common models (e.g. logistic regression), the NLL is smooth and convex in the parameters, making optimization more reliable than working with the raw product likelihood.

Part IV
Lecture 4

Q4.1 Define mean squared error and show how it can be derived using MLE. What assumptions do we make during such derivation? [10]

Mean Squared Error (MSE) is commonly used as a loss function for regression problems and can be derived from Maximum Likelihood Estimation (MLE) when we assume that the target variables, t, conditioned on the inputs, $\vec{x}$, are normally distributed with a mean equal to the output of the model, $y(\vec{x};\vec{w})$, and variance σ^2. Under this assumption, the probability distribution for t is given by $p(t|\vec{x};\vec{w}) = \mathcal{N}(t; y(\vec{x};\vec{w}), \sigma^2)$.

Applying MLE, we look for the parameters $\vec{w}$ that maximize the likelihood of the observed data, which is equivalent to minimizing the negative log-likelihood. This leads to the MSE as follows:

$$\begin{aligned}
\vec{w}_{\text{MLE}} &= \arg\max_{\vec{w}} p(\vec{t}|\vec{X};\vec{w}) = \arg\min_{\vec{w}} \sum_{i=1}^{N} -\log p(t_i|\vec{x}_i;\vec{w}) \\
&= \arg\min_{\vec{w}} -\sum_{i=1}^{N} \log\left(\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(t_i - y(\vec{x}_i;\vec{w}))^2}{2\sigma^2}\right)\right) \\
&= \arg\min_{\vec{w}} -N\log\left((2\pi\sigma^2)^{-\frac{1}{2}}\right) - \sum_{i=1}^{N} \frac{(t_i - y(\vec{x}_i;\vec{w}))^2}{2\sigma^2} \\
&= \arg\min_{\vec{w}} \frac{1}{2\sigma^2} \sum_{i=1}^{N} (y(\vec{x}_i;\vec{w}) - t_i)^2.
\end{aligned}$$

Ignoring the constant $\frac{1}{2\sigma^2}$, we obtain the familiar form of the MSE:

$$E(\vec{w}) = \frac{1}{N} \sum_{i=1}^{N} (y(\vec{x}_i;\vec{w}) - t_i)^2.$$

This derivation shows that when we assume a normal distribution for the model errors, the MLE approach naturally leads to the MSE as the loss function to be minimized.

Q4.2 Considering K-class logistic regression model, write down its parameters (including their size) and explain how we decide what classes the input data belong to (including the formula for the softmax function). [10]

To extend the binary logistic regression to a K-class case, we define the model parameters as a weight matrix $W \in \mathbb{R}^{D \times K}$, where D is the number of features and K is the number of classes. Each column $W_{*,i}$ corresponds to the weights associated with class i.

Predictions are made using the softmax function applied to the linear outputs, known as logits. For an input vector $\vec{x}$, the logits are given by $\vec{y}(\vec{x}; W) = W^T \vec{x}$, and the softmax function is defined as:

$$\text{softmax}(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

for each class i, where $\vec{z}$ is the vector of logits.

Therefore, the probability that $\vec{x}$ belongs to class i is:

$$p(C_i|\vec{x}; W) = \text{softmax}(\vec{y}(\vec{x}; W))_i = \frac{e^{\vec{x}^T W_{*,i}}}{\sum_{j=1}^{K} e^{\vec{x}^T W_{*,j}}}$$

The linear part of the model $\vec{x}^T W$ can be interpreted as logits because they represent the log odds before passing through the softmax function. The softmax function normalizes these log odds to probabilities that sum to one across all classes.

Training a K-class logistic regression model typically involves using the cross-entropy loss function, which for a given data point (x_i, t_i) is:

$$E(W) = -\sum_{i=1}^{N} \log p(C_{t_i}|\vec{x}_i; W)$$

This loss function is minimized using optimization algorithms such as minibatch stochastic gradient descent (SGD).

Q4.3 Explain the relationship between the sigmoid function and softmax. [5]

The softmax function is a generalization of the sigmoid function to the case where there are multiple classes. For binary classification ($K = 2$), the softmax function simplifies to the sigmoid function. Specifically, the sigmoid function is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

which is the probability of a single class (e.g., class 1 in binary classification).

The softmax function, which is used for multinomial logistic regression with K classes, is defined as:

$$\text{softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

for each class i. When $K = 2$, this reduces to:

$$\text{softmax}([x, 0]) = \frac{e^x}{e^x + e^0} = \frac{1}{1 + e^{-x}}$$

which is identical to the sigmoid function. Therefore, the softmax function can be seen as an extension of the sigmoid function from binary to multiclass classification, where the output for each class is the normalized exponential function of the logits, ensuring that the class probabilities sum to one.

The sigmoid function thus can be represented as a softmax function applied to a vector with two elements, where one element is the logit x and the other is zero. This connection shows the versatility of softmax as a multi-class sigmoid function.

Q4.4 Show that the softmax function is invariant towards constant shift. [5]

The main idea is that the term e^c gets canceled out.

$\text{softmax}(z_i + c) = \frac{e^{z_i+c}}{\sum_{j=1}^{n} e^{z_j+c}} = \frac{e^{z_i}e^{c}}{\sum_{j=1}^{n} e^{z_j}e^{c}} = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}} = \text{softmax}(z_i)$

Q4.5 Write down an L^2-regularized minibatch SGD algorithm for training a K-class logistic regression model, including the explicit formulas (i.e., formulas you would use to code it in `numpy`) of the loss function and its gradient. [20]

To train a K-class logistic regression model, we use the minibatch stochastic gradient descent (SGD) with L2 regularization. The loss function is the regularized negative log-likelihood, and the gradient takes into account the regularization term.

Minibatch SGD algorithm

Algorithm 3 L^2-regularized minibatch SGD for K-class logistic regression

1: **Input:** Input dataset ($X \in \mathbb{R}^{N\times D}$, $t \in \{0, 1, \ldots, K-1\}^N$), learning rate $\alpha \in \mathbb{R}^+$, regularization $\lambda \geq 0$
2: **Model:** Let w denote all parameters of the model (here $w = (W, b)$ with $W \in \mathbb{R}^{D\times K}$, $b \in \mathbb{R}^K$)
3: $w \leftarrow 0$ **or** initialize w randomly
4: **while** until convergence (or patience runs out) **do**
5: process a minibatch of examples $\mathcal{B}$
6: $g \leftarrow \nabla_w \mathcal{L}_{\mathcal{B}}(w)$ ▷ $\mathcal{L}_{\mathcal{B}}(w) = -\frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} \log(p(C_{t_i} \mid \boldsymbol{x}_i; w)) + \frac{\lambda}{2}\|W\|_F^2$
7: $w \leftarrow w - \alpha g$
8: **end while**

Loss Function

The regularized loss function for a minibatch $\mathcal{B}$ is:

$$E(\mathbf{W}) = -\frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} \log(p(C_{t_i}|\mathbf{x}_i; \mathbf{W})) + \frac{\lambda}{2}\|\mathbf{W}\|_F^2$$

where $\|\mathbf{W}\|_F^2$ is the Frobenius norm of $\mathbf{W}$, representing the L2 regularization term.

Gradient

The gradient of the loss function with L2 regularization is:

$$\nabla_{\mathbf{W}} E(\mathbf{W}) = -\frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} x_i (softmax(x_i^T \mathbf{W}) - 1_t)^T + \lambda \mathbf{W}$$

Note that 1_t is a one-hot vector with a 1 in the position corresponding to the target class t.

Q4.6 Prove that decision regions of a multiclass logistic regression are convex. [10]

To prove the convexity of decision regions in multiclass logistic regression, consider two points x_A and x_B in the same decision region R_k. The decision criterion for logistic regression is based on the linear functions $x^T W$, where W is the weight matrix. A point x is in region R_k if and only if

$$\hat{y}(x)_k = x^T W_k$$

is the largest among all class scores. For two points $x_A, x_B \in R_k$, and any $\lambda \in [0, 1]$, their convex combination $x = \lambda x_A + (1 - \lambda) x_B$ also satisfies

$$\hat{y}(x)_k = \lambda \hat{y}(x_A)_k + (1 - \lambda) \hat{y}(x_B)_k$$

Given that both $\hat{y}(x_A)_k$ and $\hat{y}(x_B)_k$ are the largest scores for their respective points, $\hat{y}(x)_k$ will also be the largest score for the convex combination, placing x in R_k. This holds for any convex combination of points in R_k, thus R_k is convex.

Q4.7 Considering a single-layer MLP with D input neurons, H hidden neurons, K output neurons, hidden activation f, and output activation a, list its parameters (including their size) and write down how the output is computed. [10]

A single-layer Multilayer Perceptron (MLP) with D input neurons, H hidden neurons, and K output neurons, uses the following parameters:

- Hidden layer weights $W^{(h)} \in \mathbb{R}^{D \times H}$
- Hidden layer biases $b^{(h)} \in \mathbb{R}^H$
- Output layer weights $W^{(y)} \in \mathbb{R}^{H \times K}$
- Output layer biases $b^{(y)} \in \mathbb{R}^K$

Algorithm 4 Forward pass of a 1-hidden-layer MLP

Require: Hidden activation $f(\cdot)$, output activation $a(\cdot)$
Require: Hidden parameters $W^{(h)} \in \mathbb{R}^{D \times H}$, $b^{(h)} \in \mathbb{R}^H$
Require: Output parameters $W^{(y)} \in \mathbb{R}^{H \times K}$, $b^{(y)} \in \mathbb{R}^K$

Single input $\boldsymbol{x} \in \mathbb{R}^D$:

$\boldsymbol{h} \leftarrow f(\boldsymbol{x}^\top W^{(h)} + b^{(h)})$ ▷ Hidden layer activations, $\in \mathbb{R}^H$
$\boldsymbol{y} \leftarrow a(\boldsymbol{h}^\top W^{(y)} + b^{(y)})$ ▷ Output predictions, $\in \mathbb{R}^K$

Batch input $X \in \mathbb{R}^{N \times D}$:

$\mathbf{1} \leftarrow (1, \ldots, 1)^\top \in \mathbb{R}^N$ ▷ all-ones column vector
$H \leftarrow f(XW^{(h)} + \mathbf{1}(b^{(h)})^\top)$ ▷ Batch hidden layer activations, $H \in \mathbb{R}^{N \times H}$
$Y \leftarrow a(HW^{(y)} + \mathbf{1}(b^{(y)})^\top)$ ▷ Batch output predictions, $Y \in \mathbb{R}^{N \times K}$

Q4.8 List the definitions of frequently used MLP output layer activations (the ones producing parameters of a Bernoulli distribution and a categorical distribution). Then write down three commonly used hidden layer activations (sigmoid, tanh, ReLU). Explain why identity is not a suitable activation for hidden layers. [10]

Output Layer Activations

- **Identity (Regression):** $\text{identity}(x) = x$
- **Sigmoid (Binary Classification):** $\sigma(x) = \frac{1}{1+e^{-x}}$
- **Softmax (K-class Classification):** $\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$

Hidden Layer Activations

- **Sigmoid:** $\sigma(x) = \frac{1}{1+e^{-x}}$
- **Tanh:** $\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} = 2\sigma(2x) - 1$
- **ReLU:** $\text{ReLU}(x) = \max(0, x)$

Why identity is not a suitable activation for hidden layers

If the hidden-layer activation is the identity $f(z) = z$, then a multilayer perceptron collapses to a single linear (affine) model, i.e., it cannot represent nonlinear functions.

Consider a 1-hidden-layer network:

$$\boldsymbol{h} = f\Big((W^{(h)})^\top \boldsymbol{x} + \boldsymbol{b}^{(h)}\Big) = (W^{(h)})^\top \boldsymbol{x} + \boldsymbol{b}^{(h)}.$$

With a linear output layer,

$$\boldsymbol{y} = (W^{(y)})^\top \boldsymbol{h} + \boldsymbol{b}^{(y)}.$$

Substituting $\boldsymbol{h}$ gives

$$\boldsymbol{y} = (W^{(y)})^\top \Big((W^{(h)})^\top \boldsymbol{x} + \boldsymbol{b}^{(h)}\Big) + \boldsymbol{b}^{(y)} = \underbrace{\big(W^{(h)} W^{(y)}\big)^\top}_{\text{single matrix}} \boldsymbol{x} + \underbrace{\Big((W^{(y)})^\top \boldsymbol{b}^{(h)} + \boldsymbol{b}^{(y)}\Big)}_{\text{single bias}}.$$

This is just an affine function of $\boldsymbol{x}$. More generally, a composition of affine maps is still affine, so adding more identity-activated hidden layers does not increase representational power.

Consequence. Using identity in hidden layers prevents the network from learning nonlinear representations and makes depth useless: the model is equivalent to a single-layer linear model (linear regression / linear classifier) in the original input space.

Q4.9 Explain the role of initialization in training MLPs. Why is it problematic to initialize all weights to zero? What is a common strategy for random initialization, and why does it typically scale with the input dimension? [10]

Role of initialization

Training an MLP with (mini)batch SGD is a nonconvex optimization problem, so the initial weights define the starting point of the optimization and strongly influence whether and how fast SGD makes progress. In particular, initialization determines the initial hidden features (the hidden layer can be viewed as "automatically constructed features").

Why initializing all weights to zero is problematic

If we initialize all weights in a layer to the same value (especially all zeros), then all hidden neurons in that layer are *symmetric*: they produce identical activations for every input and also receive identical gradients. Therefore, after each update they remain identical, meaning the network cannot learn diverse hidden features (it effectively behaves as if it had only one hidden unit repeated multiple times). Hence, the weights in an MLP must be initialized randomly to break this symmetry.

Common random initialization and why it scales with input dimension

A common strategy is:

- initialize biases to 0;
- initialize each weight matrix randomly; for a matrix mapping a layer of size M to a layer of size O, sample entries e.g. uniformly from a small interval such as

$$W_{ij} \sim \mathcal{U}\left[-\frac{1}{M}, \frac{1}{M}\right].$$

The scaling with the input dimension (fan-in) M is motivated by magnitude/variance control: a pre-activation is a sum of M terms, $z = \sum_{j=1}^{M} x_j w_j$, so its variance typically grows with M unless the weights get smaller as M increases. Choosing the weight scale to decrease with M keeps activations (and thus gradients) in a reasonable range, reducing the risk of saturation/exploding values; the exact range becomes especially important for deeper networks.

Q4.10 You have trained two models on the same dataset: (1) logistic regression and (2) a multilayer perceptron with one hidden layer of 100 neurons. The MLP achieves 95% training accuracy while logistic regression achieves 85%. However, both achieve 84% test accuracy. Interpret these results and explain what they suggest about the models and the data. [5]

Logistic regression is a *lower-capacity* (linear) model, while an MLP with a hidden layer of 100 neurons has *much higher capacity* and can represent more complex (nonlinear) decision boundaries.

Here, the MLP reaches 95% training accuracy but only 84% test accuracy, i.e. it has a large **generalization gap** ($95 - 84 = 11$ percentage points). This indicates **overfitting**: the MLP is fitting patterns specific to the training set (including noise or incidental correlations) that do not hold on unseen data.

Logistic regression achieves 85% training accuracy and 84% test accuracy, i.e. a very small gap. This suggests it is not strongly overfitting; rather, its limited capacity prevents it from fitting the training set much beyond the test performance.

Because *both* models achieve essentially the same test accuracy (84%), the results suggest that:

- the additional capacity of the MLP does *not* translate into better generalization on this dataset as trained;
- either (i) the true decision boundary is close to linear / the features are not rich enough to benefit from nonlinearity, or (ii) the dataset contains substantial noise / limited data so the achievable test accuracy is capped;
- improving the MLP would likely require stronger regularization (e.g. weight decay), early stopping, or other capacity control to reduce overfitting.

Q4.11 You are supposed to train an MLP for regression that has several numeric features as the input. How would you preprocess them? Specifically, would you use polynomial features? Explain your decision. [5]

Preprocessing of numeric features

For an MLP trained with (mini)batch SGD, it is important that input features have comparable scales; otherwise different features effectively require different learning rates and optimization becomes poorly conditioned. A common solution is to **normalize/standardize** each feature dimension using statistics computed on the *training* set and then apply the same transform to validation/test data.

Typical options (feature-wise, for feature j):

- **Standardization (zero mean, unit variance):**

$$x_{i,j}^{\text{stand}} = \frac{x_{i,j} - \mu^j}{\sigma^j}.$$

- **Min–max normalization:**

$$x_{i,j}^{\text{norm}} = \frac{x_{i,j} - \min_k x_{k,j}}{\max_k x_{k,j} - \min_k x_{k,j}}.$$

(Optionally, also handle missing values/outliers before scaling.)

Would I use polynomial features?

Usually **no**. Polynomial features are mainly a form of *feature engineering* used to increase the capacity of *linear* models, i.e., when the algorithm cannot represent nonlinear functions on its own. In contrast, an MLP already creates nonlinear features internally: the mapping into the hidden layer can be viewed as *automatically constructed features* (linear transform followed by a nonlinearity).

Adding polynomial expansions to an MLP typically:

- increases input dimensionality and training cost,
- increases effective capacity and can worsen overfitting,
- is redundant because nonlinear interactions can be learned by hidden units.

I would only consider polynomial features if I had strong domain knowledge about specific interactions and wanted to inject them explicitly (or if the model were intentionally constrained to be nearly linear).

Part V
Lecture 5

Q5.1 Considering a single-layer MLP with D input neurons, a ReLU hidden layer with H units and a softmax output layer with K units, write down the explicit formulas (i.e., formulas you would use to code it in `numpy`) for the forward pass through the MLP. [10]

Assuming an MLP with D input neurons, a ReLU hidden layer with H units, and a softmax output layer with K units, we compute the gradients of the loss L with respect to the weight

matrices $W^{(h)}, W^{(y)}$ and bias vectors $b^{(h)}, b^{(y)}$ given an input x, a target t, and using the negative log likelihood loss.

Let $x \in \mathbb{R}^D$ be the input vector, $h \in \mathbb{R}^H$ be the output of the hidden layer, and $y \in \mathbb{R}^K$ be the output of the network. The negative log likelihood loss for a correct class c is given by $L = -\log(y_c) = -\log(p(C|x))$.

Forward Pass:

$$\begin{aligned} h^{(in)} &= x^T W^{(h)} + b^{(h)} \\ h &= \mathrm{ReLU}(h^{(in)}) \\ y^{(in)} &= h^T W^{(y)} + b^{(y)} \\ y &= \mathrm{softmax}(y^{(in)}) \end{aligned}$$

Q5.2 Compute the partial derivative of $-\log \mathrm{softmax}(\boldsymbol{z})$ with respect to $\boldsymbol{z}$. Explain how this computation is used when training MLP. [20]

$$\begin{aligned} \frac{\partial}{\partial z_t}\Big(-\log \mathrm{softmax}(\boldsymbol{z})_t\Big) &= \frac{\partial}{\partial z_t}\left(-\log \frac{\exp z_t}{\sum_j \exp z_j}\right) \\ &= \frac{\partial}{\partial z_t}\left(-\log \exp z_t + \log \sum_j \exp z_j\right) \\ &= -\mathbf{1}_t + \frac{\exp(z_t)}{\sum_j \exp(z_j)} = \mathrm{softmax}(\boldsymbol{z})_t - \mathbf{1}_t. \end{aligned}$$

How this derivative is used when training an MLP (backprop intuition)

Assume the output layer of the MLP produces logits $\boldsymbol{z} \in \mathbb{R}^K$ and probabilities

$$\boldsymbol{y} = \mathrm{softmax}(\boldsymbol{z}), \qquad y_k = \frac{e^{z_k}}{\sum_j e^{z_j}}.$$

With a one-hot target $\boldsymbol{t}$ (true class t), the negative log-likelihood loss is

$$L = -\log y_t.$$

The key result used in training is the gradient w.r.t. logits (the "pre-activation" of the output layer):

$$\boxed{\frac{\partial L}{\partial z_k} = y_k - t_k}$$

(in particular, $\frac{\partial L}{\partial z_t} = y_t - 1$ and for $k \neq t$, $\frac{\partial L}{\partial z_k} = y_k$). This is exactly the error signal that starts the backward pass through the computation graph.

Intuition: boost the correct class, suppress the others. During gradient descent, we update $z_k \leftarrow z_k - \alpha \frac{\partial L}{\partial z_k}$:

- For the correct class t: $\frac{\partial L}{\partial z_t} = y_t - 1 < 0$ (unless $y_t = 1$), so

$$z_t \leftarrow z_t - \alpha(y_t - 1) = z_t + \alpha(1 - y_t),$$

  i.e., we *increase* the logit of the correct class.
- For each wrong class $k \neq t$: $\frac{\partial L}{\partial z_k} = y_k > 0$, so

$$z_k \leftarrow z_k - \alpha y_k,$$

  i.e., we *decrease* logits of incorrect classes.

Moreover, the suppression is *proportional to the current prediction*: wrong classes that the model assigns larger probability (y_k large) are pushed down more strongly. This matches the intuition that training "flips" the correct class up and pushes competing classes down according to how much they are predicted.

How it plugs into backpropagation in the MLP. Let the last affine layer be

$$\boldsymbol{z} = (W^{(y)})^\top \boldsymbol{h} + \boldsymbol{b}^{(y)},$$

where $\boldsymbol{h}$ are hidden activations. Define the output-layer error

$$\boldsymbol{\delta}^{(y)} \stackrel{\text{def}}{=} \frac{\partial L}{\partial \boldsymbol{z}} = \boldsymbol{y} - \boldsymbol{t}.$$

Then the gradients of the output-layer parameters are

$$\frac{\partial L}{\partial W^{(y)}} = \boldsymbol{h}\,(\boldsymbol{\delta}^{(y)})^\top, \qquad \frac{\partial L}{\partial \boldsymbol{b}^{(y)}} = \boldsymbol{\delta}^{(y)},$$

and the error is propagated to the hidden layer via

$$\frac{\partial L}{\partial \boldsymbol{h}} = W^{(y)} \boldsymbol{\delta}^{(y)}.$$

From there, backprop continues through the hidden activation f (e.g. ReLU) and earlier affine layers to obtain $\partial L / \partial W^{(h)}$ and $\partial L / \partial \boldsymbol{b}^{(h)}$.

[OUTDATED] Considering a single-layer MLP with D input neurons, a ReLU hidden layer with H units and a softmax output layer with K units, write down the explicit formulas of the gradient of all the MLP parameters (two weight matrices and two bias vectors), assuming input x, target t, and negative log likelihood loss. [20]

Assuming an MLP with D input neurons, a ReLU hidden layer with H units, and a softmax output layer with K units, we compute the gradients of the loss L with respect to the weight matrices $W^{(h)}, W^{(y)}$ and bias vectors $b^{(h)}, b^{(y)}$ given an input x, a target t, and using the negative log likelihood loss.

Let $x \in \mathbb{R}^D$ be the input vector, $h \in \mathbb{R}^H$ be the output of the hidden layer, and $y \in \mathbb{R}^K$ be the output of the network. The negative log likelihood loss for a correct class c is given by $L = -\log(y_c) = -\log(p(C|x))$.

Forward Pass:

$$
\begin{aligned}
h^{(in)} &= x^T W^{(h)} + b^{(h)} \\
h &= \text{ReLU}(h^{(in)}) \\
y^{(in)} &= h^T W^{(y)} + b^{(y)} \\
y &= \text{softmax}(y^{(in)})
\end{aligned}
$$

Backward Pass (Gradients):

$$
\begin{aligned}
\frac{\partial L}{\partial y_k} &= -\frac{t_k}{y_k} \\
\frac{\partial L}{\partial \mathbf{y}^{(in)}} &= \mathbf{y} - \mathbf{t} \quad (\text{since } \sum_k t_k = 1) \\
\frac{\partial L}{\partial \mathbf{W}^{(y)}} &= \mathbf{h} \left(\frac{\partial L}{\partial \mathbf{y}^{(in)}} \right)^\top \\
\frac{\partial L}{\partial \mathbf{b}^{(y)}} &= \frac{\partial L}{\partial \mathbf{y}^{(in)}} \\
\frac{\partial L}{\partial \mathbf{h}} &= \mathbf{W}^{(y)} \frac{\partial L}{\partial \mathbf{y}^{(in)}}^\top \\
\frac{\partial L}{\partial \mathbf{h}^{(in)}} &= \begin{cases} \frac{\partial L}{\partial \mathbf{h}} & \text{if } \mathbf{h}^{(in)} > 0 \\ 0 & \text{otherwise} \end{cases} \quad (\text{ReLU gradient}) \\
\frac{\partial L}{\partial \mathbf{W}^{(h)}} &= \mathbf{x} \left(\frac{\partial L}{\partial \mathbf{h}^{(in)}} \right)^\top \\
\frac{\partial L}{\partial \mathbf{b}^{(h)}} &= \frac{\partial L}{\partial \mathbf{h}^{(in)}}
\end{aligned}
$$

Gradient of Loss with respect to Output Probabilities y

$$
\frac{\partial L}{\partial \mathbf{y}} = -\frac{\mathbf{t}}{\mathbf{y}}
$$

Gradient of Output Probabilities with respect to Logits $\mathbf{y}^{(\mathbf{in})}$

The softmax function for a class k is given by $y_k = \frac{e^{y_k^{(\text{in})}}}{\sum_j e^{y_j^{(\text{in})}}}$. Its derivative with respect to the logits $y_i^{(\text{in})}$ is:

$$
\frac{\partial y_k}{\partial y_i^{(\text{in})}} = \begin{cases} y_k(1 - y_i) & \text{if } i = k, \\ -y_k y_i & \text{if } i \neq k. \end{cases}
$$

Chain Rule Application for Loss Gradient with respect to Logits

$$\begin{aligned}
\frac{\partial L}{\partial y_i^{(\text{in})}} &= \sum_k \frac{\partial L}{\partial y_k} \frac{\partial y_k}{\partial y_i^{(\text{in})}} \\
&= \frac{\partial L}{\partial y_i} \frac{\partial y_i}{\partial y_i^{(\text{in})}} + \sum_{k \neq i} \frac{\partial L}{\partial y_k} \frac{\partial y_k}{\partial y_i^{(\text{in})}} \\
&= -\frac{t_i}{y_i} y_i (1 - y_i) - \sum_{k \neq i} \frac{t_k}{y_k} (-y_k y_i) \\
&= -t_i + t_i y_i + \sum_{k \neq i} t_k y_i \\
&= -t_i + y_i \left(t_i + \sum_{k \neq i} t_k \right) \\
&= -t_i + y_i \sum_k t_k \\
&= y_i - t_i \quad (\text{since } \sum_k t_k = 1 \text{ for one-hot encoded targets})
\end{aligned}$$

Gradient of Loss with respect to Output Layer Weights $\mathbf{W}^{(y)}$

$$\begin{aligned}
\frac{\partial L}{\partial \mathbf{W}^{(y)}} &= \frac{\partial L}{\partial \mathbf{y}^{(\text{in})}} \frac{\partial \mathbf{y}^{(\text{in})}}{\partial \mathbf{W}^{(y)}} \\
&= (\mathbf{y} - \mathbf{t})^\top \mathbf{h}
\end{aligned}$$

Gradient of Loss with respect to Output Layer Biases $\mathbf{b}^{(y)}$

$$\frac{\partial L}{\partial \mathbf{b}^{(y)}} = \mathbf{y} - \mathbf{t}$$

Gradient of Loss with respect to Hidden Layer Outputs $\mathbf{h}$

$$\frac{\partial L}{\partial \mathbf{h}} = \mathbf{W}^{(y)} (\mathbf{y} - \mathbf{t})$$

Gradient of Loss with respect to Hidden Layer Pre-Activation $\mathbf{h}^{(\text{in})}$

$$\frac{\partial L}{\partial \mathbf{h}^{(\text{in})}} = \frac{\partial L}{\partial \mathbf{h}} \cdot \mathbb{I}(\mathbf{h}^{(\text{in})} > 0)$$

where $\mathbb{I}$ is the indicator function, yielding 1 for elements where the condition is true and 0 otherwise, which corresponds to the derivative of the ReLU activation function.

Gradient of Loss with respect to Hidden Layer Weights $\mathbf{W}^{(h)}$

$$\frac{\partial L}{\partial \mathbf{W}^{(h)}} = \mathbf{x}^\top \frac{\partial L}{\partial \mathbf{h}^{(\text{in})}}$$

Gradient of Loss with respect to Hidden Layer Biases $\mathbf{b}^{(h)}$

$$\frac{\partial L}{\partial \mathbf{b}^{(h)}} = \frac{\partial L}{\partial \mathbf{h}^{(\text{in})}}$$

Update Rules:

$$W^{(h)} = W^{(h)} - \alpha \frac{\partial L}{\partial W^{(h)}}$$
$$b^{(h)} = b^{(h)} - \alpha \frac{\partial L}{\partial b^{(h)}}$$
$$W^{(y)} = W^{(y)} - \alpha \frac{\partial L}{\partial W^{(y)}}$$
$$b^{(y)} = b^{(y)} - \alpha \frac{\partial L}{\partial b^{(y)}}$$

In these equations, α represents the learning rate, and the derivatives with respect to $h^{(in)}$ take into account the ReLU activation, which is zero for negative inputs and equal to the derivative of the loss with respect to h otherwise. The derivative with respect to $y^{(in)}$ is computed as the difference between the output probabilities y and the one-hot encoded target vector t.

Q5.3 Formulate the computation of MLP as a computation graph. Explain how such a graph can be used to compute the gradients of the parameters in the back-propagation algorithm. [10]

A Multi-Layer Perceptron (MLP) is a feedforward neural network that consists of an input layer, one or more hidden layers, and an output layer. The MLP processes an input through each layer by applying a series of transformations, including weighted sums and activation functions.

We can represent the computation performed by the MLP as a computation graph, where each node represents a variable or an operation, and edges represent dependencies between them.

Let's assume an MLP with one hidden layer for simplicity, though this extends to more layers. The forward pass involves the following steps:

1. Input to the first layer
2. Activation function
3. Input to the output layer
4. Output layer activation (optional)

The computation graph for this MLP can be visualized as a directed acyclic graph (DAG), where:

Nodes: Represent operations (like matrix multiplication, addition, activation functions) and variables (like weights, biases, activations). Edges: Represent dependencies between operations, such as the flow of data between layers or the relationship between weights and activations. Here's a simplified view of the graph:

Input x —-> (W1 * x + b1) —-> Activation f —-> (W2 * a1 + b2) —-> Output y

Not really sure how to explain the backpropagation part, basically you compute the gradients of the loss with respect to the parameters by applying the chain rule in reverse order through the graph. The gradients are computed layer by layer, starting from the output layer and moving backward to the input layer. The chain rule allows us to compute the gradients of the loss with respect to each parameter by multiplying the gradients of the loss with respect to the output of each node in the graph.

Q5.4 Explain the concept of dropout as a regularization technique for MLPs. How does it work during training versus at test time, and what is the intuition behind why it improves generalization? [10]

Concept. *Dropout* is a stochastic regularization method for multilayer perceptrons (MLPs) in which, during training, individual hidden units are randomly "dropped" (set to zero) with some probability. This reduces effective model capacity on each update and helps prevent overfitting.

How it works during training. Let $h \in \mathbb{R}^H$ be the activation vector of a hidden layer. Sample an elementwise mask

$$m \sim \text{Bernoulli}(p_{\text{keep}})^H,$$

and apply it:

$$\tilde{h} = m \odot h,$$

where $\odot$ denotes elementwise multiplication. Each mini-batch (or even each example) sees a different "thinned" network, so training effectively optimizes an ensemble of many subnetworks that share weights.

How it works at test time. At test time, dropout is disabled: we use the full network (no units dropped).

Intuition for improved generalization. Dropout improves generalization primarily by reducing variance and discouraging over-reliance on specific pathways:

- **Prevents co-adaptation:** neurons cannot rely on a fixed set of other neurons being present, so features must be useful in many contexts.
- **Robust, distributed representations:** information must be represented redundantly because any unit may be missing during training.
- **Approximate model averaging:** training with many random masks is akin to training a large ensemble of subnetworks and averaging them at test time, which typically generalizes better than a single overfit model.

Q5.5 Formulate Universal Approximation Theorem ('89) and explain in words what it says about multi-layer perceptron. [10]

Let $\phi(x) : \mathbb{R} \to \mathbb{R}$ be a nonconstant, bounded and nondecreasing continuous function (later also shown for $\phi = \text{ReLU}$, and more generally for many nonpolynomial activations).

For any $\epsilon > 0$ and any continuous function $f : [0,1]^D \to \mathbb{R}$, there exists $H \in \mathbb{N}$, $\mathbf{v} \in \mathbb{R}^H$, $\mathbf{b} \in \mathbb{R}^H$, and $\mathbf{W} \in \mathbb{R}^{D \times H}$ such that, defining the (one-hidden-layer) MLP

$$F(\mathbf{x}) = \mathbf{v}^T \phi(\mathbf{W}^T \mathbf{x} + \mathbf{b}) = \sum_{i=1}^{H} v_i \, \phi(\mathbf{x}^T \mathbf{W}_{*,i} + b_i),$$

(where ϕ is applied elementwise), we have for all $\mathbf{x} \in [0,1]^D$:

$$|F(\mathbf{x}) - f(\mathbf{x})| < \epsilon.$$

What it says (in words). The theorem states that a **single-hidden-layer** MLP with a suitable nonlinearity is a **universal function approximator**: by using *enough* hidden units H, it can approximate *any* continuous function on a compact domain (here $[0,1]^D$) arbitrarily well (to within any tolerance ϵ).

Why we use it / why it matters.

- **Expressive power guarantee:** it provides a theoretical justification that MLPs are not fundamentally limited in what mappings they can represent; in principle, they can represent very complex input–output relationships.
- **Existence, not construction:** it is an *existence* result—it guarantees that some parameters $(\mathbf{W}, \mathbf{b}, \mathbf{v})$ exist, but it does *not* guarantee that gradient-based training will find them, nor that the required H is practically small.
- **No generalization guarantee:** approximating f on the domain does not automatically imply good performance on finite data; regularization, architecture choices, and data still determine generalization.

Intuition behind the formula. The representation

$$F(\mathbf{x}) = \sum_{i=1}^{H} v_i \, \phi(\mathbf{x}^T \mathbf{W}_{*,i} + b_i)$$

can be viewed as a **linear combination of many nonlinear basis functions**. Each hidden unit implements a nonlinear feature $\phi(\mathbf{x}^T \mathbf{W}_{*,i} + b_i)$; by scaling and summing enough such features (choosing H large enough), the network can "tile" the input space and shape the output to match f up to error ϵ.

Q5.6 How do we search for a minimum of a function $f(\mathbf{x}) : \mathbb{R}^D \to \mathbb{R}$ subject to equality constraints $g_1(\mathbf{x}) = 0, \ldots, g_m(\mathbf{x}) = 0$? [10]

We search for a minimum of a function $f(\mathbf{x}) : \mathbb{R}^D \to \mathbb{R}$ subject to equality constraints $g_1(\mathbf{x}) = 0, \ldots, g_m(\mathbf{x}) = 0$ using the method of Lagrange multipliers. This involves finding a point $\mathbf{x} \in \mathbb{R}^D$ and a set of multipliers $\lambda_1, \ldots, \lambda_m \in \mathbb{R}$ such that the gradient of the Lagrangian function $\mathcal{L}(\mathbf{x}, \lambda)$ with respect to both $\mathbf{x}$ and λ is zero.

The Lagrangian is defined as:

$$\mathcal{L}(\mathbf{x}, \lambda) = f(\mathbf{x}) - \sum_{i=1}^{m} \lambda_i g_i(\mathbf{x}),$$

where $\nabla_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \lambda) = 0$ and $\nabla_{\lambda} \mathcal{L}(\mathbf{x}, \lambda) = 0$. This gives us a system of equations which, when solved, gives the points $\mathbf{x}$ that minimize $f(\mathbf{x})$ subject to the constraints.

Q5.7 Prove which categorical distribution with N classes has maximum entropy. [10]

We want to find a categorical distribution $\mathbf{p} = (p_1, \ldots, p_N)$ that maximizes entropy, subject to the constraints:

- $p_i \geq 0$ for all i,
- $\sum_{i=1}^{N} p_i = 1$.

The entropy $H(\mathbf{p})$ for a categorical distribution is given by:

$$H(\mathbf{p}) = -\sum_{i=1}^{N} p_i \log(p_i)$$

We form the Lagrangian $\mathcal{L}$ to include the equality constraint:

$$\mathcal{L}(\mathbf{p}, \lambda) = -\sum_{i=1}^{N} p_i \log(p_i) + \lambda \left(\sum_{i=1}^{N} p_i - 1 \right)$$

Taking the derivative of $\mathcal{L}$ with respect to p_i and setting it to zero:

$$0 = \frac{\partial \mathcal{L}}{\partial p_i} = -\log(p_i) - 1 + \lambda$$

Solving for p_i, we get:

$$p_i = e^{\lambda - 1}$$

Since all p_i must satisfy the equality constraint $\sum_{i=1}^{N} p_i = 1$, substituting p_i gives:

$$\sum_{i=1}^{N} e^{\lambda - 1} = 1$$

$$N e^{\lambda - 1} = 1$$

$$e^{\lambda - 1} = \frac{1}{N}$$

$$p_i = \frac{1}{N}$$

Therefore, each p_i is $\frac{1}{N}$, indicating that the distribution with maximum entropy is the uniform distribution.

Q5.8 Consider derivation of softmax using maximum entropy principle, assuming we have a dataset of N examples (x_i, t_i), $x_i \in \mathbb{R}^D$, $t_i \in \{1, 2, \ldots, K\}$. Formulate the three conditions we impose on the searched $\pi : \mathbb{R}^D \to \mathbb{R}^K$, and write down the Lagrangian to be minimized. [20]

Given a dataset of N examples (x_i, t_i) where $x_i \in \mathbb{R}^D$ and $t_i \in \{1, 2, \ldots, K\}$, we want to derive a softmax function using the maximum entropy principle. The softmax function $\pi(x)$ must satisfy the following conditions:

1. For all $x \in \mathbb{R}^D$ and each class k, the predicted probability $\pi(x)_k \geq 0$.
2. For each input x, the probabilities must sum up to 1: $\sum_{k=1}^{K} \pi(x)_k = 1$.
3. The expected value of the predicted distribution should match the empirical distribution: $\frac{1}{N} \sum_{i=1}^{N} \pi(x_i)_k = \frac{1}{N} \sum_{i=1}^{N} [t_i = k]$ for each class k.

The Lagrangian $\mathcal{L}$, incorporating these constraints with Lagrange multipliers λ and μ_k. We want to minimize $-\sum_{i=1}^{N} H(\pi(x_i))$ given

- for $1 \leq i \leq N$, $1 \leq k \leq K$: $\pi(x_i)_k \geq 0$,
- for $1 \leq i \leq N$: $\sum_{k=1}^{K} \pi(x_i)_k = 1$,
- for $1 \leq j \leq D$, $1 \leq k \leq K$: $\sum_{i=1}^{N} \pi(x_i)_k x_{i,j} = \sum_{i=1}^{N} [t_i = k] x_{i,j}$.

We therefore form a Lagrangian (ignoring the first inequality constraint):

$$\mathcal{L} = \sum_{i=1}^{N} \sum_{k=1}^{K} \pi(x_i)_k \log(\pi(x_i)_k) - \sum_{j=1}^{D} \sum_{k=1}^{K} \lambda_{j,k} \left(\sum_{i=1}^{N} \pi(x_i)_k x_{i,j} - [t_i = k] x_{i,j} \right) - \sum_{i=1}^{N} \beta_i \left(\sum_{k=1}^{K} \pi(x_i)_k - 1 \right).$$

Q5.9 Define precision (including true positives and others), recall, F_1 score, and F_β score (we stated several formulations for F_1 and F_β scores; any one of them will do). [10]

The confusion matrix is a table used to describe the performance of a classification model:

	Predicted Positive	**Predicted Negative**
Actual Positive	True Positives (TP)	False Negatives (FN)
Actual Negative	False Positives (FP)	True Negatives (TN)

Table 1: Confusion Matrix

Precision quantifies the number of correct positive predictions made. It is defined as:

$$\text{Precision} = \frac{\text{True Positives (TP)}}{\text{TP} + \text{False Positives (FP)}}$$

Recall measures the proportion of actual positives correctly identified. It is defined as:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{False Negatives (FN)}}$$

The F_1 score is the harmonic mean of precision and recall. It is defined as:

$$F_1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

The F_β score generalizes the F_1 score by weighing recall more heavily than precision. It is defined as:

$$F_\beta = (1 + \beta^2) \times \frac{\text{Precision} \times \text{Recall}}{(\beta^2 \times \text{Precision}) + \text{Recall}}$$

Q5.10 Explain the difference between micro-averaged and macro-averaged F_1 scores. Under what circumstances do we use them? [10]

Macro-averaged F_1. Compute the F_1 score *separately for each class* (one-vs-rest) and then average:

$$F_1^{\text{macro}} = \frac{1}{C} \sum_{c=1}^{C} F_{1,c}.$$

Each class has *equal weight*, so rare classes influence the score as much as frequent ones. Use it when performance on minority classes matters and you want a class-balanced view.

Micro-averaged F_1. Aggregate contributions over *all* classes first (sum TP/FP/FN across classes), then compute F_1:

$$P^{\text{micro}} = \frac{\sum_c \text{TP}_c}{\sum_c (\text{TP}_c + \text{FP}_c)}, \quad R^{\text{micro}} = \frac{\sum_c \text{TP}_c}{\sum_c (\text{TP}_c + \text{FN}_c)}, \quad F_1^{\text{micro}} = \frac{2P^{\text{micro}} R^{\text{micro}}}{P^{\text{micro}} + R^{\text{micro}}}.$$

This effectively weights classes by their support (number of examples), so majority classes dominate. Use it when overall instance-level performance is the priority, especially under strong class imbalance.

Q5.11 Explain (using examples) why accuracy is not a suitable metric for unbalanced target classes, e.g., for a diagnostic test for a contagious disease. [5]

Accuracy is defined as the ratio of correctly predicted observations to the total observations. In the context of unbalanced datasets, particularly in disease diagnosis where the disease prevalence is low, a model might predict "no disease" for all patients and still achieve high accuracy. For example:

$$\text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{Total Observations}}$$

Consider a dataset with 1000 individuals where only 10 have the disease. A model that predicts "no disease" for everyone would have an accuracy of:

$$\frac{0 + 990}{10 + 990} = \frac{990}{1000} = 99\%$$

Despite the high accuracy, the model fails to detect any true disease cases, demonstrating the inadequacy of accuracy as a performance metric in such scenarios. It is more informative to look at metrics such as precision, recall, and the F_1 score in cases of class imbalance.

Part VI
Lecture 6

Q6.1 Explain how is the TF-IDF weight of a given document-term pair computed. [5]

The TF-IDF weight of a document-term pair is given by:

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

where:

- $\text{TF}(t, d)$ is the term frequency, defined as the number of times term t appears in document d, normalized or not.
- $\text{IDF}(t, D)$ is the inverse document frequency, calculated as:

$$\text{IDF}(t, D) = \log \frac{N}{|\{d \in D : t \in d\}|(optionaly + 1))}$$

In this formula, N is the total number of documents in the corpus D, and $|\{d \in D : t \in d\}|$ is the number of documents where the term t appears (i.e., df_t, the document frequency of t).

The TF-IDF score increases with the number of times a word appears in the document but is offset by the frequency of the word in the corpus, which helps to adjust for the fact that some words appear more frequently in general.

Q6.2 What is Zipf's law? Explain how it can be used to provide intuitive justification for using the logarithm when computing IDF. [5]

Zipf's Law is an empirical rule that suggests that in many natural language datasets, the frequency of a word is inversely proportional to its rank in a frequency table. In simpler terms, a few words are very frequent, and many words are very rare. This means that the most frequent word in a text will appear approximately twice as often as the second most frequent word, three times as often as the third most frequent word, and so on.

Mathematically, Zipf's Law can be expressed as: $f(r) \propto \frac{1}{r^s}$, where $f(r)$ is the frequency of the word ranked r, r is the rank of the word, s is a parameter close to 1 (often approximated as 1 for natural language).

If Zipf's law holds, we can use it to justify the use of the logarithm when computing the IDF. The intuition is that the frequency of a word is inversely proportional to its rank, which means that the IDF should increase logarithmically with the rank of the word. This is because the logarithm is the inverse function of the exponential, and it helps to compress the range of values, making the IDF more manageable and interpretable. By taking the logarithm of the IDF, we can ensure that the IDF values are not skewed by the extreme frequency differences between words, aligning with the observed distribution of word frequencies in natural language datasets.

Q6.3 Define conditional entropy, mutual information, write down the relation between them, and finally prove that mutual information is zero if and only if the two random variables are independent (you do not need to prove statements about D_{KL}). [10]

Conditional entropy $H(Y|X)$ quantifies the expected value of the entropy of Y given that the value of X is known. It is defined as:

$$H(Y|X) = -\sum_{x \in X, y \in Y} P(x, y) \log P(y|x).$$

Mutual information $I(X;Y)$ measures the amount of information that one random variable contains about another random variable. It is defined as:

$$I(X;Y) = \sum_{x \in X, y \in Y} P(x, y) \log \frac{P(x, y)}{P(x)P(y)}.$$

The relationship between conditional entropy and mutual information can be expressed as:

$$I(X;Y) = H(Y) - H(Y|X).$$

This equation implies that mutual information is the reduction in uncertainty about Y due to the knowledge of X.

The mutual information is symmetrical, so

$$I(X;Y) = I(Y;X) = H(X) - H(X|Y) = H(Y) - H(Y|X).$$

Therefore

$$I(X;Y) = D_{KL}(P(X,Y)||P(X)P(Y))$$

Proof that $I(X;Y) = 0 \iff X \perp Y$. Using the definition,

$$I(X;Y) = \sum_{x,y} P(x,y) \log \frac{P(x,y)}{P(x)P(y)}.$$

Observe that this is exactly the KL-divergence between the joint distribution and the product of marginals:

$$I(X;Y) = D_{KL}\big(P(X,Y) \,\|\, P(X)P(Y)\big).$$

Now use the standard property of KL-divergence (no proof required here):

$$D_{KL}(P\|Q) \geq 0 \quad \text{and} \quad D_{KL}(P\|Q) = 0 \iff P = Q.$$

Therefore,

$$I(X;Y) = 0 \iff P(X,Y) = P(X)P(Y).$$

But $P(X,Y) = P(X)P(Y)$ is precisely the definition of independence of X and Y. Hence, $I(X;Y) = 0$ if and only if X and Y are independent.

Q6.4 Show that TF-IDF terms can be considered portions of suitable mutual information. [10]

Let $\mathcal{D}$ be a collection of documents and $\mathcal{T}$ a collection of terms. We can express the mutual information between a document d and a term t using their probabilities and the TF-IDF measure.

- The probability of selecting a document d uniformly at random from $\mathcal{D}$ is $P(d) = \frac{1}{|\mathcal{D}|}$.
- The information content of a document $I(d) = H(\mathcal{D}) = \log |\mathcal{D}|$.
- The probability of a term t occurring in a document d is $P(t|d) = \frac{|\{d \in \mathcal{D} : t \in d\}|}{|\mathcal{D}|}$.
- The information content of a term t in a document d is $I(t|d) = \log |\{d \in \mathcal{D} : t \in d\}|$.
- The difference in information content of a document with and without a term is the IDF: $I(d) - I(t|d) = \log |\mathcal{D}| - \log |\{d \in \mathcal{D} : t \in d\}| = \text{IDF}(t)$.

The mutual information $I(\mathcal{D};\mathcal{T})$ is calculated as:

$$I(\mathcal{D};\mathcal{T}) = \sum_{d,t \in d} P(d) \cdot P(t|d) \cdot (I(d) - I(t|d)).$$

Given the definitions of TF and IDF, we can write:

$$I(\mathcal{D};\mathcal{T}) = \frac{1}{|\mathcal{D}|} \sum_{d,t \in d} \text{TF}(t,d) \cdot \text{IDF}(t).$$

Thus, we can interpret the TF-IDF weight as a portion of the mutual information between the collection of documents $\mathcal{D}$ and the collection of terms $\mathcal{T}$, where each TF-IDF value corresponds to a "bit of information" for a document-term pair.

Q6.5 Explain the concept of word embedding in the context of MLP and how it relates to representation learning. [5]

Word embedding is a technique in representation learning where words from a vocabulary are associated with vectors of real numbers, effectively capturing their semantic meanings in a continuous vector space. Semantically similar words are positioned closely in this space.

In the realm of Multilayer Perceptrons (MLPs), these embeddings serve as the input layer. Each word is represented by a unique, learnable vector, rather than a high-dimensional, sparse one-hot vector. Throughout the training phase, the MLP fine-tunes these embeddings via back-propagation, based on the context in which words appear.

This approach is a key part of representation learning because it enables MLPs to internalize language subtleties directly from data, surpassing the need for manual feature engineering. It allows the network to capture complex syntactic and semantic word relationships, enhancing its performance on Natural Language Processing (NLP) tasks such as sentiment analysis, translation, and text categorization.

Word embeddings are the cornerstone of modern NLP models and are integrated into more advanced neural architectures like Recurrent Neural Networks (RNNs), Long Short-Term Memory networks (LSTMs), and Transformers, driving forward the state-of-the-art in various NLP applications.

Q6.6 Describe the skip-gram model trained using negative sampling. What is it used for? What are the input and output of the algorithm? Use an equation to describe the loss function. [10]

The skip-gram model aims to learn word embeddings that predict the context words given a target word. For a given word in the vocabulary, the model outputs a probability distribution over all words to be the 'context' words.

The Skip-gram model with negative sampling (SGNS) enhances training efficiency by altering the objective function. Instead of predicting the presence of context words among all words in the vocabulary, SGNS focuses on distinguishing the actual context words from a number of randomly sampled 'negative' words.

What it is used for. Skip-gram with negative sampling (Word2Vec) is a proxy task for *learning word embeddings*: it learns vector representations in which words that occur in similar contexts have similar vectors, so the embeddings can be reused as features in downstream NLP tasks.

Model idea (skip-gram). Given a *center* (input) word w in a sentence, the model tries to predict words in its context window c. It uses two embedding tables:

$$E \in \mathbb{R}^{|V| \times d} \quad \text{(input embeddings } e_w\text{)}, \qquad W \in \mathbb{R}^{|V| \times d} \quad \text{(output embeddings } v_c\text{)}.$$

Instead of an expensive softmax over the whole vocabulary, negative sampling turns training into binary classification of word pairs (w, c).

Input and output.

- **Input:** a center word w (typically as an index / one-hot; embedding lookup gives e_w).
- **Output during training:** for a given candidate context word c, a probability that (w, c) is a *real* co-occurrence:

$$P(\text{real} = 1 \mid w, c) = \sigma(e_w^\top v_c).$$

- **Output after training:** the learned word embeddings (typically the rows of E; in Word2Vec the output table is usually discarded).

Negative sampling training objective. For each observed positive pair (w, c_i) (where c_i is in the context window of w), sample K negative words $c_1, \ldots, c_K$ that are *not* in the window. The per-example loss is

$$\mathcal{L}(w, c_i, \{c_j\}_{j=1}^K) = -\log \sigma(e_w^\top v_{c_i}) - \sum_{j=1}^{K} \log\big(1 - \sigma(e_w^\top v_{c_j})\big),$$

(equivalently, the second term can be written as $-\sum_{j=1}^{K} \log \sigma(-e_w^\top v_{c_j})$). Typical values are small K (e.g. $K \approx 5$) for efficiency.

Example (what skip-gram predicts). Consider the sentence: *"the cat sat on the mat"* and a context window of size 2. If the center word is $w = \texttt{sat}$, then the true context words might be

$$c^+ \in \{\texttt{cat},\ \texttt{on}\} \quad \text{(and with a larger window also } \texttt{the}, \texttt{the}).$$

Skip-gram constructs positive training pairs such as

$$(\texttt{sat}, \texttt{cat}),\ (\texttt{sat}, \texttt{on}).$$

Example (negative sampling). For the positive pair $(w, c^+) = (\texttt{sat}, \texttt{cat})$, sample K negative words from a noise distribution (e.g. unigram$^{3/4}$), for instance with $K = 3$:

$$\{c_1^-, c_2^-, c_3^-\} = \{\texttt{banana},\ \texttt{river},\ \texttt{green}\}.$$

The model then solves $K+1$ binary classification subproblems:

$\sigma(e_{\texttt{sat}}^\top v_{\texttt{cat}})$ should be close to 1, $\quad \sigma(e_{\texttt{sat}}^\top v_{\texttt{banana}})$, $\sigma(e_{\texttt{sat}}^\top v_{\texttt{river}})$, $\sigma(e_{\texttt{sat}}^\top v_{\texttt{green}})$ should be close to 0.

So a single update increases the dot product for the positive pair and decreases it for the sampled negatives.

What exactly are the inputs/outputs in this example?

- **Input to the algorithm (data):** a corpus of tokenized sentences (and chosen window size, K, and a negative-sampling distribution).
- **Input to one training step:** a center word index $w = \texttt{sat}$, one positive context word $c^+ = \texttt{cat}$, and K sampled negatives $\{c_j^-\}$.
- **Output of one training step:** probabilities

  $$p^+ = \sigma(e_w^\top v_{c^+}), \qquad p_j^- = \sigma(e_w^\top v_{c_j^-}),$$

  and a scalar loss value used for backpropagation.
- **Final output after training:** the learned embedding matrix E (word vectors for all vocabulary items; often W is not kept).

Q6.7 Explain why the skip-gram model uses negative sampling instead of softmax. [5]

Skip-gram with softmax requires computing

$$p(c \mid w) = \frac{\exp(\mathbf{e}_w^\top \mathbf{v}_c)}{\sum_{c' \in V} \exp(\mathbf{e}_w^\top \mathbf{v}_{c'})},$$

so every update needs the normalization sum over the whole vocabulary V. Since $|V|$ can be very large (e.g. 10^5–10^6 word forms), this is computationally expensive.

Negative sampling avoids the full softmax by turning the problem into *binary classification* of word pairs: given a target word w and a candidate context word c, we model their co-occurrence probability independently as logistic regression

$$P(c \mid w) \approx \sigma(\mathbf{e}_w^\top \mathbf{v}_c).$$

For each observed (positive) pair (w, c) we sample K "negative" context words $c_1, \ldots, c_K$ that are not in the window, and optimize the (per-pair) loss

$$L = -\log \sigma(\mathbf{e}_w^\top \mathbf{v}_c) \; - \; \sum_{j=1}^{K} \log\bigl(1 - \sigma(\mathbf{e}_w^\top \mathbf{v}_{c_j})\bigr).$$

This reduces the cost of one update from $\mathcal{O}(|V|)$ (softmax) to $\mathcal{O}(K)$ with $K \ll |V|$ (typically $K \approx 5$), making training feasible on large corpora.

Q6.8 How would you proceed to train a part-of-speech tagger (i.e., you want to assign each word with its part of speech) if you only could use pre-trained word embeddings and MLP classifier? [5]

What we are doing (POS tagging). *Part-of-speech (POS) tagging* means: for every word in a sentence, we assign a grammatical label such as `NOUN`, `VERB`, `ADJ`, `ADV`, `PRON`, etc. Example:

"I/PRON can/VERB fish/VERB" vs. "a/DET can/NOUN"

The correct tag often depends on the *context* (neighboring words).

- **Use labeled data:** take sentences where each word already has a gold POS tag.
- **Convert words to vectors:** for each word w_t, look up its *pre-trained* embedding vector e_t. Keep these embeddings *fixed* (do not update them).
- **Add context (important for POS):** build an input vector from a small window of embeddings:
  $$x_t = [e_{t-1}; e_t; e_{t+1}]$$
  (or use a larger window; use a special `PAD` vector at sentence boundaries).
- **Train an MLP classifier:** feed x_t into an MLP and use a final softmax to predict the POS tag of the center word. Train only the MLP parameters using cross-entropy over all tokens.
- **Predict:** at test time, do the same embedding lookup + window, run the MLP, and output the most likely tag.

Part VII
Lecture 7

Q7.1 Describe k-nearest neighbors prediction, both for regression and classification. Define L_p norm and describe uniform, inverse, and softmax weighting. [10]

Regression

For regression, k-NN predicts the target value by a weighted average of the targets of the k nearest neighbors:

$$t = \frac{\sum_i w_i \cdot t_i}{\sum_j w_j}$$

Classification

For classification, k-NN uses voting among the k nearest neighbors. For uniform weights:

$$\text{class} = \text{mode}\{t_1, t_2, \ldots, t_k\}$$

With non-uniform weights, the predicted class maximizes the weighted sum of targets:

$$\text{class} = \arg\max \sum_i w_i \cdot t_{i,k}$$

Lp-norm

The L_p-norm is defined as:

$$\|x - y\|_p = \left(\sum_{i=1}^{D} |x_i - y_i|^p \right)^{1/p}$$

Weighting Methods

- Uniform: $w_i = 1$
- Inverse: $w_i = \frac{1}{\text{distance}(x, x_i)}$
- Softmax: $w_i = \frac{\exp(-\text{distance}(x, x_i))}{\sum_j \exp(-\text{distance}(x, x_j))}$

Q7.2 Show that L^2-regularization can be obtained from a suitable prior by Bayesian inference (from the MAP estimate). [10]

Assuming a Gaussian prior for model parameters $\mathbf{w}$ with zero mean and variance σ^2, the prior distribution is given as $p(\mathbf{w}_i) = \mathcal{N}(\mathbf{w}_i; 0, \sigma^2)$. Consequently, the prior over all weights $\mathbf{w}$ is $p(\mathbf{w}) = \prod_{i=1}^{N} \mathcal{N}(\mathbf{w}_i; 0, \sigma^2) = \mathcal{N}(\mathbf{w}; 0, \sigma^2 \mathbf{I})$. The maximum a posteriori (MAP) estimation is then:

$$\begin{aligned}\mathbf{w}_{\text{MAP}} &= \arg\max_{\mathbf{w}} p(\mathbf{X}|\mathbf{w})p(\mathbf{w}) \\ &= \arg\max_{\mathbf{w}} \prod_{i=1}^{N} p(\mathbf{x}_i|\mathbf{w})p(\mathbf{w}) \\ &= \arg\min_{\mathbf{w}} \sum_{i=1}^{N} \left(-\log p(\mathbf{x}_i|\mathbf{w}) - \log p(\mathbf{w})\right).\end{aligned}$$

Incorporating the Gaussian prior probability, we get the L2-regularized objective:

$$\mathbf{w}_{\text{MAP}} = \arg\min_{\mathbf{w}} \left[\sum_{i=1}^{N} -\log p(\mathbf{x}_i|\mathbf{w}) + \frac{D}{2}\log(2\pi\sigma^2) + \frac{\|\mathbf{w}\|^2}{2\sigma^2}\right],$$

which is the L2-regularization term.

Q7.3 Write down how $p(C_k|x)$ is approximated in a Naive Bayes classifier, explicitly state the Naive Bayes assumption, and show how is the prediction performed. [10]

The Naive Bayes classifier approximates the conditional probability $p(C_k|x)$ using Bayes' theorem and the naive independence assumption. This assumption states that all features x_d are independent given the class C_k. Therefore, the joint probability of the feature vector x given the class C_k can be expressed as the product of individual probabilities:

$$p(x \mid C_k) = \prod_{d=1}^{D} p(x_d \mid C_k).$$

Using Bayes' theorem, the posterior probability for class C_k given the feature vector x is then:

$$p(C_k \mid x) = \frac{p(x \mid C_k)p(C_k)}{p(x)},$$

where $p(x)$ is the evidence term, typically ignored during prediction as it remains constant across all classes.

The prediction for a new sample x is performed by choosing the class C_k that maximizes this posterior probability:

$$\hat{C} = \arg\max_{k} p(C_k \mid x) = \arg\max_{k} \left(\prod_{d=1}^{D} p(x_d \mid C_k)\right) p(C_k).$$

This approach allows for efficient computation and prediction in high-dimensional feature spaces.

Q7.4 Considering a Gaussian naive Bayes, describe how are $p(x_d|C_k)$ modeled (what distribution and which parameters does it have) and how we estimate it during fitting. [10]

In Gaussian Naive Bayes, the conditional probability $p(x_d \mid C_k)$ for a continuous feature x_d given a class C_k is modeled by a normal distribution:

$$p(x_d \mid C_k) = \mathcal{N}(x_d \mid \mu_{d,k}, \sigma^2_{d,k}).$$

The parameters $\mu_{d,k}$ and $\sigma^2_{d,k}$ of this distribution are estimated from the training data using maximum likelihood estimation (MLE). For each feature d and class k, the MLE of the mean $\mu_{d,k}$ is computed as:

$$\mu_{d,k} = \frac{1}{N_k} \sum_{i=1}^{N_k} x_{i,d},$$

where $x_{i,d}$ is the i-th training sample of feature d that belongs to class C_k, and N_k is the number of samples in class C_k.

The variance $\sigma^2_{d,k}$ is estimated as:

$$\sigma^2_{d,k} = \frac{1}{N_k} \sum_{i=1}^{N_k} (x_{i,d} - \mu_{d,k})^2.$$

In practice, to avoid the issue of zero variance, a smoothing term α is often added to the variance estimate:

$$\sigma^2_{d,k} = \frac{1}{N_k + \alpha} \sum_{i=1}^{N_k} (x_{i,d} - \mu_{d,k})^2 + \alpha.$$

The smoothing term α is a hyperparameter that can be tuned using cross-validation.

Q7.5 Considering a Bernoulli naive Bayes, describe how are $p(x_d|C_k)$ modeled (what distribution and which parameters does it have) and how we estimate it during fitting. [10]

In Bernoulli Naive Bayes, the probability of a binary feature x_d given a class C_k, denoted as $p(x_d|C_k)$, is modeled using a Bernoulli distribution with parameter $p_{d,k}$. This parameter represents the probability of feature d being present in a sample of class C_k.

$$p(x_d \mid C_k) = p_{d,k}^{x_d} \cdot (1 - p_{d,k})^{(1-x_d)}.$$

The likelihood of class C_k given the feature vector x is then:

$$p(C_k \mid x) \propto \left(\prod_{d=1}^{D} p_{d,k}^{x_d} \cdot (1 - p_{d,k})^{(1-x_d)} \right) p(C_k),$$

where D is the number of binary features.

Taking the logarithm, we obtain:

$$\log p(C_k \mid x) + c = \log p(C_k) + \sum_d \left(x_d \log \frac{p_{d,k}}{1 - p_{d,k}} + \log(1 - p_{d,k}) \right) = b_k + x^T w_k,$$

where c is a constant that does not depend on C_k and is not needed for prediction.

The prediction is made by:

$$\arg\max_k \log p(C_k \mid x) = \arg\max_k b_k + x^T w_k.$$

The parameter $p_{d,k}$ is estimated during training as the relative frequency of the feature d in samples of class C_k:

$$p_{d,k} = \frac{1}{N_k} \sum_{i=1}^{N_k} x_{i,d},$$

where $x_{i,d}$ is the presence or absence of feature d in the i-th sample, and N_k is the total number of samples in class C_k.

To prevent zero probabilities and to handle unseen features in the training data, smoothing is applied. The smoothed estimate for $p_{d,k}$ with Laplace smoothing is:

$$p_{d,k} = \frac{\sum_{i=1}^{N_k} x_{i,d} + \alpha}{N_k + 2\alpha},$$

where α is a smoothing parameter, typically set to 1.

Q7.6 What measures can we take to prevent numeric instabilities in the Naive Bayes classifier, particularly if the probability density is too high in Gaussian Naive Bayes and there are zero probabilities in Bernoulli Naive Bayes? [10]

General numerical stability (all Naive Bayes variants). Instead of multiplying many probabilities (which underflows/overflows), compute posteriors in the *log-domain*:

$$\log p(C_k \mid \mathbf{x}) = \log p(C_k) + \sum_{d=1}^{D} \log p(x_d \mid C_k) + \text{const},$$

and predict

$$\hat{k} = \arg\max_k \left[\log p(C_k) + \sum_{d=1}^{D} \log p(x_d \mid C_k) \right].$$

(Optionally, if normalized probabilities are needed, use a log-sum-exp normalization.)

Gaussian Naive Bayes: avoid too sharp Gaussians (too high density). In Gaussian NB, very small estimated variances $\sigma_{d,k}^2$ make the normal density extremely peaked, which can cause numerical issues. A standard fix is *variance smoothing*:

$$\sigma_{d,k}^2 \leftarrow \sigma_{d,k}^2 + \alpha,$$

for a small $\alpha > 0$ (often chosen relative to the overall feature variances), which prevents $\sigma_{d,k}^2$ from becoming too small and stabilizes the likelihood computation.

Bernoulli Naive Bayes: avoid zero/one probabilities. If a binary feature is always 1 (or always 0) in class C_k, the MLE gives $p_{d,k} = 1$ (or 0), which makes some test examples get probability 0 and yields $\log 0$. Use *Laplace/additive smoothing* (a pseudo-count $\alpha > 0$):

$$p_{d,k} = \frac{n_{d,k} + \alpha}{N_k + 2\alpha},$$

where $n_{d,k}$ is the number of training examples in class k with feature $d = 1$, and N_k is the number of training examples in class k. This guarantees $0 < p_{d,k} < 1$ and removes zero probabilities.

Q7.7 What is the difference between discriminative and (classical) generative models? [5]

Discriminative models. Discriminative models learn *the decision rule directly*, i.e. they model the conditional distribution

$$p(y \mid \mathbf{x})$$

(or an equivalent scoring function for predicting y from $\mathbf{x}$). They focus on separating classes and do not need to model how the features $\mathbf{x}$ are generated.

(Classical) generative models. Classical generative models learn a model of how data are produced: they model the class prior and the class-conditional distribution,

$$p(y) \quad \text{and} \quad p(\mathbf{x} \mid y),$$

so that the posterior used for classification is obtained via Bayes' rule:

$$p(y \mid \mathbf{x}) \propto p(y)\, p(\mathbf{x} \mid y).$$

Because they model $p(\mathbf{x} \mid y)$, they can (in principle) also *generate* or score samples $\mathbf{x}$ given a class y.

Part VIII
Lecture 8

Q8.1 Prove that independent discrete random variables are uncorrelated. [10]

Given two discrete random variables X and Y, they are said to be independent if the joint probability distribution can be expressed as the product of their marginal distributions:

$$P(X = x, Y = y) = P(X = x)P(Y = y) \quad \text{for all } x \text{ and } y.$$

The covariance of X and Y is defined as:

$$\text{Cov}(X, Y) = \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])].$$

For independent variables, the expectation of the product is the product of the expectations:

$$\begin{aligned}\text{Cov}(X, Y) &= \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])] \\ &= \mathbb{E}[XY] - \mathbb{E}[X]\mathbb{E}[Y] - \mathbb{E}[X]\mathbb{E}[Y] + \mathbb{E}[X]\mathbb{E}[Y] \\ &= \mathbb{E}[XY] - \mathbb{E}[X]\mathbb{E}[Y].\end{aligned}$$

Since X and Y are independent:

$$\mathbb{E}[XY] = \mathbb{E}[X]\mathbb{E}[Y].$$

Substituting this into the covariance formula gives:

$$\text{Cov}(X, Y) = \mathbb{E}[X]\mathbb{E}[Y] - \mathbb{E}[X]\mathbb{E}[Y] = 0.$$

Therefore, if X and Y are independent, their covariance is zero, implying that they are uncorrelated.

Q8.2 Give an example of two random variables that are dependent but uncorrelated. [5]

Let $X \sim \text{Unif}[-1, 1]$ and define $Y = |X|$.

Dependent. Y is completely determined by X, so X and Y cannot be independent.

Uncorrelated.

$$\text{Cov}(X, Y) = \mathbb{E}[XY] - \mathbb{E}[X]\mathbb{E}[Y].$$

By symmetry, $\mathbb{E}[X] = 0$. Also,

$$\mathbb{E}[XY] = \mathbb{E}[X|X|] = \int_{-1}^{1} x|x| \cdot \frac{1}{2}\, dx = 0,$$

because $x|x|$ is an odd function on $[-1, 1]$. Hence $\text{Cov}(X, Y) = 0$, so they are uncorrelated, yet dependent.

Q8.3 Write down the definition of covariance and Pearson correlation coefficient ρ, including its range. [10]

The covariance between two random variables X and Y is a measure of the joint variability of X and Y. It is defined as:

$$\text{Cov}(X, Y) = \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])]$$

where $\mathbb{E}$ denotes the expected value.

The Pearson correlation coefficient, denoted as ρ or r, is defined as:

$$\rho = \frac{\text{Cov}(X, Y)}{\sqrt{\text{Var}(X)\text{Var}(Y)}}$$

$$r = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2 \sum_i (y_i - \bar{y})^2}}$$

where:

- ρ is used when the full expectation is computed (population Pearson correlation coefficient);
- r is used when estimating the coefficient from data (sample Pearson correlation coefficient);
- $\bar{x}$ and $\bar{y}$ are sample estimates of the respective means.

The range of the Pearson correlation coefficient is from -1 to 1, inclusive. A value of 1 implies a perfect positive linear relationship between variables, -1 implies a perfect negative linear relationship, and 0 implies no linear relationship.

Q8.4 Explain how are the Spearman's rank correlation coefficient and the Kendall rank correlation coefficient computed (no need to describe the Pearson correlation coefficient). [10]

Assume we have paired observations $\{(x_i, y_i)\}_{i=1}^n$.

Spearman's rank correlation coefficient ρ

- Replace values by their **ranks**: $R_i = \text{rank}(x_i)$ and $S_i = \text{rank}(y_i)$ (ties are typically handled by assigning the average rank).
- Compute Pearson correlation **on the ranks**:

$$\rho_{\text{Spearman}} = \frac{\sum_{i=1}^{n}(R_i - \bar{R})(S_i - \bar{S})}{\sqrt{\sum_{i=1}^{n}(R_i - \bar{R})^2}\sqrt{\sum_{i=1}^{n}(S_i - \bar{S})^2}}.$$

- If there are **no ties**, an equivalent shortcut is

$$\rho_{\text{Spearman}} = 1 - \frac{6\sum_{i=1}^{n} d_i^2}{n(n^2-1)}, \qquad d_i = R_i - S_i.$$

Kendall rank correlation coefficient τ

- Consider all unordered pairs (i, j) with $i < j$.
- A pair is **concordant** if $(x_i - x_j)(y_i - y_j) > 0$ (they move in the same direction), and **discordant** if $(x_i - x_j)(y_i - y_j) < 0$ (they move in opposite directions).
- Let C be the number of concordant pairs and D the number of discordant pairs. Then

$$\tau = \frac{C - D}{\binom{n}{2}} = \frac{1}{\binom{n}{2}} \sum_{i<j} \text{sign}(x_j - x_i)\,\text{sign}(y_j - y_i).$$

- With ties, τ as defined above has a smaller attainable range; tie-corrected variants exist.

Both coefficients range from -1 to 1. A coefficient of 1 implies a perfect agreement, -1 implies perfect disagreement, and 0 implies the absence of association.

Q8.5 Describe setups or tasks where a correlation coefficient might be a good evaluation metric. [5]

A correlation coefficient is a good evaluation metric when we care about whether the model outputs *track* the target values (or preserve their ordering), rather than matching the exact scale.

- **Learning to rank (e.g., document retrieval):** we care about the *ordering* of items, not the absolute scores. Rank correlations such as Spearman's ρ or Kendall's τ are appropriate.
- **Similarity evaluation (embeddings):** evaluate word/sentence embeddings by correlating embedding similarities (e.g. cosine similarity) with human similarity ratings for word/sentence pairs (often Pearson or Spearman).
- **Inter-annotator agreement for continuous labels:** when annotators provide real-valued scores (e.g. rating sentiment strength), correlation between annotators indicates consistency/reliability.
- **Validating automatic metrics against human judgment:** for subjective tasks (e.g. machine translation quality, grammar checking), we often judge an automatic metric by how strongly it correlates with human ratings.

Q8.6 Describe under what circumstance correlation can be used to assess validity of evaluation metrics. Name examples of tasks. What data do you need besides the model predictions and the targets? [10]

When correlation is used (metric validity / meta-evaluation). Correlation is used to assess the *validity* of an evaluation metric when:

- the task quality is at least partly **subjective** or **multi-correct** (there are many acceptable outputs), so comparing only to a single reference/target is imperfect; and
- we want to know whether an **automatic metric** (computed from predictions and references) agrees with **human judgments** of quality.

In this setting, a metric is considered more valid if its scores are strongly correlated with human scores (or if it induces a similar ranking of systems).

Examples of tasks.

- **Machine translation** (validating BLEU/chrF/COMET etc. against human adequacy/fluency ratings).
- **Text summarization** (validating ROUGE/BERTScore etc. against human quality, coherence, faithfulness).
- **Image captioning / text generation** (validating CIDEr/SPICE and similar against human ratings).
- **Dialogue / open-ended generation** (validating automatic metrics against human preference or quality scores).

What extra data you need (besides predictions and targets). To compute such a correlation you need **human evaluation data** aligned with the same items/systems, e.g.:

- **Human quality scores** per output (continuous ratings) *or* **human rankings / pairwise preferences**.
- Often **outputs from multiple systems/models** (or multiple checkpoints) to create enough variability for a meaningful correlation, measured either at:
  - **system-level** (one score per system) or
  - **segment-level** (one score per example/sentence).

Then compute the correlation between automatic metric scores and human judgments (Pearson for linear agreement, Spearman/Kendall for rank agreement).

Q8.7 Define Mean Reciprocal Rank (MRR) and explain for what tasks it is used. Describe a scenario where you would prefer MRR over Spearman/Kendall correlation. [10]

Definition. Given a set of queries $\{q_i\}_{i=1}^{N}$ and, for each query q_i, a ranked list of items returned by a system, let rank_i be the position (1 = best) of the *first relevant* item for query i. The *reciprocal rank* for query i is

$$\text{RR}_i = \frac{1}{\text{rank}_i},$$

and the *Mean Reciprocal Rank* is

$$\text{MRR} = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{\text{rank}_i}.$$

(If a query has no relevant item retrieved, one typically uses $\text{RR}_i = 0$.)

What tasks it is used for. MRR is used in **ranking / retrieval** tasks where each query is expected to have one (or a small number of) correct/relevant answers and we care about **how early** the first relevant item appears, e.g.:

- information retrieval / search (first relevant document),
- question answering (first correct answer passage),
- recommendation / candidate ranking (first relevant item),
- entity linking / synonym or definition retrieval.

When prefer MRR over Spearman/Kendall. Prefer MRR when the evaluation is **query-wise** and relevance is **binary** (relevant / not relevant), and the goal is to optimize **top-of-the-list success**. Example scenario: a QA system returns a ranked list of candidate answers and the user will look only at the top few. MRR directly rewards placing the *first correct* answer at rank 1 (score 1), rank 2 (score 1/2), etc.

In contrast, Spearman/Kendall measure **agreement between two rankings** (e.g. metric vs. human ranking, or predicted vs. true ordering), which is not the primary objective when we simply want the first relevant item as high as possible.

Q8.8 Define Cohen's κ and explain what it is used for when preparing data for machine learning. [10]

Cohen's kappa coefficient (κ) is a statistical measure used to evaluate the inter-annotator agreement for qualitative (categorical) items. It is defined as:

$$\kappa = \frac{p_o - p_e}{1 - p_e}$$

where p_o is the relative observed agreement among raters, and p_e is the hypothetical probability of chance agreement.

This metric is utilized in machine learning to assess the consistency of annotations provided by different human experts. It serves multiple purposes:

- **Data Reliability:** Ensures that the data labels used for training machine learning models are consistent and reliable.
- **Annotator Performance:** Helps in evaluating the performance of annotators and can be used to filter out unreliable annotations.
- **Cultural Insights:** Low values of κ might indicate cultural differences or subjectivity in the data, providing insights into potential biases.
- **Model Benchmarking:** Sets a benchmark for machine learning performance, as achieving high accuracy beyond IAA is often unrealistic and might indicate overfitting or data leakage.

By quantifying the agreement level, Cohen's kappa allows for more informed decisions in the data curation process, ultimately leading to the development of more robust machine learning models.

Q8.9 Explain the relationship between inter-annotator agreement and expected model performance. Why is it suspicious if a model achieves performance significantly above the inter-annotator agreement? What does this suggest about the model and the data? [10]

Relationship (IAA as a performance ceiling). *Inter-annotator agreement (IAA)* measures how consistently humans assign labels/ratings to the same items, i.e., how well-defined the task is and how reliable the labels are. If annotators often disagree, the targets contain irreducible ambiguity/noise, so a supervised model trained on these targets cannot (in general) achieve arbitrarily high test performance: **IAA sets a natural upper bound (ceiling) on attainable ML performance**.

Why performance well above IAA is suspicious. If a model scores *significantly* above IAA, it is unlikely to be genuine "super-human" ability; rather, it often indicates the model is exploiting something other than the intended signal. The slides explicitly note that **performance over IAA is suspicious and is more likely overfitting to the way the data is curated than super-human performance**.

What it suggests about the model and the data. Performance $\gg$ IAA typically suggests issues such as:

- **Dataset artifacts / shortcuts:** the model learns spurious cues correlated with the label (annotation patterns, formatting, metadata, template effects), not the underlying phenomenon the task intends to measure.
- **Data leakage or contamination:** overlap/near-duplicates between train and test, or other leakage that makes the test set easier than it should be.
- **Evaluation mismatch / target bias:** the model may be evaluated against a *single* annotator's labels and can "learn that annotator"; since IAA measures agreement between humans, matching one annotator can exceed human-human agreement without truly solving the task.
- **IAA not representative:** IAA might have been computed on a harder subset, different annotator pool, or with a different protocol/metric.

Practical implication. Use IAA to identify confusing items/unreliable annotators and to understand the realistic headroom for models; if model performance exceeds IAA, investigate curation/leakage/artifacts before claiming a breakthrough.

Q8.9 [OUTDATED] Assuming you have collected data for classification by letting people annotate data instances. How do you estimate a reasonable range for classifier performance? [5]

To estimate a reasonable range for classifier performance, you can use several methods. Cross-validation is a popular approach, where you split your data into k subsets and train/test the model k times, averaging the performance. Bootstrap sampling is another method that repeatedly samples data with replacement, training on one set and testing on the out-of-bag instances, which helps estimate the model's variability. The holdout method involves splitting the data into a training and a test set, training on one and testing on the other, then repeating with different splits. You can calculate confidence intervals around your performance metric (like accuracy) from multiple runs to understand variability. Finally, examining bias-variance trade-offs with

learning curves helps assess if your model is underfitting or overfitting, which can further refine your estimate of performance.

Part IX
Lecture 9

Q9.1 Considering an averaging ensemble of M models, prove the relation between the average mean squared error of the ensemble and the average error of the individual models, assuming the model errors have zero means and are uncorrelated. Use a formula to explain what uncorrelated errors mean in this context. [20]

Let $y_i(x)$ be the prediction of model i for an input x with true target t, and $\varepsilon_i(x)$ be the error of model i such that $y_i(x) = t + \varepsilon_i(x)$. The mean squared error (MSE) of model i is:

$$\mathbb{E}\left[(y_i(x) - t)^2\right] = \mathbb{E}\left[\varepsilon_i^2(x)\right].$$

For an ensemble of M models, the ensemble prediction is the average of individual predictions:

$$y_{\text{ensemble}}(x) = \frac{1}{M}\sum_{i=1}^{M} y_i(x).$$

The MSE of the ensemble is then:

$$\mathbb{E}\left[(y_{\text{ensemble}}(x) - t)^2\right] = \mathbb{E}\left[\left(\frac{1}{M}\sum_{i=1}^{M}\varepsilon_i(x)\right)^2\right].$$

Assuming that errors $\varepsilon_i(x)$ are uncorrelated and have zero means, we have:

$$\mathbb{E}\left[\varepsilon_i(x)\varepsilon_j(x)\right] = 0 \quad \text{for} \quad i \neq j.$$

Therefore, the MSE of the ensemble simplifies to:

$$\mathbb{E}\left[\left(\frac{1}{M}\sum_{i=1}^{M}\varepsilon_i(x)\right)^2\right] = \frac{1}{M^2}\mathbb{E}\left[\left(\sum_{i=1}^{M}\varepsilon_i(x)\right)^2\right] + \frac{1}{M^2}\sum_{i,j}\mathbb{E}\left[\varepsilon_i(x)\varepsilon_j(x)\right] = \frac{1}{M}\mathbb{E}\left[\frac{1}{M}\sum_{i}\varepsilon_i^2(x)\right],$$

Hence, the average MSE of the ensemble is $\frac{1}{M}$ times the average MSE of the individual models.

Q9.2 Explain knowledge distillation: what it is used for, describe how it is done. What is the loss function? How does it differ from standard training? [10]

What it is and what it is used for. *Knowledge distillation* trains a smaller/faster **student** model to mimic a stronger **teacher** model (often a large model or an ensemble). It is used when the teacher is too slow/large for deployment, but we want to keep most of its performance.

How it is done (procedure).

1. Train (or choose) a high-quality teacher model.
2. Run the teacher on the training data (and optionally extra unlabeled data) to obtain the **full output distribution** for each input:

$$p_{\text{teacher}}(y \mid x).$$

3. Train the student on the same inputs so that its output distribution matches the teacher's:

$$p_{\text{student}}(y \mid x) \approx p_{\text{teacher}}(y \mid x).$$

Loss function. A standard distillation objective is the cross-entropy between the teacher and student distributions:

$$\mathcal{L}_{\text{distill}} = \sum_{(x,\cdot)\in\mathcal{D}} H(p_{\text{teacher}}(\cdot \mid x),\, p_{\text{student}}(\cdot \mid x)) = -\sum_{(x,\cdot)\in\mathcal{D}} \sum_{y} p_{\text{teacher}}(y \mid x)\, \log p_{\text{student}}(y \mid x).$$

(Equivalently, this is minimizing a KL divergence up to an additive constant.)

How it differs from standard training. In standard supervised training, the target is typically a **one-hot** label distribution $\delta_{y=y^*}$, so the loss is

$$\mathcal{L}_{\text{standard}} = -\sum_{(x,y^*)\in\mathcal{D}} \log p_{\text{student}}(y^* \mid x).$$

In distillation, the target is the teacher's **soft** distribution $p_{\text{teacher}}(\cdot \mid x)$, which provides richer supervision (e.g., it tells the student not only the top class, but also which other classes are plausible), making it easier for the smaller model to learn the teacher's behavior.

Q9.3 Describe the difference between voting (hard voting) and averaging (soft voting) in classification ensembles. Assuming, you have classication into three classes, give an example of classifier outputs where hard voting and soft voting differ. [10]

Hard voting (majority vote). Each base classifier outputs a *single class label* (its arg max class). The ensemble prediction is the class with the most votes:

$$\hat{y} = \arg\max_{c\in\{1,2,3\}} \sum_{m=1}^{M} \mathbb{I}\left[\hat{y}^{(m)} = c\right].$$

Soft voting (probability averaging). Each base classifier outputs a *probability vector* over classes. The ensemble averages these probabilities (and then takes arg max):

$$\hat{y} = \arg\max_{c\in\{1,2,3\}} \frac{1}{M} \sum_{m=1}^{M} p^{(m)}(y = c \mid x).$$

Soft voting uses the *confidence* information; hard voting discards it.

Example where they differ (3 classes). Suppose we have $M = 3$ classifiers with probability outputs:

$$p^{(1)} = (0.34,\ 0.33,\ 0.33) \ \Rightarrow\ \hat{y}^{(1)} = 1,$$
$$p^{(2)} = (0.34,\ 0.33,\ 0.33) \ \Rightarrow\ \hat{y}^{(2)} = 1,$$
$$p^{(3)} = (0.01,\ 0.99,\ 0.00) \ \Rightarrow\ \hat{y}^{(3)} = 2.$$

Hard voting: votes are (2 for class 1, 1 for class 2), so the ensemble predicts class 1.
Soft voting: average the probabilities:

$$\bar{p} = \frac{1}{3}\Big((0.34, 0.33, 0.33) + (0.34, 0.33, 0.33) + (0.01, 0.99, 0.00)\Big) = (0.23,\ 0.55,\ 0.22),$$

so the ensemble predicts class 2.

Hence, hard voting chooses class 1 (majority labels) while soft voting chooses class 2 (high-confidence classifier dominates).

Q9.4 List and explain three common heuristics used to control the growth of decision trees. Explain what problem it helps prevent and why. [10]

Decision trees can keep splitting until leaves are (nearly) pure, which often leads to **overfitting** (high variance): the tree starts fitting noise and dataset-specific quirks. The following heuristics *pre-prune* the tree by limiting its complexity:

- **Maximum tree depth:** do not split nodes deeper than a chosen depth $d_{\max}$. This limits how many sequential rules the tree can form and prevents very specific, fragile decision paths.
- **Minimum examples to split:** only split a node if it contains at least $n_{\min}$ training examples. This prevents creating splits (and leaves) supported by very few samples, which are statistically unreliable and typically reflect noise.
- **Maximum number of leaf nodes:** keep splitting only until the tree has at most $L_{\max}$ leaves. This directly caps the number of regions the input space is partitioned into, controlling model capacity similarly to limiting depth.

What problem this prevents and why. These heuristics prevent **overfitting** by restricting the tree's size/complexity. A fully grown tree can achieve very low training error by memorizing the training set, but such detailed rules generalize poorly. By stopping growth early, we reduce variance (increase stability) at the cost of a small increase in bias, which typically improves test performance.

Q9.5 In a regression decision tree, state what values are kept in internal nodes, define the squared error criterion and describe how is a leaf split during training (without discussing splitting constraints). [10]

Assume training data $X \in \mathbb{R}^{N \times D}$ and targets $t \in \mathbb{R}^N$. For any node T, denote by I_T the set of training-example indices that fall into T.

What is stored in internal nodes. An internal node stores a *split rule*: a selected feature (dimension) $d \in \{1, \dots, D\}$ and a split value s, which route an example $i \in I_T$ to

$$T_L : \ x_{i,d} \leq s \qquad \text{or} \qquad T_R : \ x_{i,d} > s,$$

(i.e., the node stores (d, s) and pointers to children T_L, T_R).

Prediction in a leaf. A leaf T predicts the average target value of the examples inside it:

$$t_T = \frac{1}{|I_T|} \sum_{i \in I_T} t_i.$$

Squared error criterion. The (non-averaged) squared error criterion for a node T is

$$c_{\text{SE}}(T) \ = \ \sum_{i \in I_T} \big(t_i - t_T\big)^2,$$

which is proportional to the variance in the node times $|I_T|$.

How a leaf is split during training. To split a leaf T, try candidate splits by iterating over

- a feature d (loop over $d = 1, \dots, D$),
- a split value s (loop over candidate values, typically unique values of the feature in the node),

which induces child index sets

$$I_{T_L}(d, s) = \{i \in I_T : x_{i,d} \leq s\}, \qquad I_{T_R}(d, s) = \{i \in I_T : x_{i,d} > s\}.$$

For each candidate, compute the post-split criterion

$$c_{\text{SE}}(T_L) + c_{\text{SE}}(T_R)$$

(or equivalently the criterion difference)

$$\Delta(d, s) = \big(c_{\text{SE}}(T_L) + c_{\text{SE}}(T_R)\big) - c_{\text{SE}}(T),$$

and choose (d, s) that minimizes $\Delta(d, s)$ (i.e., decreases the criterion the most). Then replace the leaf T by an internal node with this split and create the two new leaves T_L, T_R, each predicting its own mean t^{T_L}, t^{T_R}.

Q9.6 Explain the CART algorithm for constructing a decision tree. Explain the relationship between the loss function that is optimized during the decision tree construction and the splitting criterion that is during the node splitting. [10]

CART (Classification and Regression Trees): tree construction. We are given training data $X \in \mathbb{R}^{N \times D}$ and targets t (real-valued for regression, categorical for classification). A decision tree partitions the training set into leaf regions. For a node T, let I_T denote the set of training indices that fall into T.

1. **Initialize:** start with a single leaf T_{root} containing all examples $I_{T_{\text{root}}} = \{1, \dots, N\}$.

2. **Leaf prediction:** each leaf T predicts the best constant parameter for that leaf (regression: a scalar t^T; classification: a distribution p_T).

3. **Greedy splitting loop:** repeatedly choose a leaf T and split it into two children T_L, T_R using a binary rule "feature d and split value s":

$$I_{T_L}(d,s) = \{i \in I_T : x_{i,d} \le s\}, \qquad I_{T_R}(d,s) = \{i \in I_T : x_{i,d} > s\}.$$

   Evaluate candidate splits (d, s) and pick the one that improves the criterion the most (see below), then replace T by an internal node storing (d, s) and create leaves T_L, T_R.

Loss function ⇒ node criterion and splitting criterion. Fix a tree structure (i.e., a fixed partition into leaves). Training can be viewed as minimizing a loss over leaf parameters:

$$\min_{\{\theta^T\}} \sum_{\text{leaves } T} \sum_{i \in I_T} \ell(\theta^T, t_i),$$

where θ^T is the leaf parameter (regression: $\theta^T = t^T$; classification: $\theta^T = p_T$).

For each leaf T, define the *minimum reachable loss* in that leaf:

$$c_T \stackrel{\text{def}}{=} \min_{\theta} \sum_{i \in I_T} \ell(\theta, t_i).$$

Then the minimum reachable loss of the whole tree equals $\sum_{\text{leaves } T} c_T$.

Key relationship: when we split a leaf T into T_L, T_R, the best achievable total loss changes from c_T to $c_{T_L} + c_{T_R}$. Therefore the *splitting criterion* is exactly the increase/decrease in the minimum reachable loss:

$$\Delta(d,s) \;=\; c_{T_L} + c_{T_R} - c_T.$$

CART chooses the split (d, s) that *minimizes* $\Delta(d, s)$, i.e., yields the largest decrease in the minimum reachable loss.

Concrete criteria (examples).

- **Regression:** with squared loss, the optimal leaf prediction is $t^T = \frac{1}{|I_T|} \sum_{i \in I_T} t_i$ and

$$c_T \equiv c_{\text{SE}}(T) = \sum_{i \in I_T} (t_i - t^T)^2.$$

- **Classification:** with NLL loss, the optimal leaf distribution is the empirical distribution $p_T(k)$, giving an entropy-based node loss

$$c_T \equiv c_{\text{entropy}}(T) = -|I_T| \sum_{k:\, p_T(k) \neq 0} p_T(k) \log p_T(k),$$

  and a commonly used alternative impurity is the Gini criterion

$$c_T \equiv c_{\text{Gini}}(T) = |I_T| \sum_k p_T(k)\big(1 - p_T(k)\big).$$

In all cases, the split decision is greedy and is driven by minimizing $c_{T_L} + c_{T_R} - c_T$, which is directly derived from the underlying loss being minimized.

Q9.7 In a binary classification decision tree, state what values are kept in internal nodes, define the Gini index and describe how is a node split during training (without discussing splitting constraints). [10]

Assume training data $X \in \mathbb{R}^{N \times D}$ and binary targets $t_i \in \{0, 1\}$. For any node T, let I_T be the set of training-example indices belonging to T.

What is stored in internal nodes. Each internal node stores a *split*: a chosen feature (dimension) $d \in \{1, \ldots, D\}$ and a split value s, which routes an example $i \in I_T$ to

$$T_L : \; x_{i,d} \leq s \qquad \text{or} \qquad T_R : \; x_{i,d} > s,$$

(i.e., the node stores (d, s) and pointers to the children T_L, T_R).

Class distribution in a node. Let $n_T(0)$ and $n_T(1)$ be the counts of labels 0 and 1 in T, and define the empirical class probabilities

$$p_T(k) = \frac{n_T(k)}{|I_T|}, \qquad k \in \{0, 1\}.$$

A leaf typically predicts the most frequent class in the leaf (equivalently $\arg\max_k p_T(k)$).

Gini index (Gini impurity). The Gini criterion for a node T is

$$c_{\text{Gini}}(T) \stackrel{\text{def}}{=} |I_T| \sum_{k \in \{0,1\}} p_T(k)\big(1 - p_T(k)\big).$$

For binary classification this simplifies to

$$c_{\text{Gini}}(T) = |I_T|\, p_T(1)\big(1 - p_T(1)\big) \quad (\text{since } p_T(0) = 1 - p_T(1)).$$

How a node is split during training. To split a leaf node T, CART tries candidate splits by looping over

- feature $d = 1, \ldots, D$,
- split value s (typically all unique values of feature d among examples in I_T),

creating the child index sets

$$I_{T_L}(d, s) = \{i \in I_T : x_{i,d} \leq s\}, \qquad I_{T_R}(d, s) = \{i \in I_T : x_{i,d} > s\}.$$

For each candidate, compute the post-split criterion and its difference:

$$c_{\text{Gini}}(T_L) + c_{\text{Gini}}(T_R) \qquad \text{or} \qquad \Delta(d, s) = c_{\text{Gini}}(T_L) + c_{\text{Gini}}(T_R) - c_{\text{Gini}}(T).$$

Choose (d, s) that minimizes $\Delta(d, s)$ (equivalently, decreases the criterion the most), store (d, s) in T, and create the two new leaves T_L, T_R with their own empirical probabilities p_{T_L}, p_{T_R}.

Q9.8 In a K-class classification decision tree, state what values are kept in internal nodes, define the entropy criterion and describe how is a node split during training (without discussing splitting constraints). [10]

In a K-class classification decision tree, the internal nodes hold the decision rules, typically a feature and a threshold that partitions the dataset into subsets.

The entropy criterion for a node T, often used as a measure of impurity or disorder within the node, is defined as:

$$C_{\text{entropy}}(T) = -|I_T| \sum_{\substack{k=1 \\ p_T(k)\neq 0}}^{K} p_T(k) \log p_T(k),$$

where $p_T(k)$ represents the proportion of class k instances within node T, and $|I_T|$ is the number of instances in node T.

The process of splitting a node during training involves:

1. For each feature, calculate the potential splits and the resulting entropy.
2. The split that results in the largest decrease in entropy (highest information gain) is chosen.
3. This process is recursively applied to each new node until the stopping criteria are met.

Q9.9 For binary classification, derive the Gini index from a squared error loss. [20]

Consider a binary classification setting with a set of training examples belonging to a leaf node T. Let $n_T(0)$ denote the number of examples with target value 0, $n_T(1)$ the number of examples with target value 1, and p_T the proportion of examples with target value 1 in T, i.e., $p_T = \frac{n_T(1)}{n_T(0)+n_T(1)}$.

The squared error loss $L(p)$ for a prediction p is defined as:

$$L(p) = \sum_{i\in I_T} (p - t_i)^2,$$

where t_i is the target value for the i-th example.

Minimizing the squared error loss, we find that the optimal prediction p is the average target value in T, i.e., $p = p_T$. The loss for this prediction is:

$$L(p_T) = \sum_{i\in I_T} (p_T - t_i)^2 = n_T(0)(p_T - 0)^2 + n_T(1)(p_T - 1)^2.$$

Expanding the terms, we get:

$$\begin{aligned} &= \frac{n_T(0)n_T(1)^2}{(n_T(0) + n_T(1))^2} + \frac{n_T(1)n_T(0)^2}{(n_T(0) + n_T(1))^2} \\ &= \frac{(n_T(1)) + n_T(0))n_T(0)n_T(1)}{(n_T(0) + n_T(1))(n_T(0) + n_T(1))} \\ &= (n_T(0) + n_T(1))(1 - p_T)p_T = |T| \cdot p_T(1 - p_T). \end{aligned} \tag{1}$$

which is proportional to the Gini impurity measure $G(T) = 2p_T(1 - p_T)$ for a binary classification.

Q9.10 For K-class classification, derive the entropy criterion from a non-averaged NLL loss. [20]

Given a set of training examples I_T corresponding to a leaf node T in a decision tree for K-class classification, let $n_T(k)$ denote the count of examples in T with target class k. The probability of class k in T is given by $p_T(k) = \frac{n_T(k)}{|I_T|}$.

The non-averaged negative log-likelihood loss for a distribution p over K classes is defined as:

$$L(p) = \sum_{i \in I_T} -\log p_{t_i},$$

where p_{t_i} is the predicted probability of the true class t_i for the i-th example.

To minimize the NLL loss, we take its derivative with respect to p_k and set it to zero, subject to the constraint $\sum_k p_k = 1$. This yields the minimizing condition $p_k = p_T(k)$.

The value of the loss with respect to p_T then simplifies to:

$$L(p_T) = \sum_{i \in I_T} -\log p_{t_i} = -\sum_{k:p_T(k)\neq 0} n_T(k) \log p_T(k).$$

Using the definition of entropy $H(p_T)$ for the distribution p_T, we have:

$$H(p_T) = -\sum_{k:p_T(k)\neq 0} p_T(k) \log p_T(k),$$

which implies that the NLL loss is equal to the size of I_T times the entropy of p_T:

$$L(p_T) = |I_T| \cdot H(p_T).$$

This concludes the derivation, showing that minimizing the non-averaged NLL loss is equivalent to minimizing the entropy of the predicted class distribution within a leaf node of a decision tree.

Q9.11 Describe how is a random forest trained (including bagging and a random subset of features) and how is prediction performed for regression and classification. [10]

A random forest is an ensemble learning method that operates by constructing a multitude of decision trees during training and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees.

Training

The training process involves the following steps:

1. **Bootstrap Aggregating (Bagging):** For each tree, a bootstrap sample is drawn from the training data. This means that for a training set of size N, M samples are drawn with replacement to form a training set for the tree.

2. **Random Feature Selection:** When splitting nodes during the construction of the trees, instead of searching for the best split among all features, a random subset of features is selected, and the best split is found within this subset. This introduces diversity among the trees and is key to the success of random forests.

3. **Tree Construction:** Decision trees are constructed to the maximum depth without pruning. Each tree is grown on a different bootstrap sample of the data, and at each node, a different random subset of features is considered for splitting.

4. **Ensemble Creation:** Steps 1 to 3 are repeated to create a forest of decision trees, typically ranging from tens to hundreds of trees.

Prediction

For prediction, the responses from all trees in the forest are aggregated:

- **Regression:** The final prediction is the average of the predictions from all individual trees.
- **Classification:** Each tree gives a vote for the class, and the class receiving the majority of votes becomes the model's prediction. In the case of a tie, one class may be randomly selected, or the tie may be broken based on the class distributions.

The random forest algorithm leverages the power of multiple decision trees to reduce variance and avoid overfitting, providing robust predictions for both regression and classification tasks.

Part X
Lecture 10

Q10.1 Explain the main differences between random forests and gradient-boosted decision trees. [5]

Random forests and gradient-boosted decision trees (GBDT) are both ensemble learning methods that combine multiple decision trees, but they differ in how they build and combine these trees. In random forests, trees are built independently, with each tree trained on a random subset of the data and features, and their predictions are averaged (for regression) or voted on (for classification) to produce the final result. This approach reduces variance by averaging out errors from individual trees. In contrast, GBDT builds trees sequentially, where each new tree is trained to correct the errors (residuals) made by the previous trees. The predictions of all trees are combined through weighted sums, where the final result is more sensitive to recent corrections. GBDT often yields better predictive performance but is more prone to overfitting and computationally more expensive compared to random forests.

Q10.2 Explain the intuition for second-order optimization using Newton's root-finding method or Taylor expansions. [10]

Intuition via Newton's root-finding. Newton's method was originally designed to find a root of a 1D function $f : \mathbb{R} \to \mathbb{R}$. At the current point x, we approximate f by its *tangent line*:

$$f(x') \approx f(x) + f'(x)(x' - x).$$

We then choose x' so that this linear approximation becomes zero, i.e. $f(x') \approx 0$, which yields the update

$$x' = x - \frac{f(x)}{f'(x)}.$$

To find a *minimum* of f, we can instead find a root of the derivative $f'(x) = 0$. Applying the same idea to f' gives

$$x' = x - \frac{f'(x)}{f''(x)}.$$

This is a **second-order** update because it uses the second derivative (curvature),

Intuition via a second-order Taylor approximation. Near x, the function can be approximated by a quadratic (parabola):

$$f(x+\varepsilon) \approx f(x) + \varepsilon f'(x) + \frac{1}{2}\varepsilon^2 f''(x).$$

Instead of taking a small fixed step like gradient descent, we *minimize this local quadratic model*. Differentiate w.r.t. ε and set to zero:

$$0 \approx \frac{\partial}{\partial \varepsilon}\Big(f(x) + \varepsilon f'(x) + \tfrac{1}{2}\varepsilon^2 f''(x)\Big) = f'(x) + \varepsilon f''(x),$$

so

$$\varepsilon^* = -\frac{f'(x)}{f''(x)}, \qquad x' = x + \varepsilon^* = x - \frac{f'(x)}{f''(x)}.$$

Thus Newton's method is: *fit a local parabola and jump directly to its minimum.*

Why second-order information helps (main idea). The second derivative $f''(x)$ tells us how *steep/flat* the function is locally:

- if curvature is large ($f''(x)$ big), Newton takes a smaller step;
- if curvature is small ($f''(x)$ small), Newton takes a larger step.

So it behaves like gradient descent with an *adaptive learning rate* $1/f''(x)$, often converging in far fewer steps near an optimum.

Multivariate form (for completeness). For $f : \mathbb{R}^D \to \mathbb{R}$, the quadratic approximation uses the Hessian H, giving the Newton step

$$\mathbf{w} \leftarrow \mathbf{w} - H(\mathbf{w})^{-1}\nabla f(\mathbf{w}),$$

i.e. we precondition the gradient by the inverse curvature matrix.

Q10.3 Write down the loss function that we optimize in gradient-boosted decision trees while constructing t tree. Then, define g_i and h_i and show the value w_τ of optimal prediction in node τ and the criterion used during node splitting. Explain how the loss formulation relates to Taylor's expansions. [20]

We use an additive model

$$y(x_i) = \sum_{j=1}^{T} y_j(x_i; w_j),$$

where w_j are the (leaf-value) parameters of tree j.

Objective while constructing the t-th tree. When trees $1, \dots, t-1$ are fixed, we optimize only w_t:

$$E^{(t)}(w_t; w_{1..t-1}) = \sum_i \ell\Big(t_i,\ y^{(t-1)}(x_i) + y_t(x_i; w_t)\Big) + \frac{\lambda}{2}\|w_t\|^2,$$

where $y^{(t-1)}(x_i) = \sum_{j=1}^{t-1} y_j(x_i; w_j)$.

Second-order (Taylor) approximation and definitions of g_i, h_i. Let y denote the current prediction argument of the loss. Define

$$g_i = \left.\frac{\partial \ell(t_i, y)}{\partial y}\right|_{y=y^{(t-1)}(x_i)}, \qquad h_i = \left.\frac{\partial^2 \ell(t_i, y)}{\partial y^2}\right|_{y=y^{(t-1)}(x_i)}.$$

Using a second-order Taylor expansion of $\ell(t_i, y^{(t-1)}(x_i) + y_t(x_i))$ around $y^{(t-1)}(x_i)$,

$$\ell\Big(t_i, y^{(t-1)}(x_i) + y_t(x_i)\Big) \approx \ell\Big(t_i, y^{(t-1)}(x_i)\Big) + g_i\, y_t(x_i) + \frac{1}{2} h_i\, y_t(x_i)^2.$$

Dropping the constant term $\sum_i \ell(t_i, y^{(t-1)}(x_i))$, the approximate objective is

$$\tilde{E}^{(t)}(w_t) \approx \sum_i \left[g_i\, y_t(x_i) + \frac{1}{2} h_i\, y_t(x_i)^2 \right] + \frac{\lambda}{2}\|w_t\|^2 + \text{const.}$$

Leaf/node form and optimal w_τ. Let τ be a leaf (node) of tree t, I_τ the indices of examples in τ, and let the tree predict a constant w_τ in that leaf, i.e. $y_t(x_i) = w_\tau$ for all $i \in I_\tau$. Then

$$\tilde{E}^{(t)}(w_t) \approx \sum_\tau \left[\left(\sum_{i \in I_\tau} g_i \right) w_\tau + \frac{1}{2} \left(\sum_{i \in I_\tau} h_i \right) w_\tau^2 \right] + \frac{\lambda}{2} \sum_\tau w_\tau^2 + \text{const.}$$

Taking derivative w.r.t. w_τ and setting to zero gives the optimal leaf value

$$w_\tau^* = -\frac{\sum_{i \in I_\tau} g_i}{\lambda + \sum_{i \in I_\tau} h_i}.$$

Splitting criterion (gain / best split). Define the aggregated statistics in a node τ:

$$G_\tau = \sum_{i \in I_\tau} g_i, \qquad H_\tau = \sum_{i \in I_\tau} h_i.$$

Substituting w_τ^* back into the approximate objective yields the (minimum reachable) score

$$\tilde{E}^{(t)}(w^*) \approx -\frac{1}{2} \sum_\tau \frac{G_\tau^2}{\lambda + H_\tau} + \text{const.}$$

Therefore, when considering a split of a node τ into τ_L, τ_R, the improvement (gain) is

$$\text{Gain} = \frac{1}{2} \left(\frac{G_{\tau_L}^2}{\lambda + H_{\tau_L}} + \frac{G_{\tau_R}^2}{\lambda + H_{\tau_R}} - \frac{G_\tau^2}{\lambda + H_\tau} \right),$$

and CART-style training chooses the split maximizing Gain (equivalently minimizing the post-split value of the approximate loss).

How this relates to Taylor expansions. The whole construction comes from replacing the true loss (after adding tree t) by its *second-order Taylor approximation* around the current ensemble prediction $y^{(t-1)}(x_i)$. The coefficients g_i and h_i are exactly the first and second derivatives of the loss at that point, so the split criterion is derived from minimizing a local quadratic model of the loss.

Q10.4 List and explain three common techniques used in gradient boosting (beyond the basic algorithm) for preventing overfitting. [10]

Gradient boosting can overfit because later trees may start correcting *noise* in the training set. Common regularization techniques used in practice are:

- **Data subsampling (stochastic gradient boosting / bagging):** when training a tree, use only a random fraction of the training instances (often around 0.5) or a bootstrapped sample. This injects randomness, reduces correlation between trees, and lowers variance (less tendency to fit training-specific noise).
- **Feature subsampling:** when building a tree (or when evaluating splits), consider only a random subset of input features. This again decorrelates trees and prevents the ensemble from repeatedly exploiting the same strong-but-spurious predictors.
- **Shrinkage (learning rate):** after fitting a tree, scale its contribution by a learning rate α:
$$y^{(t)}(x) = y^{(t-1)}(x) + \alpha\, y_t(x).$$
Smaller α makes each tree have a weaker effect, so the model improves more gradually and leaves "room" for future trees, which typically improves generalization (at the cost of needing more trees).

Q10.5 For binary classification with gradient boosted decision trees, write down how the prediction is computed and define the per-example loss function. [10]

In gradient boosting, the trees predict the *linear score* (logit)

$$y(x_i) \;=\; \sum_{t=1}^{T} y_t(x_i; w_t),$$

and the predicted probability of class 1 is obtained using the logistic sigmoid

$$p_i \;=\; \sigma(y(x_i)) \;=\; \frac{1}{1+e^{-y(x_i)}}.$$

A hard class prediction can then be

$$\hat{t}_i \;=\; \mathbb{I}\big[p_i \geq \tfrac{1}{2}\big]\,.$$

The per-example loss is the Bernoulli negative log-likelihood (binary cross-entropy):

$$\ell(t_i, y(x_i)) = -\log\Big(\sigma(y(x_i))^{t_i}\big(1-\sigma(y(x_i))\big)^{1-t_i}\Big) = -t_i \log\sigma(y(x_i)) - (1-t_i)\log\big(1-\sigma(y(x_i))\big),$$

where $t_i \in \{0, 1\}$.

Q10.6 For a K-class classification, describe how to perform prediction with a gradient boosted decision tree trained for T time steps (how the individual trees perform prediction and how are the KT trees combined to produce the predicted categorical distribution). [10]

For multiclass classification, we model the full categorical distribution using a generalized linear model with a softmax output.

How individual trees predict. For each boosting *timestep* $t \in \{1, \dots, T\}$ we train K *trees*, one per class. The k-th tree at time t produces a single real-valued output (a contribution to the linear score / logit of class k), denoted

$$y_{t,k}(x_i; w_{t,k}).$$

(As in a regression tree, the prediction is the constant value stored in the leaf reached by x_i.)

How the KT trees are combined. We sum the contributions over timesteps separately for each class to obtain the K logits:

$$y_k(x_i) \;=\; \sum_{t=1}^{T} y_{t,k}(x_i; w_{t,k}), \qquad k = 1, \dots, K.$$

Equivalently, we form the logit vector

$$y(x_i) \;=\; \Big(\sum_{t=1}^{T} y_{t,1}(x_i; w_{t,1}),\; \dots,\; \sum_{t=1}^{T} y_{t,K}(x_i; w_{t,K})\Big).$$

Predicted categorical distribution. Finally, we convert the logits to a categorical distribution with softmax:

$$p(y = k \mid x_i) \;=\; \mathrm{softmax}(y(x_i))_k \;=\; \frac{\exp(y_k(x_i))}{\sum_{j=1}^{K} \exp(y_j(x_i))}.$$

The predicted class is $\hat{y}_i = \arg\max_k p(y = k \mid x_i)$ (equivalently $\arg\max_k y_k(x_i)$).

Q10.7 What type of data are gradient boosted decision trees good for as opposed to multilayer perceptron? Explain the intuition why it is the case. [5]

Gradient boosted decision trees (GBDTs) are optimal for structured, tabular data where input features have high predictive power and clear interpretability. They can capture non-linear relationships and feature interactions without extensive preprocessing.

- Good for lower-dimensional data with meaningful features.
- Handles mixed data types (continuous, categorical).
- Requires less data preprocessing.

Multilayer perceptrons (MLPs), or neural networks, are better suited for high-dimensional data such as images or text. They excel in learning hierarchical feature representations, essential in domains where raw features are not individually informative.

- When one feature does not mean much alone.
- Ideal for high-dimensional data (images, text).
- Capable of complex feature extraction.
- Benefits from pre-trained networks.

The choice of model depends on the dataset characteristics and the problem at hand.

Part XI
Lecture 11

Q11.1 Formulate SVD decomposition of matrix X, describe properties of individual parts of the decomposition. Explain what the reduced version of SVD is. [10]

Given a matrix $X \in \mathbb{R}^{m \times n}$, the singular value decomposition (SVD) of X is a factorization of the form:

$$X = U\Sigma V^T$$

where:

- $U \in \mathbb{R}^{m \times m}$ is an orthogonal matrix whose columns are the left singular vectors of X,
- $\Sigma \in \mathbb{R}^{m \times n}$ is a diagonal matrix with non-negative real numbers on the diagonal known as singular values, sorted in descending order,
- $V \in \mathbb{R}^{n \times n}$ is an orthogonal matrix whose columns are the right singular vectors of X.

Properties:

- The columns of U and V are orthonormal bases for the column space and row space of X, respectively.
- The non-zero singular values in Σ are the square roots of the non-zero eigenvalues of both $X^T X$ and XX^T.

Reduced SVD:
The reduced version of SVD is used when we want to approximate X by a matrix of lower rank k, which is less than the original rank r. It can be expressed as:

$$\tilde{X} = U_k \Sigma_k V_k^T$$

where U_k and V_k contain only the first k columns of U and V, and Σ_k contains only the top k singular values. This approximation minimizes the Frobenius norm $\|X - \tilde{X}\|_F$ among all rank-k approximations.

Q11.2 Formulate the Eckart-Young theorem. Provide an interpretation of what the theorem says and why it is useful. [10]

The Eckart-Young theorem states that given a matrix $X \in \mathbb{R}^{n \times m}$ and its Singular Value Decomposition (SVD), the best rank-k approximation X_k of X in terms of the Frobenius norm is obtained by retaining the first k singular values and corresponding singular vectors. Formally:

$$X_k = \sigma_1 u_1 v_1^T + \ldots + \sigma_k u_k v_k^T$$

where σ_i are the singular values, and u_i, v_i are the left and right singular vectors, respectively. This approximation minimizes the Frobenius norm of the difference between X and X_k:

$$\|X - X_k\|_F \leq \|X - B\|_F$$

for any $B \in \mathbb{R}^{n \times m}$ of rank k. The Frobenius norm is the square root of the sum of the absolute squares of its elements, and it can also be expressed as the square root of the trace of $X^T X$. It has the important property that multiplying by an orthonormal matrix does not change the norm:

$$\|UA\|_F = \sqrt{\text{trace}((UA)^T UA)} = \sqrt{\text{trace}(A^T U^T UA)} = \sqrt{\text{trace}(A^T A)} = \|A\|_F$$

The norm is invariant under orthogonal transformations, and the best strategy to approximate X while preserving most of its norm is to remove the smallest singular values.

Q11.3 Given a data matrix X, explain how to compute the PCA of dimension M using the SVD decomposition. [10]

Principal Component Analysis (PCA) can be computed using Singular Value Decomposition (SVD) of the data matrix X. For a data matrix $X \in \mathbb{R}^{n \times m}$, where each row is a data point and each column is a feature, PCA is performed as follows:

1. Center the data by subtracting the mean of each feature from the data matrix, resulting in the mean-centered matrix $\tilde{X} = X - \bar{x}$.
2. Compute the SVD of $\tilde{X}$, which is given by $\tilde{X} = U\Sigma V^T$.
3. The columns of V (right singular vectors) correspond to the principal components of X.
4. To reduce the dimensionality to M, select the first M columns of V, and the first M singular values from Σ.
5. The projection of X onto the M-dimensional subspace is given by $X_M = XV_M$, where V_M is the matrix containing the first M columns of V.

This works because the singular values in Σ represent the amount of variance captured by each principal component, and the columns of V are the directions along which the variance is maximized. By taking the first M components, we retain the features that capture the most variance in the data.

$$\|\mathbf{X} - \bar{\mathbf{x}}\|_F^2 = \text{trace}\left((\mathbf{X} - \bar{\mathbf{x}})^T (\mathbf{X} - \bar{\mathbf{x}})\right) = N \sum_{i=1}^{D} \text{Var}(\mathbf{X}_{:,i})$$

Approximating the matrix in terms of Frobenius norm means keeping the most variance from the data. Components are ordered by how much variability in the data they capture.

$$\text{Let } \mathbf{S} = \frac{1}{N}(\mathbf{X} - \bar{\mathbf{x}})^T (\mathbf{X} - \bar{\mathbf{x}}),$$
$$\text{then PCA of } \mathbf{X} \text{ involves the eigenvectors of } \mathbf{S},$$
$$\text{denoted by the } \mathbf{V} \text{ matrix in the SVD of } \mathbf{X} - \bar{\mathbf{x}}.$$

Q11.4 Describe how SVD can be used in recommender systems. What are the advantages of using SVD instead of the full user-interaction matrix? [10]

Let $R \in \mathbb{R}^{m \times n}$ be a user–item interaction matrix (rows = users, columns = items; entries are ratings, likes, clicks, etc.). Such matrices are typically very large, sparse, and noisy.

Using SVD for recommendation (latent-factor modelling)

Compute an SVD (or a truncated/low-rank variant):

$$R = U\Sigma V^\top \approx U_k \Sigma_k V_k^\top,$$

where $k \ll \min(m, n)$ and Σ_k keeps only the largest singular values (dropping small ones acts as denoising).

From the truncated factors, define low-dimensional embeddings (latent vectors) for users and items, e.g.

$$P := U_k \Sigma_k^{1/2} \in \mathbb{R}^{m \times k}, \qquad Q := V_k \Sigma_k^{1/2} \in \mathbb{R}^{n \times k}.$$

Then a predicted preference score is

$$\hat{r}_{ui} = p_u^\top q_i \quad \text{(optionally plus user/item/global bias terms)}.$$

Recommendations for user u are obtained by ranking items i by $\hat{r}_{ui}$, and similarity search can be performed directly in the k-dimensional latent space (often described as "eigenusers" and "eigencontent").

Advantages over the full interaction matrix

- **Dimensionality reduction:** represent each user/item with only k numbers instead of n or m.
- **Compression / memory:** store U_k, Σ_k, V_k with cost $O(k(m+n))$ rather than $O(mn)$.
- **Noise reduction:** discarding small singular values removes variance likely due to noise/outliers.
- **Best low-rank approximation:** the truncated SVD gives the optimal rank-k approximation in Frobenius norm (Eckart–Young), so it preserves as much signal/energy as possible for a given k.
- **Faster computation:** scoring and nearest-neighbour similarity become operations in $\mathbb{R}^k$; many downstream tasks (ranking, clustering) are cheaper.
- **Generalization to unseen pairs:** the low-rank model can infer missing entries (unobserved user–item pairs) via latent structure, rather than relying on exact overlap in sparse raw interactions.

Note. In real systems, "pure" SVD is often adapted (e.g., regularization, handling implicit feedback, alternative loss functions), but the core idea remains low-rank latent factors derived from SVD-like matrix factorization.

Q11.5 Given a data matrix X, write down the algorithm for computing the PCA of dimension M using the power iteration algorithm. [20]

Algorithm 5 Computing PCA — The Power Iteration Algorithm

Input: Matrix X, desired number of dimensions M.

- Compute the mean μ of the examples (the rows of X).
- Compute the covariance matrix $S \leftarrow \frac{1}{N}(X-\mu)^T(X-\mu)$.
- for i in $\{1, 2, \ldots, M\}$:
  - Initialize v_i randomly.
  - Repeat until convergence (or for a fixed number of iterations):
    - $v_i \leftarrow Sv_i$
    - $\lambda_i \leftarrow \|v_i\|$
    - $v_i \leftarrow v_i/\lambda_i$
  - $S \leftarrow S - \lambda_i v_i v_i^T$
- Return $(X-\mu)V$, where the columns of V are $v_1, v_2, \ldots, v_M$.

Q11.6 List at least two applications of SVD or PCA. [5]

Singular Value Decomposition (SVD) and Principal Component Analysis (PCA) are widely used in various applications. One common application is image compression, where SVD or PCA is used to reduce the dimensionality of image data by retaining only the most significant components, effectively reducing storage requirements without sacrificing much image quality. Another application is topic modeling in natural language processing (NLP), where PCA or SVD is applied to text data, such as in Latent Semantic Analysis (LSA), to extract underlying patterns in word usage and identify the most important topics in a corpus of documents. These techniques are essential for simplifying complex data and uncovering hidden structures in a wide range of domains.

Also, PCA is often used in data visualization to reduce high-dimensional data to 2 or 3 dimensions for plotting, allowing for better visualization of the data distribution and relationships between data points.

Q11.7 Describe the K-means algorithm, including the `kmeans++` initialization. What is it used for? What is the loss function that the algorithm optimizes? What can you say about the algorithm convergence? [20]

What it is used for. K-means is an *unsupervised* clustering method: given data points $x_1, \ldots, x_N \in \mathbb{R}^D$ and a chosen number of clusters K, it partitions the points into K groups and represents each group by a *cluster center* (centroid).

Model and loss function. Let $z_{i,k} \in \{0, 1\}$ indicate whether point x_i is assigned to cluster k (with $\sum_{k=1}^{K} z_{i,k} = 1$), and let cluster centers be $\mu_1, \ldots, \mu_K$. K-means minimizes the within-

cluster sum of squared distances

$$J(\{z_{i,k}\}, \{\mu_k\}) = \sum_{i=1}^{N} \sum_{k=1}^{K} z_{i,k} \|x_i - \mu_k\|^2.$$

Algorithm 6 K-means (Lloyd's algorithm)

Input: Input points $x_1, \ldots, x_N$, number of clusters K.
Output: Cluster centers $\mu_1, \ldots, \mu_K$ and assignments $z_{i,k}$.

Loss: $J = \sum_{i=1}^{N} \sum_{k=1}^{K} z_{i,k} \|x_i - \mu_k\|^2$.

- Initialize $\mu_1, \ldots, \mu_K$ as K random input points (or using `kmeans++`, Alg. **??**).
- Repeat until convergence (or until patience runs out):
  - Compute the best possible $z_{i,k}$ (minimizes J for fixed μ):

$$z_{i,k} = \begin{cases} 1 & \text{if } k = \arg\min_j \|x_i - \mu_j\|^2, \\ 0 & \text{otherwise.} \end{cases}$$

  - Compute the best possible μ_k (minimizes J for fixed z):

$$\mu_k = \arg\min_{\mu} \sum_i z_{i,k} \|x_i - \mu\|^2 = \frac{\sum_i z_{i,k} x_i}{\sum_i z_{i,k}}.$$

`kmeans++` initialization. Instead of picking all initial centers uniformly at random, `kmeans++` chooses:

- the first center uniformly at random from the data;
- each subsequent center x with probability proportional to $D(x)^2$, where $D(x)$ is the distance from x to the *nearest already chosen* center.

This typically yields better starting points; moreover, it can be shown to achieve an $O(\log K)$ approximation ratio *in expectation*.

Convergence properties. Each iteration *does not increase* the objective:

- updating assignments (nearest-center choice) decreases J or keeps it the same;
- updating centers (means) decreases J or keeps it the same.

Therefore, K-means converges to a *local optimum* (not necessarily the global one) and is sensitive to initialization; a common practice is to run it multiple times with different initializations and keep the solution with the lowest J.

Q11.8 Name at least two clustering algorithms. What is their main principle? How do they differ? [10]

K-means Clustering

Approach: Partitional clustering algorithm that divides data into k clusters by minimizing the variance within each cluster. How it works: The algorithm iteratively assigns points to the nearest cluster center (centroid) and updates the centroid based on the mean of points in each

cluster. This process is repeated until convergence. Advantages: Simple, fast, and efficient for large datasets with well-separated spherical clusters. Disadvantages: Requires specifying the number of clusters k in advance, sensitive to initial centroid placement, and struggles with non-spherical or overlapping clusters.

Hierarchical Clustering

Approach: Builds a hierarchy of clusters either in a bottom-up (agglomerative) or top-down (divisive) manner. How it works: In agglomerative hierarchical clustering, each data point starts as its own cluster, and pairs of clusters are merged based on a distance metric until all points are in a single cluster. In divisive clustering, all points start in one cluster, and the algorithm recursively splits them. Advantages: Does not require pre-specifying the number of clusters, can produce a dendrogram to visualize the relationships between clusters. Disadvantages: Computationally expensive, particularly for large datasets, and sensitive to noise and outliers.

Comparison

K-means is efficient and works well for spherical clusters with a known number of clusters but struggles with complex data structures. Hierarchical clustering produces a detailed tree-like structure of clusters and doesn't require the number of clusters to be defined in advance, but it can be slow for large datasets.

Part XII
Lecture 12

Q12.1 Considering statistical hypothesis testing, define type I errors and type II errors (in terms of the null hypothesis). Finally, define what a significance level is. [10]

In hypothesis testing, we begin with a null hypothesis, H_0, which is a statement we assume to be true until evidence suggests otherwise. An alternative hypothesis, H_1, is considered if there is sufficient evidence against H_0.

Type I and Type II Errors

- **Type I Error**: Also known as a false positive, occurs when H_0 is true, but we incorrectly reject it.
- **Type II Error**: Also known as a false negative, occurs when H_0 is false, but we fail to reject it.

The probability of making a Type I error is denoted by α, and is known as the significance level of the test. It is the threshold below which we reject H_0, commonly set at 5%.

Significance Level

The significance level α is the probability of rejecting the null hypothesis H_0 when it is actually true. It defines the sensitivity of our hypothesis test.

$$\alpha = P(\text{Type I Error}) = P(\text{Reject } H_0 | H_0 \text{ is true})$$

The lower the value of α, the less likely we are to make a Type I error, but the higher the chance of a Type II error.

Confusion Matrix

A confusion matrix helps to visualize the performance of a statistical test:

	H_0 True	H_1 True
Reject H_0	Type I Error (False Positive)	Correct Decision (True Positive)
Fail to Reject H_0	Correct Decision (True Negative)	Type II Error (False Negative)

Q12.2 Explain what a test statistic and a p-value are. [5]

Test Statistic

A **test statistic** is a standardized value derived from sample data during a hypothesis test. It is calculated to assess the strength of the evidence against the null hypothesis. Mathematically, it can be represented as:

$$T = \frac{\text{Sample Estimate} - \text{Population Parameter}}{\text{Standard Error}}$$

It quantifies how far our sample statistic deviates from what we would expect if the null hypothesis H_0 were true, under the assumption of the null hypothesis.

P-value

The **p-value** is the probability of obtaining a test statistic at least as extreme as the one actually observed, given that the null hypothesis is true. It is a measure of the evidence against the null hypothesis provided by the sample. A smaller p-value indicates stronger evidence against H_0. It is given by:

$$p\text{-value} = P(T \geq t | H_0 \text{ is true})$$

for a right-tailed test, or

$$p\text{-value} = P(T \leq t | H_0 \text{ is true})$$

for a left-tailed test, or

$$p\text{-value} = P(|T| \geq |t| | H_0 \text{ is true})$$

for a two-sided symmetric test. where t is the observed value of the test statistic.

A p-value less than the chosen significance level α (commonly 0.05) leads to the rejection of the null hypothesis.

Q12.3 Write down the steps of a statistical hypothesis test, including a definition of a p-value. [10]

The procedure for performing a statistical hypothesis test is as follows:

1. Formulate the null hypothesis H_0, which is a statement of no effect or no difference, and optionally the alternative hypothesis H_1, which is what you aim to support.
2. Choose the appropriate test statistic that will measure the degree of agreement between the sample data and the null hypothesis.
3. Compute the observed value of the test statistic from the sample data.
4. Calculate the p-value, which is the probability of observing a test statistic as extreme as, or more extreme than, the observed value, assuming the null hypothesis is true.
5. Decide whether to reject or not reject the null hypothesis by comparing the p-value to the chosen significance level α. Common choices for α include 5%, 1%, 0.5%, or 0.1%. If the p-value is less than or equal to α, reject H_0; otherwise, do not reject H_0.

P-value Definition

The p-value is defined as the probability of obtaining a test statistic at least as extreme as the one that was actually observed, under the assumption that the null hypothesis is true. It is used as a tool to decide whether to reject or not reject the null hypothesis. More detailed explanation in Q12.2.

Q12.4 Explain the differences between a one-sample test, two-sample test, and a paired test. [10]

- **One-sample test:** This test is used when we want to compare the sample mean from a single group to a known value or theoretical expectation. Common applications include testing whether the mean of a single distribution is equal to, greater than, or less than a certain value, or performing goodness of fit tests to determine if the data come from a specified distribution.
- **Two-sample test:** This type of test is applicable when two independent samples are taken from different populations, and we want to compare their means. The key assumption is that the two samples are independent of each other.
- **Paired test:** Paired tests are used when the samples are not independent but are paired in some meaningful way. For example, measurements before and after a treatment on the same subjects, or the same subjects measured under two different conditions. The test involves computing the differences between the paired measurements and then performing a one-sample test on these differences.

In a one-sample test, the null hypothesis typically states that the mean of the distribution is equal to a target value. For two-sample tests, the null hypothesis commonly states that the means of the two populations are equal. In a paired test, the null hypothesis usually states that the mean difference between the paired observations is zero.

Q12.5 When considering multiple comparison problem, define the family-wise error rate, and prove the Bonferroni correction, which allows limiting the family-wise error rate by a given α. [10]

Family-Wise Error Rate (FWER)

The family-wise error rate (FWER) is the probability of making one or more Type I errors when performing multiple hypotheses tests. Formally, for a family of m hypotheses tests, the FWER is defined as:

$$FWER = P\left(\bigcup_{i=1}^{m}(p_i \leq \alpha)\right)$$

where p_i is the p-value for the i-th hypothesis test and α is the significance level.

Bonferroni Correction

The Bonferroni correction is a method to control the FWER. It involves adjusting the significance level α by the number of hypotheses m. The corrected significance level is α/m.

Proof: Using the union bound (Boole's inequality), we have:

$$FWER = P\left(\bigcup_{i=1}^{m}(p_i \leq \alpha)\right) \leq \sum_{i=1}^{m} P(p_i \leq \alpha)$$

For independent tests, assuming a uniform distribution under the null hypothesis, the probability of a single test yielding a p-value less than α is exactly α. With the Bonferroni correction, we have:

$$P(p_i \leq \frac{\alpha}{m}) = \frac{\alpha}{m}$$

Therefore, the sum of probabilities for m independent tests is:

$$\sum_{i=1}^{m} P(p_i \leq \frac{\alpha}{m}) = m \cdot \frac{\alpha}{m} = \alpha$$

This proves that the Bonferroni correction ensures that the FWER is at most α.

Q12.6 For a trained model and a given test set with N examples and metric E, write how to estimate 95% confidence intervals using bootstrap resampling. [10]

Given a test set $\{(x_1, t_1), \ldots, (x_N, t_N)\}$, a trained model with predictions $\{y(x_1), \ldots, y(x_N)\}$, and a performance metric E, we estimate the 95% confidence intervals of the model's performance using bootstrap resampling as follows:

1. Initialize an empty list called *performances* to store the sampled performances.
2. Repeat R times, where R is a large number (commonly 1000 or more for better approximation):
   (a) Generate a bootstrap sample by sampling N examples from the test set *with replacement*.
   (b) For each sampled example, obtain the corresponding model prediction.
   (c) Compute the performance metric E for the bootstrap sample.
   (d) Append the computed performance to the *performances* list.
3. Sort the list of *performances*.
4. The 95% confidence interval is estimated by selecting the 2.5th percentile as the lower bound and the 97.5th percentile as the upper bound from the sorted *performances* list.

Formally, the 95% confidence interval CI is given by:

$$CI = \left(performances\left[\frac{R}{40}\right], performances\left[\frac{39R}{40}\right]\right)$$

This interval estimates the range within which the true performance metric of the model lies with 95% confidence.

Q12.7 For two trained models and a given test set with N examples and metric E, explain how to perform a paired bootstrap test that the first model is better than the other. [10]

Given a test set $\{(x_1, t_1), \ldots, (x_N, t_N)\}$ and predictions from two models $\{y(x_1), \ldots, y(x_N)\}$ and $\{z(x_1), \ldots, z(x_N)\}$, we aim to test the hypothesis that the first model y performs better than the second model z using the paired bootstrap test with the following steps:

1. Initialize an empty list called *differences* to store the differences in performances.
2. For a number of resamplings R, repeat:
   (a) Generate a bootstrap sample by sampling N examples from the test set *with replacement*.
   (b) For each sampled example, obtain the corresponding predictions from both models y and z.
   (c) Calculate the performance of both models on the sampled data using the metric E.
   (d) Compute the difference in performance between the two models for each bootstrap sample and append it to the *differences* list.
3. The significance of the test is determined by the proportion of differences that are less than or equal to zero (indicating no improvement of model y over model z).
4. This proportion represents the estimated probability that model y is not better than model z. If this proportion is small (typically less than a chosen significance level such as 0.05), we reject the null hypothesis and conclude that model y is significantly better than model z.

Unfortunately, the value returned by the algorithm is not really a p-value. The reason is that the distribution of differences was obtained **under the true distribution**. However, to perform the statistical test, we require the distribution of the test statistic **under the null hypothesis**. Nevertheless, you can encounter such paired bootstrap tests *"in the wild"*.

Q12.8 For two trained models and a given test set with N examples and metric E, explain how to perform a random permutation test that the first model is better than the other with a significance level of α. [10]

Given two trained models and a test set $\{(x_1, t_1), \ldots, (x_N, t_N)\}$, with model predictions $\{y(x_1), \ldots, y(x_N)\}$ and $\{z(x_1), \ldots, z(x_N)\}$, we perform a random permutation test to determine if the first model is significantly better than the second at a significance level α with the following algorithm:

1. Initialize an empty list called *differences* to store the differences in performance.
2. Repeat for a number of resamplings R:
   (a) For each test set example, randomly permute the model predictions, effectively assigning the prediction of model y or model z to each test case.
   (b) Compute the performance of the permuted predictions using the metric E.
   (c) Record the performance and append it to the list *performances*.

3. Return the ratio of the performances which are greater than or equal to the performance of the model y.

4. The p-value is then given by this proportion. If the p-value is less than or equal to the significance level α, we reject the null hypothesis that there is no difference in performance, suggesting that the first model is significantly better.

(The calculation of the p-value is not exactly as I say here, because the algorithm actually returns β)

Q12.9 Explain why the paired bootstrap test does not produce a true p-value, even though it can be useful for model comparison. What is the fundamental difference between the distribution obtained through bootstrap resampling and the distribution required for proper hypothesis testing? [10]

TODO

Part XIII
Lecture 13

Q13.1 Explain the difference between deontological and utilitarian ethics. List examples on how these theoretical frameworks can be applied in machine learning ethics. [10]

Deontological Ethics

Deontological Ethics focuses on the inherent nature of actions rather than their consequences. It emphasizes adherence to predefined rules and principles, such as the Universal Declaration of Human Rights, the Ten Commandments, or Kant's Categorical Imperative. In the context of machine learning (ML), it translates to principles like beneficence, non-malevolence, privacy, non-discrimination, autonomy, and informed consent.

Utilitarian Ethics

Utilitarian Ethics, on the other hand, is an ethical theory that emphasizes the maximization of overall happiness or well-being, thus focusing on the consequences of actions. It promotes actions that lead to the greatest overall positive impact.

Examples

In ML ethics, **deontological frameworks** might lead to ethical problems in:

- Problem definition: Some tasks may not align with fundamental ethical principles.
- Data collection: Issues like privacy invasion or non-consensual data usage.
- Model development: Ensuring models do not discriminate or violate user autonomy.

Utilitarian frameworks in ML ethics might consider:

- Model evaluation: Metrics should account for overall happiness or harm reduction.

- Model deployment: Using models in ways that maximize social good while minimizing potential harm or feedback loops that might disadvantage certain groups.

Examples:

- A deontological approach might reject any form of user data exploitation, even if it improves the performance of a recommendation system, on the principle of user autonomy.
- A utilitarian approach may justify the use of personal data if the resulting system significantly benefits a large number of users, thereby increasing overall utility.

Both approaches have their merits and challenges when applied to ML ethics. Deontological ethics provide clear guidelines but can be rigid and may lead to conflicts between principles. Utilitarian ethics offer flexibility and quantifiability but may overlook individual rights and face difficulties in defining collective well-being.

Q13.2 List a few examples of potential ethical problems related to data collection. [5]

1. **Representation Bias**: Data may not be representative of the entire population, often excluding minorities or economically disadvantaged groups.
2. **Internet Data Misrepresentation**: Data collected from the internet might disproportionately represent the views and behaviors of those who have access and are more vocal online, skewing perceptions of the general population.
3. **Historical Bias**: Data reflecting past inequalities may perpetuate these biases when used to train modern machine learning systems.
4. **Exploitation in Crowdsourcing**: Individuals hired to collect or label data, often in low-income countries, may be underpaid and work under poor conditions, which could lead to monotonous work that causes psychological harm.
5. **Non-transparent Data Collection**: Users may unknowingly provide personal data when using online services, without a clear understanding or explicit consent of how their data will be used or the implications of its use.

Q13.3 List a few examples of potential ethical problems that can originate in model evaluation. [5]

1. **Incomplete Metrics:** Evaluation metrics may not fully capture the desired outcomes. For instance, while translation fluency metrics may seem adequate, they might overlook ingrained gender biases.
2. **Macro-averaging Oversights:** Using macro-averaging in evaluation can obscure poor performance for specific user groups, often minorities, thus perpetuating discrimination.
3. **Human Resource Algorithms:** Employment recommendation systems optimized for precision may inadvertently discriminate based on gender, age, ethnicity, etc., since the system's recall—indicating the full scope of candidates, including potentially overlooked qualified individuals—is not visible.
4. **Data Mismatch:** When training and testing data do not align, minority languages could be disproportionately classified as hate speech or not safe for work, as highlighted in the paper "The Risk of Racial Bias in Hate Speech Detection" (Sap et al., ACL 2019).

5. **Feedback Loops:** Recommender systems can create feedback loops where predictions influence user behavior, which then feeds back into the training data. This can lead to echo chambers and self-affirmative groups. A notable example is how YouTube's recommendation algorithms facilitated the discovery of a category of home videos of scantily clad children by pedophiles.

Q13.4 List at least one example of an ethical problem that can originate in model design or model development. [5]

One ethical problem that can arise in model design or development is bias and discrimination in predictive models. For example, if a machine learning model is trained on biased historical data, it may perpetuate or even exacerbate existing inequalities. In hiring algorithms, if the training data reflects gender or racial biases, the model might unfairly favor certain demographic groups over others. This can lead to discriminatory practices, such as hiring or lending decisions that disadvantage certain groups, violating principles of fairness and equality. Ensuring fairness and mitigating bias in model design is crucial to avoid these ethical issues.

Q13.5 Under what circumstances could train-test mismatch be an ethical problem? [5]

Train-test mismatch can become an ethical problem when a model is trained on data that doesn't accurately represent the population it will be applied to, leading to unfair or harmful outcomes. For instance, if a model is trained on data from one demographic group (e.g., based on gender, age, or ethnicity) but then tested or deployed on a broader or different group, it may fail to generalize well, leading to biased predictions. This can result in unjust decisions, such as in healthcare or criminal justice, where individuals from underrepresented groups are unfairly treated or disadvantaged because the model doesn't account for their specific characteristics. In such cases, the mismatch between training and testing data can reinforce existing inequalities, causing harm to individuals who are misrepresented or overlooked by the model, and raising significant ethical concerns regarding fairness, accountability, and transparency.

Q13.6 Chose one ethical issue with deploying ML systems and describe it from deontological and utalitarian perspective. [5]

Ethical issue (deployment): ***Using ML risk scores for criminal sentencing (e.g., recidivism prediction / COMPAS).*** An ML system is deployed to estimate a defendant's risk of reoffending and this score influences judicial decisions. Such systems can be opaque and can exhibit disparate impact across protected groups.

Deontological (rule-/rights-based) perspective. From deontological ethics, the core question is whether the deployment violates duties and rights (e.g., fairness, non-discrimination, transparency, autonomy/informed consent). If the system is non-transparent and treats similar cases differently across ethnicities, it violates the right to a fair trial and equality before the law, so deploying it is morally wrong *regardless* of any efficiency gains.

Utilitarian (consequence-based) perspective. From utilitarianism, the decision depends on overall consequences: who is affected and how, and whether benefits outweigh harms. Even if the system saves time/money for the state, the aggregate harm from wrongful or biased outcomes

(loss of justice, increased discrimination, societal distrust) can outweigh those benefits; therefore deployment is morally wrong if it decreases overall well-being or increases harm.

Part XIV
The End

Contributing

Github repo: `https://github.com/Desperadus/mff-ml-exam-prep`

If you find any errors in this document, please create a pull request. Contributions are welcome! If you decide to do so feel free to add your name to the list of contributors below.

Contributors

- **luk27official (Lukáš Polák)** - added 2024/25 questions
- **Jakub-Kos (Jakub Kos)** - added 2025/26 questions