1 ML tasks

1.1 a) Supervised learning

The dataset is a collection of labeled data points:

$$D = \{(\vec{x}_i, y_i)\}_{i=1,\dots,n}$$

The components are often referred to by different names:

Goal: Learn the conditional probability distribution $p(y|\vec{x})$. Y can be univariate or multivariate.

1.1.1 Classification

The output variable y is nominal (categorical).

1.1.2 Regression

The output variable y is numeric.

1.2 b) Unsupervised learning

Here we have no labels, just data points $\vec{x}_i$. The goal is to learn the underlying structure of the data.

Goal: Learn $p(\vec{y})$.

1.2.1 Dimensionality reduction

Examples of algorithms include PCA, MDS, t-SNE, UMAP, etc.

1.3 c) Semi-supervised learning

This approach uses a combination of labeled and unlabeled data. Using the unlabeled data can help identify better decision boundaries than using the labeled data alone.

1.4 d) Reinforcement learning

Key concepts include:

- Delayed reward
- Exploration vs. exploitation trade-off

2 Models

2.1 a) Language/model class $\mathcal{H}$

A model is a set of hypotheses $h \in \mathcal{H}$. This represents our assumptions about the Data Generating Process (DGP).

2.2 b) Learning (fitting/training)

This is the process of expressing a preference for a hypothesis h from the model class $\mathcal{H}$. The result can be:

- A single element h
- A set of elements
- A distribution over all elements (Bayesian approach)

2.3 Bad examples of models

- **The set of all possible models:** Leads to overfitting.
- **Memorization:** Return y if there is an exact match for x, otherwise fail. This is useless for new data.
- **Assuming a linear relationship:** This might not be true for the underlying data.

2.4 Parametric vs. Non-parametric learning

3 K-Nearest Neighbours (k-NN)

The core assumption is that similar points have similar labels. For a new data point, k-NN finds the 'k' closest points in the training data.

- **Classification:** The prediction is the majority class among the k neighbors.
- **Regression:** The prediction is the average of the values of the k neighbors.

Weights can be added based on distance, so that closer neighbors have more influence on the prediction.

3.1 Distance Metrics

- **Euclidean:** Standard distance between two points. The formula is given by:

$$\|x - y\|_2 = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

- **Cosine:** Measures the cosine of the angle between two non-zero vectors. Cosine similarity is calculated as:

$$\cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

  Cosine distance is then typically defined as $1 - \cos(\theta)$.

- **Jaccard:** Used for comparing sets.

4 Curse of dimensionality

In high-dimensional spaces, Euclidean distance can become meaningless. As the number of dimensions (d) increases, the volume of the space grows exponentially. This causes data points to become sparse and far apart from each other.

Consider a hypercube. To capture a fixed fraction of the data, say $a = 0.5$, the required side length of the hypercube, $r = \sqrt[d]{a}$, approaches 1 as the dimension d increases.

- In 1 dimension, to capture 50% of the data, you need to cover 50% of the length ($r = 0.5$).
- In 2 dimensions, you need a square with side length $r = \sqrt{0.5} \approx 0.71$.
- In high dimensions, the hypercube must cover almost the entire range in each dimension to capture the same fraction of data.

The result is that in high dimensions, nearly all data points are near the "edges" of the space, making the concept of "closeness" less useful. This negatively affects distance-based algorithms like k-NN.

5 Bayes Optimal Classifier

If we know the true data generating process $p(y|x)$, the optimal classifier $h^*(x)$ is the one that picks the most likely class for a given input x.

$$h^*(x) = \underset{y}{\operatorname{argmax}}\ p(y|x)$$

The error of this classifier is the Bayes error rate:

$$\epsilon_{OPT} = P(y \neq h^*(x)) = 1 - p(y = h^*(x)|x)$$

5.1 1-NN vs Bayes Optimal

For $n \to \infty$, the error of the 1-NN classifier is bounded by twice the Bayes optimal error rate. For a two-class problem, with y^* being the optimal class label:

$$\epsilon_{1NN} \leq 2\epsilon_{OPT}$$

The derivation from the lecture notes shows:

$$\epsilon_{1NN} = (1 - p(y^*|x))p(y^*|x) + p(y^*|x)(1 - p(y^*|x))$$

$$\leq 2(1 - p(y^*|x)) = 2\epsilon_{OPT}$$

6 Linear Regression

The goal of linear regression is to model the relationship between a dependent variable y and one or more independent variables x. The model takes the form of a linear equation:

$$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots$$

where $\hat{y}$ is the predicted value.

The core assumption is that the observed value y is the predicted value $\hat{y}$ plus some normally distributed error term ϵ. This can be expressed as a conditional probability:

$$p(y|x; \beta, \sigma^2) = \mathcal{N}(\hat{y}(x, \beta), \sigma^2)$$

6.1 Maximum Likelihood Estimation (MLE)

Given a dataset, we want to find the parameters β that are most likely to have produced the observed data. We do this using Maximum Likelihood Estimation. The error (or residual) for each data point is $e = y - \hat{y}$. Assuming the errors are independent and identically distributed according to a normal distribution, the probability density of an error is:

$$p(e) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{e^2}{2\sigma^2}\right)$$

Maximizing the likelihood of the data is equivalent to minimizing the sum of the squared errors.

6.2 Assumptions

- The relationship between y and x is linear, with additive noise.
- The independent variables x_i are constants, measured without error.
- The error term e is normally distributed.
- The error term e is independent of x (homoscedasticity, or constant variance).

6.3 Finding the optimal β

To find the optimal parameters β, we want to maximize the log-likelihood of our data. The likelihood is the product of the probabilities of each individual data point:

$$L(\beta) = p(\mathbf{y}|X, \beta) = \prod_{i=1}^{n} p(y_i|\vec{x}_i, \beta)$$

It is easier to work with the log-likelihood, $l(\beta) = \log L(\beta)$. Taking the log converts the product into a sum:

$$l(\beta) = \sum_{i=1}^{n} \log\left(\frac{1}{\sigma\sqrt{2\pi}}\right) - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

To maximize $l(\beta)$, we need to minimize the sum of squared errors (SSE), also known as the cost function $J(\beta)$:

$$J(\beta) = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \sum_{i=1}^{n}(y_i - \beta^T\vec{x}_i)^2$$

There are two main approaches to find the β that minimizes this function.

6.3.1 a) Gradient Descent

Gradient descent is an iterative optimization algorithm that finds the minimum of a function. We start with an initial guess for β and repeatedly move in the opposite direction of the gradient of the cost function. The update rule for each parameter β_j is:

$$\beta_j \leftarrow \beta_j - \alpha\frac{\partial}{\partial\beta_j}J(\beta)$$

where α is the learning rate. The partial derivative is:

$$\frac{\partial}{\partial\beta_j}J(\beta) = \frac{\partial}{\partial\beta_j}(\hat{y}(\beta) - y)^2 = 2(\hat{y}(\beta) - y) \cdot x_j$$

- **Stochastic Gradient Descent (SGD):** The parameters are updated using only one data instance at a time.

$$\beta_j \leftarrow \beta_j - \alpha(\hat{y}(\beta) - y)x_j$$

- **Batch Gradient Descent:** The parameters are updated using the entire dataset. The update rule averages the gradients over all N instances.

$$\beta_j \leftarrow \beta_j - \alpha\frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)x_{ij}$$

6.3.2 b) Closed-Form Solution (Normal Equation)

For linear regression, a direct closed-form solution exists. The cost function in matrix form is:

$$J(\beta) = (\mathbf{y} - X\beta)^T(\mathbf{y} - X\beta)$$

We can find the minimum by taking the derivative with respect to β and setting it to zero:

$$\frac{\partial J(\beta)}{\partial\beta} = -2X^T(\mathbf{y} - X\beta) = 0$$

Solving for β gives the normal equation:

$$\hat{\beta} = (X^TX)^{-1}X^T\mathbf{y}$$

This solution gives the global minimum directly. However, this method requires computing the inverse of the matrix X^TX, which is not always possible. This matrix is non-invertible if there is multicollinearity (i.e., features are linearly dependent) or if the number of features is greater than the number of data instances.

7 Trees and Forests

7.1 Decision Trees (CART)

Classification and Regression Trees (CART) work by recursively partitioning the input space into rectangular regions using axis-parallel splits. The goal is to create regions that are as "pure" as possible.

7.1.1 Splitting Criteria

To find the best split at each node, we want to maximize the information gain, which is the reduction in impurity after the split.

- **Regression:** The impurity is typically the variance of the target variable within the node. The goal is to minimize the weighted average variance of the child nodes.

$$\text{Gain} = \text{var}(D) - \left(\frac{|D_L|}{|D|}\text{var}(D_L) + \frac{|D_R|}{|D|}\text{var}(D_R) \right)$$

- **Classification:** Impurity measures how mixed the classes are in a node. Common measures are:
  - **Gini Impurity:** $G(p) = \sum_c p_c(1 - p_c) = 1 - \sum_c p_c^2$.
  - **Entropy:** $H(p) = -\sum_c p_c \log_2(p_c)$.

Once the tree is grown, predictions are made by taking the mean (for regression) or the majority class (for classification) of the training instances in the leaf node where the new data point falls.

7.1.2 Challenges with Decision Trees

- **Instability:** Small changes in the training data can result in a completely different tree structure.
- **Stopping criteria:** Deciding when to stop splitting (e.g., maximum depth, minimum samples per leaf) is crucial to avoid overfitting.
- **Interpretability:** While small trees are interpretable, large trees are not.
- **Assumptions:** They can model non-linear relationships but struggle with simple additive structures.

8 Bagging (Bootstrap Aggregating)

Bagging is an ensemble technique designed to improve the stability and accuracy of machine learning algorithms like decision trees. The process is as follows:

1. Create multiple bootstrap samples from the original dataset by sampling with replacement.
2. Train a separate model on each bootstrap sample.
3. The final prediction is the average (for regression) or a majority vote (for classification) of all the individual models' predictions.

Bagging helps to reduce the variance of a model and lowers the influence of outliers. A useful feature is the Out-of-Bag (OOB) error, where the data points not selected in a bootstrap sample are used to evaluate the model, removing the need for a separate validation set. However, bagging is most effective for unstable models; it provides little benefit for stable models.

9 Random Forest

A Random Forest is an enhancement of bagging that specifically uses decision trees as the base models. It adds an extra layer of randomness to further de-correlate the trees. In addition to the bootstrap sampling of data points, a Random Forest also samples a random subset of features at each split in the tree-building process.

- **Includes:** Bootstrap sampling (like bagging) and random feature selection for each split.
- **Benefits:** The added randomness makes the trees less correlated with each other, which generally leads to a more robust and accurate model.
- **Properties:** Random forests are generally fast to train (as each split considers fewer features) and are considered a very strong baseline model for many prediction tasks.

10 Variance of Averaged Random Variables

The variance of an average of random variables depends on their correlation.

- **Independent and Identically Distributed (IID):** For n IID random variables x_i each with variance σ^2, the variance of their average $\bar{x}$ is:

$$\text{Var}(\bar{x}) = \text{Var}\left(\frac{1}{n}\sum_{i=1}^{n} x_i \right) = \frac{1}{n^2}\sum_{i=1}^{n}\text{Var}(x_i) = \frac{1}{n^2} n\sigma^2 = \frac{\sigma^2}{n}$$

  As the number of variables n increases, the variance of the average approaches zero. This is the principle behind bagging.

- **Correlated Variables:** If the variables are not independent but have an average pairwise correlation ρ, the variance of the average is:

$$\text{Var}(\bar{x}) = \rho\sigma^2 + \frac{1-\rho}{n}\sigma^2$$

  If $\rho > 0$, the variance does not go to zero as $n \to \infty$. It is lower-bounded by $\rho\sigma^2$. This highlights a limitation of bagging where models (trees) can be correlated.

11 Model Evaluation

Model evaluation is a critical part of the machine learning workflow. It is unavoidable and can be costly, but it serves as a crucial protection against issues like overfitting by providing feedback on how a model will perform on unseen data. Common mistakes in model evaluation include:

- Answering the wrong question.
- Optimizing for the wrong metric.
- Using an inappropriate model for the data.
- Employing a flawed evaluation procedure, such as not accounting for temporal dependencies in data.
- Re-inventing the wheel with custom inference when standard procedures exist.

It's good practice to build a reusable evaluation "harness" that can be applied to multiple models. This ensures that models are compared fairly under the same conditions and allows for tracking improvements from incremental changes. A critical rule is to ensure that any pre-processing or post-processing steps are part of the model itself, not the evaluation harness. Otherwise, a saved model will be incomplete and perform poorly in production.

11.1 Bias-Variance Decomposition

The expected squared error of a model can be decomposed into bias and variance.

$$\mathbb{E}_D[(\hat{y} - y)^2] = \underbrace{(\mathbb{E}[\hat{y}] - y)^2}_{\text{Bias}^2} + \underbrace{\mathbb{E}[(\mathbb{E}[\hat{y}] - \hat{y})^2]}_{\text{Variance}}$$

- **Bias:** A measure of systematic error; how far the average prediction is from the true value. High bias (underfitting) occurs when a model is too simple. To combat this, one can increase the number of features or use more complex models.
- **Variance:** A measure of how much a model's prediction changes for different training sets. High variance (overfitting) occurs when a model is too complex and captures noise in the training data. To reduce variance, one can use regularization, add more training data, or use techniques like bagging.

There is a fundamental trade-off between bias and variance: decreasing one often increases the other.

12 Measures of Predictive Performance

The choice of performance measure depends on the machine learning task.

- **Classification:** Accuracy, precision, recall, F1-score, AUC.
- **Regression:** Mean Squared Error (MSE), Root Mean Squared Error (RMSE), absolute error, R^2.
- **Predictive Probability Distributions:** Log-likelihood (cross-entropy), quadratic score (Brier score).

13 Learning Paradigms

13.1 Empirical Risk Minimization (ERM)

ERM is a foundational principle in machine learning. It involves defining a loss function $l(\text{true}, \text{predicted})$ and finding a model h from a hypothesis class $\mathcal{H}$ that minimizes the average loss over the training data.

The **true risk** (or generalization error) is the expected loss over the true data distribution, which is unknown:

$$R(h) = \mathbb{E}_{xy}[l(y, h(x))]$$

The **empirical risk** is the average loss on the training set:

$$R_n(h) = \frac{1}{n}\sum_{i=1}^{n} l(y_i, h(x_i))$$

The ERM principle states that we should choose the hypothesis $\hat{h}$ that minimizes the empirical risk:

$$\hat{h} = \underset{h\in\mathcal{H}}{\operatorname{argmin}} R_n(h)$$

The optimal predictor depends on the chosen loss function:

- **Quadratic Loss** ($l(y, \hat{y}) = (y - \hat{y})^2$): The predictor that minimizes this loss is the **mean**.
- **Absolute Loss** ($l(y, \hat{y}) = |y - \hat{y}|$): The optimal predictor is the **median**.
- **0-1 Loss** ($l(y, \hat{y}) = \mathbb{I}(y \neq \hat{y})$): The optimal predictor is the **mode** (majority class).
- **Log Loss** ($l(y, \hat{p}) = -\log \hat{p}(y)$): The optimal predictor is the **relative frequencies** of the classes.

13.1.1 ERM with Log Loss is equivalent to MLE

For a parametric model $p(y|\theta)$, using ERM with log loss means we want to find:

$$\hat{\theta}_{ERM} = \underset{\theta}{\operatorname{argmin}} \left[-\frac{1}{n}\sum_{i=1}^{n} \log p(y_i|\theta) \right]$$

This is equivalent to maximizing the term inside the brackets:

$$\hat{\theta}_{ERM} = \underset{\theta}{\operatorname{argmax}} \left[\frac{1}{n}\sum_{i=1}^{n} \log p(y_i|\theta) \right]$$

Since $\frac{1}{n}$ and log are monotonic, this is the same as maximizing the likelihood:

$$\hat{\theta}_{ERM} = \underset{\theta}{\operatorname{argmax}} \left[\log \prod_{i=1}^{n} p(y_i|\theta) \right] = \underset{\theta}{\operatorname{argmax}} \left[\prod_{i=1}^{n} p(y_i|\theta) \right] = \hat{\theta}_{MLE}$$

Thus, minimizing the empirical risk under log-loss is the same as finding the maximum likelihood estimate.

13.2 Approximation-Estimation Decomposition

The excess generalization error of a learned model $\hat{h}_n$ can be decomposed:

$$R(\hat{h}_n) - R(h^*) = \underbrace{(R(h_{opt}) - R(h^*))}_{\text{Approximation Error}} + \underbrace{(R(\hat{h}_n) - R(h_{opt}))}_{\text{Estimation Error}}$$

- **Approximation Error:** How close the best possible model in our hypothesis class (h_{opt}) is to the true best model (h^*). This is related to bias. A simple model like linear regression might have high approximation error if the true relationship is non-linear.
- **Estimation Error:** How close our learned model ($\hat{h}_n$) is to the best model in its class (h_{opt}). This error arises from having a finite amount of data. It is related to variance. A complex model like a deep neural network might have low approximation error but high estimation error, meaning it might not generalize well.

13.3 Consistency of ERM

Consistency is a desirable property of an estimator. For ERM, we consider two types of convergence as the sample size $n \to \infty$:

- The empirical risk of a fixed hypothesis h converges to its true risk: $R_n(h) \to R(h)$.
- The true risk of the hypothesis $\hat{h}_n$ selected by ERM converges to the true risk of the best hypothesis in the class: $R(\hat{h}_n) \to R(h_{opt})$.

This means that with enough data, ERM will select a model that is nearly as good as the best possible model within the chosen class. Generalization error bounds provide a more formal statement, often relating the true risk to the empirical risk via a complexity term.

13.4 Structural Risk Minimization (SRM)

SRM extends ERM by adding a penalty term for model complexity. This is a form of regularization.

$$\hat{h} = \underset{h\in\mathcal{H}}{\operatorname{argmin}} \left(R_n(h) + \lambda C(h)\right)$$

Here, $C(h)$ is a measure of the complexity of the model h (e.g., related to its VC-dimension), and λ is a regularization parameter that controls the trade-off between fitting the data and keeping the model simple. This approach aims to minimize an upper bound on the true risk, not just the empirical risk.

13.5 Vapnik-Chervonenkis (VC) Dimension

The VC-dimension of a hypothesis class $\mathcal{H}$, denoted $VC(\mathcal{H})$, is a measure of its capacity or complexity. It is defined as the size of the largest set of points that can be "shattered" by $\mathcal{H}$, meaning it can perfectly classify the points for every possible labeling. For example, the VC-dimension of linear classifiers in 2D is 3.

14 Choosing a Loss Function

The choice of loss function is a critical modeling decision.

- If possible, do not train a model with one loss function and evaluate it with another.
- Understand the consequences of your chosen loss function.
- When in doubt, using Log Loss (equivalent to MLE) is often a principled and robust choice.

15 Scoring Rules

When evaluating probabilistic forecasts, we need a way to measure how good a predicted probability distribution is, given an observed outcome. This is the role of a scoring rule. A scoring rule $S(p, y)$ assigns a score to a predicted distribution p when the outcome y is observed.

We want to maximize this score. Here are some examples:

- **Logarithmic Score (Log Loss):** $S_{\log}(p, y) = \log p(y)$
- **Quadratic (Brier) Score:** $S_{\text{quad}}(p, y) = 2p(y) - \sum_i p(y_i)^2$. This is an alternative formulation to the more common loss version, which is $L_{\text{quad}}(p, y) = \sum_i (p(y_i) - \delta_{iy})^2$ where δ_{iy} is 1 if y_i is the true class and 0 otherwise.
- **0-1 Score:** $S_{0-1}(p, y) = 1$ if $y = \text{mode}(p)$, and 0 otherwise. This rule only cares if the most likely predicted outcome was correct.

15.1 Proper Scoring Rules

A key concept for scoring rules is whether they are *proper*. A scoring rule is considered **proper** if a forecaster maximizes their expected score by reporting their true belief about the distribution. A rule is **strictly proper** if the true distribution is the *unique* distribution that maximizes the expected score.

15.1.1 Why This Matters

Using a non-proper scoring rule can lead to paradoxical situations. A model might achieve a better score by reporting a distribution different from what it has actually learned. For instance, if the true distribution for three classes is $p = [0.6, 0.2, 0.2]$, a model predicting this distribution should be rewarded most highly. However, with a non-proper rule, it might be possible to get a better score by predicting a biased distribution, like $[1, 0, 0]$. This means the scoring rule is not incentivizing honesty.

15.1.2 Examples of Scoring Rules

- **Log Score:** This is a strictly proper scoring rule. It heavily penalizes a model for assigning a very low probability to an event that then occurs (since $\log(p) \to -\infty$ as $p \to 0$). Because it is strictly proper, it is a very reliable metric for comparing probabilistic models.
- **Quadratic (Brier) Score:** This is also a strictly proper scoring rule. It is less sensitive to extreme errors than the log score.
- **0-1 Score (Accuracy):** This rule is **not** proper. Multiple different probability distributions can yield the same mode, and thus the same score. It doesn't differentiate between a model that predicts $[0.51, 0.49]$ and one that predicts $[0.99, 0.01]$ if the first class is correct.
- **Absolute Error and MSE (for the mean):** These are also **not** proper for evaluating full distributions, as they only depend on the mean of the distribution, not its full shape.

16 Estimating Generalization Error (Risk)

The ultimate goal of a model is to perform well on new, unseen data from the Data Generating Process (DGP). The model's performance on this future data is its **generalization error** or **risk**. Since the DGP is unknown, we cannot calculate this error directly. Instead, we must estimate it using the finite data we have.

This is analogous to estimating a population parameter, like the mean $\theta = E[x]$, using a sample statistic, like the sample mean $\hat{\theta} = \bar{x}$. We want our estimate of the risk to be unbiased and have low variance.

We have a learning algorithm A which takes a dataset D_n of size n and produces a hypothesis (a trained model) h_n. We want to estimate the true risk of this model:

$$R(h_n) = \mathbb{E}_{xy \sim \text{DGP}}[l(y, h_n(x))]$$

16.1 Methods for Estimating Risk

16.1.1 1. Independent Test Set (Hold-out Estimation)

The conceptually simplest method is to acquire a new, large, independent test set $D'_{m'}$ from the same DGP. We can then calculate the empirical risk on this test set:

$$\hat{R}(h_n) = \frac{1}{m'} \sum_{(x_i, y_i) \in D'_{m'}} l(y_i, h_n(x_i))$$

This estimate is an unbiased estimator of the true risk $R(h_n)$.

In practice, we rarely have the luxury of acquiring a separate test set. Instead, we split our available data D_n into a training set D_{train} and a test set D_{test}. We train the model on D_{train} to get h_{train} and then estimate its risk on D_{test}:

$$\hat{R}(h_{train}) = \frac{1}{|D_{test}|} \sum_{(x_i, y_i) \in D_{test}} l(y_i, h_{train}(x_i))$$

This estimate is an unbiased estimate of the risk of h_{train}, but it is a **pessimistic** estimate of the risk of the final model h_n (which would be trained on all of D_n). This is because h_{train} was trained on less data than h_n and is therefore expected to perform worse.

Another common mistake is to evaluate the model on the same data it was trained on. This gives the **training error**, which is an optimistically biased (i.e., too low) estimate of the true risk.

16.1.2 Estimating the Uncertainty of the Risk Estimate

A single risk estimate, like 83% accuracy, is just a point estimate. We also need to know the uncertainty of this estimate. The risk estimate $\hat{R}$ is an average of the losses l_i over the test set. By the Central Limit Theorem (CLT), we can approximate its variance:

$$\sigma^2_{\hat{R}} \approx \frac{\sigma^2_l}{m_{\text{test}}}$$

Since we don't know the true variance of the losses σ^2_l, we estimate it from the test sample itself:

$$\hat{\sigma}^2_l = \frac{1}{m_{\text{test}} - 1} \sum_{i=1}^{m_{\text{test}}} (l_i - \bar{l})^2$$

The standard error of our risk estimate is then $\text{SE}(\hat{R}) = \sqrt{\hat{\sigma}^2_l / m_{\text{test}}}$. The problem is that to get a low-variance (i.e., reliable) estimate, we need a large test set (m_{test}), which leaves less data for training a good model.

16.1.3 2. Cross-Validation (CV)

When data is limited, holding out a large portion for testing is wasteful. Cross-validation is a resampling method that uses the data more efficiently to get a more reliable estimate of generalization error.

- **k-Fold Cross-Validation:** The data is partitioned into k equal-sized folds. For each fold, the model is trained on the other $k - 1$ folds and tested on the held-out fold. The CV estimate of the risk is the average of the risks from the k test folds.
- **Leave-One-Out CV (LOOCV):** This is the extreme case of k-fold CV where $k = n$, the number of data points. It provides a nearly unbiased estimate of the risk but can be computationally very expensive and the resulting estimator can have high variance.

- **Monte Carlo CV (Repeated Random Sub-sampling):** This involves creating many random splits of the data into training and testing sets and averaging the results. Unlike k-fold CV, the test sets can overlap.

16.1.4 Practical Considerations for CV

- **Stratified CV:** When dealing with classification problems, especially with imbalanced classes, it is important to ensure that each fold has the same class distribution as the original dataset. This is called stratification and helps to reduce variance in the risk estimate.
- **Nested CV:** This is a more advanced technique used when model selection (e.g., hyperparameter tuning) must be performed as part of the evaluation. It involves an inner CV loop for hyperparameter tuning within each fold of an outer CV loop used for risk estimation. This provides an unbiased estimate of the performance of the model selection procedure itself.

17 Feature Selection and Model Regularization

Working with datasets that have many features presents several challenges:

- A higher risk of overfitting the model to the training data.
- Increased computational cost for training.
- The presence of redundant, correlated, or irrelevant features can degrade model performance and make the model difficult to interpret.

To address these issues, we use techniques to reduce the complexity of the model. These techniques often improve model quality by eliminating noise and can significantly increase interpretability. The main approaches are:

1. **Feature Selection:** Choosing a subset of the original features. This approach directly improves interpretability.
2. **Feature Transformation (Dimensionality Reduction):** Creating a new, smaller set of features by combining the original ones. Examples include PCA, MDS, and t-SNE.
3. **Regularization:** A technique that simplifies the model by penalizing complexity, which can implicitly perform feature selection. Examples include L1/L2 regularization or limiting the depth of a decision tree.

18 Feature Selection Approaches

The goal of feature selection is to find a small subset of features that are most useful for the modeling task. Since checking every possible combination of features is computationally infeasible , we use heuristic approaches, which can be categorized as follows:

- **Filter Methods:** These methods rank features based on their statistical properties, independent of the model being trained. They are computationally fast.
- **Wrapper Methods:** These methods use a specific machine learning model to evaluate the quality of different feature subsets. They are more computationally expensive but can lead to better performance for the chosen model.
- **Embedded Methods:** In these methods, feature selection is an integral part of the model training process. Regularization techniques are a prime example.

18.1 Filter Methods

Filter methods evaluate features individually or against the target variable.

18.1.1 Univariate Measures

These measures assess each feature independently.

- **Correlation (e.g., Pearson's r):** Measures the linear dependence between a continuous feature and a continuous target.
- **Mutual Information (Information Gain):** Measures the dependency between two variables, capable of capturing non-linear relationships. It can be used for discrete or continuous variables (though binning might be required for continuous ones).
- **Statistical Tests (e.g., ANOVA):** Used to check the statistical significance between a mixed set of feature and target types.

Mutual Information (MI) MI measures how much information about one variable is provided by another. It's defined in terms of entropy (H):

$$I(X,Y) = H(Y) - H(Y|X) = H(X) - H(X|Y)$$

If variables X and Y are independent, $I(X,Y) = 0$. If Y is a deterministic function of X, then $I(X,Y) = H(Y)$. However, a major drawback of univariate methods like MI is that they evaluate each feature in isolation and can miss feature interactions. For instance, two features might be useless on their own but highly predictive when combined.

18.1.2 Relief Algorithm

The Relief algorithm is a more advanced filter method that can detect feature interactions. It estimates the quality of features by how well they distinguish between instances that are near to each other. For each randomly selected instance X_i, Relief finds its nearest neighbor from the same class (nearest hit, H_i) and its nearest neighbor from a different class (nearest miss, M_i). It then updates the weight W for each feature A based on how much its value differs for the hit versus the miss:

$$W(A) \leftarrow W(A) - \text{diff}(A, X_i, H_i)^2 + \text{diff}(A, X_i, M_i)^2$$

The 'diff' function calculates the distance between the values of feature A for two instances. The idea is that a good feature should have similar values for instances of the same class (small difference with hit) and different values for instances of different classes (large difference with miss).

Concrete Example for Relief Let's assume we have a simple dataset with two features (F1, F2) and a binary class (0 or 1).

- Instance $X = (F1 = 3, F2 = 5, Class = 1)$
- Nearest Hit $H = (F1 = 4, F2 = 6, Class = 1)$
- Nearest Miss $M = (F1 = 8, F2 = 2, Class = 0)$

Let's initialize the weights for F1 and F2 to zero: $W(F1) = 0, W(F2) = 0$. We assume the 'diff' function is the simple absolute difference normalized by the range of the feature. Let's say range is 10 for both.

1. **Update for F1:**
   - $\text{diff}(F1, X, H) = |3 - 4|/10 = 0.1$
   - $\text{diff}(F1, X, M) = |3 - 8|/10 = 0.5$
   - $W(F1) \leftarrow 0 - (0.1)^2 + (0.5)^2 = -0.01 + 0.25 = 0.24$
2. **Update for F2:**

- $\text{diff}(F2, X, H) = |5 - 6|/10 = 0.1$
- $\text{diff}(F2, X, M) = |5 - 2|/10 = 0.3$
- $W(F2) \leftarrow 0 - (0.1)^2 + (0.3)^2 = -0.01 + 0.09 = 0.08$

After this one instance, F1 has a higher weight (0.24) than F2 (0.08), indicating it's more relevant. This process is repeated for many random instances to get a stable estimate of the feature weights.

18.2 Wrapper Methods

Wrapper methods wrap the feature selection process around a specific model, treating it as a black box that provides a quality score. The goal is to search the space of all possible feature subsets to find the one that results in the best model performance. Since this space is huge, heuristic search strategies are used, such as:

- **Forward Selection:** Start with an empty set and greedily add the feature that improves performance the most at each step.
- **Backward Elimination:** Start with all features and greedily remove the feature that results in the smallest performance drop.

19 Regularization

Regularization is a technique used to prevent overfitting by adding a penalty term to the model's objective function. This penalty discourages overly complex models by constraining the magnitude of the model's parameters.

19.1 L2 Regularization (Ridge Regression)

For linear regression, L2 regularization adds a penalty proportional to the sum of the squared coefficient values (β_i). The objective function becomes:

$$\min_{\beta} \sum_{i=1}^{n} (y_i - \beta^T x_i)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

The hyperparameter $\lambda \geq 0$ controls the strength of the regularization. As $\lambda \to \infty$, all coefficients are pushed towards zero. L2 regularization has a closed-form solution:

$$\hat{\beta}_{L2} = (X^T X + \lambda I)^{-1} X^T y$$

This penalty term ensures that the matrix is always invertible, solving a potential issue with ordinary least squares. It's crucial to normalize the features before applying regularization to ensure they are penalized fairly.

19.2 L1 Regularization (Lasso Regression)

L1 regularization adds a penalty proportional to the sum of the absolute values of the coefficients:

$$\min_{\beta} \sum_{i=1}^{n} (y_i - \beta^T x_i)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

L1 regularization has a powerful property: it can force some feature coefficients to be exactly zero, effectively performing feature selection. This leads to sparse models. Unlike Ridge, Lasso has no closed-form solution and must be solved with numerical optimization methods.

19.3 Geometric Interpretation

The difference between L1 and L2 can be visualized. Regularization is equivalent to constraining the coefficient vector β to lie within a certain region.

- **L2 (Ridge):** The constraint region is a circle (in 2D) or a hypersphere. The solution is the point where the elliptical contours of the sum-of-squares error term touch this circle. This is unlikely to happen at an axis, so coefficients are shrunk but rarely to zero.
- **L1 (Lasso):** The constraint region is a diamond (in 2D) or a polyhedron. The elliptical contours are much more likely to hit one of the sharp corners of the diamond, which lie on the axes. When the solution lies on an axis, one of the feature coefficients is zero.

19.4 Bayesian Interpretation of Regularization

Regularization can be interpreted from a Bayesian perspective as placing a prior distribution on the model parameters. The regularized solution corresponds to the Maximum A Posteriori (MAP) estimate.

$$\underbrace{p(\beta|D)}_{\text{Posterior}} \propto \underbrace{p(D|\beta)}_{\text{Likelihood}} \cdot \underbrace{p(\beta)}_{\text{Prior}}$$

Maximizing the log-posterior is equivalent to minimizing the negative log-posterior:

$$\min_{\beta} [-\log(p(D|\beta)) - \log(p(\beta))]$$

- The negative log-likelihood corresponds to the error term (e.g., sum of squared errors).
- The negative log-prior corresponds to the regularization penalty term.
- **L2 (Ridge):** This is equivalent to placing a Gaussian (Normal) prior on the coefficients: $\beta_j \sim \mathcal{N}(0, \tau^2)$.
- **L1 (Lasso):** This is equivalent to placing a Laplace prior on the coefficients: $p(\beta_j) \propto \exp(-|\beta_j|/b)$. The sharper peak of the Laplace distribution at zero places more mass on coefficients being exactly zero, leading to sparsity.

20 Generalizing Linear Regression

Standard linear regression assumes the response variable y is continuous and the errors ϵ_i are normally distributed: $y_i = \beta^T x_i + \epsilon_i$, where $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$.

However, this model is not suitable for classification problems where the outcome is binary (0 or 1). A naive approach of fitting a line and setting a threshold has several problems:

- The linear predictor $\beta^T x$ can produce values outside the [0, 1] range required for probabilities.
- It is not obvious how to convert the output to a valid probability.
- The model can be heavily skewed by outliers and perform poorly on imbalanced data.

20.1 Logistic Regression

To address these issues, we use a function that maps the output of a linear model to the (0, 1) interval. The standard choice for this is the **logistic function** (or standard sigmoid):

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

This function takes any real number z and maps it to a value between 0 and 1. A useful property of the logistic function is its simple derivative:

$$\frac{d}{dz}\sigma(z) = \sigma(z)(1-\sigma(z))$$

In logistic regression, we model the probability of a positive class as:

$$p(y=1|x;\beta) = h_\beta(x) = \sigma(\beta^T x) = \frac{1}{1+e^{-\beta^T x}}$$

Since there are only two outcomes, the probability of the negative class is $p(y=0|x;\beta) = 1 - h_\beta(x)$. We can write this compactly for a single observation (x_i, y_i) where $y_i \in \{0,1\}$:

$$p(y_i|x_i;\beta) = h_\beta(x_i)^{y_i}(1-h_\beta(x_i))^{1-y_i}$$

This is the probability mass function of a Bernoulli distribution.

20.1.1 Parameter Estimation via Maximum Likelihood (MLE)

To find the optimal parameters β, we maximize the likelihood of observing our entire dataset. Assuming the observations are independent, the likelihood is the product of the individual probabilities:

$$L(\beta) = p(\mathbf{y}|X;\beta) = \prod_{i=1}^{n} p(y_i|x_i;\beta) = \prod_{i=1}^{n} h_\beta(x_i)^{y_i}(1-h_\beta(x_i))^{1-y_i}$$

It is easier to work with the log-likelihood, $l(\beta) = \log L(\beta)$:

$$l(\beta) = \sum_{i=1}^{n} [y_i \log h_\beta(x_i) + (1-y_i)\log(1-h_\beta(x_i))]$$

This is also known as the negative of the binary cross-entropy loss. Substituting $h_\beta(x_i) = \sigma(\beta^T x_i) = \frac{e^{\beta^T x_i}}{1+e^{\beta^T x_i}}$ and $1-h_\beta(x_i) = \frac{1}{1+e^{\beta^T x_i}}$ leads to:

$$\begin{aligned} l(\beta) &= \sum_{i=1}^{n} \left[y_i \log\left(\frac{e^{\beta^T x_i}}{1+e^{\beta^T x_i}}\right) + (1-y_i)\log\left(\frac{1}{1+e^{\beta^T x_i}}\right)\right] \\ &= \sum_{i=1}^{n} \left[y_i(\beta^T x_i - \log(1+e^{\beta^T x_i})) - (1-y_i)\log(1+e^{\beta^T x_i})\right] \\ &= \sum_{i=1}^{n} \left[y_i \beta^T x_i - \log(1+e^{\beta^T x_i})\right] \end{aligned}$$

Unlike linear regression, there is no closed-form solution for β. We must use an iterative optimization algorithm like gradient descent. A key property is that the log-likelihood function for logistic regression is **concave**, which guarantees that gradient descent will find the global maximum.

20.1.2 Interpretation of Coefficients

The logistic regression model can be rearranged to understand the meaning of the coefficients:

$$\frac{p(x)}{1-p(x)} = e^{\beta^T x} \implies \log\left(\frac{p(x)}{1-p(x)}\right) = \beta^T x = \beta_0 + \beta_1 x_1 + \dots$$

The term $\log(p/(1-p))$ is the **log-odds** or **logit**. This shows that a one-unit increase in a feature x_j, holding all other features constant, changes the log-odds of the outcome by β_j. Equivalently, the odds are multiplied by a factor of e^{β_j}.

21 Generalized Linear Models (GLMs)

GLMs extend linear regression to handle response variables with different distributions. A GLM has three components:

1. **Random Component:** Specifies the probability distribution of the response variable y, which must be a member of the exponential family (e.g., Normal, Poisson, Bernoulli, Gamma).
2. **Systematic Component:** A linear predictor, which is a linear combination of the features: $\eta = \beta^T x$.
3. **Link Function (g):** A function that links the expected value of the response, $\mu = E[y]$, to the linear predictor: $g(\mu) = \eta$.

The general form of a distribution in the exponential family is:

$$p(y|\theta,\phi) = \exp\left(\frac{y\theta - b(\theta)}{a(\phi)} + c(y,\phi)\right)$$

For these distributions, it can be shown that $E[y] = \mu = b'(\theta)$.

The link function's role is to map the range of μ (e.g., $(0,\infty)$ for Poisson, $(0,1)$ for Bernoulli) to the entire real line $\mathbb{R}$. The **canonical link** is a special choice where $g(\mu) = \theta$.

21.1 Examples of GLMs

22 Models for Categorical Dependent Variables

22.1 Direct Extension (Softmax Regression)

This approach is a direct generalization of binary logistic regression to handle multiple classes.

For a given feature vector $x \in \mathbb{R}^k$, the model first computes a linear score z_j for each class j from $1, \dots, m$:

$$z_j = w_j^T x + b_j$$

Here, $w_j \in \mathbb{R}^k$ is the weight vector and b_j is the bias term for class j.

To ensure that the model is identifiable (i.e., has a unique solution), one class is typically designated as a reference class. For this reference class, the parameters are fixed, usually to zero (e.g., $w_m = \vec{0}$ and $b_m = 0$).

The scores are then converted into probabilities using the **softmax function**, which normalizes the scores so that they sum to one:

$$P(y=j|x) = \frac{\exp(z_j)}{\sum_{k=1}^{m}\exp(z_k)}$$

The total number of parameters to be learned for this model is $(m-1)(k+1)$, accounting for the k weights and 1 bias for each of the $m-1$ non-reference classes.

22.2 One-vs-Rest (OvR) Decomposition

The One-vs-Rest (also known as One-vs-All) strategy decomposes the single multi-class problem into multiple binary classification problems.

For a problem with m classes, this method involves training m independent binary classifiers. For each class j, a binary logistic regression model is trained to distinguish class j (treated as the positive class) from all other $m-1$ classes combined (treated as the negative class).

After training, to make a prediction for a new input, all m classifiers produce an output probability. Since these probabilities are not guaranteed to sum to one, the outputs are scaled to obtain normalized class probabilities:

$$P(y=j|x) = \frac{p_j(y=j|x)}{\sum_{l=1}^{m} p_l(y=l|x)}$$

where $p_j(y=j|x)$ is the probability output from the binary classifier trained for class j.

The total number of parameters to be learned for this model is $(m)(k+1)$, as each of the m classes has k weights and 1 bias term.

22.3 Ordinal Regression

This model is used when the categorical outcome has a natural ordering (e.g., 'small', 'medium', 'large'). Instead of modeling the probability of each class, ordinal models typically model the cumulative probability, $p(y \leq j)$. The most common is the **proportional odds model** or **cumulative logit model**.

It assumes a single linear score $u_i = \beta^T x_i$ and a series of ordered thresholds or cut-points $\{b_1, b_2, \ldots, b_{m-1}\}$. The cumulative probability is then:

$$p(y_i \leq j) = F(b_j - u_i)$$

where F is a cumulative distribution function (CDF), typically the logistic function. The probability of being in a specific category j is the difference between two cumulative probabilities:

$$p(y_i = j) = p(y_i \leq j) - p(y_i \leq j-1) = F(b_j - u_i) - F(b_{j-1} - u_i)$$

This model is more parsimonious than multinomial regression, as it only estimates one set of feature coefficients β and $m-1$ cut-points, making it useful for smaller datasets. The cut-points must satisfy the constraint $b_1 < b_2 < \cdots < b_{m-1}$.

The total number of parameters to be learned for this model is $k + (m-2)$, where k is the number of features.

23 Boosting

Boosting is a powerful ensemble method that, similar to bagging, combines multiple models to produce a single, superior predictive model. However, its approach is fundamentally different.

- **Bagging** builds multiple independent models in parallel on different bootstrap samples of the data and averages their predictions.
- **Boosting** builds models sequentially, where each new model is trained to correct the errors made by the previous ones.

Boosting is designed to work with **weak learners**—models that are only slightly better than random guessing (e.g., a decision tree with a single split, called a "stump"). The goal is to combine many of these weak learners, which typically have high bias, into a single strong learner with low bias. Using strong learners with boosting often doesn't work well and can lead to overfitting.

24 The AdaBoost Algorithm

AdaBoost (Adaptive Boosting) is the archetypal boosting algorithm, designed for binary classification with labels $y \in \{-1, 1\}$. The core idea is to iteratively re-weight the data samples. In each iteration, the algorithm places more weight on instances that were misclassified by the previous learner, forcing the new learner to focus on these "hard" examples.

The algorithm proceeds as follows:

1. **Initialization:** Assign an equal weight to each of the n training samples: $w_i = 1/n$ for $i = 1, \ldots, n$.
2. **Iterate** for $m = 1, \ldots, M$:
   (a) Fit a weak classifier $G_m(x)$ to the training data using the weights w_i.
   (b) Compute the weighted error of the classifier:
   $$\text{err}_m = \frac{\sum_{i=1}^n w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^n w_i}$$
   where $I(\cdot)$ is the indicator function.
   (c) Compute the classifier's weight (or importance factor):
   $$\alpha_m = \frac{1}{2} \log\left(\frac{1 - \text{err}_m}{\text{err}_m}\right)$$
   This value is large and positive for accurate classifiers and can be negative for classifiers that perform worse than random.
   (d) Update the sample weights:
   $$w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))]$$
   After this step, re-normalize the weights so they sum to 1. This update increases the weight of misclassified samples.
3. **Final Prediction:** The final prediction is a weighted majority vote of all the weak classifiers:
   $$G(x) = \text{sign}\left(\sum_{m=1}^M \alpha_m G_m(x)\right)$$

25 A General View: Forward Stagewise Additive Modeling (FSAM)

AdaBoost can be understood as a special case of a more general framework called Forward Stagewise Additive Modeling. The goal of FSAM is to build an additive model of the form:

$$f(x) = \sum_{m=1}^M \beta_m b(x; \gamma_m)$$

where $b(x; \gamma_m)$ are basis functions (our weak learners). The objective is to minimize a loss function $L(y, f(x))$ over the training data.

The "stagewise" approach builds the model iteratively. At each step m, we add a new weak learner to the ensemble without changing the parameters of the learners already added.

1. Initialize $f_0(x) = 0$.
2. For $m = 1, \ldots, M$:
   (a) Find the next weak learner G_m and its coefficient β_m by solving:
   $$(\beta_m, G_m) = \underset{\beta, G}{\text{argmin}} \sum_{i=1}^n L(y_i, f_{m-1}(x_i) + \beta G(x_i))$$
   (b) Update the model: $f_m(x) = f_{m-1}(x) + \beta_m G_m(x)$.

25.1 Deriving AdaBoost from FSAM

AdaBoost is equivalent to FSAM when using the **exponential loss function**: $L(y, f(x)) = e^{-yf(x)}$.

At step m, we need to solve:

$$(\beta_m, G_m) = \underset{\beta, G}{\text{argmin}} \sum_{i=1}^n e^{-y_i[f_{m-1}(x_i) + \beta G(x_i)]}$$

We can rewrite the sum as:

$$\sum_{i=1}^n e^{-y_i f_{m-1}(x_i)} e^{-y_i \beta G(x_i)}$$

The term $e^{-y_i f_{m-1}(x_i)}$ is independent of β and G. We can treat it as a weight $w_i^{(m)}$ for each sample. The problem becomes:

$$(\beta_m, G_m) = \underset{\beta, G}{\text{argmin}} \sum_{i=1}^n w_i^{(m)} e^{-y_i \beta G(x_i)}$$

Since $y_i, G(x_i) \in \{-1, 1\}$, the term $y_i G(x_i)$ is 1 if the classification is correct and -1 if it's incorrect. We can split the sum:

$$\sum_{y_i = G(x_i)} w_i^{(m)} e^{-\beta} + \sum_{y_i \neq G(x_i)} w_i^{(m)} e^{\beta}$$

This can be rewritten using the total weight of misclassified points, $\text{err}_m = \sum w_i^{(m)} I(y_i \neq G(x_i))$:

$$= (1 - \text{err}_m) e^{-\beta} + \text{err}_m e^{\beta}$$

Now we solve for G_m and β_m separately.

1. **Find G_m:** For any fixed $\beta > 0$, the term e^{β} is larger than $e^{-\beta}$. To minimize the expression, we must choose the function G that minimizes its coefficient, which is the weighted error err_m. Thus, the optimal weak learner is:

$$G_m = \underset{G}{\text{argmin}} \sum_{i=1}^{n} w_i^{(m)} I(y_i \neq G(x_i))$$

2. **Find β_m:** With G_m (and thus err_m) fixed, we find the optimal β_m by taking the derivative of the expression with respect to β and setting it to zero:

$$\frac{\partial}{\partial \beta}[(1 - \text{err}_m) e^{-\beta} + \text{err}_m e^{\beta}] = -(1 - \text{err}_m) e^{-\beta} + \text{err}_m e^{\beta} = 0$$

$$e^{2\beta} = \frac{1 - \text{err}_m}{\text{err}_m} \implies \beta_m = \frac{1}{2} \log\left(\frac{1 - \text{err}_m}{\text{err}_m}\right)$$

This is exactly the formula for α_m in AdaBoost. The weight update rule in FSAM is $w_i^{(m+1)} = w_i^{(m)} e^{-y_i \beta_m G_m(x_i)}$. This is also equivalent to the AdaBoost update, since $-y_i G_m(x_i)$ is -1 for correct predictions and $+1$ for incorrect ones, making the exponent $\pm\beta_m$, which matches AdaBoost's $\alpha_m \cdot I(y_i \neq G_m(x_i))$ if we adjust for the sign convention.

26 Gradient Boosting

Gradient Boosting is a further generalization of boosting that can be used with any differentiable loss function, not just the exponential loss. It is particularly useful for regression problems.

The core idea is to reframe boosting as a gradient descent procedure in function space. At each step, instead of fitting a new learner to re-weighted data, we fit a learner to the **negative gradient** of the loss function with respect to the current model's prediction. These negative gradients are called **pseudo-residuals**.

The algorithm is as follows:

1. Initialize the model with a constant value, typically the mean of the target variable: $f_0(x) = \underset{\gamma}{\text{argmin}} \sum_{i=1}^{n} L(y_i, \gamma)$.

2. For $m = 1, \ldots, M$:

   (a) **Compute pseudo-residuals:** For each sample i, calculate the negative gradient of the loss function:

$$r_{im} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f = f_{m-1}}$$

   (b) **Fit a weak learner:** Train a weak learner $h_m(x)$ to predict the pseudo-residuals $\{(x_i, r_{im})\}_{i=1}^{n}$.

   (c) **Find optimal step size:** Find the best coefficient γ_m for the new learner by performing a line search to minimize the overall loss:

$$\gamma_m = \underset{\gamma}{\text{argmin}} \sum_{i=1}^{n} L(y_i, f_{m-1}(x_i) + \gamma h_m(x_i))$$

   (d) **Update the model:** Add the new weak learner to the ensemble:

$$f_m(x) = f_{m-1}(x) + \gamma_m h_m(x)$$

For example, in regression with squared error loss $L(y, f(x)) = \frac{1}{2}(y - f(x))^2$, the negative gradient is simply the ordinary residual, $r_i = y_i - f(x_i)$. In this case, gradient boosting iteratively fits new models to the residuals of the previous ensemble.

27 Artificial Neural Networks (ANNs)

The core idea behind Artificial Neural Networks is to learn features or representations from the data in an automated way. This is done by applying a series of linear combinations followed by non-linear transformations, repeated over multiple layers. The heavy reliance on matrix multiplications is why GPUs, which excel at parallelized matrix operations, are so crucial for training modern neural networks.

27.1 Motivation

Traditional "shallow" machine learning methods like kernel methods implicitly transform data into a higher-dimensional space where it might become linearly separable:

$$y(x) = w^T \phi(x) + b$$

The drawback is that computing the kernel matrix can be computationally expensive, often scaling quadratically with the number of data points.

Neural networks, in contrast, learn the transformation $\phi(x)$ directly as part of the model:

$$\phi(x) = \sigma(Ux + c)$$

This approach scales linearly with the number of data points, making it more suitable for large datasets. The network learns to warp the feature space to make the problem easier, for example, by making non-linearly separable data linearly separable in a transformed space.

27.2 Mathematical Model of a Neuron

A single artificial neuron is a simple computational unit. It receives multiple inputs $x_1, x_2, \ldots, x_k$, computes a weighted sum, adds a bias b, and then applies a non-linear activation function σ.

$$z = \sum_{i=1}^{k} w_i x_i + b \quad \text{or in vector form,} \quad z = w^T x + b$$

$$y = \sigma(z)$$

The output y is then passed as an input to other neurons in the next layer.

28 Network Architecture

A neural network is formed by organizing these neurons into layers. A typical feed-forward network consists of:

- An **input layer** that receives the raw data.
- One or more **hidden layers** where intermediate computations and feature learning occur.
- An **output layer** that produces the final prediction.

28.1 Activation Functions

The activation function σ introduces non-linearity into the model, which is crucial for learning complex patterns. Without it, a multilayer network would be equivalent to a single linear model. Common activation functions include:

- **Sigmoid:** $\sigma(z) = \frac{1}{1+e^{-z}}$. Maps values to the (0, 1) range.

- **Hyperbolic Tangent (Tanh):** $\sigma(z) = \tanh(z)$. Maps values to the (-1, 1) range.
- **Rectified Linear Unit (ReLU):** $\sigma(z) = \max(0, z)$. A very popular choice due to its simplicity and effectiveness in combating the vanishing gradient problem.

28.2 Output Layer

The structure of the output layer depends on the task:

- **Regression:** The output layer typically has a single neuron with a linear activation function to produce a continuous value.
- **Classification:** For multi-class classification, a **softmax** layer is often used. The softmax function takes the raw outputs (logits) for each class and converts them into a probability distribution that sums to 1:

$$P(y = j|x) = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

28.3 The Power of ANNs: Universal Approximation

A key theoretical result is the Universal Approximation Theorem. It states that a neural network with a single hidden layer containing a finite number of neurons (using a non-linear activation function like ReLU) can approximate any continuous function to an arbitrary degree of accuracy. A ReLU network, for instance, creates a piecewise linear approximation of the target function.

$$h(x) = \sum_d w_d \max(0, u_d x + c_d)$$

While one layer is theoretically sufficient, it might need to be impractically large. In practice, deeper networks (with multiple layers) can often learn the same function more efficiently (with fewer total neurons) and are easier to train.

29 Training Neural Networks: Backpropagation

The weights of a neural network are learned by minimizing a cost function J (e.g., mean squared error) using an optimization algorithm like gradient descent. The core of this process is calculating the gradient of the cost function with respect to every weight and bias in the network. This is done efficiently using the **backpropagation** algorithm, which is essentially a systematic application of the chain rule of calculus.

Let's denote the activation of a layer as $a^{(l)} = \sigma(z^{(l)})$ and the pre-activation as $z^{(l)} = W^{(l)}a^{(l-1)} + b^{(l)}$. For the final layer L, the partial derivative of the cost J with respect to a weight $w_{jk}^{(L)}$ in that layer is:

$$\frac{\partial J}{\partial w_{jk}^{(L)}} = \frac{\partial J}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial w_{jk}^{(L)}}$$

The "error" from the output layer is then propagated backward to the preceding layers. For a weight $w_{ij}^{(l-1)}$ in a hidden layer, its gradient depends on the gradients of all the neurons it connects to in the next layer:

$$\frac{\partial J}{\partial a_k^{(l-1)}} = \sum_j \frac{\partial J}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial a_k^{(l-1)}}$$

This process is applied recursively from the last layer to the first. Modern deep learning libraries like TensorFlow and PyTorch automate this entire gradient calculation process, allowing developers to define complex network architectures without manually deriving the gradients.

29.1 Optimization

The gradients calculated via backpropagation are used to update the model's weights.

- **Stochastic Gradient Descent (SGD):** Instead of calculating the gradient over the entire dataset (which is computationally expensive), SGD approximates the gradient using a single data point or a small "mini-batch" of points at each step.
- **Learning Rate:** The learning rate is a hyperparameter that controls the step size during optimization. It is often adjusted (decayed) over the course of training.
- **Initialization:** Weights are typically initialized to small random values from a normal distribution to break symmetry and allow different neurons to learn different features.

While SGD can be noisy, it is computationally efficient and can help the model escape shallow local minima in the loss landscape.

30 Training Neural Networks: Practical Considerations

30.1 Initialization

How we initialize the weights of a neural network is crucial for effective training. A bad initialization can lead to slow convergence or getting stuck in poor local minima.

- **Zero Initialization:** Initializing all weights to zero is a mistake. Since all neurons in a layer would have the same inputs and weights, they would compute the same gradients and undergo the same updates. The network would fail to learn, as no symmetry is broken.
- **Random Initialization:** We must initialize weights randomly. However, the scale of these random values matters. If weights are too large, the pre-activation values ($z = w^T x + b$) can be very large, pushing activation functions like sigmoid or tanh into their flat, saturated regions. In these regions, the gradients are close to zero, effectively stalling the learning process. This is known as the **vanishing gradient problem** and leads to "dead neurons" that do not update.

To combat this, specialized initialization schemes are used to ensure that the variance of the outputs of a layer remains similar to the variance of its inputs.

- **Xavier/Glorot Initialization:** Used for activation functions like sigmoid and tanh. It draws weights from a distribution with zero mean and a variance of $1/n_{in}$ (or $2/(n_{in} + n_{out})$).
  - Normal: $W \sim \mathcal{N}(0, \sqrt{2/(n_{in} + n_{out})})$
  - Uniform: $W \sim U(-\sqrt{6/(n_{in} + n_{out})}, +\sqrt{6/(n_{in} + n_{out})})$
- **He Initialization:** Designed for the ReLU activation function. It uses a variance of $2/n_{in}$.

30.2 Optimization

The process of finding the optimal weights is guided by an optimizer.

- **Mini-batch Gradient Descent:** Instead of using the entire dataset (batch GD) or a single sample (SGD), we compute the gradient on a small "mini-batch" of data. This provides a good balance between the stability of batch GD and the efficiency of SGD. Performance is typically evaluated on a separate validation set periodically (e.g., every X batches).
- **Learning Rate Schedules:** The learning rate is arguably the most important hyperparameter. A fixed learning rate is often suboptimal. It's common practice to use a schedule:

  - **Step Decay:** Start with a high learning rate and decrease it by a factor at specific epochs.
  - **Continuous Decay:** Gradually decrease the learning rate over time.

- **Advanced Optimizers:** Modern optimizers adapt the learning rate for each parameter individually.
  - **Momentum:** Accumulates a "velocity" of past gradients to accelerate descent in consistent directions and dampen oscillations.
  - **AdaGrad:** Adapts the learning rate for each parameter, decreasing it for parameters with frequently occurring features.
  - **RMSProp:** A modification of AdaGrad that introduces a "forgetting" factor to prevent the learning rate from decaying to zero too quickly.
  - **Adam (Adaptive Moment Estimation):** The most common optimizer in practice. It combines the ideas of Momentum and RMSProp.

30.3 Efficiency

Training large models requires significant computational resources. Efficiency can be improved by using lower-precision numerical formats for weights and activations.

- **FP32 (Single Precision):** The standard 32-bit floating-point format.
- **FP16 (Half Precision):** Uses 16 bits. This reduces memory usage, allowing for larger models or batch sizes, and can speed up computation on modern GPUs with specialized hardware (Tensor Cores).
- **Mixed Precision Training:** A technique that uses FP16 for most operations to gain speed and memory benefits, while strategically using FP32 for certain operations to maintain numerical stability.

30.4 Regularization

These are techniques used to prevent overfitting.

- **Penalty-based:** L1 and L2 regularization (also known as weight decay) add a penalty to the loss function based on the magnitude of the weights.
- **Ensembling:** Training multiple models with different initializations and averaging their predictions.
- **Dropout:** During training, randomly sets a fraction of neuron activations to zero at each update step. This prevents neurons from co-adapting too much and forces the network to learn more robust features.
- **Batch Normalization:** Normalizes the activations within a mini-batch to have a mean of 0 and a variance of 1. It helps stabilize and accelerate training by reducing "internal covariate shift." A typical layer sequence becomes: Weight Layer $\rightarrow$ Batch Norm Layer $\rightarrow$ Activation Layer.

31 Neural Network Architectures

31.1 Convolutional Neural Networks (CNNs)

CNNs are the standard architecture for computer vision tasks. They are designed to be invariant to translation and scale by using shared weights and local connectivity.

- **Convolutional Layer:** Instead of a fully connected layer, a convolutional layer uses a set of small filters (kernels) that slide across the input image (or feature map). Each filter is specialized to detect a specific pattern (e.g., an edge, a corner). The output is a set of new feature maps.
- **Pooling Layer:** This layer downsamples the feature maps, reducing their spatial dimensions. This helps to make the representation more robust to small shifts and distortions. Common pooling operations are Max Pooling and Average Pooling.
- **Typical Structure:** A typical CNN consists of several blocks of [Conv $\rightarrow$ ReLU $\rightarrow$ Pooling], followed by one or more fully connected layers at the end for classification.

31.2 Transformer Architectures (e.g., GPT)

Transformers have become the dominant architecture for NLP and are increasingly used in other domains. They were designed to handle sequential data without the sequential processing limitations of Recurrent Neural Networks (RNNs).

- **Language Modeling:** The fundamental task is often language modeling: predicting the next token (word or sub-word) in a sequence given all previous tokens. The loss function is typically cross-entropy (minimized by maximizing the log-likelihood of the data).
- **Pre-training and Fine-tuning:** Large models like GPT (Generative Pre-trained Transformer) are first pre-trained on a massive amount of text data in a self-supervised way. They are then fine-tuned on a smaller, task-specific dataset (e.g., for question answering).
- **Self-Attention:** The core innovation of the Transformer is the self-attention mechanism. This allows the model to weigh the importance of all other tokens in the input sequence when processing a specific token. It addresses the issue that the meaning of a word can depend on distant words in the sequence, not just its immediate neighbors.

31.2.1 The Self-Attention Mechanism

For each token in the input sequence, we create three vectors: a **Query** (Q), a **Key** (K), and a **Value** (V).

1. A "relevance" score is computed between the Query vector of the current token and the Key vector of every other token, typically using a dot product.
2. These scores are scaled (by $\sqrt{d_k}$) and passed through a softmax function to get attention weights. These weights represent how much attention the current token should pay to every other token.
3. The final output for the token is a weighted average of all the Value vectors, using the attention weights.

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

This mechanism allows for rich communication between all parts of the sequence. **Multi-Head Attention** applies this process multiple times in parallel with different learned transformations of Q, K, and V, allowing the model to focus on different types of relationships simultaneously.

A standard Transformer block consists of a multi-head attention layer followed by a feed-forward network, with residual connections and layer normalization applied after each sub-layer.

32 Motivation: From Linear Models to Kernels

Linear models, like logistic regression, are defined by a linear decision boundary:

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

While simple, these models can be made much more powerful. If the data is not linearly separable in its original feature space, we can often make it separable by creating new, non-linear features from the original ones. For example, we can add polynomial terms:

$$z(x) = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1 x_2 + w_4 x_1^2 + w_5 x_2^2$$

This is equivalent to first mapping the data into a higher-dimensional feature space using a function Φ, and then applying a linear model in that new space.

$$z(x) = w^T \Phi(x)$$

The problem with this explicit feature expansion is that the number of new features can grow exponentially, making it computationally expensive or even infeasible.

33 The Kernel Trick

The kernel trick provides an elegant solution for non-linear problems. Many machine learning algorithms can be formulated to depend only on the inner products (dot products) of feature vectors, e.g., $\langle \Phi(\mathbf{x}_i), \Phi(\mathbf{x}_j) \rangle$.

A **kernel** is a function $K(\mathbf{x}_i, \mathbf{x}_j)$ that computes this inner product in a high-dimensional feature space directly from the original vectors, without ever explicitly performing the feature mapping Φ.

$$K(\mathbf{x}_i, \mathbf{x}_j) = \langle \Phi(\mathbf{x}_i), \Phi(\mathbf{x}_j) \rangle$$

This allows us to work with very high or even infinite-dimensional feature spaces efficiently. The **Representer Theorem** states that for a wide range of models, the optimal solution $f(\mathbf{x})$ can be written as a weighted combination of kernel evaluations against the training data points:

$$f(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i K(\mathbf{x}, \mathbf{x}_i)$$

33.1 A Concrete Example: The Polynomial Kernel

To demonstrate how a kernel function works as a computational shortcut, consider original vectors in $\mathbb{R}^2$, such as $\mathbf{x}_i = (x_{i1}, x_{i2})$ and $\mathbf{x}_j = (x_{j1}, x_{j2})$.

Let's define a feature map Φ that takes our 2D data into a 6D space:

$$\Phi(\mathbf{x}) = (x_1^2, x_2^2, \sqrt{2}x_1x_2, \sqrt{2}x_1, \sqrt{2}x_2, 1)$$

33.1.1 Method 1: The Hard Way (Explicit High-Dimensional Calculation)

First, we apply the map Φ to our vectors and then compute their dot product in $\mathbb{R}^6$.

$$\begin{aligned}\langle \Phi(\mathbf{x}_i), \Phi(\mathbf{x}_j) \rangle &= (x_{i1}^2)(x_{j1}^2) + (x_{i2}^2)(x_{j2}^2) + (\sqrt{2}x_{i1}x_{i2})(\sqrt{2}x_{j1}x_{j2}) \\ &\quad + (\sqrt{2}x_{i1})(\sqrt{2}x_{j1}) + (\sqrt{2}x_{i2})(\sqrt{2}x_{j2}) + (1)(1) \\ &= x_{i1}^2x_{j1}^2 + x_{i2}^2x_{j2}^2 + 2x_{i1}x_{i2}x_{j1}x_{j2} \\ &\quad + 2x_{i1}x_{j1} + 2x_{i2}x_{j2} + 1\end{aligned}$$

This calculation is cumbersome and computationally expensive as it requires explicit transformation into $\mathbb{R}^6$.

33.1.2 Method 2: The Kernel Trick

For the specific feature map Φ above, there is an equivalent kernel function, the polynomial kernel of degree 2:

$$K(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i^T \mathbf{x}_j + 1)^2$$

Let's evaluate this using only the original 2D vectors. The dot product in the original space is $\mathbf{x}_i^T \mathbf{x}_j = x_{i1}x_{j1} + x_{i2}x_{j2}$.

$$\begin{aligned}K(\mathbf{x}_i, \mathbf{x}_j) &= ((x_{i1}x_{j1} + x_{i2}x_{j2}) + 1)^2 \\ &= (x_{i1}x_{j1})^2 + (x_{i2}^2x_{j2}^2) + 1^2 + 2(x_{i1}x_{j1})(x_{i2}x_{j2}) \\ &\quad + 2(x_{i1}x_{j1})(1) + 2(x_{i2}x_{j2})(1) \\ &= x_{i1}^2x_{j1}^2 + x_{i2}^2x_{j2}^2 + 2x_{i1}x_{j1}x_{i2}x_{j2} \\ &\quad + 2x_{i1}x_{j1} + 2x_{i2}x_{j2} + 1\end{aligned}$$

Both methods yield the exact same result. The kernel function $K(\mathbf{x}_i, \mathbf{x}_j)$ provides the result of the 6-dimensional dot product by performing a much simpler calculation in the original 2D space. This allows us to leverage the power of high-dimensional feature spaces without ever paying the computational cost of explicit transformation.

34 Support Vector Machines (SVMs)

SVMs are a classic example of a model that benefits greatly from kernels.

34.1 Hard-Margin SVM (Linearly Separable Case)

The goal of an SVM is to find the hyperplane that separates the two classes with the maximum possible margin. The margin is the distance from the decision boundary to the closest data point from either class.

For a data point x_i with label $y_i \in \{-1, 1\}$, the decision boundary is $w^T x + b = 0$. We require all data points to be on or outside the margin, which we can set to have a width of 1 on each side:

$$y_i(w^T x_i + b) \geq 1 \quad \forall i$$

The total width of the margin is $2/||w||$. To maximize the margin, we must minimize $||w||$, which is equivalent to minimizing $\frac{1}{2}||w||^2$. The optimization problem is:

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

34.1.1 The Dual Formulation

This constrained optimization problem can be solved by reformulating it using Lagrange multipliers.

1. We begin with the Lagrangian, which incorporates the constraint using multipliers $\alpha_i \geq 0$:

$$\mathcal{L}(w, b, \alpha) = \frac{1}{2}||w||^2 - \sum_{i=1}^{n} \alpha_i [y_i(w^T x_i + b) - 1]$$

2. To find the minimum with respect to w and b, we take the partial derivatives and set them to zero:

- $\frac{\partial \mathcal{L}}{\partial w} = w - \sum_{i=1}^{n} \alpha_i y_i x_i = 0 \implies w = \sum_{i=1}^{n} \alpha_i y_i x_i$
- $\frac{\partial \mathcal{L}}{\partial b} = -\sum_{i=1}^{n} \alpha_i y_i = 0$

3. We substitute these results back into the Lagrangian to eliminate w and b. We start by substituting w into the term $-\sum \alpha_i y_i w^T x_i$ and use the fact that $\sum \alpha_i y_i = 0$:

$$-\sum_i \alpha_i [y_i(w^T x_i + b) - 1] = -\sum_i \alpha_i y_i w^T x_i - b\sum_i \alpha_i y_i + \sum_i \alpha_i =$$

$$-\sum_{i,j} \alpha_i \alpha_j y_i y_j x_j^T x_i + \sum_i \alpha_i$$

4. We then combine this with the first term of the Lagrangian, $\frac{1}{2}||w||^2 = \frac{1}{2}\sum_{i,j}\alpha_i\alpha_j y_i y_j x_i^T x_j$.

$$L(\alpha) = \frac{1}{2}\sum_{i,j}\alpha_i\alpha_j y_i y_j x_i^T x_j - \sum_{i,j}\alpha_i\alpha_j y_i y_j x_i^T x_j + \sum_{i=1}^{n}\alpha_i$$

5. Simplifying this gives the final dual problem. We want to maximize this function with respect to α. Replacing the dot product $x_i^T x_j$ with the kernel function $K(x_i, x_j)$ yields the final kernelized form:

$$\max_{\alpha} L(\alpha) = \sum_{i=1}^{n}\alpha_i - \frac{1}{2}\sum_{i,j}\alpha_i\alpha_j y_i y_j K(x_i, x_j)$$

subject to the constraints: $\alpha_i \geq 0$ and $\sum \alpha_i y_i = 0$.

The non-zero α_i coefficients correspond to the data points that lie exactly on the margin. These points are called the **support vectors**.

34.2 Soft-Margin SVM

For data that is not perfectly linearly separable, the soft-margin SVM introduces slack variables $\xi_i \geq 0$. These variables allow some data points to be within the margin or misclassified. The constraint is relaxed to:

$$y_i(w^T x_i + b) \geq 1 - \xi_i$$

The objective function is modified to penalize these violations:

$$\min_{w,b,\xi} \frac{1}{2}||w||^2 + C\sum_{i=1}^{n}\xi_i$$

The hyperparameter $C > 0$ controls the trade-off between maximizing the margin and minimizing the classification error. The dual formulation is very similar, with the only change being an upper bound on the α_i coefficients: $0 \leq \alpha_i \leq C$.

34.3 Common Kernel Functions

- **Linear Kernel:** $K(x, z) = x^T z$.
- **Polynomial Kernel:** $K(x, z) = (x^T z + c)^d$.
- **Radial Basis Function (RBF) Kernel:** $K(x, z) = \exp(-\gamma||x - z||^2)$.

34.4 Kernelized Ridge Regression

The kernel trick can also be applied to Ridge Regression.

1. The primal objective function is:

$$\min_{w}\sum_{i=1}^{n}(y_i - w^T\Phi(x_i))^2 + \lambda\|w\|^2$$

2. The Representer Theorem states that the optimal weight vector w can be expressed as a linear combination of the mapped training instances:

$$w = \sum_{j=1}^{n}\alpha_j\Phi(x_j)$$

3. We substitute this expression for w back into the objective function.
   - The regularization term becomes $\lambda\|w\|^2 = \lambda\alpha^T K\alpha$.
   - The loss term $\sum_{i=1}^{n}(y_i - w^T\Phi(x_i))^2$ becomes $(y - K\alpha)^T(y - K\alpha)$.
4. The full objective function, now in terms of α, is:

$$J(\alpha) = (y - K\alpha)^T(y - K\alpha) + \lambda\alpha^T K\alpha$$

5. By taking the gradient with respect to α and setting it to zero, we can find the closed-form solution for α:

$$\alpha = (K + \lambda I)^{-1}y$$

6. The final prediction for a new input x_{new} is then given by:

$$f(x_{\text{new}}) = \sum_{i=1}^{n}\alpha_i K(x_i, x_{\text{new}})$$

Unlike SVMs, all α_i are generally non-zero, meaning all training points are required for prediction.

35 Feature Importance

Understanding which features are most influential in a model's predictions is a fundamental part of explainable AI (XAI).

35.1 Feature Importance in Linear Models

For a simple linear regression model, one might initially assume that the feature with the largest coefficient is the most important. For example, given the model:

$$f(x) = -0.23x_1 + 2.37x_2 + 1.3$$

It appears that x_2 is the most important feature. However, this can be misleading because the importance of a coefficient depends on the scale of its corresponding feature. To make a fair comparison, the features must be on the same scale (e.g., by standardizing them before training). After scaling, the model's coefficients might reveal a different story, for instance, showing that x_1 has a larger impact.

35.2 Detailed Feature Importance: Partial Dependence Plots (PDPs)

While a single number for global feature importance is useful, it doesn't show *how* the prediction changes as the feature's value changes. Partial Dependence Plots visualize this relationship. The core idea is to see how a model's prediction changes based on one feature, while keeping all other feature values fixed at a constant. This shows the marginal effect of a feature on the prediction.

35.3 Local Feature Importance

Local explanations focus on understanding a single prediction for a specific data instance. For a linear model, this is straightforward. The contribution of each feature is simply its coefficient multiplied by its value for that instance. For the instance $(x_1 = 0.1, x_2 = 0.0001)$, the prediction $f(x) = 1.28$ is broken down as follows:

$$f(x) = \underbrace{-0.23 \cdot 0.1}_{\text{contrib of } x_1} + \underbrace{2.37 \cdot 0.0001}_{\text{contrib of } x_2} + \underbrace{1.3}_{\text{intercept/bias}} =$$

$$= -0.023 + 0.00237 + 1.3 \approx 1.28$$

This is a model-specific method as it relies directly on the model's coefficients.

35.4 Interpretability Beyond Linear Models

The techniques above are simple for linear models because these models assume linearity, additivity, and no feature interactions. These ideas can be extended to Generalized Additive Models (GAMs) of the form $f(x) = \sum f_k(x_k) + \beta_0$, but how do we achieve interpretability for complex, non-additive "black-box" models like AdaBoost or deep neural networks?

To do this, we need a model-agnostic way to simulate the absence of a feature to see its impact.

- **Retraining:** One could retrain the model without a feature, but this is computationally very expensive and creates an entirely new model.
- **Marginalizing:** Instead of fixing other features to a constant (like in PDPs), a more robust method is to average the model's predictions over the distribution of the other features. This avoids creating unrealistic data points. The global importance of a feature can then be measured by the variance or standard deviation of these marginal predictions.

36 Shapley Values in XAI

Shapley values offer a principled, model-agnostic approach to local feature importance, rooted in cooperative game theory. They provide a way to fairly distribute the "payout" (the model's prediction) among the "players" (the features).

36.1 Core Concepts

The contribution of a set of features S is the difference between the prediction made using those features, $f(S)$, and the baseline prediction made with no features, $f(\emptyset)$.

$$\Delta(S) = f(S) - f(\emptyset)$$

The total "payout" to be distributed is $\Delta(N)$, where N is the set of all features. The Shapley value ϕ_i for a feature i is its average marginal contribution across all possible subsets (or "coalitions") of features.

36.2 The Shapley Value Formula

The Shapley value for feature i is calculated as:

$$\phi_i = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(p - |S| - 1)!}{p!} \left(\Delta(S \cup \{i\}) - \Delta(S)\right)$$

where p is the total number of features, and the sum is over all subsets S that do not contain feature i. The term $(\Delta(S \cup \{i\}) - \Delta(S))$ is the marginal contribution of feature i when it is added to the subset S.

36.3 Simple SHAP Value Example (Two Features)

Let the model $f(x_1, x_2)$ be defined as:

$$f(0,0) = 0,$$
$$f(1,0) = 10,$$
$$f(0,1) = 20,$$
$$f(1,1) = 30.$$

We explain the prediction for input $x = (1, 1)$.

36.3.1 Step 1: Baseline Value

The baseline is the average prediction over all possible inputs:

$$\text{baseline} = \frac{0 + 10 + 20 + 30}{4} = 15$$

36.3.2 Step 2: SHAP Value for x_1

Subset $\emptyset$:

No features known:

$$\mathbb{E}[f(\cdot)] = 15$$

Add $x_1 = 1$: average over $x_2 \in \{0, 1\}$

$$\mathbb{E}[f(1, \cdot)] = \frac{f(1,0) + f(1,1)}{2} = \frac{10 + 30}{2} = 20$$

Contribution: $20 - 15 = 5$

Subset $\{x_2\}$:

Given $x_2 = 1$, x_1 unknown:

$$\mathbb{E}[f(\cdot, 1)] = \frac{f(0,1) + f(1,1)}{2} = \frac{20 + 30}{2} = 25$$

Add $x_1 = 1$: $f(1,1) = 30$ Contribution: $30 - 25 = 5$

SHAP for x_1:

$$\phi_1 = \frac{1}{2}(5 + 5) = 5$$

36.3.3 Step 3: SHAP Value for x_2

Subset $\emptyset$:

$$\mathbb{E}[f(\cdot)] = 15, \quad \mathbb{E}[f(\cdot, x_2 = 1)] = \frac{20 + 30}{2} = 25$$

Contribution: $25 - 15 = 10$

Subset $\{x_1\}$:

$$\mathbb{E}[f(1, \cdot)] = \frac{10 + 30}{2} = 20, \quad f(1,1) = 30$$

Contribution: $30 - 20 = 10$

SHAP for x_2:

$$\phi_2 = \frac{1}{2}(10 + 10) = 10$$

36.3.4 Final Result

$$\phi_1 = 5, \quad \phi_2 = 10$$

$$\text{Prediction: } f(1,1) = 30, \quad \text{Baseline: } 15$$

$$\phi_1 + \phi_2 = 15 = f(1,1) - \text{baseline}$$

36.4 Axioms and Computation

Shapley values are the only attribution method that simultaneously satisfies four desirable properties: Efficiency (Additivity), Symmetry, Dummy, and Linearity.

- **Efficiency/Additivity:** The sum of the Shapley values for all features equals the total difference between the model's prediction and the baseline prediction.
- **Symmetry:** If two features have the same contribution to all possible coalitions, they have the same Shapley value.
- **Dummy:** If a feature has no effect on the prediction, its Shapley value is zero.

The exact computation of Shapley values is very expensive ($O(2^p)$), as it requires evaluating the model for all possible subsets of features. Therefore, in practice, they are approximated using methods like SHAP (SHapley Additive exPlanations), which uses clever sampling and kernel-based approaches to estimate the values efficiently.

37 A Review of Empirical Risk Minimization (ERM)

Before diving into Bayesian inference, it's useful to recall the frequentist approach, which is often framed as Empirical Risk Minimization (ERM). This is the underlying principle for many machine learning algorithms, including those solved with Maximum Likelihood Estimation (MLE).

37.1 The ERM Framework

The goal in ERM is to learn a function $h : X \to Y$ from a predefined set of functions (the hypothesis space, $\mathcal{H}$) that minimizes a chosen loss function L over the training data $D = \{(x_i, y_i)\}_{i=1}^n$. The chosen hypothesis $\hat{h}$ is the one that minimizes the total loss on the training set:

$$\hat{h}_{\text{ERM}} = \underset{h \in \mathcal{H}}{\text{argmin}} \sum_{i=1}^{n} L(h(x_i), y_i)$$

The core idea is to select a single best model from the hypothesis space based on its performance on the training data. If the true data-generating function h_{TRUE} is within our hypothesis space $\mathcal{H}$, then as we get more data, $\hat{h}_{\text{ERM}}$ will converge towards h_{TRUE}.

37.2 Issues with the ERM/MLE Approach

While powerful, this approach has several limitations:

1. **No built-in quantification of uncertainty:** ERM provides a single point estimate for the model parameters (e.g., β_{ERM}). It doesn't naturally tell us how certain we are about this estimate. While we can use techniques like bootstrapping or calculating standard errors and confidence intervals to estimate uncertainty, their interpretation can be somewhat awkward.

2. **Overfitting:** ERM can easily overfit the training data. The common solution is to introduce regularization, such as the L2 penalty in Ridge Regression, which adds a term to the loss function.

$$\beta_{\text{ERM, L2}} = \underset{\beta}{\text{argmin}} \sum_{i=1}^{n} (\beta^T x_i - y_i)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

3. **Messy Decision Theory:** The choice of loss function is tied directly to the optimization problem. If we want to evaluate our model using a different metric, we can't easily do so without reformulating and re-solving the entire problem. The learning and decision-making processes are coupled.

38 Bayesian Inference

38.1 The Core Idea

Bayesian inference offers a different perspective. Instead of selecting a single best model, the core idea is to determine a **probability distribution over all possible models** (or parameters) in the hypothesis space. We use data to update our beliefs about which models are more or less probable.

This is achieved using Bayes' Theorem to find the **posterior distribution** $p(h|D)$:

$$\underbrace{p(h|D)}_{\text{Posterior}} = \frac{\overbrace{p(D|h)}^{\text{Likelihood}} \cdot \overbrace{p(h)}^{\text{Prior}}}{\underbrace{p(D)}_{\text{Evidence}}} \propto p(D|h) \cdot p(h)$$

- **Prior** $p(h)$**:** Our belief about the probability of a hypothesis h *before* seeing the data.
- **Likelihood** $p(D|h)$**:** The probability of observing the data D if hypothesis h were true. This is the same likelihood function used in MLE.
- **Posterior** $p(h|D)$**:** Our updated belief about the probability of hypothesis h *after* seeing the data.

38.2 Example: Bayesian Linear Regression

Let's assume the standard linear model setup, where the error variance σ^2 is known.

- **Likelihood:** The likelihood of the data is given by the normal distribution:

$$p(Y|X, \beta, \sigma^2) = \prod_{i=1}^{n} p(y_i|x_i, \beta, \sigma^2) =$$

$$\prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(\beta^T x_i - y_i)^2}{2\sigma^2}\right)$$

- **Prior:** We must specify a prior belief about the parameters β. A common choice is a zero-mean Gaussian prior, which expresses a belief that smaller coefficient values are more likely. This acts as a form of regularization.

$$p(\beta) = \prod_{j=1}^{p} \frac{1}{\sqrt{2\pi\tau^2}} \exp\left(-\frac{\beta_j^2}{2\tau^2}\right)$$

where τ^2 is the prior variance.

38.2.1 Deriving the Posterior

The posterior is proportional to the likelihood times the prior. It's easiest to work with the log-posterior:

$$\begin{aligned}\log p(\beta|X, Y) &= \log p(Y|X, \beta) + \log p(\beta) + \text{const.} \\ &= -\frac{1}{2\sigma^2} \sum_{i=1}^{n} (\beta^T x_i - y_i)^2 - \frac{1}{2\tau^2} \sum_{j=1}^{p} \beta_j^2 + \text{const.}\end{aligned}$$

Notice that maximizing this log-posterior (known as Maximum A Posteriori or MAP estimation) is equivalent to minimizing the L2-regularized (Ridge) regression loss function. Using a Laplace prior instead of a Gaussian one would be equivalent to L1 (Lasso) regularization.

Because the prior and likelihood are both Gaussian, the posterior will also be a Gaussian distribution. By performing a procedure called "completing the square", the log-posterior can be rewritten into the quadratic form of a multivariate Gaussian:

$$\log p(\beta|X, Y) = -\frac{1}{2}(\beta - \mu_{\text{post}})^T \Sigma_{\text{post}}^{-1} (\beta - \mu_{\text{post}}) + \text{const.}'$$

This tells us that the posterior distribution is $\beta|Y, X \sim \mathcal{N}(\mu_{\text{post}}, \Sigma_{\text{post}})$, with posterior covariance $\Sigma_{\text{post}}^{-1} = \frac{1}{\sigma^2} X^T X + \frac{1}{\tau^2} I$ and posterior mean $\mu_{\text{post}} = \Sigma_{\text{post}} \left(\frac{1}{\sigma^2} X^T Y\right)$.

38.3 Pros and Cons of Bayesian Inference

- **Pros:**
  - Provides a full posterior distribution, offering a natural and conceptually simple way to quantify uncertainty.
  - Robust to overfitting due to model averaging.
  - Decouples inference (finding the posterior) from decision-making (using the posterior to make a prediction). We can use the same posterior to make decisions based on different loss functions (e.g., use the posterior mean for MSE, posterior median for MAE).
- **Cons:**
  - Requires specifying a prior, which can be subjective.
  - The computational complexity is often much higher. MLE is an optimization problem, whereas Bayesian inference is an integration problem (to find the evidence term $p(D)$), which is typically harder.

39 Computation in Bayesian Inference

For many models, especially simpler ones with conjugate priors, the posterior distribution can be calculated analytically. However, for most non-trivial models, this is not possible, and we must use computational methods to approximate it.

1. **Analytical (Closed Form):** This method provides an exact, unbiased solution for the posterior distribution. It is very fast to compute. However, a closed-form solution does not exist for many complex models.

2. **Markov Chain Monte Carlo (MCMC):** A class of algorithms that perform integration by sampling. They draw a sequence of samples from a Markov chain whose stationary distribution is the desired posterior distribution $p(\theta|D)$. This provides an unbiased but not exact estimate of the posterior. A key drawback is that it can be very slow and computationally intensive.

3. **Approximate Inference:** These methods approximate the true (and often intractable) posterior $p(\theta|D)$ with a simpler, tractable distribution $q(\theta)$. These methods are typically biased.

    - **Laplace Approximation:** A fast and simple method that approximates the posterior with a Gaussian distribution. It finds the mode (peak) of the posterior distribution and then uses the curvature (the Hessian matrix of the log-posterior) at that mode to define the covariance of the approximating Gaussian. The result is a biased approximation of the true posterior.
    - **Variational Inference (VI):** Another powerful approximation method that turns the integration problem into an optimization problem. VI finds the best approximation from a family of distributions by minimizing the Kullback-Leibler (KL) divergence to the true posterior. Like the Laplace approximation, it is fast but produces a biased estimate.

40 Laplace Approximation

The Laplace approximation provides a way to approximate a posterior probability distribution with a Gaussian distribution. This is particularly useful when the posterior is unimodal but not a standard distribution that is easy to work with. The core idea is to find the mode of the distribution (its peak) and then fit a Gaussian centered at that mode, where the variance of the Gaussian is determined by the curvature of the log-posterior at the mode.

The steps to compute the Laplace approximation are as follows:

1. **Define the log-probability:** Write down the logarithm of the target probability distribution, which we denote as $l(x) = \log p(x)$. Working with the log-probability is often easier than working with the probability directly.

2. **Find the mode:** Solve for the point $\hat{x}$ where the first derivative of the log-probability is zero. This point, $\hat{x}$, is the mode (the peak) of the distribution.

$$l'(\hat{x}) = 0$$

3. **Compute the curvature at the mode:** Calculate the second derivative of the log-probability, $l''(x)$, and evaluate it at the mode, $\hat{x}$. This value tells us about the sharpness of the peak.

    - If $l''(\hat{x})$ is strongly negative, the peak is sharp $\Rightarrow$ this corresponds to a small variance.
    - If $l''(\hat{x})$ is near zero, the peak is flat $\Rightarrow$ this corresponds to a large variance.

4. **Calculate the approximate variance:** Form the approximate variance, σ^2, by taking the negative inverse of the second derivative at the mode.

$$\sigma^2 = (-l''(\hat{x}))^{-1}$$

5. **Form the Gaussian approximation:** Write down the final Gaussian distribution, $q(x)$, which approximates $p(x)$. This distribution is centered at the mode $\hat{x}$ with the variance σ^2 you just calculated.

$$q(x) = \mathcal{N}(x \mid \hat{x}, \sigma^2)$$

6. **Use the approximation:** Approximate any integral or expectation that was originally under $p(x)$ by using the corresponding, and much simpler, integral or expectation under the new Gaussian distribution $q(x)$.

41 Unsupervised Machine Learning

Unsupervised learning deals with unlabeled data, aiming to find hidden patterns or intrinsic structures within the data itself. The general process involves defining a model structure (parameterized by Θ) and optimizing a loss or criterion function $L(\Theta)$ to find the best parameters that describe the data's structure.

41.1 Clustering

Clustering is the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar to each other than to those in other groups.

41.1.1 Types of Clustering Algorithms

There are several families of clustering algorithms, each with different strengths.

- **Partitional Clustering:** These algorithms divide the dataset into a pre-specified number of non-overlapping clusters.
    - **k-Means:** Assigns points to the cluster with the nearest centroid (mean). This can be problematic for data types where a mean is not well-defined (e.g., text, images).
    - **k-Medoids:** Similar to k-means, but uses an actual data point (a medoid) as the center of the cluster, making it more robust and suitable for arbitrary data types.
- **Hierarchical Clustering:** Builds a hierarchy of clusters, either agglomerative (bottom-up, merging clusters) or divisive (top-down, splitting clusters). It doesn't require the number of clusters to be specified beforehand.
- **Density-Based Clustering:** Finds clusters based on areas of high density separated by areas of low density. A popular example is **DBSCAN**.
- **Model-Based Clustering:** Assumes the data is a mixture of a finite number of probability distributions (e.g., Gaussians). It uses an algorithm like Expectation-Maximization (EM) to fit a **Gaussian Mixture Model (GMM)** to the data.

41.2 Data Maps (Dimensionality Reduction)

This family of unsupervised methods aims to represent high-dimensional data in a lower-dimensional space (an embedding or "data map"), often for visualization or as a pre-processing step for other ML models.

41.2.1 Principal Component Analysis (PCA)

PCA is a linear technique that finds a new set of orthogonal axes (principal components) that capture the maximum amount of variance in the data.

Derivation: Assume the data matrix X is centered (mean of zero). The projection of the data onto a direction vector u_1 is $z = Xu_1$. The variance of this projected data is:

$$\text{Var}(z) = \frac{1}{n}||Xu_1||^2 = \frac{1}{n}(Xu_1)^T(Xu_1) = \frac{1}{n}u_1^T X^T X u_1 = u_1^T S u_1$$

where $S = \frac{1}{n}X^T X$ is the covariance matrix of the data. Our goal is to find the direction u_1 that maximizes this variance, subject to the constraint that u_1 is a unit vector ($u_1^T u_1 = 1$). We can solve this using a Lagrange multiplier λ:

$$\mathcal{L}(u_1, \lambda) = u_1^T S u_1 - \lambda(u_1^T u_1 - 1)$$

Taking the derivative with respect to u_1 and setting it to zero gives:

$$\frac{\partial \mathcal{L}}{\partial u_1} = 2Su_1 - 2\lambda u_1 = 0 \implies Su_1 = \lambda u_1$$

This is the fundamental eigenvector equation. It shows that the optimal direction u_1 is the eigenvector of the covariance matrix S. The variance captured by this component is equal to its corresponding eigenvalue:

$$\text{Var}(z) = u_1^T S u_1 = u_1^T(\lambda u_1) = \lambda(u_1^T u_1) = \lambda$$

To capture the most variance, we choose the eigenvector corresponding to the largest eigenvalue. Subsequent components are the eigenvectors corresponding to the next largest eigenvalues.

Interpretation: The elements of an eigenvector u_k are called the **loadings**, and they indicate how much each original feature contributes to that principal component. A **scree plot** can be used to visualize the percentage of variance explained by each component, helping to decide how many components to retain. A key limitation of PCA is that it is a linear method and will fail to capture non-linear structures in the data.

41.2.2 Multidimensional Scaling (MDS)

MDS is a technique that aims to create a low-dimensional embedding that preserves the pairwise distances between points from the original high-dimensional space. It minimizes a loss function like:

$$L = \sum_{i,j}(d_{ij} - D_{ij})^2$$

where d_{ij} is the original distance and D_{ij} is the distance in the new embedding. A weakness of this approach is that it gives equal weight to preserving both small and large distances. In many cases, we care more about preserving the local structure of nearby points.

41.2.3 t-Distributed Stochastic Neighbor Embedding (t-SNE)

t-SNE is a powerful non-linear technique for visualization, created by Geoffrey Hinton. It improves upon a predecessor called SNE.

- **SNE** first converts high-dimensional Euclidean distances into conditional probabilities representing similarities. The similarity of point x_j to x_i is the probability that x_i would pick x_j as its neighbor if neighbors were picked in proportion to their probability density under a Gaussian centered at x_i.
- A problem with using a Gaussian in the low-dimensional space is that it can lead to a "crowding problem," where clusters become too dense and overlap.
- **t-SNE** solves this by using a heavy-tailed Student's t-distribution in the low-dimensional space. This allows dissimilar points to be placed further apart in the map, helping to separate clusters more clearly.

41.2.4 Autoencoders

An autoencoder is a type of neural network used for unsupervised learning, typically for dimensionality reduction. It consists of two parts:

- An **encoder** network that compresses the high-dimensional input data X into a low-dimensional latent representation, often called a "bottleneck".
- A **decoder** network that attempts to reconstruct the original input X from the low-dimensional representation.

The network is trained to minimize the reconstruction error (e.g., Mean Squared Error) between the output Y and the original input X. After training, the encoder part of the network can be used on its own to generate the low-dimensional embeddings for the data.