Machine Learning: Module 3 Notes

Dr. Likhit Nayak

October 26, 2025

1 Generative model for discrete data

Generative models learn the joint probability distribution $p(x, y)$ of the input features x and the corresponding labels y. By modeling how data is generated for each class, they can then use Bayes' theorem to predict the most likely label for a new input. This contrasts with discriminative models, which learn the decision boundary or the conditional probability $p(y|x)$ directly.

1.1 Key Concepts

- **Generative Model:** A model that learns the joint probability distribution $p(x, y)$ of the data and labels. Classification is performed by using this joint distribution to calculate the conditional probability $p(y|x)$ via Bayes' theorem.
- **Bayesian Concept Learning:** A probabilistic approach to learning where Bayes' theorem is used to compute the posterior probability of each hypothesis in a hypothesis space, given the training data. The prediction for a new instance is often made by considering the most probable hypothesis.
- **Naive Bayes Classifier:** A specific type of generative classifier based on Bayes' theorem. It simplifies the learning process by making a 'naive' assumption that all features are conditionally independent of each other, given the class label.
- **Prior Probability ($P(h)$ or $P(y)$):** The probability of a hypothesis or class before observing any data. It represents our initial belief.
- **Likelihood ($P(D|h)$ or $P(X|y)$):** The probability of observing the data D (or feature vector X) given that a specific hypothesis h (or class y) is true.
- **Posterior Probability ($P(h|D)$ or $P(y|X)$):** The updated probability of a hypothesis or class after the data has been observed. It is calculated using Bayes' theorem.
- **Maximum a Posteriori (MAP) Estimation:** A decision rule that selects the hypothesis or class with the highest posterior probability.

1.2 Bayesian Concept Learning

Bayesian concept learning provides a framework for reasoning about hypotheses under uncertainty. It uses probability theory to quantify the plausibility of different hypotheses given the observed data.

The foundation of this approach is **Bayes' Theorem**:

$$P(h \mid D) = \frac{P(D \mid h)P(h)}{P(D)}$$

Each term in the theorem has a specific meaning in the context of learning:

- h is a specific hypothesis from the hypothesis space $\mathcal{H}$.
- D represents the observed training data.

- $P(h)$ is the **prior probability** of hypothesis h. This term captures any background knowledge about the domain, suggesting that some hypotheses are more likely than others even before seeing any data. If no such knowledge exists, a uniform prior is often assumed, i.e., $P(h)$ is the same for all $h \in \mathcal{H}$.
- $P(D|h)$ is the **likelihood** of the data D given the hypothesis h. It quantifies how well the hypothesis explains the data. In a noise-free setting, the likelihood is often 1 if the data is consistent with the hypothesis and 0 otherwise.
- $P(h|D)$ is the **posterior probability** of hypothesis h given the data D. This is the result of the learning process—our updated belief in the hypothesis after observing the data.
- $P(D)$ is the **marginal likelihood** or **evidence** of the data. It is calculated by summing over all hypotheses: $P(D) = \sum_{h' \in \mathcal{H}} P(D \mid h')P(h')$. It serves as a normalization constant, ensuring that the posterior probabilities sum to 1.

The goal is to find the most probable hypothesis given the data. This is known as the **Maximum a Posteriori (MAP)** hypothesis, h_{MAP}.

$$h_{MAP} = \arg\max_{h \in \mathcal{H}} P(h \mid D) = \arg\max_{h \in \mathcal{H}} \frac{P(D \mid h)P(h)}{P(D)}$$

Since $P(D)$ is constant for all hypotheses, it can be dropped from the maximization:

$$h_{MAP} = \arg\max_{h \in \mathcal{H}} P(D \mid h)P(h)$$

If we assume a uniform prior (i.e., $P(h)$ is constant), the MAP hypothesis simplifies to the **Maximum Likelihood (ML)** hypothesis, h_{ML}, which chooses the hypothesis that best explains the data:

$$h_{ML} = \arg\max_{h \in \mathcal{H}} P(D \mid h)$$

1.3 Naive Bayes Classifier

The Naive Bayes classifier applies these Bayesian principles to classification tasks. Given a feature vector $X = (x_1, x_2, \ldots, x_d)$, the goal is to predict the class y from a set of possible classes $Y = \{y_1, y_2, \ldots, y_K\}$.

The decision rule is to choose the class that is most probable given the observed features:

$$y_{predict} = \arg\max_{y_k \in Y} P(y_k \mid X)$$

Using Bayes' theorem, we can rewrite the posterior probability as:

$$P(y_k \mid X) = \frac{P(X \mid y_k)P(y_k)}{P(X)}$$

As $P(X)$ is the same for all classes, our decision rule becomes:

$$y_{predict} = \arg\max_{y_k \in Y} P(X \mid y_k)P(y_k)$$

1.3.1 The Naive Independence Assumption

Calculating the class-conditional probability of the feature vector, $P(X \mid y_k) = P(x_1, x_2, \ldots, x_d \mid y_k)$, is difficult without a very large amount of data. The Naive Bayes classifier simplifies this by making a strong assumption: all features are conditionally independent of each other, given the class.

Mathematically, this assumption is:

$$P(X \mid y_k) = P(x_1, x_2, \ldots, x_d \mid y_k) = \prod_{i=1}^{d} P(x_i \mid y_k)$$

Substituting this into the decision rule gives the final Naive Bayes classification formula:

$$y_{predict} = \arg\max_{y_k \in Y} \left[P(y_k) \prod_{i=1}^{d} P(x_i \mid y_k) \right]$$

1.3.2 Parameter Estimation and Smoothing

To use the Naive Bayes classifier, we need to estimate the prior probabilities $P(y_k)$ and the class-conditional probabilities (likelihoods) $P(x_i \mid y_k)$ from the training data.

- **Prior Estimation:** The prior probability for each class is estimated as the relative frequency of that class in the training data. If N_k is the number of training instances of class y_k and N is the total number of instances:
$$\hat{P}(y_k) = \frac{N_k}{N}$$
- **Likelihood Estimation (Categorical Features):** The likelihood is estimated as the frequency of a feature value v within instances of a specific class. If $N_{ik,v}$ is the count of instances where feature x_i has value v for class y_k:
$$\hat{P}(x_i = v \mid y_k) = \frac{N_{ik,v}}{N_k}$$

1.3.3 Algorithm

Training Phase:

1. For each class $y_k \in Y$, calculate the prior probability $\hat{P}(y_k) = N_k/N$.
2. For each class y_k, for each feature x_i, and for each possible value v of that feature, calculate the likelihood $\hat{P}(x_i = v \mid y_k)$ using frequency counts and Laplace smoothing.
3. Store all calculated prior and likelihood probabilities.

Prediction Phase (for a new instance $X_{new} = (x_1, \ldots, x_d)$):

1. For each class $y_k \in Y$, calculate a score proportional to the posterior probability. To avoid numerical underflow from multiplying many small probabilities, it is common to use the sum of log-probabilities:
$$\text{score}(y_k) = \log(\hat{P}(y_k)) + \sum_{i=1}^{d} \log(\hat{P}(x_i \mid y_k))$$
2. Predict the class with the highest score:
$$\hat{y} = \arg\max_{y_k \in Y} \text{score}(y_k)$$

1.4 Example Question

Question:

Given the following dataset, predict whether to play tennis for a new day with the conditions: "(Outlook=Sunny, Temp=Cool, Humidity=High, Windy=True)". Use a Naive Bayes classifier.

Training Data:

Outlook	**Temp**	**Humidity**	**Windy**	**Play**
Sunny	Hot	High	False	No
Sunny	Hot	High	True	No
Overcast	Hot	High	False	Yes
Rainy	Mild	High	False	Yes
Rainy	Cool	Normal	False	Yes
Rainy	Cool	Normal	True	No
Overcast	Cool	Normal	True	Yes
Sunny	Mild	High	False	No
Sunny	Cool	Normal	False	Yes
Rainy	Mild	Normal	False	Yes
Sunny	Mild	Normal	True	Yes
Overcast	Mild	High	True	Yes
Overcast	Hot	Normal	False	Yes
Rainy	Mild	High	True	No

Solution:

Step 1: Calculate Prior Probabilities

First, we calculate the prior probability of each class based on the training data.
Total instances $N = 14$.

- Number of 'Yes' instances: $N_{\text{Yes}} = 9$
- Number of 'No' instances: $N_{\text{No}} = 5$

The prior probabilities are:

$$P(\text{Play=Yes}) = \frac{N_{\text{Yes}}}{N} = \frac{9}{14}$$

$$P(\text{Play=No}) = \frac{N_{\text{No}}}{N} = \frac{5}{14}$$

Step 2: Calculate Likelihoods (No Smoothing)

Next, we calculate the conditional probability of each feature value given each class for the query instance $X = $ (Outlook=Sunny, Temp=Cool, Humidity=High, Windy=True).

For class Play=Yes ($N_{\textbf{Yes}} = 9$):

- $P(\text{Outlook=Sunny} \mid \text{Yes}) = \frac{\text{count(Sunny, Yes)}}{N_{\text{Yes}}} = \frac{2}{9}$
- $P(\text{Temp=Cool} \mid \text{Yes}) = \frac{\text{count(Cool, Yes)}}{N_{\text{Yes}}} = \frac{3}{9}$
- $P(\text{Humidity=High} \mid \text{Yes}) = \frac{\text{count(High, Yes)}}{N_{\text{Yes}}} = \frac{3}{9}$
- $P(\text{Windy=True} \mid \text{Yes}) = \frac{\text{count(True, Yes)}}{N_{\text{Yes}}} = \frac{3}{9}$

For class Play=No ($N_{\textbf{No}} = 5$):

- $P(\text{Outlook=Sunny} \mid \text{No}) = \frac{\text{count(Sunny, No)}}{N_{\text{No}}} = \frac{3}{5}$
- $P(\text{Temp=Cool} \mid \text{No}) = \frac{\text{count(Cool, No)}}{N_{\text{No}}} = \frac{1}{5}$
- $P(\text{Humidity=High} \mid \text{No}) = \frac{\text{count(High, No)}}{N_{\text{No}}} = \frac{4}{5}$
- $P(\text{Windy=True} \mid \text{No}) = \frac{\text{count(True, No)}}{N_{\text{No}}} = \frac{3}{5}$

Step 3: Make Prediction

Now, we calculate a score for each class proportional to the posterior probability.

Score(Play=Yes):

$$\begin{aligned}\text{Score(Yes)} &\propto P(\text{Yes}) \times P(\text{Sunny}|\text{Yes}) \times P(\text{Cool}|\text{Yes}) \times P(\text{High}|\text{Yes}) \times P(\text{True}|\text{Yes}) \\ &= \frac{9}{14} \times \frac{2}{9} \times \frac{3}{9} \times \frac{3}{9} \times \frac{3}{9} \\ &= \frac{486}{91854} \approx 0.00529\end{aligned}$$

Score(Play=No):

$$\begin{aligned}\text{Score(No)} &\propto P(\text{No}) \times P(\text{Sunny}|\text{No}) \times P(\text{Cool}|\text{No}) \times P(\text{High}|\text{No}) \times P(\text{True}|\text{No}) \\ &= \frac{5}{14} \times \frac{3}{5} \times \frac{1}{5} \times \frac{4}{5} \times \frac{3}{5} \\ &= \frac{180}{8750} \approx 0.02057\end{aligned}$$

Step 4: Conclusion

Finally, we compare the two scores to make our prediction.

$$0.02057(\text{No}) > 0.00529(\text{Yes})$$

Since the score for 'No' is higher, the model predicts that **Play = No** for the given day.

2 SVM for Classification

Support Vector Machines (SVMs) are a powerful class of supervised learning models used for classification, regression, and outlier detection. For classification, an SVM is a discriminative classifier that works by finding an optimal hyperplane that separates data points of different classes in a high-dimensional space.

2.1 Key Concepts

- **Hyperplane:** In an N-dimensional space, a hyperplane is an $(N-1)$-dimensional flat subspace. In the context of binary classification, it serves as a decision boundary to separate the two classes.
- **Margin:** The margin is the distance between the separating hyperplane and the nearest data points from either class. A larger margin implies a more confident and robust classification.
- **Optimal Hyperplane (Maximal Margin Classifier):** The core idea of SVM is to find the specific hyperplane that has the largest possible margin between the classes. This hyperplane is considered optimal because it is the most generalizable to unseen data.
- **Support Vectors:** These are the data points that lie closest to the hyperplane, on the margin boundaries. They are the critical data points that "support" or define the position and orientation of the optimal hyperplane. If any non-support vector points are removed, the optimal hyperplane would not change.
- **Hard Margin SVM:** This is the formulation used when the training data is perfectly linearly separable. It finds a hyperplane that separates the data with no misclassifications.
- **Soft Margin SVM:** This is a more flexible and practical extension that allows for some data points to be misclassified or to fall inside the margin. This is achieved by introducing slack variables and a regularization parameter, C, which controls the trade-off between maximizing the margin and minimizing the classification error.

- **Kernel Trick:** A powerful mathematical technique that allows SVMs to classify data that is not linearly separable. It works by implicitly mapping the original data into a higher-dimensional space where a linear separator may exist, without ever having to explicitly compute the coordinates of the data in this new space. This is done using a kernel function.

2.2 Mathematical Formulation: The Hard Margin SVM

Let's consider a binary classification problem with a training dataset of n points $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)$, where $\mathbf{x}_i \in \mathbb{R}^d$ is a feature vector and $y_i \in \{-1, 1\}$ is the class label.

The Hyperplane: A separating hyperplane can be defined by the equation:

$$\mathbf{w} \cdot \mathbf{x} + b = 0$$

where $\mathbf{w}$ is a weight vector normal (perpendicular) to the hyperplane and b is a bias term.

Maximizing the Margin: The two parallel hyperplanes that form the margins can be described by:

$$\begin{aligned} \mathbf{w} \cdot \mathbf{x} + b &= 1 \quad \text{(for the positive class)} \\ \mathbf{w} \cdot \mathbf{x} + b &= -1 \quad \text{(for the negative class)} \end{aligned}$$

The distance between these two hyperplanes, which is the margin, is given by $\frac{2}{\|\mathbf{w}\|}$. To maximize this margin, we need to minimize $\|\mathbf{w}\|$, which is equivalent to minimizing $\frac{1}{2}\|\mathbf{w}\|^2$ for mathematical convenience (it's a quadratic programming problem).

The Primal Optimization Problem We want to find the $\mathbf{w}$ and b that minimize $\frac{1}{2}\|\mathbf{w}\|^2$ subject to the constraint that all data points are correctly classified and lie outside the margin. This can be expressed as:

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1, \quad \forall i = 1, \ldots, n$$

So, the complete primal optimization problem for a hard-margin SVM is:

$$\min_{\mathbf{w}, b} \frac{1}{2}\|\mathbf{w}\|^2$$

subject to

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1, \quad i = 1, \ldots, n$$

2.3 The Dual Problem and Lagrange Multipliers

The original, or *primal*, problem in SVM is to find the widest possible "street" (the margin) between two classes. This is a constrained optimization problem: we want to maximize the margin, subject to the constraint that all data points are correctly classified and are outside this "street". Solving this directly is computationally challenging, especially when the data is high-dimensional. The dual formulation offers two major advantages:

1. It turns the optimization problem into one that often can be solved more efficiently, as we will see.
2. It introduces the dot product of data points $(\mathbf{x}_i \cdot \mathbf{x}_j)$, which is the key to using the "kernel trick" for non-linear classification.

2.3.1 The Lagrangian: Combining Objective and Constraints

To move from the primal to the dual, we use the method of **Lagrange multipliers**. The Lagrangian function, $L(\mathbf{w}, b, \boldsymbol{\alpha})$, is a single function that cleverly merges our original goal with our constraints.

$$L(\mathbf{w}, b, \boldsymbol{\alpha}) = \underbrace{\frac{1}{2}\|\mathbf{w}\|^2}_{\text{Original Objective}} \underbrace{- \sum_{i=1}^{n} \alpha_i [y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1]}_{\text{Constraints with Multipliers}}$$

- $\frac{1}{2}\|\mathbf{w}\|^2$: This is what we want to minimize. Minimizing this term is equivalent to maximizing the margin $2/\|\mathbf{w}\|$.
- α_i: These are the Lagrange multipliers. We introduce one multiplier for each data point $(\mathbf{x}_i, y_i)$ in our training set. The constraint is that $\alpha_i \geq 0$.
- $y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1 \geq 0$: This is the mathematical representation of the constraint that every data point must be on or outside the correct margin. By subtracting the entire sum in the Lagrangian, we are essentially penalizing any violation of this constraint.

2.3.2 Finding the Optimal Point by Differentiation

To solve the optimization, we treat the Lagrangian as a function and find its minimum with respect to the primal variables, $\mathbf{w}$ and b. We do this in the standard calculus way: take the partial derivatives and set them to zero.

Derivative with respect to w:

$$\frac{\partial L}{\partial \mathbf{w}} = \mathbf{w} - \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i = 0 \implies \mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i$$

This first result is profoundly important. It tells us that the optimal weight vector $\mathbf{w}$ (which defines the orientation of our decision boundary) is simply a **linear combination of the input data vectors $\mathbf{x}_i$**. The α_i values act as weights for each data vector in this sum.

Derivative with respect to b:

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

This second result gives us a new constraint that the solution for the α_i values must satisfy. It's a condition on the weighted sum of the class labels.

2.3.3 Constructing the Dual Problem

Now, we substitute these two results back into our original Lagrangian. This process eliminates $\mathbf{w}$ and b, leaving us with an optimization problem that depends *only* on the Lagrange multipliers $\boldsymbol{\alpha}$. This new problem is the dual:

$$\max_{\boldsymbol{\alpha}} \quad W(\boldsymbol{\alpha}) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j)$$

$$\text{subject to} \quad \alpha_i \geq 0, \quad \text{and} \quad \sum_{i=1}^{n} \alpha_i y_i = 0$$

Our task has transformed from minimizing a function of $\mathbf{w}$ and b to maximizing a function of $\boldsymbol{\alpha}$. This is a Quadratic Programming Problem (QPP) that can be solved with standard optimizers.

2.3.4 Support Vectors

The Karush-Kuhn-Tucker (KKT) conditions are a set of rules that must be satisfied by the optimal solution of a constrained optimization problem. One of these KKT conditions, known as **complementary slackness**, states that for our optimal solution:

$$\alpha_i[y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1] = 0 \quad \text{for all } i = 1, \dots, n$$

Let's analyze this condition: for this equation to be true, for any given data point $\mathbf{x}_i$, at least one of its two factors must be zero.

1. **Case 1: The vast majority of data points.** For most data points, they will lie correctly classified and far away from the margin. For these points, $y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1 > 0$. Since this term is not zero, the KKT condition forces its corresponding Lagrange multiplier to be zero: $\boldsymbol{\alpha_i = 0}$.
2. **Case 2: The critical data points.** Some points may lie exactly on the margin line. For these points, $y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1 = 0$. For the KKT condition to hold, their corresponding Lagrange multiplier α_i is allowed to be non-zero: $\boldsymbol{\alpha_i > 0}$.

A **support vector** is any data point $\mathbf{x}_i$ for which its corresponding optimal Lagrange multiplier α_i is strictly positive ($\alpha_i > 0$). Based on the Karush-Kuhn-Tucker (KKT) conditions, this has two profound and equivalent implications:

- Geometrically, support vectors are the data points that lie exactly on one of the margin hyperplanes. They are the points closest to the decision boundary.
- Mathematically, from the equation $\mathbf{w} = \sum \alpha_i y_i \mathbf{x}_i$, we can see that only the support vectors (where $\alpha_i > 0$) contribute to defining the weight vector $\mathbf{w}$. All other points have $\alpha_i = 0$ and have no say in where the decision boundary is placed.

In essence, these few critical points are "supporting" the entire structure of the decision boundary. If you were to remove any data point that is *not* a support vector, the solution would not change. But if you move a support vector, the optimal decision boundary itself will move.

2.4 Handling Non-Separable Data: The Soft Margin SVM

If the data is not linearly separable, the hard margin SVM has no solution. The soft margin SVM relaxes the constraints by introducing slack variables $\xi_i \geq 0$. The constraint becomes $y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 - \xi_i$. The optimization problem is modified to penalize points that violate the margin:

$$\begin{aligned} \min_{\mathbf{w},b,\boldsymbol{\xi}} \quad & \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n}\xi_i \\ \text{subject to} \quad & y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 - \xi_i, \quad i = 1, \dots, n \\ & \xi_i \geq 0, \quad i = 1, \dots, n \end{aligned}$$

The parameter $C > 0$ is a hyperparameter that controls the trade-off. A large C imposes a high penalty for margin violations, leading to a narrower margin, while a small C allows for more violations, resulting in a wider margin.

The dual problem for the soft margin SVM is very similar, with an additional constraint on the Lagrange multipliers:

$$0 \leq \alpha_i \leq C$$

2.5 Non-Linear Classification: The Kernel Trick

The true power of SVMs comes from their ability to handle non-linearly separable data. The idea is to map the data $\mathbf{x}$ to a higher-dimensional feature space using a mapping $\phi(\mathbf{x})$, where the data becomes linearly separable. Notice that the dual formulation only depends on the dot product of feature vectors, $\mathbf{x}_i \cdot \mathbf{x}_j$. The kernel trick allows us to replace this dot product with a **kernel function** $K(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_j)$. This computes the dot product in the high-dimensional space without ever explicitly performing the mapping ϕ.

2.6 Example Question

Question:

Find the maximal margin hyperplane for the following 2D data:

- Class +1: (3, 1), (3, -1)
- Class -1: (1, 1), (1, -1)

Solution:

By inspecting the data, we can see that it is linearly separable and the separating hyperplane should be a vertical line between $x = 1$ and $x = 3$. The optimal hyperplane will be equidistant from these points, so we can guess its equation is $x = 2$. Let's verify this using the SVM formulation.

Let the hyperplane be $\mathbf{w} \cdot \mathbf{x} + b = 0$. From our guess, $\mathbf{w} = (1, 0)$ and $b = -2$. All four points lie on the margin boundaries, so they are all support vectors. Let's check the constraint $y_i(\mathbf{w} \cdot \mathbf{x}_i + b) = 1$ for the support vectors.

- For point (3, 1) with label $y = +1$:

$$+1 \times ((1, 0) \cdot (3, 1) - 2) = 1 \times (1(3) + 0(1) - 2) = 3 - 2 = 1$$

  The constraint is satisfied.

- For point (3, -1) with label $y = +1$:

$$+1 \times ((1, 0) \cdot (3, -1) - 2) = 1 \times (1(3) + 0(-1) - 2) = 3 - 2 = 1$$

  The constraint is satisfied.

- For point (1, 1) with label $y = -1$:

$$-1 \times ((1, 0) \cdot (1, 1) - 2) = -1 \times (1(1) + 0(1) - 2) = -1 \times (1 - 2) = 1$$

  The constraint is satisfied.

- For point (1, -1) with label $y = -1$:

$$-1 \times ((1, 0) \cdot (1, -1) - 2) = -1 \times (1(1) + 0(-1) - 2) = -1 \times (1 - 2) = 1$$

  The constraint is satisfied.

Since all constraints are met exactly as equalities, our guess for the hyperplane is correct. The optimal hyperplane is $\mathbf{w} = (1, 0)$ and $b = -2$, with the equation:

$$x - 2 = 0$$

The margin boundaries are $x - 2 = 1$ (i.e., $x = 3$) and $x - 2 = -1$ (i.e., $x = 1$). The geometric margin is $\frac{2}{\|\mathbf{w}\|} = \frac{2}{\sqrt{1^2+0^2}} = 2$. This is the distance between the lines $x = 1$ and $x = 3$. The objective function value is $\frac{1}{2}\|\mathbf{w}\|^2 = \frac{1}{2}(1^2) = 0.5$, which is minimized.

3 Reproducing Kernels

3.1 Key Concepts

- **Feature Map (ϕ):** A function that maps input data from its original space $\mathcal{X}$ to a higher-dimensional feature space $\mathcal{F}$. The idea is that data which is not linearly separable in $\mathcal{X}$ may become linearly separable in $\mathcal{F}$. For an input vector $\mathbf{x} \in \mathcal{X}$, the mapped vector is $\phi(\mathbf{x}) \in \mathcal{F}$.

- **Kernel Function (K):** A function $K(\mathbf{x}, \mathbf{z})$ that computes the dot product of two vectors $\mathbf{x}$ and $\mathbf{z}$ in the feature space $\mathcal{F}$, without ever explicitly calculating the mapping ϕ. That is, $K(\mathbf{x}, \mathbf{z}) = \langle \phi(\mathbf{x}), \phi(\mathbf{z}) \rangle = \phi(\mathbf{x})^T \phi(\mathbf{z})$.
- **The Kernel Trick:** The core idea of replacing all instances of dot products in a machine learning algorithm (like the dual form of an SVM) with a kernel function. This allows the algorithm to operate in a high-dimensional feature space without incurring the computational cost of explicitly mapping the data points into that space. This is especially powerful for feature spaces that are infinite-dimensional.
- **Reproducing Kernel Hilbert Space (RKHS):** A vector space of functions equipped with an inner product. For every valid kernel function, there exists a unique RKHS where the kernel acts as the reproducing kernel. This means the dot product in the RKHS corresponds to an evaluation of the kernel function.
- **Mercer's Theorem:** A mathematical theorem that provides the necessary and sufficient conditions for a function $K(\mathbf{x}, \mathbf{z})$ to be a valid kernel. A symmetric function K is a valid kernel if and only if the Gram matrix generated from any set of data points is positive semi-definite.

3.2 The Kernel Trick in Support Vector Machines

The power of kernels is most evident in the context of Support Vector Machines. The optimization problem for a linear SVM is typically solved in its dual form, which is expressed in terms of dot products of the input data points.

The dual Lagrangian to be maximized is:

$$L_D(\boldsymbol{\alpha}) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j (\mathbf{x}_i^T \mathbf{x}_j)$$

subject to $\sum_{i=1}^{m} \alpha_i y_i = 0$ and $0 \leq \alpha_i \leq C$ for all i.

Notice the term $(\mathbf{x}_i^T \mathbf{x}_j)$. This is a dot product in the original input space. When the data is not linearly separable, we can apply a feature mapping $\phi : \mathcal{X} \to \mathcal{F}$. In this new feature space, the dot product becomes $(\phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j))$. The dual objective function transforms to:

$$L_D(\boldsymbol{\alpha}) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j (\phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j))$$

The **kernel trick** is to replace the computationally expensive inner product $\phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)$ with a kernel function $K(\mathbf{x}_i, \mathbf{x}_j)$:

$$L_D(\boldsymbol{\alpha}) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j)$$

Similarly, the decision function for a new point $\mathbf{z}$ becomes:

$$f(\mathbf{z}) = \text{sign}\left(\sum_{i=1}^{m} \alpha_i y_i K(\mathbf{x}_i, \mathbf{z}) + b \right)$$

where the sum is typically only over the support vectors (for which $\alpha_i > 0$). By using a kernel, we can find a non-linear decision boundary in the original space, corresponding to a linear hyperplane in the high-dimensional feature space, without ever needing to know what ϕ is.

3.3 Common Kernel Functions

- **Linear Kernel:** This is the simplest kernel, representing the dot product in the original space. It is used when the data is already linearly separable.

  $$K(\mathbf{x}, \mathbf{z}) = \mathbf{x}^T \mathbf{z}$$

  The corresponding feature map is the identity: $\phi(\mathbf{x}) = \mathbf{x}$.

- **Polynomial Kernel:** This kernel maps data to a feature space of polynomial combinations of the original features. It is useful for problems where the decision boundary is polynomial.

$$K(\mathbf{x}, \mathbf{z}) = (\gamma \mathbf{x}^T \mathbf{z} + r)^d$$

  Here, d is the degree of the polynomial, γ is a scaling factor, and r is a constant offset.

- **Gaussian (Radial Basis Function - RBF) Kernel:** This is one of the most popular and powerful kernels. It can handle complex, non-linear relationships and maps the data to an infinite-dimensional feature space.

$$K(\mathbf{x}, \mathbf{z}) = \exp(-\gamma \|\mathbf{x} - \mathbf{z}\|^2)$$

  The parameter γ controls the width of the Gaussian. A small γ means the influence of a single training example is far-reaching (smoother decision boundary), while a large γ results in a more localized influence (more complex, potentially overfitted boundary).

3.4 Conditions for a Valid Kernel: Mercer's Theorem

Not any function can be used as a kernel. A function must correspond to a dot product in some feature space. Mercer's theorem provides a formal criterion to check this.

Theorem (Mercer): Let $\mathcal{X}$ be a compact subset of $\mathbb{R}^n$. A symmetric function $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a valid kernel if and only if for any finite set of points $\{\mathbf{x}_1, \ldots, \mathbf{x}_m\} \subset \mathcal{X}$, the matrix $\mathbf{K} \in \mathbb{R}^{m \times m}$ defined by $K_{ij} = K(\mathbf{x}_i, \mathbf{x}_j)$ is positive semi-definite.

A matrix $\mathbf{K}$ is positive semi-definite if for any non-zero vector $\mathbf{c} \in \mathbb{R}^m$, we have $\mathbf{c}^T \mathbf{K} \mathbf{c} \geq 0$. This theorem is crucial because it allows us to design and validate new kernel functions without having to explicitly find the corresponding feature map ϕ.

3.5 Example Question

Question:

Consider a dataset with four points in $\mathbb{R}^2$:

- Class +1: $\mathbf{x}_1 = (1, 1)$, $\mathbf{x}_2 = (-1, -1)$
- Class -1: $\mathbf{x}_3 = (1, -1)$, $\mathbf{x}_4 = (-1, 1)$

This dataset is not linearly separable in $\mathbb{R}^2$ (it's the classic XOR problem). Show that by using the polynomial kernel $K(\mathbf{x}, \mathbf{z}) = (\mathbf{x}^T \mathbf{z})^2$, the data becomes linearly separable in the corresponding feature space.

Solution:

Step 1: Find the feature map ϕ for the kernel.

Let $\mathbf{x} = (x_1, x_2)$ and $\mathbf{z} = (z_1, z_2)$. The kernel function is:

$$K(\mathbf{x}, \mathbf{z}) = (\mathbf{x}^T \mathbf{z})^2 = (x_1 z_1 + x_2 z_2)^2 = x_1^2 z_1^2 + 2 x_1 x_2 z_1 z_2 + x_2^2 z_2^2$$

We need to find a mapping $\phi(\mathbf{x})$ such that $K(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x})^T \phi(\mathbf{z})$. We can rewrite the expanded form as a dot product:

$$K(\mathbf{x}, \mathbf{z}) = (x_1^2, \sqrt{2} x_1 x_2, x_2^2) \cdot (z_1^2, \sqrt{2} z_1 z_2, z_2^2)$$

Thus, the feature map is $\phi(\mathbf{x}) = (x_1^2, \sqrt{2} x_1 x_2, x_2^2)$. This maps the 2D input data into a 3D feature space.

Step 2: Map the data points into the feature space.

Now we apply this mapping to our four data points:

- $\phi(\mathbf{x}_1) = \phi(1,1) = (1^2, \sqrt{2}(1)(1), 1^2) = (1, \sqrt{2}, 1)$
- $\phi(\mathbf{x}_2) = \phi(-1,-1) = ((-1)^2, \sqrt{2}(-1)(-1), (-1)^2) = (1, \sqrt{2}, 1)$
- $\phi(\mathbf{x}_3) = \phi(1,-1) = (1^2, \sqrt{2}(1)(-1), (-1)^2) = (1, -\sqrt{2}, 1)$
- $\phi(\mathbf{x}_4) = \phi(-1,1) = ((-1)^2, \sqrt{2}(-1)(1), 1^2) = (1, -\sqrt{2}, 1)$

Step 3: Check for linear separability in the new space.

Let the mapped points be $\mathbf{z}_i = \phi(\mathbf{x}_i)$.

- Class +1 points: $\mathbf{z}_1 = (1, \sqrt{2}, 1)$, $\mathbf{z}_2 = (1, \sqrt{2}, 1)$
- Class -1 points: $\mathbf{z}_3 = (1, -\sqrt{2}, 1)$, $\mathbf{z}_4 = (1, -\sqrt{2}, 1)$

In this 3D feature space, the points of class +1 have their second coordinate as $+\sqrt{2}$, while the points of class -1 have it as $-\sqrt{2}$. All points lie on the plane defined by their first coordinate being 1 and third coordinate being 1.

We can easily separate these two classes with the hyperplane defined by the equation $z_2 = 0$. For example, a linear classifier with weight vector $\mathbf{w} = (0,1,0)$ and bias $b = 0$ would classify:

- For class +1: $\mathbf{w}^T\mathbf{z}_{1,2} + b = (0,1,0)^T(1, \sqrt{2}, 1) = \sqrt{2} > 0$.
- For class -1: $\mathbf{w}^T\mathbf{z}_{3,4} + b = (0,1,0)^T(1, -\sqrt{2}, 1) = -\sqrt{2} < 0$.

Since a separating hyperplane exists, the data is linearly separable in the feature space. This demonstrates the power of the kernel method to handle non-linear data by implicitly projecting it into a higher-dimensional space where it becomes linearly separable.

4 SVM for Regression (SVR)

Support Vector Regression (SVR) is an adaptation of Support Vector Machines (SVM) to solve regression problems. While classification SVMs aim to find a hyperplane that best separates two classes, SVR aims to find a hyperplane that best fits a set of continuous-valued data points. The core idea is to tolerate errors within a certain threshold, known as the epsilon-insensitive tube, and find a function that is as flat as possible.

4.1 Key Concepts

- **Hyperplane in Regression:** In SVR, the goal is to find a function, typically linear of the form $f(x) = \mathbf{w} \cdot \mathbf{x} + b$, that best predicts the target values y_i.
- **Epsilon-Insensitive Tube (ϵ-tube):** This is a central concept in SVR. It refers to a tube of width 2ϵ around the regression function $f(x)$. The loss for any data point (x_i, y_i) is zero if its predicted value $f(x_i)$ is within a distance of ϵ from the true value y_i. That is, if $|y_i - f(x_i)| \leq \epsilon$. Errors are only penalized for points that fall outside this tube.
- **Slack Variables (ξ_i, ξ_i^*):** To handle points that lie outside the ϵ-tube (i.e., non-zero errors), two sets of slack variables are introduced. ξ_i measures the distance of a point above the upper boundary of the tube, and ξ_i^* measures the distance of a point below the lower boundary.
- **Support Vectors:** In SVR, support vectors are the data points that lie on or outside the boundary of the ϵ-insensitive tube. These are the only points that influence the position and orientation of the regression hyperplane. Points inside the tube have no effect on the final model.

- **Regularization Parameter (C):** This hyperparameter controls the trade-off between the flatness of the function (minimizing the norm of the weight vector,$\|w\|^2$) and the tolerance for errors outside the ϵ-tube. A large value of C imposes a higher penalty on errors, forcing the model to fit the training data more strictly, potentially leading to overfitting. A smaller C allows for a larger margin and a flatter, simpler model.
- **Kernel Trick:** For non-linear regression problems, the kernel trick is used to map the input data into a higher-dimensional feature space where a linear regression model can be effectively applied. This is done implicitly by replacing dot products in the algorithm with a kernel function $K(x_i, x_j)$.

4.2 Mathematical Formulation

4.2.1 Primal Form

The objective of SVR is to find a function $f(x) = \mathbf{w} \cdot \mathbf{x} + b$ that is as flat as possible while fitting the data. Minimizing $\|w\|^2$ corresponds to maximizing the flatness of the function. This leads to the following constrained optimization problem, known as the primal form:

$$\min_{\mathbf{w},b,\xi,\xi^*} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{N}(\xi_i + \xi_i^*)$$

subject to:

$$y_i - (\mathbf{w} \cdot \mathbf{x}_i + b) \le \epsilon + \xi_i$$
$$(\mathbf{w} \cdot \mathbf{x}_i + b) - y_i \le \epsilon + \xi_i^*$$
$$\xi_i, \xi_i^* \ge 0 \quad \text{for } i = 1, \ldots, N$$

Let's break down the objective function we are trying to minimize:

- **The Function** $f(x) = \mathbf{w} \cdot \mathbf{x} + b$: This is the linear regression function we are trying to find. $\mathbf{w}$ is the weight vector and b is the bias term.
- **Minimizing** $\frac{1}{2}\|\mathbf{w}\|^2$: This is the key to making the function "as flat as possible." A smaller norm of the weight vector ($\|\mathbf{w}\|$) corresponds to a less complex function that is less sensitive to small changes in the input features. This is a form of regularization that helps prevent overfitting.
- **The ϵ-insensitive Tube**: The first two constraints define a "tube" or margin of tolerance around our regression line. They state that the error for any data point i must be within this tolerance, ϵ.
  - The first constraint, $y_i - f(\mathbf{x}_i) \le \epsilon$, handles points that are *above* the tube.
  - The second constraint, $f(\mathbf{x}_i) - y_i \le \epsilon$, handles points that are *below* the tube.

  Essentially, if a point's prediction is within a distance of ϵ from its true value y_i, we consider the error to be zero.
- **Slack Variables ξ_i and ξ_i^***: In reality, not all data points will fall neatly inside our tube. The slack variables ξ_i (for points above the tube) and ξ_i^* (for points below the tube) measure the magnitude of the error for points that lie *outside* the ϵ-tube. They are the "soft margin" component, allowing for some errors.
- **The Penalty Term** $C\sum(\xi_i + \xi_i^*)$: This part of the objective function represents the penalty for all errors. The parameter $C > 0$ is a hyperparameter that controls the trade-off between the flatness of our function ($\|\mathbf{w}\|^2$) and the amount of error we are willing to tolerate.
  - A **small** C makes the penalty for errors smaller, prioritizing a flatter function (larger margin) even if it means mis-predicting some training points.
  - A **large** C imposes a high penalty on errors, forcing the model to fit the training data more closely, potentially at the cost of a less flat function (a more complex model).

4.2.2 The Dual Formulation: A More Practical Approach

Solving the primal problem directly can be computationally challenging. The dual formulation, derived using Lagrange multipliers, is often more efficient and, critically, it enables the use of the kernel trick for non-linear problems.

$$\max_{\alpha,\alpha^*} -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}(\alpha_i-\alpha_i^*)(\alpha_j-\alpha_j^*)(\mathbf{x}_i\cdot\mathbf{x}_j)-\epsilon\sum_{i=1}^{N}(\alpha_i+\alpha_i^*)+\sum_{i=1}^{N}y_i(\alpha_i-\alpha_i^*)$$

subject to:

$$\sum_{i=1}^{N}(\alpha_i-\alpha_i^*)=0$$
$$0\le\alpha_i,\alpha_i^*\le C$$

Key insights from the dual formulation:

- **Lagrange Multipliers** α_i, α_i^*: Each data point $(\mathbf{x}_i, y_i)$ gets a pair of non-negative Lagrange multipliers, α_i and α_i^*. These multipliers are what we solve for in the dual problem. They represent the "importance" of each data point in defining the regression function.
- **The Kernel Trick Entry Point**: Notice the term $(\mathbf{x}_i \cdot \mathbf{x}_j)$. The entire optimization problem now depends only on the dot products between data points, not on the individual weights $\mathbf{w}$. This is what allows us to later substitute this dot product with a kernel function $K(\mathbf{x}_i, \mathbf{x}_j)$ to handle non-linear data.
- **The Weight Vector w**: A profound result from the optimization process (specifically, the Karush-Kuhn-Tucker or KKT conditions) is that the weight vector can be expressed as:

  $$\mathbf{w}=\sum_{i=1}^{N}(\alpha_i-\alpha_i^*)\mathbf{x}_i$$

  This shows that $\mathbf{w}$ is simply a linear combination of the input data points.
- **Support Vectors**: The KKT conditions also dictate that the multipliers α_i, α_i^* will be non-zero *only* for the data points that lie on or outside the ϵ-tube boundary. These points, for which $(\alpha_i - \alpha_i^*) \neq 0$, are the **support vectors**. They are the only points that contribute to the final solution for $\mathbf{w}$. All points inside the tube will have $\alpha_i = \alpha_i^* = 0$.

4.3 The SVR Algorithm

4.3.1 Training Phase

This is the learning stage where the model finds the optimal parameters from the data.

1. **Input:** We begin with our labeled training data, a choice of kernel function (e.g., Linear, Polynomial, RBF), and values for the hyperparameters C and ϵ. The choice of these parameters is critical for model performance and is often done via cross-validation.
2. **Construct QP Problem:** We use the input data and parameters to set up the dual objective function and its constraints. This is now a standard Quadratic Programming (QP) problem, which involves maximizing a quadratic function subject to linear constraints.
3. **Solve for Multipliers:** We use a specialized numerical optimization algorithm, known as a QP solver, to find the optimal values for the Lagrange multipliers α_i and α_i^*.
4. **Identify Support Vectors:** After solving, we inspect the multipliers. Any data point $\mathbf{x}_i$ whose corresponding multipliers α_i or α_i^* are non-zero is identified as a support vector. These are the critical points that define our model.

5. **Compute Bias Term:** The bias term b is calculated using a support vector $\mathbf{x}_k$ that lies exactly on the boundary of the ϵ-tube. For such a point, its corresponding multiplier will be between 0 and C (i.e., $0 < \alpha_k < C$ or $0 < \alpha_k^* < C$) and its slack variable will be zero. We can then rearrange the boundary condition equations to solve for b. It's common practice to calculate b for all such support vectors and then average the results.

4.3.2 Prediction Phase

Once the model is trained, making predictions is straightforward.

1. **Input:** We take a new, unseen data point, $\mathbf{x}_{new}$.
2. **Compute Prediction:** We use the prediction function with the parameters we just learned. The key here is that the summation is only performed over the set of support vectors (SVs), as all other points have $(\alpha_i - \alpha_i^*) = 0$:
$$y_{pred} = f(\mathbf{x}_{new}) = \sum_{i \in \text{SVs}} (\alpha_i - \alpha_i^*) K(\mathbf{x}_i, \mathbf{x}_{new}) + b$$
This makes the prediction phase very efficient, as we only need to consider the (often small) subset of training points that are support vectors.

5 Regression and Classification Trees

Decision Trees are a non-parametric supervised learning method used for both classification and regression tasks. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. The model is represented as a tree structure, where each internal node represents a test on an attribute, each branch represents the outcome of the test, and each leaf node represents a class label (in classification) or a continuous value (in regression).

5.1 Key Concepts

- **Decision Tree:** A hierarchical, flowchart-like structure where each internal node denotes a test on an attribute, each branch represents an outcome of the test, and each leaf node (terminal node) holds a class label or a continuous value.
- **Root Node:** The top-most node in the tree, representing the entire dataset, which gets partitioned into two or more homogeneous sets.
- **Splitting:** The process of dividing a node into two or more sub-nodes based on a certain feature and split point.
- **Internal Node (Decision Node):** A node that splits into further sub-nodes. It represents a decision rule on a feature.
- **Leaf Node (Terminal Node):** Nodes that do not split further. They represent the final outcome or prediction for a given set of input features.
- **Pruning:** The process of removing sections of the tree (sub-nodes) that provide little predictive power, in order to simplify the model and prevent overfitting.
- **Impurity:** A measure of the homogeneity of the labels at a node. A node is considered 'pure' if all its samples belong to the same class. Common impurity measures are Entropy and Gini Index.
- **Information Gain:** The reduction in impurity or entropy achieved by splitting a dataset on a particular attribute. The attribute with the highest information gain is chosen for the split.

5.2 Classification Trees

Classification trees are used when the target variable is categorical. The tree is built by recursively partitioning the data into subsets based on feature values, with the goal of making the resulting subsets as 'pure' as possible.

5.2.1 Impurity Measures

To decide which feature to split on at each step, we need a measure of how 'mixed' a group of samples is. This is called impurity.

- **Entropy:** A measure of uncertainty or randomness in a set of samples S. For a set with c classes, the entropy is defined as:

$$H(S) = -\sum_{i=1}^{c} p_i \log_2(p_i)$$

  where p_i is the proportion of samples in set S that belong to class i. Entropy is 0 if all samples belong to one class (pure node) and is maximized when the classes are perfectly mixed (e.g., 50/50 for two classes).

- **Gini Index (Gini Impurity):** Measures the frequency with which any element from the set would be incorrectly labeled if it was randomly labeled according to the distribution of labels in the subset. It is defined as:

$$G(S) = 1 - \sum_{i=1}^{c} p_i^2$$

  Like entropy, the Gini index is 0 for a pure node and reaches a maximum for a perfectly mixed node.

5.2.2 ID3 Algorithm

Information Gain (IG) is the primary criterion used by algorithms like ID3 to construct a decision tree. It measures the expected reduction in entropy caused by partitioning the data on an attribute A. The attribute with the highest information gain is chosen for the split.

The Information Gain for a split on attribute A is calculated as:

$$IG(S, A) = H(S) - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} H(S_v)$$

Where:

- $H(S)$ is the entropy of the parent node (set S).
- Values(A) is the set of all possible values for attribute A.
- S_v is the subset of S for which attribute A has value v.
- $|S_v|$ is the number of samples in the subset S_v.
- $|S|$ is the total number of samples in the parent set S.

5.2.3 CART Algorithm (Classification and Regression Trees)

The core principle of CART is **recursive partitioning**. The algorithm starts with the entire dataset at the root node and recursively splits the data into two more homogeneous child nodes. The "best" split is chosen by evaluating every feature and every possible split point for that feature, using a specific impurity metric. This process continues until a stopping criterion is met, at which point a node becomes a terminal **leaf node**.

In a classification setting, the goal is to predict a categorical label. The CART algorithm constructs a tree where each leaf node represents a class label. To decide which split is best, CART for classification uses the **Gini Impurity** metric.

- A Gini Impurity of **0** indicates that the node is perfectly "pure"—all samples in the node belong to a single class.
- A Gini Impurity of **0.5** (for a binary classification problem) indicates the worst-case scenario where the samples are split evenly between the two classes.

When considering a potential split, the algorithm calculates the weighted Gini Impurity of the two resulting child nodes (left and right). The algorithm seeks to find the feature and split point that minimizes the weighted average impurity of the children nodes.

Once the tree is built, a new, unseen data point is classified by traversing the tree from the root down to a leaf node based on the feature values of the data point. The prediction is the **majority class** (or mode) of the training samples that ended up in that particular leaf node.

5.3 Regression Trees

Regression trees are used when the target variable is continuous. The structure and building process are similar to classification trees, but the splitting criterion and prediction method are different. The structure of the CART algorithm remains the same, but the impurity metric and prediction method are adapted for continuous outcomes.

5.3.1 The Splitting Criterion: Sum of Squared Errors (SSE)

For regression trees, the concept of "impurity" is replaced with a measure of variance. The most common metric used is the **Sum of Squared Errors (SSE)** or Mean Squared Error (MSE). The goal is to create splits that result in child nodes where the target values are as close to their mean as possible. For a given node t, let S_t be the set of training samples in that node, and let $\bar{y}_t$ be the mean of the target values for these samples. The SSE for the node is:

$$SSE(t) = \sum_{i \in S_t} (y_i - \bar{y}_t)^2$$

where $\bar{y}_t = \frac{1}{|S_t|} \sum_{i \in S_t} y_i$.

When evaluating a split, the algorithm calculates the combined SSE of the two potential child nodes. The best split is the one that **minimizes** the total SSE of the children:

$$\text{Minimize:} \quad SSE(t_L) + SSE(t_R)$$

This is equivalent to maximizing the reduction in variance.

5.3.2 Prediction in Regression Trees

Making a prediction for a new data point follows the same traversal process as in the classification tree. The data point is passed down the tree until it reaches a leaf node. The prediction is then the **mean** of the target values of all the training samples contained within that leaf node.

5.4 Overfitting, Pruning, and Stopping Criteria

Decision trees are prone to overfitting, meaning they can learn the training data too well, including its noise, and fail to generalize to new, unseen data. A fully grown tree can have leaves that correspond to single training instances.

To combat overfitting, two main strategies are used:

1. **Pre-pruning (Early Stopping):** This involves stopping the tree-building process before it becomes overly complex. Common stopping criteria include:

- Limiting the maximum depth of the tree.
- Setting a minimum number of samples required to split a node.
- Stopping if the leaf node is a pure node

2. **Post-pruning:** This involves growing the tree to its full complexity and then trimming branches in a bottom-up fashion.

6 Random Forest

A Random Forest is a powerful and versatile supervised machine learning algorithm that belongs to the family of ensemble methods. It operates by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees. It corrects for the decision tree's habit of overfitting to its training set.

6.1 Key Concepts

- **Ensemble Learning:** The core principle is that a committee of individual models, when combined, can produce a more accurate and robust prediction than any single model. Random Forest is an ensemble of decision trees.
- **Bagging (Bootstrap Aggregating):** This is the technique used to generate different training subsets for each tree. For a training set of size N, a bootstrap sample is created by randomly drawing N samples *with replacement*. On average, a bootstrap sample contains about 63.2% of the original instances, with some instances appearing multiple times.
- **Feature Randomness:** This is the key innovation of Random Forest over simple bagging of decision trees. When building each tree, at every split point, the algorithm does not consider all available features. Instead, it selects a random subset of features and finds the optimal split only within that subset. This process de-correlates the trees, reducing the variance of the overall model.
- **Aggregation:** Once the forest of trees is trained, a new instance is passed through all the trees. The final prediction is determined by combining the outputs of all trees.
  - For **classification tasks**, the final prediction is the class that receives the most votes (majority vote).
  - For **regression tasks**, the final prediction is the average of the predictions from all the individual trees.
- **Out-of-Bag (OOB) Error:** Since each tree is trained on a bootstrap sample, a portion of the original data (the out-of-bag samples) is left out. The OOB error is an unbiased estimate of the test set error, calculated by making predictions for each data point using only the trees that did not have that data point in their bootstrap sample.
- **Feature Importance:** Random Forest provides a robust way to measure the importance of each feature. The importance is typically calculated by measuring how much the model's prediction error (usually the OOB error) increases when that feature's values are randomly permuted.

6.2 Algorithm

The Random Forest algorithm can be broken down into two phases: training the forest and making predictions.

6.2.1 Training Algorithm

Let the training data be $D = \{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\}$, where x_i is a feature vector and y_i is the label. We want to build a forest with B trees. Let the number of features be p. A common choice for the number of features to sample at each split, m_{try}, is $\sqrt{p}$ for classification and $p/3$ for regression.

Steps:

1. For $b = 1$ to B (for each tree in the forest):
   (a) Create a bootstrap sample D_b by drawing N samples with replacement from the original training data D.
   (b) Grow a decision tree T_b on the bootstrap sample D_b. To grow the tree, for each node:
      i. Randomly select m_{try} features from the total p features.
      ii. Pick the best feature and split point among the selected m_{try} features using a criterion like Gini impurity or Information Gain.
      iii. Split the node into two child nodes.
   (c) Continue this process recursively until a stopping criterion is met (e.g., the node is pure, or a maximum depth is reached). The trees are grown to their maximum possible size and are not pruned.
2. Output the ensemble of trees $\{T_1, T_2, \ldots, T_B\}$.

6.2.2 Prediction Algorithm

To make a prediction for a new instance x', we aggregate the predictions from all the trees in the forest.

- **For Classification:** Let $\hat{C}_b(x')$ be the class prediction of the b-th tree. The final prediction is the majority vote:
$$\hat{C}_{\text{rf}}(x') = \text{majority\_vote}\{\hat{C}_1(x'), \hat{C}_2(x'), \ldots, \hat{C}_B(x')\}$$
- **For Regression:** Let $\hat{T}_b(x')$ be the value predicted by the b-th tree. The final prediction is the average of all predictions:
$$\hat{T}_{\text{rf}}(x') = \frac{1}{B}\sum_{b=1}^{B} \hat{T}_b(x')$$

6.3 Mathematical Details and Properties

6.3.1 Variance Reduction

The primary benefit of Random Forest is the reduction in variance compared to a single decision tree, thus preventing overfitting.

6.3.2 Out-of-Bag (OOB) Error Estimation

The OOB error provides an unbiased estimate of the generalization error without requiring a separate validation set. The procedure is as follows:

1. For each instance (x_i, y_i) in the original dataset D, identify the set of trees that did not use (x_i, y_i) in their training. Let's call this set $T_{\text{oob},i}$.
2. To obtain the OOB prediction for x_i, aggregate the predictions $\hat{y}_{\text{oob},i}$ only from the trees in $T_{\text{oob},i}$.
3. The overall OOB error is the average of the errors for all instances:
$$E_{\text{OOB}} = \frac{1}{N}\sum_{i=1}^{N} L(y_i, \hat{y}_{\text{oob},i})$$
   where L is a loss function (e.g., 0-1 loss for classification, squared error for regression).