R23 **III B.Tech I Semester**

Department of Artificial Intelligence and Machine Learning
Subject: Advanced Machine Learning 23A03351T
Unit-I

Bias and Variance:

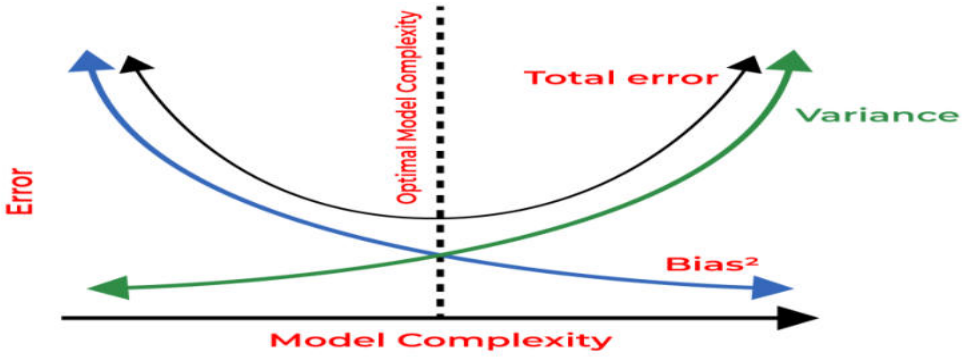
- **Bias** refers to the errors which occur when we try to fit a statistical model on real-world data which does not fit perfectly well on some mathematical model. If we use a way too simplistic a model to fit the data then we are more probably face the situation of **High Bias** (underfitting) refers to the case when the model is unable to learn the patterns in the data at hand and perform poorly.
- **Variance** shows the error value that occurs when we try to make predictions by using data that is not previously seen by the model. There is a situation known as **high variance** (overfitting) that occurs when the model learns noise that is present in the data.

Finding a proper balance between the two is also known as the **Bias-Variance Tradeoff** which helps us to design an accurate model.

Bias Variance tradeoff

The **Bias-Variance Tradeoff** refers to the balance between bias and variance which affect predictive model performance. Finding the right tradeoff is important for creating models that generalize well to new data.

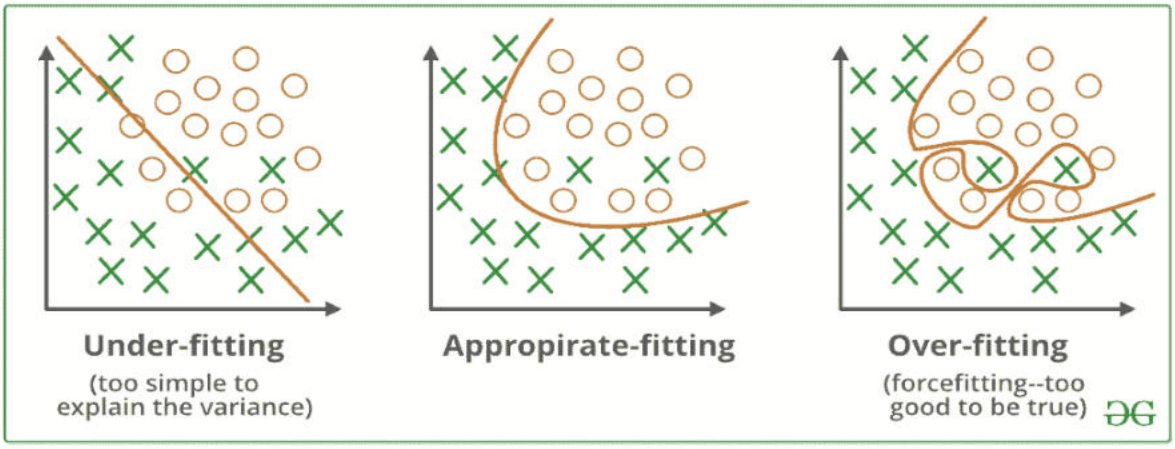
- The **bias-variance tradeoff** shows the inverse relationship between bias and variance. When one decreases, the other tends to increase and vice versa.
- Finding the right balance is important. An overly simple model with high bias won't capture the underlying patterns while an overly complex model with high variance will fit the noise in the data.



Overfitting and Underfitting:

Overfitting and **underfitting** are terms used to describe the performance of machine learning models in relation to their ability to generalize from the training data to unseen data.

Department of Artificial Intelligence and Machine Learning
Subject: Advanced Machine Learning 23A03351T
Unit-I



Overfitting happens when a machine learning model learns the training data too well including the noise and random details. This makes the model to perform poorly on new, unseen data because it memorizes the training data instead of understanding the general patterns.
For example, if we only study last week's weather to predict tomorrow's i.e our model might focus on one-time events like a sudden rainstorm which won't help for future predictions.

Underfitting is the opposite problem which happens when the model is too simple to learn even the basic patterns in the data. An underfitted model performs poorly on both training and new data. To fix this we need to make the model more complex or add more features.
For example if we use only the average temperature of the year to predict tomorrow's weather hence the model misses important details like seasonal changes which results in bad predictions.

Department of Artificial Intelligence and Machine Learning
Subject: Advanced Machine Learning 23A03351T
Unit-I

Ensemble Learning

Ensemble learning is a method where we use many small models instead of just one. Each of these models may not be very strong on its own, but when we put their results together, we get a better and more accurate answer. It's like asking a group of people for advice instead of just one person—each one might be a little wrong, but together, they usually give a better answer.

Types of Ensembles Learning in Machine Learning

There are three main types of ensemble methods:

1. **Bagging (Bootstrap Aggregating):**
   Models are trained independently on different random subsets of the training data. Their results are then combined—usually by averaging (for regression) or voting (for classification). This helps reduce variance and prevents overfitting.
2. **Boosting:**
   Models are trained one after another. Each new model focuses on fixing the errors made by the previous ones. The final prediction is a weighted combination of all models, which helps reduce bias and improve accuracy.
3. **Stacking (Stacked Generalization):**
   Multiple different models (often of different types) are trained, and their predictions are used as inputs to a final model, called a meta-model. The meta-model learns how to best combine the predictions of the base models, aiming for better performance than any individual model.

1. Bagging Algorithm

Bagging classifier can be used for both regression and classification tasks. Here is an overview of Bagging classifier algorithm**:**

- **Bootstrap Sampling:** Divides the original training data into 'N' subsets and randomly selects a subset with replacement in some rows from other subsets. This step ensures that the base models are trained on diverse subsets of the data and there is no class imbalance.
- Base Model Training: For each bootstrapped sample we train a base model independently on that subset of data. These weak models are trained in parallel to increase computational efficiency and reduce time consumption. We can use different base learners i.e. different ML models as base learners to bring variety and robustness.
- **Prediction Aggregation:** To make a prediction on testing data combine the predictions of all base models. For classification tasks it can include majority voting or weighted majority while for regression it involves averaging the predictions.
- **Out-of-Bag (OOB) Evaluation**: Some samples are excluded from the training subset of particular base models during the bootstrapping method. These "out-of-bag" samples can be used to estimate the model's performance without the need for cross-validation.
- **Final Prediction:** After aggregating the predictions from all the base models, Bagging produces a final prediction for each instance.

Python pseudo code for Bagging Estimator implementing libraries:

1. Importing Libraries and Loading Data

- **BaggingClassifier:** for creating an ensemble of classifiers trained on different subsets of data.
- **DecisionTreeClassifier:** the base classifier used in the bagging ensemble.
- **load_iris:** to load the Iris dataset for classification.
- **train_test_split:** to split the dataset into training and testing subsets.
- **accuracy_score**: to evaluate the model's prediction accuracy.

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

2. Loading and Splitting the Iris Dataset

- **data = load_iris():** loads the Iris dataset, which includes features and target labels.
- **X = data.data:** extracts the feature matrix (input variables).
- **y = data.target:** extracts the target vector (class labels).
- **train_test_split(...):** splits the data into training (80%) and testing (20%) sets, with random_state=42 to ensure reproducibility.

```
data = load_iris()
X = data.data
y = data.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

3. Creating a Base Classifier

Decision tree is chosen as the base model. They are prone to overfitting when trained on small datasets making them good candidates for bagging.

- **base_classifier = DecisionTreeClassifier()**: initializes a Decision Tree classifier, which will serve as the base estimator in the Bagging ensemble.

```
base_classifier = DecisionTreeClassifier()
```

4. Creating and Training the Bagging Classifier

- A **BaggingClassifier** is created using the decision tree as the base classifier.
- **n_estimators = 10** specifies that 10 decision trees will be trained on different bootstrapped subsets of the training data.

```
bagging_classifier = BaggingClassifier(base_classifier, n_estimators=10, random_state=42)
bagging_classifier.fit(X_train, y_train)
```

5. Making Predictions and Evaluating Accuracy

- The trained bagging model predicts labels for test data.
- The accuracy of the predictions is calculated by comparing the predicted labels (**y_pred**) to the actual labels (**y_test**).

```
y_pred = bagging_classifier.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

Output:

Accuracy: 1.0

2. Boosting Algorithm

Boosting is an ensemble technique that combines multiple weak learners to create a strong learner. Weak models are trained in series such that each next model tries to correct errors of the previous model until the entire training dataset is predicted correctly. One of the most well-

known boosting algorithms is AdaBoost (Adaptive Boosting). Here is an overview of Boosting algorithm:

- **Initialize Model Weights**: Begin with a single weak learner and assign equal weights to all training examples.
- **Train Weak Learner**: Train weak learners on these dataset.
- **Sequential Learning**: Boosting works by training models sequentially where each model focuses on correcting the errors of its predecessor. Boosting typically uses a single type of weak learner like decision trees.
- **Weight Adjustment**: Boosting assigns weights to training datapoints. Misclassified examples receive higher weights in the next iteration so that next models pay more attention to them.

Python pseudo code for boosting Estimator implementing libraries:

1. Importing Libraries and Modules

- **AdaBoostClassifier from sklearn.ensemble:** for building the AdaBoost ensemble model.
- **DecisionTreeClassifier from sklearn.tree:** as the base weak learner for AdaBoost.
- **load_iris from sklearn.datasets:** to load the Iris dataset.
- **train_test_split from sklearn.model_selection:** to split the dataset into training and testing sets.
- **accuracy_score from sklearn.metrics:** to evaluate the model's accuracy.

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

2. Loading and Splitting the Dataset

- **data = load_iris(): loads the Iris dataset, which includes features and target labels.**
- **X = data.data: extracts the feature matrix (input variables).**
- **y = data.target: extracts the target vector (class labels).**
- **train_test_split(...): splits the data into training (80%) and testing (20%) sets, with random_state=42 to ensure reproducibility.**

```
data = load_iris()
X = data.data
y = data.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

3. Defining the Weak Learner

We are creating the base classifier as a decision tree with maximum depth 1 (a decision stump). This simple tree will act as a weak learner for the **AdaBoost algorithm**, which iteratively improves by combining many such weak learners.

```
base_classifier = DecisionTreeClassifier(max_depth=1)
```

4. Creating and Training the AdaBoost Classifier

- **base_classifier: The weak learner used in boosting.**
- **n_estimators = 50: Number of weak learners to train sequentially.**
- **learning_rate = 1.0: Controls the contribution of each weak learner to the final model.**
- **random_state = 42: Ensures reproducibility.**

```
adaboost_classifier = AdaBoostClassifier(
   base_classifier, n_estimators=50, learning_rate=1.0, random_state=42
)
adaboost_classifier.fit(X_train, y_train)
```

5. Making Predictions and Calculating Accuracy

We are calculating the accuracy of the model by comparing the true labels **y_test** with the predicted labels **y_pred**. The accuracy_score function returns the proportion of correctly predicted samples. Then, we print the accuracy value.

```
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

Output:

Accuracy: 1.0

Benefits of Ensemble Learning in Machine Learning

Ensemble learning is a versatile approach that can be applied to machine learning model for: -

- **Reduction in Overfitting**: By aggregating predictions of multiple model's ensembles can reduce overfitting that individual complex models might exhibit.
- **Improved Generalization**: It generalizes better to unseen data by minimizing variance and bias.
- **Increased Accuracy**: Combining multiple models gives higher predictive accuracy.
- **Robustness to Noise**: It mitigates the effect of noisy or incorrect data points by averaging out predictions from diverse models.
- **Flexibility**: It can work with diverse models including decision trees, neural networks and support vector machines making them highly adaptable.
- **Bias-Variance Tradeoff**: Techniques like bagging reduce variance, while boosting reduces bias leading to better overall performance.

There are various ensemble learning techniques we can use as each one of them has their own pros and cons.

Ensemble Learning Techniques

Technique	Category	Description
Random Forest	Bagging	Random forest constructs multiple decision trees on bootstrapped subsets of the data and aggregates their predictions for final output, reducing overfitting and variance.

Department of Artificial Intelligence and Machine Learning
Subject: Advanced Machine Learning 23A03351T
Unit-I

Technique	Category	Description
Random Subspace Method	Bagging	Trains models on random subsets of input features to enhance diversity and improve generalization while reducing overfitting.
Gradient Boosting Machines (GBM)	Boosting	Gradient Boosting Machines sequentially builds decision trees, with each tree correcting errors of the previous ones, enhancing predictive accuracy iteratively.
Extreme Gradient Boosting (XGBoost)	Boosting	XGBoost do optimizations like tree pruning, regularization, and parallel processing for robust and efficient predictive models.
AdaBoost (Adaptive Boosting)	Boosting	AdaBoost focuses on challenging examples by assigning weights to data points. Combines weak classifiers with weighted voting for final predictions.
CatBoost	Boosting	CatBoost specialize in handling categorical features natively without extensive preprocessing with high predictive accuracy and automatic overfitting handling.

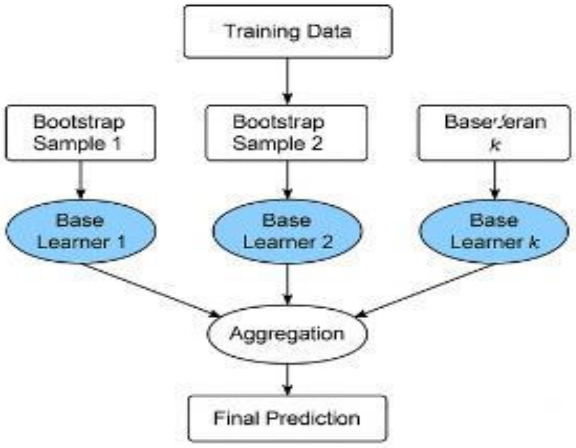
Bagging

Bagging (Bootstrap Aggregating) is an **ensemble learning** technique in machine learning that improves the **accuracy and stability** of models by reducing **variance** and avoiding **overfitting**, especially in high-variance models like decision trees.

Definition:
Bagging stands for **Bootstrap Aggregating**. It involves:

- **Generating multiple versions** of a training dataset using **bootstrap sampling** (random sampling with replacement).
- **Training separate models** (often the same type, like decision trees) on each of these datasets.
- **Aggregating their predictions** (averaging for regression, majority vote for classification).

Workflow of Bagging Algorithm (Step-by-Step):



1. **Bootstrap Sampling**: Create multiple datasets (say, k datasets) from the original training data using sampling with replacement.
2. **Model Training**: Train a base learner (e.g., decision tree) on each dataset independently.
3. **Aggregation**:
   - **Classification**: Use **majority voting** to decide the final output.
   - **Regression**: Use **averaging** of all predictions to give the final output.

Uses of Bagging:

- Reduces **overfitting** by averaging out predictions.
- Decreases **model variance** (good for unstable models).
- Improves **generalization**.

Common Algorithms That Use Bagging:

- **Random Forest** is a prime example: it's a bagging method using decision trees with added randomness in feature selection.

Advantages of Bagging:

- Reduces variance, thus improving model stability.
- Works well with high-variance, low-bias models.
- Easy to implement and parallelize.

Limitations:

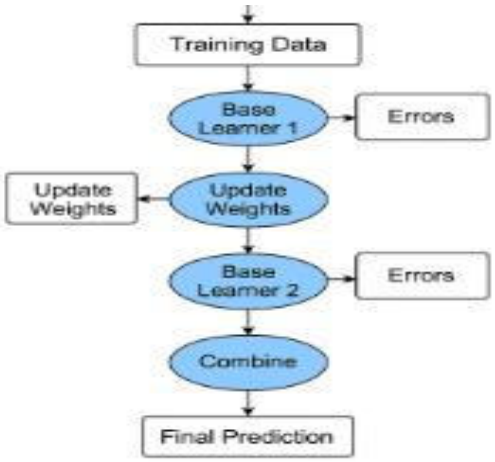
- Doesn't help much if the base model is already low in variance (like linear regression).
- May not reduce bias.
- Can be computationally expensive.

Boosting

Boosting is an **ensemble learning** method that combines multiple **weak learners** to form a **strong learner**. It builds models **sequentially**, where each model learns from the **errors of the previous ones**, improving overall performance.

Definition:
Boosting refers to a family of algorithms that **convert weak models (like shallow decision trees)** into a strong model by focusing more on **misclassified** data points during each iteration.

Working Steps of Boosting:



1. **Initialize** the model by training a weak learner on the original dataset.
2. **Compute Errors**: Measure the performance of the model.
3. **Update Weights**: Increase weights of **incorrectly predicted** samples.
4. **Train Next Learner**: The next model focuses more on the **harder examples**.
5. **Combine Models**: Final prediction is a **weighted sum** of all weak learners.

Key Concepts:

- **Sequential training**
- Focus on **difficult** samples
- Reduces both **bias** and **variance**
- Final prediction is based on the **weighted majority vote** (classification) or **weighted average** (regression)

Popular Boosting Algorithms:

Algorithm	**Key Feature**
AdaBoost	Adjusts weights of samples
Gradient Boosting	Optimizes loss function via gradients
XGBoost	Optimized, fast version of gradient boosting
LightGBM	Faster training, uses histogram-based techniques

Algorithm	Key Feature
CatBoost	Handles categorical features efficiently

Advantages of Boosting:

- High accuracy
- Handles both bias and variance
- Performs well on imbalanced data

Limitations:

- Prone to **overfitting** if not regularized
- **Sequential** → difficult to parallelize
- Slower than bagging

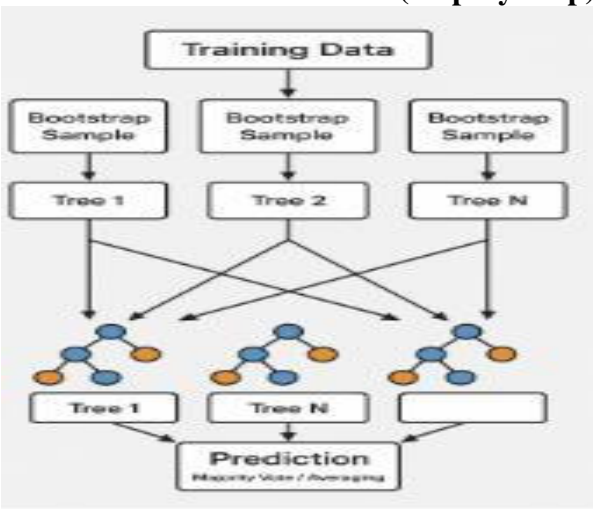
Random Forest Algorithm

- Random Forest is a **supervised ensemble learning algorithm**.
- It is used for both **classification and regression** tasks.
- It builds multiple **decision trees** and merges them together to get a more accurate and stable prediction.

A **Random Forest** is a collection (ensemble) of **Decision Trees** where:

- Each tree is trained on a different subset of the data using **bootstrap sampling** (bagging).
- At each node, only a random subset of features is considered for splitting.
- Final output is based on **majority voting** (classification) or **averaging** (regression).

Workflow of Random Forest (Step-by-Step)



Step 1: Bootstrap Sampling

- Create **N different subsets** (with replacement) from the training data.
- Each subset is used to train one decision tree.

Step 2: Build Decision Trees

- For each tree:
  - Choose a random subset of features at each split (feature bagging).
  - Grow trees **fully** without pruning.

Step 3: Aggregate Results

- For **Classification**: Each tree votes → final class = **majority vote**.
- For **Regression**: Average the outputs from all trees.

Key Terms

Term	Description
Bootstrap Sampling	Sampling with replacement from the dataset
Feature Bagging	Randomly selecting a subset of features at each split
Ensemble Learning	Combining multiple models for better performance
Majority Voting	Used in classification
Averaging	Used in regression

Advantages

- **Reduces overfitting** compared to individual decision trees.
- Works well with both **categorical and numerical** features.
- Can handle **missing values** and maintain accuracy.
- **Robust to outliers and noise**.
- Can give **feature importance scores**.

Disadvantages

- **Computationally intensive** (training many trees).
- Less interpretable than a single decision tree.
- Slower in real-time predictions (due to ensemble size).

Applications of Ramdom Forest:

- Medical diagnosis (e.g., cancer prediction)
- Financial risk analysis
- Credit scoring
- Image classification
- Fraud detection

Python Code Example

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

# Load data
X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

# Build model
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)

# Accuracy
from sklearn.metrics import accuracy_score
print("Accuracy:", accuracy_score(y_test, y_pred))
```

Parameters of Random Forest (Sklearn)

Parameter	Description
n_estimators	Number of trees
max_features	Number of features to consider at each split
max_depth	Maximum depth of the tree
min_samples_split	Minimum samples required to split an internal node

Parameter	Description
bootstrap	Whether bootstrap samples are used

Comparison with Other Algorithms

Feature	Decision Tree	Bagging	Random Forest	Boosting
Overfitting Risk	High	Low	Low	Medium
Interpretability	High	Low	Medium	Low
Accuracy	Medium	High	High	Very High
Training Speed	Fast	Moderate	Slow	Slow

AdaBoost Algorithm

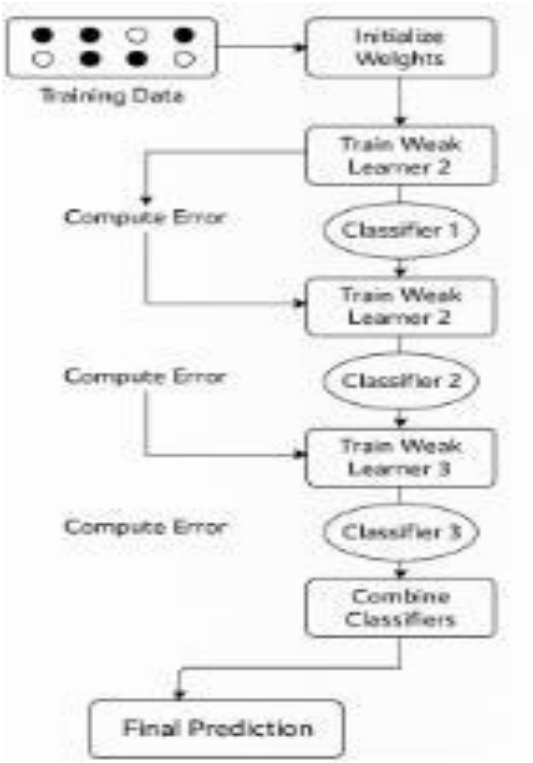
AdaBoost (Adaptive Boosting) is a **Boosting ensemble technique** that combines **multiple weak learners** (usually decision stumps — trees with one split) to form a **strong classifier**.

- It focuses on **instances that were previously misclassified**.
- Learners are added **sequentially**, and each one tries to **correct the mistakes** of the previous ones.

Key Idea:

Increase the **weights** of incorrectly classified data points so that subsequent models focus more on those **"hard" cases**.

Workflow of AdaBoost:



Step-by-Step:

1. **Initialize Weights**:
   - Assign equal weights to all training samples.
2. **Train a Weak Learner**:
   - Train a classifier (e.g., a decision stump) on the weighted data.
3. **Calculate Error**:
   - Compute the weighted error of the learner:

$$\text{Error} = \sum w_i \cdot I(y_i \neq h(x_i))$$

   - where I is an indicator function.
4. **Compute Learner's Weight**:
   - A classifier with lower error gets **higher importance**:

$$\alpha = \frac{1}{2} \log \left(\frac{1 - \text{Error}}{\text{Error}} \right)$$

5. **Update Weights** of Samples:
   - Increase weights of **misclassified** samples.
   - Decrease weights of correctly classified samples:

$$w_i = w_i \cdot e^{\pm \alpha}$$

   - Normalize weights.
6. **Repeat**:
   - Train next learner on updated weights.
   - Repeat steps for **T rounds** (number of estimators).
7. **Final Prediction**:
   - Combine all classifiers using their weights:

$$H(x) = \text{sign} \left(\sum_{t=1}^{T} \alpha_t h_t(x) \right)$$

Key Notations:

- x_i: Input features of sample i
- y_i: Label of sample i, typically $y_i \in \{-1, +1\}$
- $h_t(x_i)$: Prediction of the weak learner t on input x_i
- w_i: Weight of sample i
- α_t: Weight assigned to weak classifier h_t
- T: Total number of weak learners

AdaBoost Code Example (Python)

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

# Load data
X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

# Base weak learner: Decision stump
base = DecisionTreeClassifier(max_depth=1)

# AdaBoost model
model = AdaBoostClassifier(base_estimator=base, n_estimators=50, learning_rate=1.0)
model.fit(X_train, y_train)
```

```
# Accuracy
print("Accuracy:", model.score(X_test, y_test))
```

Advantages of AdaBoost

Feature	**Benefit**
Improves weak learners	Combines simple models to perform well
Versatile	Works for binary and multi-class classification
Feature importance	Can give feature significance
No need for data pre-processing	Robust to outliers and noise

Disadvantages

- Sensitive to **noisy data** and **outliers**
- Not suitable for large datasets with **many irrelevant features**
- Harder to interpret compared to individual trees

Applications

- Face detection (e.g., Viola-Jones algorithm)
- Fraud detection
- Text classification
- Bioinformatics

Comparison: AdaBoost vs Bagging vs Random Forest

Feature	**AdaBoost**	**Bagging**	**Random Forest**
Base Learners	Sequential	Parallel	Parallel
Focus	Hard samples	Variance reduction	Random features & samples
Output	Weighted vote	Majority vote	Majority vote

Gradient Boosting Algorithm

Gradient Boosting is an **ensemble learning** technique that builds a **strong predictive model** by combining multiple **weak learners** (typically decision trees), **trained sequentially** to correct the errors made by previous models.
It uses the idea of **minimizing a loss function** by applying **gradient descent**.
Key Idea:
Each new learner is trained to **predict the residuals (errors)** of the previous learners, thereby improving the model step by step.
Workflow of Gradient Boosting (Step-by-Step):

Step 1: Initialize the Model

- Use a constant value that minimizes the loss function.
- For regression with MSE:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} (y_i - \gamma)^2 = \bar{y}$$

Step 2: Iterate for T steps (number of trees)

For $t = 1$ to T:

1. Compute Residuals (Negative Gradient):

$$r_i^{(t)} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F=F_{t-1}(x)}$$

These are the **pseudo-residuals**.

2. **Train a Weak Learner** (e.g., decision tree) to predict the residuals:

$$h_t(x) \approx r_i^{(t)}$$

3. **Compute Step Size** (learning rate multiplied by a fitting coefficient):

$$\gamma_t = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, F_{t-1}(x_i) + \gamma h_t(x_i))$$

4. **Update the Model**:

$$F_t(x) = F_{t-1}(x) + \eta \gamma_t h_t(x)$$

where η is the **learning rate** (controls how much we trust each learner)

Key Terms

Term	Description
Weak Learner	Typically a decision tree (shallow)
Loss Function	Measures error (MSE, Log Loss, etc.)
Learning Rate η\etaη	Shrinks the contribution of each tree
Residuals	Errors the model tries to fix
Additive Model	Combines learners in a stage-wise manner

Loss Functions

- **Regression**:
  - MSE: $L(y, F(x)) = (y - F(x))^2$
- **Classification**:
  - Log Loss: $L(y, F(x)) = \log(1 + e^{-2yF(x)})$

Gradient Boosting Code in Python

```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

# Load data
X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

# Model
gb_model = GradientBoostingClassifier(n_estimators=100, learning_rate=0.1, max_depth=3)
gb_model.fit(X_train, y_train)

# Accuracy
print("Accuracy:", gb_model.score(X_test, y_test))
```

Advantages of Gradient Boosting

- High prediction accuracy
- Handles both regression and classification

- Works with many types of loss functions
- Feature importance ranking

Disadvantages

- Can overfit if not tuned properly
- Training is **slower** due to sequential nature
- Requires careful parameter tuning (learning rate, depth, etc.)

Comparison: AdaBoost and Gradient Boosting

Feature	**AdaBoost**	**Gradient Boosting**
Loss Optimization	Based on exponential loss	Any differentiable loss
Weighting	Adjusts sample weights	Fits to residuals
Robustness to Outliers	Lower	Higher
Tuning Needed	Less	More (learning rate, depth)

XGBoost Algorithm

XGBoost (Extreme Gradient Boosting) is an advanced implementation of the **Gradient Boosting algorithm**. It is designed to be **highly efficient**, **flexible**, and **portable**, with state-of-the-art performance.

XGBoost = Gradient Boosting + Regularization + Speed + Flexibility

It is **robust**, **scalable**, and **tunable**, and often outperforms other models in structured/tabular data tasks.

The uses of XGBoost:

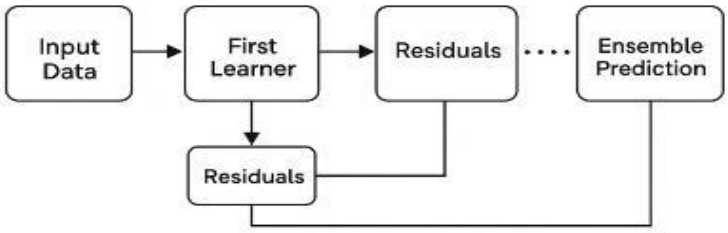
- **Fast** and **parallelizable**
- Handles **missing values**
- Includes **regularization** (to prevent overfitting)
- Excellent performance in **Kaggle competitions**
- Scales well to **large datasets**

Core Idea

Like Gradient Boosting, XGBoost builds trees **sequentially**, where each new tree corrects the **errors of the previous ensemble** by minimizing a loss function using **gradient descent**. XGBoost enhances this process with:

- **Second-order optimization** (using both gradient and hessian)
- **Regularization**
- **Tree pruning**
- **Cache-aware computing**

Workflow of XGBoost (Step-by-Step)



Step 1: Objective Function

XGBoost minimizes a regularized objective function:

$$\mathcal{L}(t) = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t)}) + \sum_{k=1}^{t} \Omega(f_k)$$

- l: Loss function (e.g., MSE, log loss)
- f_k: Each tree in the ensemble
- $\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_j w_j^2$
  T: number of leaves, w_j: leaf weights
  γ, λ: regularization parameters

Step 2: Second-Order Taylor Approximation

The loss is approximated with gradients and hessians:

$$\mathcal{L}^{(t)} \approx \sum_{i=1}^{n} \left[g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right] + \Omega(f_t)$$

- $g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)})$ → Gradient
- $h_i = \partial^2_{\hat{y}^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)})$ → Hessian

Step 3: Structure Score for Splits

For a split node with instances III:

$$\text{Score} = \frac{1}{2} \left[\frac{(\sum_{i \in I} g_i)^2}{\sum_{i \in I} h_i + \lambda} \right] - \gamma$$

Choose the split with the **highest score**.

Step 4: Tree Building

- Add trees **greedily** to minimize loss.
- Trees are built **depth-wise** or **loss-wise**, not leaf-wise like LightGBM.
- Stop growing when score improvement < threshold.

Step 5: Prediction Update

Update prediction:

$$\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + \eta f_t(x_i)$$

- η: Learning rate

Advantages of XGBoost

Advantage	**Description**
Speed	Parallel and fast due to efficient CPU use
Accuracy	Often better than other ML models
Regularization	Controls overfitting via λ,γ\lambda, \gammaλ,γ
Handles Missing Values	Smart split-finding for missing data
Built-in Cross-Validation	Available in API

Disadvantages

- **Complex to tune** (many hyperparameters)
- Can **overfit** on small data if not regularized
- Not ideal for **image or sequential data** (use CNNs or RNNs instead)

XGBoost Code Example (Python)

```
import xgboost as xgb
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

```
# Load data
X, y = load_breast_cancer(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train model
model = xgb.XGBClassifier(n_estimators=100, learning_rate=0.1, max_depth=3)
model.fit(X_train, y_train)

# Predict and evaluate
y_pred = model.predict(X_test)
print("Accuracy:", accuracy_score(y_test, y_pred))
```

Common Parameters

Parameter	**Meaning**
n_estimators	Number of boosting rounds
max_depth	Maximum tree depth
learning_rate	Shrinks contribution of each tree
subsample	Fraction of training data per tree
colsample_bytree	Feature sampling per tree
lambda	L2 regularization
gamma	Minimum loss reduction to make a split

Department of Artificial Intelligence and Machine Learning
Subject: Advanced Machine Learning 23A03351T
Unit-I

Stacking

Stacking (Stacked Generalization) is an ensemble learning technique that **combines multiple different models** (called *base learners*) and trains a **meta-model** to make the final prediction.

Unlike bagging or boosting (which use the same type of learners), **stacking uses diverse models** (e.g., decision trees, SVMs, neural networks).

Workflow of Stacking:

Step-by-Step Process:

1. **Train Base Learners**
    - Train several different machine learning models on the training dataset.
    - These models can be of **different types** (e.g., logistic regression, random forest, SVM).
2. **Generate Base Predictions**
    - Each base learner makes predictions on:
        - Either the validation set (during cross-validation),
        - Or directly on the test set.
3. **Train Meta-Learner**
    - A new model (called a **meta-model** or **blender**) is trained using the **predictions of base models** as features.
    - Its goal is to learn how to best combine the outputs of base models.
4. **Final Prediction**
    - The meta-model takes the predictions from base learners and makes the final decision.

Illustration (Simple Example)
Assume you have 3 base learners:

- Model 1: Logistic Regression
- Model 2: Decision Tree
- Model 3: K-Nearest Neighbors

Let the predictions from these models for a data point be:
Model 1: 0.6
Model 2: 0.8
Model 3: 0.7
These become the **features** for the **meta-model**, which might output a final prediction of 0.75.

Use of Stacking:

- Combines **strengths of multiple models**
- Can **reduce generalization error**
- Works well when base models are **diverse** and not highly correlated

Mathematically:

Suppose you have input features X and target variable y. The process is:

- Train base learners $h_1(x), h_2(x), \ldots, h_n(x)$
- Collect predictions $z_i = h_i(x)$
- Train meta-learner $H(z_1, z_2, \ldots, z_n)$ on these predictions

Final prediction:

$$\hat{y} = H(h_1(x), h_2(x), ..., h_n(x))$$

Example in Python (with scikit-learn)

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.ensemble import StackingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Load data
X, y = load_breast_cancer(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

# Define base learners
base_learners = [
    ('dt', DecisionTreeClassifier()),
    ('svc', SVC(probability=True))
]

# Define meta-learner
meta_model = LogisticRegression()

# Build stacking model
stacked_model = StackingClassifier(estimators=base_learners, final_estimator=meta_model)
stacked_model.fit(X_train, y_train)

# Predict and evaluate
y_pred = stacked_model.predict(X_test)
print("Accuracy:", accuracy_score(y_test, y_pred))
```

Advantages of Stacking

Benefit	Description
Combines model strengths	Leverages diversity to improve performance
Reduces generalization error	Less likely to overfit than a single model
Flexible	Works with any combination of models

Disadvantages

Limitation	Description
More complex	Requires training multiple models
Risk of overfitting	If meta-model is too complex or base models are similar
Slower to train	Compared to single-model methods

Department of Artificial Intelligence and Machine Learning
Subject: Advanced Machine Learning 23A03351T
Unit-I

Blending in Machine Learning

Blending is an ensemble technique used to combine the predictions of multiple machine learning models using a **validation dataset** and a **meta-model** (usually a simple one like logistic regression or linear regression).

It's very similar to **stacking**, but with a few key differences in how data is split and how the meta-model is trained.

How Does Blending Work?

Steps:

1. **Split the dataset into 3 parts**:
   - **Training set**: For training base models
   - **Validation set**: For generating predictions from base models
   - **Test set**: For final evaluation
2. **Train Base Models**:
   - Use the **training set** to train multiple models (e.g., SVM, Random Forest, XGBoost)
3. **Predict on Validation Set**:
   - Use base models to make predictions on the **validation set**
   - These predictions become **input features** for the meta-model
4. **Train Meta-Model**:
   - Train a **simple model** (e.g., logistic regression) using:
     - Inputs: Predictions of base models on the validation set
     - Targets: True values from the validation set
5. **Final Prediction**:
   - Use base models to predict on the **test set**
   - Meta-model uses these to make final predictions

How It Differs from Stacking

Feature	Blending	Stacking
Data Split	Train/Validation/Test split	Usually uses cross-validation
Meta-model trained on	Validation set predictions	Out-of-fold predictions from cross-validation
Simplicity	Easier to implement	More robust but complex
Risk of Overfitting	Higher (due to smaller validation set)	Lower (thanks to cross-validation)

Why Use Blending?

- **Simpler implementation**
- Useful when you're in a time crunch (e.g., in competitions)
- Easy to apply when you want to **combine different models quickly**

Blending Illustration Example

Imagine this:

- You train 3 models on your training data:
  - Logistic Regression → outputs 0.6
  - Random Forest → outputs 0.7
  - XGBoost → outputs 0.8

These outputs are used as **features** for a meta-model (e.g., linear regression), which may combine them like:

$$\text{Final prediction} = 0.2 \cdot LR + 0.3 \cdot RF + 0.5 \cdot XGB$$

Giving a final prediction = 0.2×0.6 + 0.3×0.7 + 0.5×0.8 = **0.73**

Advantages of Blending

Benefit	**Description**
Simple to implement	No need for complex cross-validation setups
Fast to train	Meta-model trained on small dataset
Good for competitions	Useful in last-minute model improvement

Disadvantages

Drawback	**Description**
High risk of overfitting	Meta-model trained on small validation set
Not as robust	Compared to stacking with cross-validation
Wastes data	Validation data not used in base model training

Small Python Example (Pseudo-code Style)

```
# Step 1: Split data
X_train, X_valid, y_train, y_valid = train_test_split(X, y, test_size=0.2)

# Step 2: Train base models
model1 = LogisticRegression().fit(X_train, y_train)
model2 = RandomForestClassifier().fit(X_train, y_train)

# Step 3: Predict on validation set
pred1 = model1.predict_proba(X_valid)[:, 1]
pred2 = model2.predict_proba(X_valid)[:, 1]

# Step 4: Stack predictions and train meta-model
meta_X = np.column_stack((pred1, pred2))
meta_model = LogisticRegression().fit(meta_X, y_valid)

# Step 5: Predict on test set
```

```
final_pred = meta_model.predict(np.column_stack((
   model1.predict_proba(X_test)[:, 1],
   model2.predict_proba(X_test)[:, 1]
)))
```

Mathematical Formulation of Blending

Let's define:

- X_{train}, y_{train}: Training data
- X_{valid}, y_{valid}: Validation data
- X_{test}: Test data
- $M_1, M_2, \ldots, M_n$: Base learners/models
- M_{meta}: Meta-model

Step 1: Train Base Models

Each base model M_i is trained on $(X_{\text{train}}, y_{\text{train}})$.

$$\hat{y}_i^{\text{valid}} = M_i(X_{\text{valid}}) \quad \text{for } i = 1, 2, ..., n$$

These predictions are collected into a **meta-feature matrix**:

$$Z_{\text{valid}} = \begin{bmatrix} \hat{y}_1^{\text{valid}} & \hat{y}_2^{\text{valid}} & \cdots & \hat{y}_n^{\text{valid}} \end{bmatrix}$$

Step 2: Train Meta-Model

The meta-model M_{meta} is trained on Z_{valid} and y_{valid}:

$$M_{\text{meta}} : Z_{\text{valid}} \rightarrow y_{\text{valid}}$$

Step 3: Final Prediction on Test Set

Each base model makes predictions on X_{test}:

$$\hat{y}_i^{\text{test}} = M_i(X_{\text{test}})$$

$$Z_{\text{test}} = \begin{bmatrix} \hat{y}_1^{\text{test}} & \hat{y}_2^{\text{test}} & \cdots & \hat{y}_n^{\text{test}} \end{bmatrix}$$

The meta-model then makes the final prediction:

$$\hat{y}^{\text{final}} = M_{\text{meta}}(Z_{\text{test}})$$

Special Case: Linear Meta-Model

If the meta-model is **linear**, then:

$$\hat{y}^{\text{final}} = \sum_{i=1}^{n} w_i \cdot \hat{y}_i^{\text{test}} + b$$

Where:

- w_i are the learned weights from the meta-model (e.g., linear regression)
- b is the intercept term

Example with 3 Models

Let's say:

- $\hat{y}_1 = 0.65$ from Model 1
- $\hat{y}_2 = 0.75$ from Model 2
- $\hat{y}_3 = 0.85$ from Model 3

Meta-model (linear regression) gives:

$$\hat{y}^{\text{final}} = 0.2 \cdot \hat{y}_1 + 0.3 \cdot \hat{y}_2 + 0.5 \cdot \hat{y}_3 = 0.2 \cdot 0.65 + 0.3 \cdot 0.75 + 0.5 \cdot 0.85 = 0.775$$

Department of Artificial Intelligence and Machine Learning
Subject: Advanced Machine Learning 23A03351T
Unit-I

Regularization Methods in Machine Learning

Regularization is a technique used to reduce **overfitting** by adding a penalty term to the loss function of a machine learning model. This discourages the model from becoming too complex or sensitive to noise in the training data.

Need to Use Regularization:

- Prevents **overfitting**
- Improves **generalization** to unseen data
- Controls the **complexity** of the model

Benefits of Regularization

Now, let's see various benefits of regularization which are as follows:

1. **Prevents Overfitting:** Regularization helps models focus on underlying patterns instead of memorizing noise in the training data.
2. **Improves Interpretability:** L1 (Lasso) regularization simplifies models by reducing less important feature coefficients to zero.
3. **Enhances Performance:** Prevents excessive weighting of outliers or irrelevant features helps in improving overall model accuracy.
4. **Stabilizes Models:** Reduces sensitivity to minor data changes which ensures consistency across different data subsets.
5. **Prevents Complexity:** Keeps model from becoming too complex which is important for limited or noisy data.
6. **Handles Multicollinearity:** Reduces the magnitudes of correlated coefficients helps in improving model stability.
7. **Allows Fine-Tuning:** Hyperparameters like alpha and lambda control regularization strength helps in balancing bias and variance.
8. **Promotes Consistency:** Ensures reliable performance across different datasets which reduces the risk of large performance shifts.

Common Regularization Methods

1. L1 Regularization (Lasso)

- Adds the **absolute value** of coefficients to the loss function.
- Encourages **sparsity** (sets some weights to **zero**), leading to feature selection.

◆ Loss Function:

$$L = \text{Loss}(y, \hat{y}) + \lambda \sum_{j=1}^{n} |w_j|$$

- λ: regularization parameter (higher = more penalty)
- w_j: model weights

2. L2 Regularization (Ridge)

- Adds the **square** of coefficients to the loss function.

- Keeps all features but shrinks weights.

◆ Loss Function:

$$L = \text{Loss}(y, \hat{y}) + \lambda \sum_{j=1}^{n} w_j^2$$

- Less aggressive than L1; doesn't zero out weights.

3. Elastic Net Regularization

- Combines both **L1** and **L2** penalties.
- Useful when there are **many correlated features**.

◆ Loss Function:

$$L = \text{Loss}(y, \hat{y}) + \lambda_1 \sum_{j=1}^{n} |w_j| + \lambda_2 \sum_{j=1}^{n} w_j^2$$

4. Dropout (in Neural Networks)

- Randomly sets a fraction of neurons to 0 during training.
- Reduces **co-adaptation** of neurons.

Intuition:

During each training iteration:

- Drop units with a probability p
- Forces the network to not rely too much on specific paths

5. Early Stopping

- Stop training when the model's performance on the **validation set** starts to degrade.
- Prevents overfitting without modifying the loss function.

6. Data Augmentation & Noise Injection

- Add noise to input data or intermediate layers to make the model more robust.

Cross-Validation Strategies

Cross-validation (CV) is a statistical method used to estimate the performance of machine learning models. It helps detect overfitting and ensures that the model generalizes well to unseen data.

Use of Cross-Validation:

- To assess **model stability** and **robustness**
- To detect **overfitting** or **underfitting**
- To choose the best **model hyperparameters**

Common Cross-Validation Strategies

1. Hold-Out Validation

- Split dataset into:
  - **Training set**: to train the model
  - **Test set**: to evaluate the model

Limitation:

- High variance depending on how data is split

2. K-Fold Cross-Validation

- Divide data into **K equal parts (folds)**
- Train the model on K−1 folds, validate on the remaining fold
- Repeat K times, each fold used once as validation
- Final performance = **mean of K results**

Example:

For K=5

Fold	Train On	Validate On
1	2,3,4,5	1
2	1,3,4,5	2
3	1,2,4,5	3
4	1,2,3,5	4
5	1,2,3,4	5

3. Stratified K-Fold Cross-Validation

- Like K-Fold but **preserves the percentage of samples for each class** in every fold.
- Useful for **imbalanced datasets**.

4. Leave-One-Out Cross-Validation (LOOCV)

- Special case of K-Fold with K=n (number of samples)
- Train on all data **except one sample**, test on that one
- Repeat for all samples

Limitation:

- Computationally **expensive** for large datasets

5. Repeated K-Fold Cross-Validation

- Repeats K-Fold CV **multiple times with different random splits**
- Reduces variance in performance estimation

6. Group K-Fold Cross-Validation

- Ensures that the **same group** (e.g., from the same patient or user) does **not appear** in both training and validation sets.
- Ideal for grouped or clustered data

7. Time Series Split (Rolling Forecast Origin)

- For time series data where order matters
- Avoids data leakage by ensuring that **future data is not used to predict the past**

Example:

Fold	Train On	Validate On
1	1, 2	3
2	1, 2, 3	4
3	1, 2, 3, 4	5

Advantages and Disadvantages of Cross-Validation Strategies

Strategy	Best For	Advantages	Disadvantages
Hold-Out	Quick checks	Simple and fast	High variance
K-Fold	General-purpose	Balanced, less bias	Can be slow for large K
Stratified K-Fold	Imbalanced classification	Maintains class distribution	More complex
LOOCV	Small datasets	Uses almost all data to train	Very slow for large datasets
Repeated K-Fold	Stability checking	Reduces random bias	Slower than standard K-Fold
Group K-Fold	Grouped data (e.g., patients)	Prevents data leakage	Requires group identifiers
Time Series Split	Time-based data	Respects time order	Needs careful setup

Unit-II

Linear and Non-linear Classification in Machine Learning

Classification in machine learning involves assigning input data to predefined categories. Based on the nature of the decision boundary, classification models are generally categorized into two types:

Linear Classification

Definition:

Linear classification uses a linear decision boundary (hyperplane) to separate different classes in the feature space.

Key Characteristics:

- Assumes that classes can be separated by a straight line (in 2D), a plane (in 3D), or a hyperplane (in higher dimensions).
- Fast and computationally efficient.
- Works well when data is linearly separable.

Common Algorithms:

- **Logistic Regression**
- **Linear Support Vector Machine (SVM)**
- **Perceptron**

Mathematical Representation:

A linear classifier makes predictions using:

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

Where:

- w = weight vector
- x = input feature vector
- b = bias
- Decision rule: If $f(x) > 0$, class 1; else class 0.

Advantages:

- Simpler and faster to train.
- Less prone to overfitting when the dataset is small or simple.

Limitations:

- Cannot handle complex patterns or non-linear data distributions.

Non-linear Classification

Definition:

Non-linear classification uses complex decision boundaries (curves, irregular shapes) to separate classes that are not linearly separable.

Key Characteristics:

- Suitable for data where linear boundaries are insufficient.
- Capable of modeling intricate relationships between features and output.

Common Algorithms:

- **Kernel SVM** (e.g., RBF kernel)
- **Decision Trees**
- **Random Forest**
- **K-Nearest Neighbors (KNN)**
- **Neural Networks**

Mathematical Approach (Example: Kernel Trick in SVM):

Transforms data into higher-dimensional space to find a linear separator in that space:

$$K(x, x') = \phi(x)^T \phi(x')$$

Where ϕ is a non-linear transformation function.

Advantages:

- Flexible and powerful for complex datasets.
- Can model data with curves, clusters, or multiple class regions.

Limitations:

- More computationally expensive.
- Higher risk of overfitting without proper regularization.

Comparison Table

Feature	**Linear Classification**	**Non-linear Classification**
Decision Boundary	Straight line / hyperplane	Curved or complex boundaries
Speed	Faster	Slower (depends on algorithm)
Interpretability	High	Often lower
Use Case	Linearly separable data	Complex, real-world data
Risk of Overfitting	Low	High (if not regularized)

Usages of linear and non-linear models

- Use **linear models** when data is simple and fast performance is required.
- Use **non-linear models** when the data exhibits complex patterns or clusters.

Kernel Trick

The **kernel trick** allows algorithms (like SVM) to operate in **high-dimensional feature spaces** without **explicitly transforming the data**.
It computes the inner product of two vectors **in a transformed feature space** using a **kernel function**:

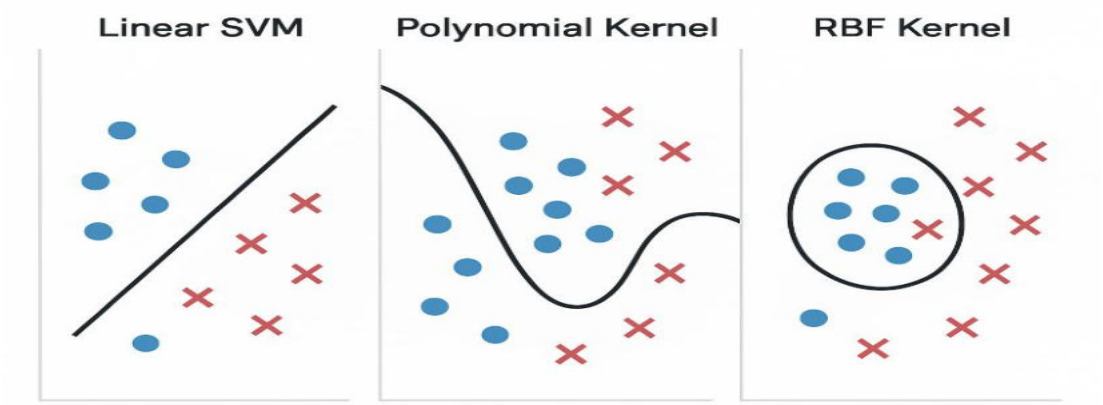
$$K(x, x') = \phi(x)^T \phi(x')$$

Where:

- x, x': input vectors
- ϕ: non-linear mapping function (not computed explicitly)
- $K(x, x')$: kernel function

Uses of Kernel Trick

- To **handle non-linear classification problems**.
- Avoids high computational cost of explicitly mapping data to a higher-dimensional space.



1. Polynomial Kernel
Formula:

$$K(x, x') = (x^T x' + c)^d$$

Where:

- x, x': input vectors
- $c \geq 0$: a constant controlling influence of higher-order terms
- d: degree of the polynomial

Interpretation:

- Allows SVM to learn polynomial decision boundaries.
- d=2: quadratic kernel
- d=3: cubic kernel, etc.

Use Case:

- When the relationship between features and labels is polynomial (e.g., circular or parabolic boundaries).

Example:

If $x = [x_1, x_2]$, and $d = 2$, then the kernel represents:

$$K(x, x') = (x_1 x_1' + x_2 x_2' + c)^2$$

This implicitly introduces terms like $x_1^2, x_2^2, x_1 x_2$, etc., for modeling.

2. RBF (Radial Basis Function) Kernel / Gaussian Kernel

Formula:

$$K(x, x') = \exp\left(-\frac{||x - x'||^2}{2\sigma^2}\right)$$

Or, more commonly with gamma (γ):

$$K(x, x') = \exp(-\gamma ||x - x'||^2)$$

Where:

- $\gamma = \frac{1}{2\sigma^2}$
- $||x - x'||^2$: squared Euclidean distance between vectors
- γ: controls the smoothness (spread)

Interpretation:

- Projects data into **infinite-dimensional space**.
- Measures **similarity**: closer vectors = higher similarity (near 1), far = near 0.

Use Case:

- Excellent for complex, non-linear problems.
- Default kernel in many SVM implementations.

Polynomial vs. RBF Kernel: Comparison

Feature	Polynomial Kernel	RBF Kernel
Decision Boundary	Curved, polynomial-shaped	Flexible, can model complex shapes
Control Parameter	Degree d, constant c	Gamma γ
Feature Space	Finite-dimensional (depends on d)	Infinite-dimensional
Speed	Slower for high degree	Often faster for large datasets
Overfitting Risk	High with large d	High with large γ
Usage	When relationship is polynomial	When complex non-linear boundaries exist

Custom Kernels

In machine learning, particularly **Support Vector Machines (SVMs)**, a **custom kernel** is a user-defined function that computes the similarity between two data points, enabling SVM to operate in an implicitly transformed feature space suited to specific data characteristics.

What is a Kernel? (Quick Recap)

A **kernel function** $K(x, x')$ computes:

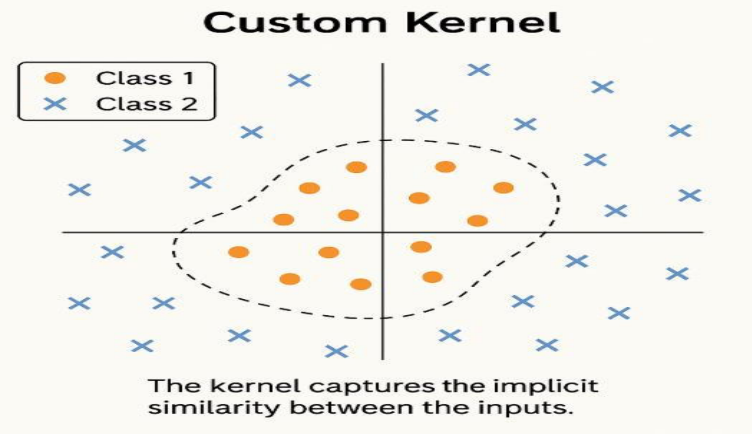
$$K(x, x') = \phi(x)^T \phi(x')$$

- $\phi(x)$: implicit mapping of input data into a higher-dimensional space
- Kernel functions help perform classification or regression on non-linear data without explicitly mapping it

Custom Kernel: Definition

A **custom kernel** is any user-defined function K(x,x′) that:

1. Measures similarity between inputs.
2. Must satisfy **Mercer's condition** (i.e., the kernel matrix must be positive semi-definite).



Steps to Define a Custom Kernel

1. **Understand your data's structure**: Determine what kind of similarity works best.
2. **Design a kernel function**: Create a function K(x,x′) that models similarity.
3. **Ensure positive semi-definiteness**: The kernel matrix K formed by $K(x_i, x_j)$ should be symmetric and positive semi-definite.
4. **Plug into SVM**: Use it via a machine learning library (like scikit-learn).

Examples of Custom Kernels

1. String Kernel (for text data)

Measures similarity based on the number of common substrings.

$$K(x, x') = \sum_{\text{substrings } s \in x, x'} \lambda^{|s|} \cdot \text{count}_x(s) \cdot \text{count}_{x'}(s)$$

Used in **bioinformatics** and **text classification**.

2. Histogram Intersection Kernel (for image histograms)

$$K(x, x') = \sum_{i=1}^{n} \min(x_i, x'_i)$$

Used in **image retrieval** and **object recognition**.

3. Custom Combination Kernel
You can combine kernels like:

$$K(x, x') = \alpha \cdot K_1(x, x') + \beta \cdot K_2(x, x')$$

Where:

- K_1, K_2: predefined kernels (e.g., linear + RBF)
- α, β: tuning parameters

This approach is called **Multiple Kernel Learning (MKL)**.

Python Example: Custom Kernel with scikit-learn

```
from sklearn.svm import SVC
from sklearn.metrics.pairwise import pairwise_kernels
import numpy as np

# Define custom kernel (e.g., sigmoid + RBF hybrid)
def custom_kernel(X, Y):
    return np.tanh(X @ Y.T + 1) + np.exp(-0.5 * np.linalg.norm(X[:, None] - Y, axis=2)**2)

# Example training
X = [[1, 2], [2, 3], [3, 4]]
y = [0, 1, 0]

# Train SVM with custom kernel
clf = SVC(kernel=custom_kernel)
clf.fit(X, y)
```

Benefits of Custom Kernels

- Tailored to **domain-specific** data
- Enables use of **non-vector** data (strings, graphs, trees)
- Often improves **performance** for complex or structured data

Challenges

- Must ensure **validity** (positive semi-definiteness)
- May require **domain expertise**
- Slower for large datasets unless optimized

Soft Margin SVMs (Support Vector Machines)

Soft Margin SVMs are an extension of hard margin SVMs that allow some misclassification in the training data. They are crucial when the data is **not linearly separable**.

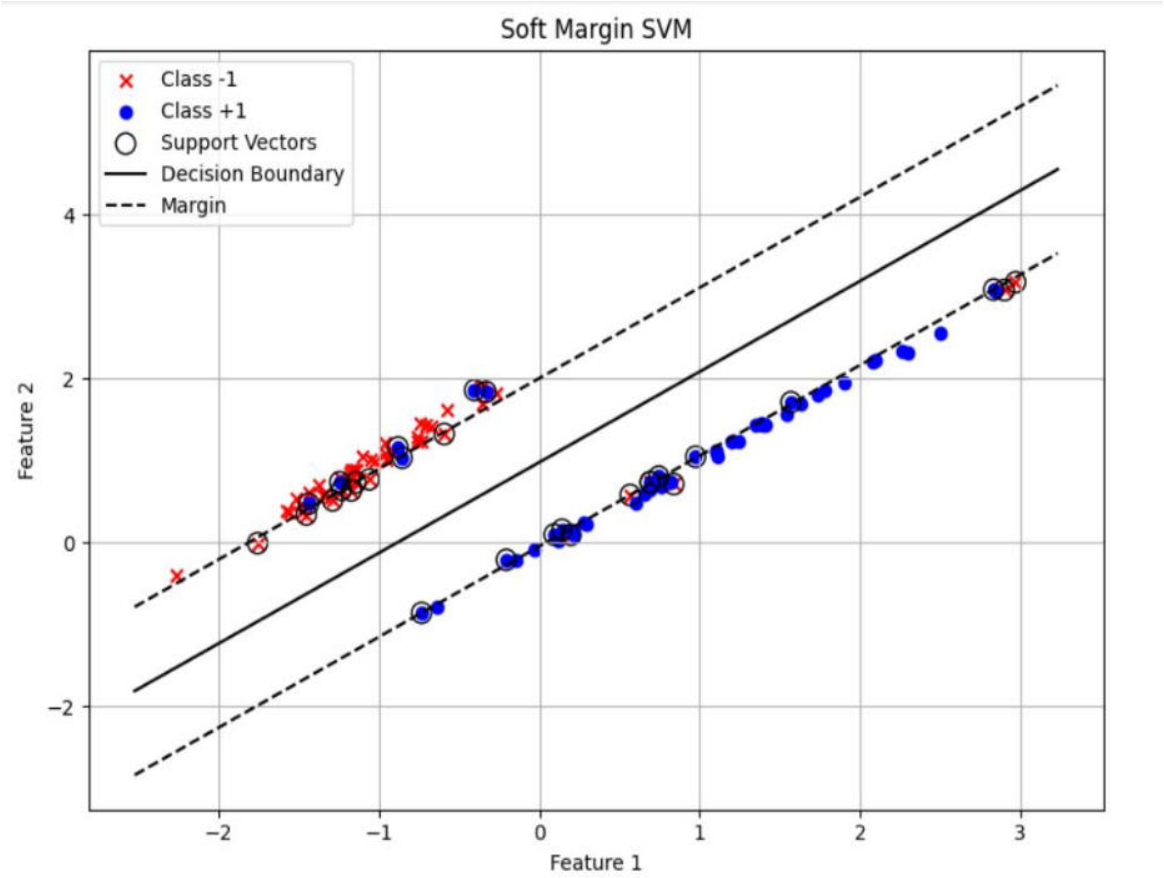
Why Soft Margin?
In real-world data:

- Classes may **overlap**
- Outliers may **exist**
- Perfect separation is often **impossible**

A **Hard Margin SVM** requires perfect separation → not practical.
Soft Margin SVM introduces **flexibility** by allowing some errors (slack).

The idea is to **balance two goals**:

1. **Maximize the margin**
2. **Minimize the classification error (misclassified or inside-margin points)**



This diagram shows

- Two classes of points (`+1` and `-1`)
- The **decision boundary** (solid black line)
- The **margin boundaries** (dashed lines)
- The **support vectors** (highlighted with black borders)

Mathematical Formulation
Objective Function:

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

Subject to:

$$y_i(w^T x_i + b) \geq 1 - \xi_i \quad \text{and} \quad \xi_i \geq 0$$

Where:

- w: weight vector
- b: bias
- ξ_i: slack variables (measure violations of the margin)
- C: penalty parameter (regularization)
- $y_i \in \{-1, +1\}$: class labels

Explanation of Terms

$\|w\|^2$	Margin width (smaller $\|w\|$ = wider margin)
ξ_i	Slack variable — allows a point to be inside margin or misclassified
C	Regularization parameter: trade-off between margin width and classification error

Interpretation of Parameters

- Slack variable ξ_i:
  - $\xi_i = 0$: point is correctly classified outside margin
  - $0 < \xi_i < 1$: point is inside margin but on correct side
  - $\xi_i > 1$: point is misclassified
- Penalty parameter C:
  - Large C: penalize misclassifications heavily (hard margin behavior)
  - Small C: allow more flexibility/misclassifications (more generalization)

Advantages

- Works well for **non-separable data**
- **Balances** accuracy and generalization
- Less sensitive to **noise/outliers**

Dual Form (for Kernel Trick)

The dual form (used with kernels) is:

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

Subject to:

$$0 \leq \alpha_i \leq C, \quad \sum_{i=1}^{n} \alpha_i y_i = 0$$

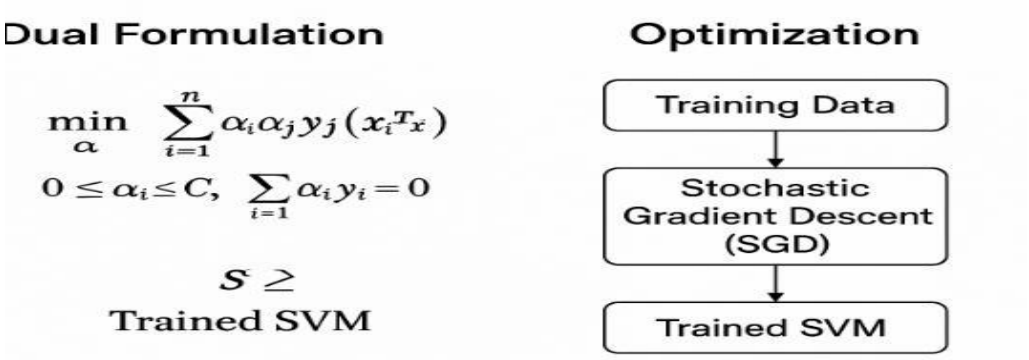
Where:

- α_i: Lagrange multipliers
- $K(x_i, x_j)$: Kernel function (e.g., linear, RBF, polynomial)

Soft Margin vs. Hard Margin

Feature	**Hard Margin SVM**	**Soft Margin SVM**
Assumes data	Perfectly separable	Overlapping/Noisy allowed
Slack variable ξ	Not used	Used
Regularization C	Not needed	Critical for performance

Dual Form and Optimization of Support Vector Machines (SVM)



1. Primal Form of SVM (Soft Margin)

We want to find the hyperplane that best separates the classes, allowing some margin of error.

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

Subject to:

$$y_i(w^T x_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0$$

2. Dual Formulation

Instead of directly solving the primal form, we derive the **dual form** using **Lagrange multipliers**. This lets us:

- Work with inner products $x_i^T x_j$
- Easily apply the **kernel trick**

◆ **Dual Problem:**

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

Subject to:

$$0 \leq \alpha_i \leq C, \quad \sum_{i=1}^{n} \alpha_i y_i = 0$$

Where:

- α_i: Lagrange multipliers
- C: regularization parameter
- y_i: class labels
- x_i: input vectors

3. Optimization Algorithm

After deriving the dual, we solve it using **Quadratic Programming (QP)**. For large datasets, we use iterative optimization algorithms:

Common Methods:

- **Sequential Minimal Optimization (SMO)**
- **Stochastic Gradient Descent (SGD)** (for linear SVMs)

- **LibSVM** (widely used library for kernel SVMs)

4. Reconstructing the Classifier
Once we solve for α\alphaα, the decision function becomes:

$$f(x) = \sum_{i=1}^{n} \alpha_i y_i K(x_i, x) + b$$

Only data points with $\alpha_i > 0$ are **support vectors.**

Advantages of Dual Form
- Allows use of **kernel functions** to handle non-linear data
- Efficient when the number of **features > number of samples**
- Helps identify **support vectors** clearly

Support Vector Regression (SVR): Formulation, Optimization & Practical Tips

Support Vector Regression extends the **maximum-margin principle** of SVM classification to regression tasks, using a **ε-insensitive loss** to tolerate small errors while maintaining model sparsity.

1. Understanding the ε-Insensitive Tube
SVR trains a function:

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

that stays **within an ε-tube** around true targets y_i . Deviations less than ε are ignored:

$$L_\varepsilon(y, f(x)) = \max\big(0,\ |y - f(x)| - \varepsilon\big)$$

This loss ensures only points that lie **outside the tube** affect the model (support vectors). Smaller ε → more SVs → more complexity; l
arger ε → fewer SVs → smoother fit

2. Primal Optimization (with Slack Variables)

To allow some predictions outside the tube, we add slack variables ξ_i^+, ξ_i^-:

$$\min_{w,b,\xi^+,\xi^-} \ \frac{1}{2}||w||^2 + C\sum_i (\xi_i^+ + \xi_i^-)$$

Subject to:

$$\begin{cases} y_i - w^T x_i - b \le \varepsilon + \xi_i^+ \\ w^T x_i + b - y_i \le \varepsilon + \xi_i^- \\ \xi_i^+, \xi_i^- \ge 0 \end{cases}$$

Here, C controls the trade-off between **flatness** ($\frac{1}{2}||w||^2$) and **penalties** for violating the ε-zone

3. Dual Formulation — Enabling Kernelization

The dual problem introduces **two sets of Lagrange multipliers** α_i and α_i^*, and transforms into:

$$\max_{\alpha,\alpha^*}\Big[-\tfrac{1}{2}\sum_{i,j}(\alpha_i - \alpha_i^*)(\alpha_j - \alpha_j^*)\langle x_i, x_j\rangle - \varepsilon\sum_i(\alpha_i + \alpha_i^*) + \sum_i y_i(\alpha_i - \alpha_i^*)\Big]$$

$$\text{s.t.} \quad \sum_i(\alpha_i - \alpha_i^*) = 0, \quad 0 \le \alpha_i, \alpha_i^* \le C$$

The regression function becomes:

$$f(x) = \sum_{i=1}^{n}(\alpha_i - \alpha_i^*)\, K(x_i, x) + b$$

Note only support vectors (with non-zero α differences) contribute to the prediction.

4. Kernel Trick & Non-Linear SVR

SVR allows non-linear regression by using kernel functions in place of the inner product:

- **Linear:** $K(x, x') = x^T x'$
- **Polynomial:** $(\gamma\, x^T x' + \text{coef0})^d$
- **RBF:** $\exp(-\gamma\, ||x - x'||^2)$

This enables mapping to high-dimensional (even infinite-dimensional) feature spaces without explicit transformation.

5. Practical Hyperparameters & Effects

Parameter	Function	Effect on model
C	Regularization strength	High C → fits training error tightly (risk overfitting); low C → smoother function
ε (epsilon)	Width of the insensitive tube	Large ε → few support vectors & high bias; small ε → more SVs & high variance
Kernel type (and γ, degree, coef0)	Determines geometry of decision surface	Critical for capturing non-linear trends
shrinking, tol, cache_size	Optimization details affecting runtime and convergence	

In **scikit-learn's** SVR implementation:

- Defaults: kernel='rbf', C=1.0, ε=0.1, gamma='scale'
- Complexity: more than quadratic in #samples → consider LinearSVR for very large datasets.
- Default behavior: ε-insensitive loss, standard RBF kernel setting, based on libsvm

6. Key Advantages & Limitations

Advantages

- **Sparse** representation: only support vectors matter (efficient inference).
- Built-in regularization → robust generalization.
- Works naturally with non-vector data using custom kernels (e.g., string, histogram-based).
- **Non-parametric**: the model adapts complexity to the data.

Limitations

- **Computationally expensive** for n ≫ 10,000, especially with non-linear kernels.
- Tree/trend coverage depends heavily on kernel and hyperparameters.
- Less interpretable than linear methods in regression settings.

7. Example: Training an SVR in Python

```
from sklearn.svm import SVR
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error

param_grid = {
    'C': [0.1, 1, 10],
    'epsilon': [0.01, 0.1, 0.5],
    'kernel': ['rbf', 'poly'],
    'gamma': ['scale', 'auto'],  # for rbf and poly
    'degree': [2, 3]             # only for poly
```

```
}

svr = SVR()
grid = GridSearchCV(svr, param_grid, cv=5, scoring='neg_mean_squared_error', verbose=2)
grid.fit(X_train, y_train)

best = grid.best_estimator_
y_pred = best.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
print(f"Best params: {grid.best_params_}, Test MSE: {mse:.3f}")
```

This grid search approach effectively tunes both **C** and **ε** parameters—crucial for getting the right bias-variance trade-off.

8. Why It Works: Sparsity & Support Vectors

Due to the ε-insensitive loss and capped multipliers ($0 \le \alpha_i, \alpha_i^* \le C$), **most training points have zero coefficients** unless they lie outside the ε-tube. Only a subset—called **support vectors**—affects the resulting model. This makes SVR flexible yet interpretable in terms of critical examples.

9. Common Extensions & Variants

- **Least-Squares SVR (LSSVR)**: uses squared ε-insensitive loss → fully dense solutions (all points become support vectors) → faster to solve but less sparse.
- **Adaptive ε-SVR ($A\varepsilon_i$-SVR)**: allows varying ε per sample for better handling of heteroskedasticity or irregular data distributions.
- **Twin Support Vector Regression (TSVR)**: separates data using two proximal hyperplanes → solved via smaller QPs → improved scalability in some cases.

TL;DR (Too Long; Didn't Read)

- SVR is the regression counterpart of SVM, optimizing margin while tolerating small errors via the ε-Tube.
- Has both **primal** and **dual** formulations; dual enables **kernelization** for complex non-linear relationships.
- Hyperparameters **C**, **ε**, and **kernel** critically affect bias–variance and generalization.
- Efficient for moderately sized datasets; less scalable for very large data unless using approximate methods like LinearSVR or kernel approximations.

Kernel PCA (Principal Component Analysis) for Non-linear Dimensionality Reduction

What is Kernel PCA?

Kernel PCA is an extension of classical **Principal Component Analysis (PCA)** that allows **non-linear** dimensionality reduction using the **kernel trick**.

- It maps input data to a **high-dimensional feature space**, where **linear PCA** is performed.
- In this feature space, **non-linear structures** in the original data can be captured effectively.

Usage of Kernel PCA

- **Classical PCA** assumes **linear correlations** between variables.
- **Kernel PCA** works well when:
  - Data lies on a **non-linear manifold**
  - There are **non-linear patterns** (e.g., spirals, concentric circles)

Core Idea

1. Map data $x \in \mathbb{R}^n$ to feature space via non-linear function $\phi(x)$
2. Compute dot products $\phi(x_i)^T \phi(x_j)$ using a **kernel function** $K(x_i, x_j)$
3. Perform PCA in this **feature space**

Kernel PCA Algorithm Steps

Step 1: Choose a Kernel Function

Examples:

- **RBF** (Gaussian):

$$K(x, x') = \exp\left(-\frac{\|x - x'\|^2}{2\sigma^2}\right)$$

- **Polynomial**:

$$K(x, x') = (x^T x' + c)^d$$

Step 2: Compute the Kernel (Gram) Matrix

$$K_{ij} = K(x_i, x_j)$$

Step 3: Center the Kernel Matrix

Center the matrix to ensure zero-mean data in feature space:

$$K_c = K - \mathbf{1}K - K\mathbf{1} + \mathbf{1}K\mathbf{1}$$

Where:

- 1: n×n matrix with all values = 1/n
- This operation centers the data in the feature space **without computing ϕ(x)**
- Subtract the row mean: K−1K
- Subtract the column mean: −K1
- Add the total mean: +1K1

➢ **Step 4: Eigen Decomposition**

Solve the eigenvalue problem:

$$K_c \alpha = \lambda \alpha$$

Where:

- α: eigenvector (principal component in kernel space)
- λ: eigenvalue (variance explained by that component)

We find the **top k** eigenvectors corresponding to the **largest eigenvalues**.

Step 5: Project Data

Project data onto the first k principal components:

$$x_i' = \sum_{j=1}^{n} \alpha_j K(x_i, x_j)$$

Applications of Kernel PCA

- **Data visualization** (in 2D or 3D)
- **Preprocessing** for classification (e.g., SVM)
- **Non-linear feature extraction**
- **Image and signal denoising**

Comparison: PCA vs. Kernel PCA

Feature	**PCA**	**Kernel PCA**
Type of method	Linear	Non-linear (via kernel trick)
Basis computation	Covariance matrix	Kernel (Gram) matrix
Captures curved manifolds	Not Allowed	Allowed
Output interpretability	Easy	Harder

Example (Python - scikit-learn)

```
from sklearn.decomposition import KernelPCA
import matplotlib.pyplot as plt

# Sample: circular data
from sklearn.datasets import make_circles
X, y = make_circles(n_samples=400, factor=0.3, noise=0.05)

# Apply Kernel PCA
kpca = KernelPCA(n_components=2, kernel='rbf', gamma=15)
X_kpca = kpca.fit_transform(X)

# Plot result
plt.scatter(X_kpca[:, 0], X_kpca[:, 1], c=y)
plt.title("Kernel PCA with RBF Kernel")
plt.show()
```

Practical Issues with **Kernel Methods**

Computational Complexity

Problem:

- Kernel methods involve computing the **Kernel (Gram) Matrix** of size n×n, where n is the number of training samples.

Importance:

- Time complexity: $O(n^2)$
- Memory usage: $O(n^2)$
- For large datasets (e.g., n>10,000), this becomes **infeasible**.

Choice of Kernel Function

Problem:

- There is **no universal kernel** that works best for all problems.

Issues:

- Requires **domain knowledge** or trial-and-error.
- Poor kernel choice ⇒ underfitting or overfitting.
- Common kernels: **RBF**, **polynomial**, **sigmoid**, but:
  - RBF: may perform poorly if data has multiple scales
  - Polynomial: sensitive to degree and coefficients

Hyperparameter Tuning

Problem:

- Most kernels have **tunable parameters** (e.g., σ in RBF, degree in polynomial).

Impact:

- Incorrect values can drastically reduce performance.
- Grid search or cross-validation is computationally expensive.

Lack of Interpretability

Problem:

- Kernel methods operate in **high-dimensional feature spaces** that are implicit and abstract.

Impact:

- Hard to understand **what features** are being used.
- Difficult to interpret decision boundaries or transformed data.
- Not ideal for explainable AI scenarios.

No Sparse Solutions for Some Kernels

Problem:

- SVM with **linear kernels** can result in **sparse models** (fewer support vectors).
- Non-linear kernels often yield **dense** solutions:
  - Many support vectors ⇒ slower predictions.

Scalability to Large Datasets

Problem:

- Kernel methods don't scale well with large-scale problems.
- Batch learning setup is inefficient for streaming or real-time data.

Numerical Stability

Problem:

- Kernel matrix K must be **positive semi-definite** (PSD).
- In practice, due to floating-point errors or bad kernels, K can become **non-PSD**, leading to instability in algorithms (e.g., eigen decomposition in Kernel PCA).

Pre-image Problem (in Kernel PCA)

Problem:

- After projecting to lower-dimensional space (e.g., with Kernel PCA), it's **non-trivial to recover the original input** from the projection.

Importance:

- Limits reconstruction, denoising, or generative tasks.

Overfitting Risk

Problem:

- Rich kernel spaces can **overfit** small datasets if regularization is not properly handled.

Implementation Complexity

Problem:

- More complex than linear models.
- Requires careful coding and parameter control to avoid pitfalls.

Summary Table

Issue	**Description**
Computational Complexity	Needs $O(n^2)$ memory and time
Kernel Selection	No one-size-fits-all kernel
Parameter Tuning	Sensitive to kernel parameters
Interpretability	Hard to understand inner workings
Scalability	Poor performance on big data
Numerical Instability	Kernel matrix may be non-PSD
Pre-image Problem	Can't easily go back to input space
Overfitting	Risky with small datasets
Dense Solutions	Many support vectors slow down prediction

Applications in Text and Image Classification

Applications in Text Classification

1. **Spam Detection**
   - Classify emails as spam or not spam
   - Use **TF-IDF** (Term Frequency-Inverse Document Frequency) or **Bag-of-Words** features
   - Kernels: Linear, Polynomial
2. **Sentiment Analysis**
   - Classify movie/product reviews as positive or negative
   - Kernels handle non-linear sentiment patterns
3. **Topic Categorization**
   - News articles categorized into politics, sports, etc.
   - **Linear kernel** often works well with high-dimensional sparse data
4. **Language Detection**
   - Identify the language of text samples
   - Kernel trick helps with capturing n-gram relationships

Applications in Image Classification

1. **Object Recognition**
   - Classify objects like cars, animals, etc., in images
   - Use **RBF kernel** or **Histogram Intersection Kernel**
2. **Facial Recognition**
   - Match a given face to identities
   - Kernel methods project face data to higher dimensions for better separation
3. **Digit Recognition (e.g., MNIST** Modified National Institute of Standards and Technology)
   - Classify handwritten digits
   - SVM with **RBF kernel** achieves high accuracy
4. **Texture Classification**
   - Identify types of surfaces or materials in images
   - Uses specialized kernels (e.g., pyramid match kernels)

Benefits of Kernel Methods in Text/Image:

- Work well with high-dimensional data
- Don't require deep architectures
- Can be very accurate with good kernel choice and tuning