Chapter 2:
Supervised Learning

Ph.D. Nguyen Thi Khanh Tien

tiennntk@ut.edu.vn

Introduction to Supervised Learning

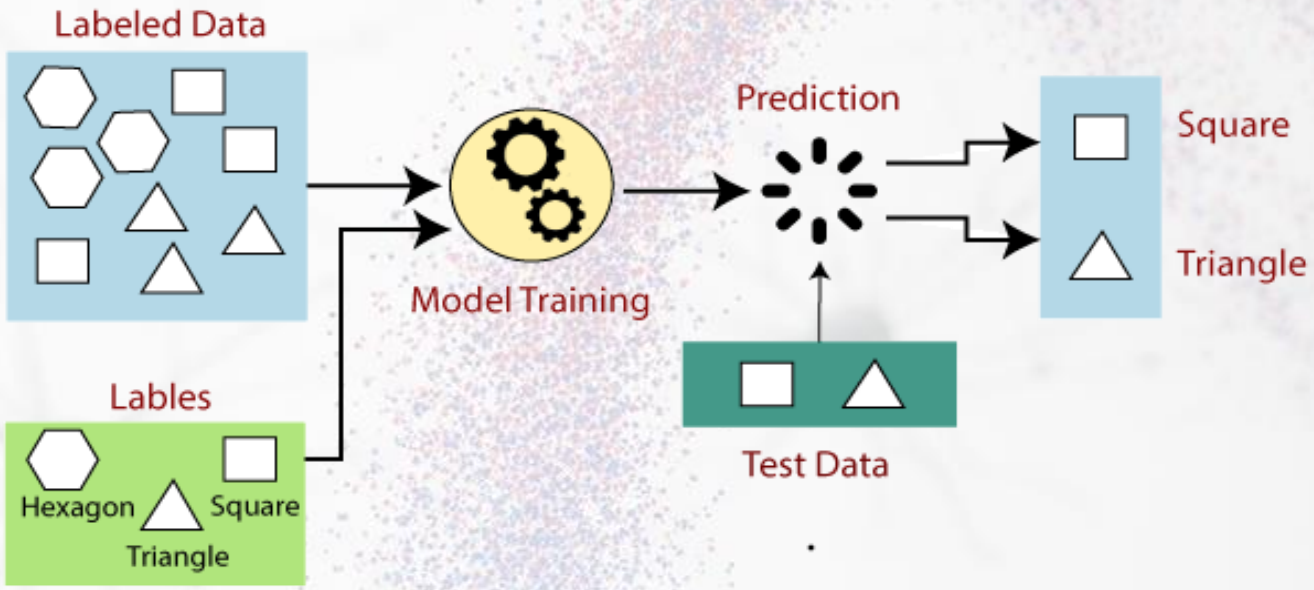
Supervised learning is a machine learning technique where a model is trained on labeled data. This means that the model is provided with both input data and its corresponding correct output. By analyzing these examples, the model learns to predict the output for new, unseen data.

Supervised learning algorithms aim to identify a pattern or relationship between the input and output variables, effectively mapping input data (x) to output data (y). This learned mapping function can then be used to make predictions on new data.

Real-world applications of supervised learning include tasks such as risk assessment, image classification, fraud detection, spam filtering, and more

Example:

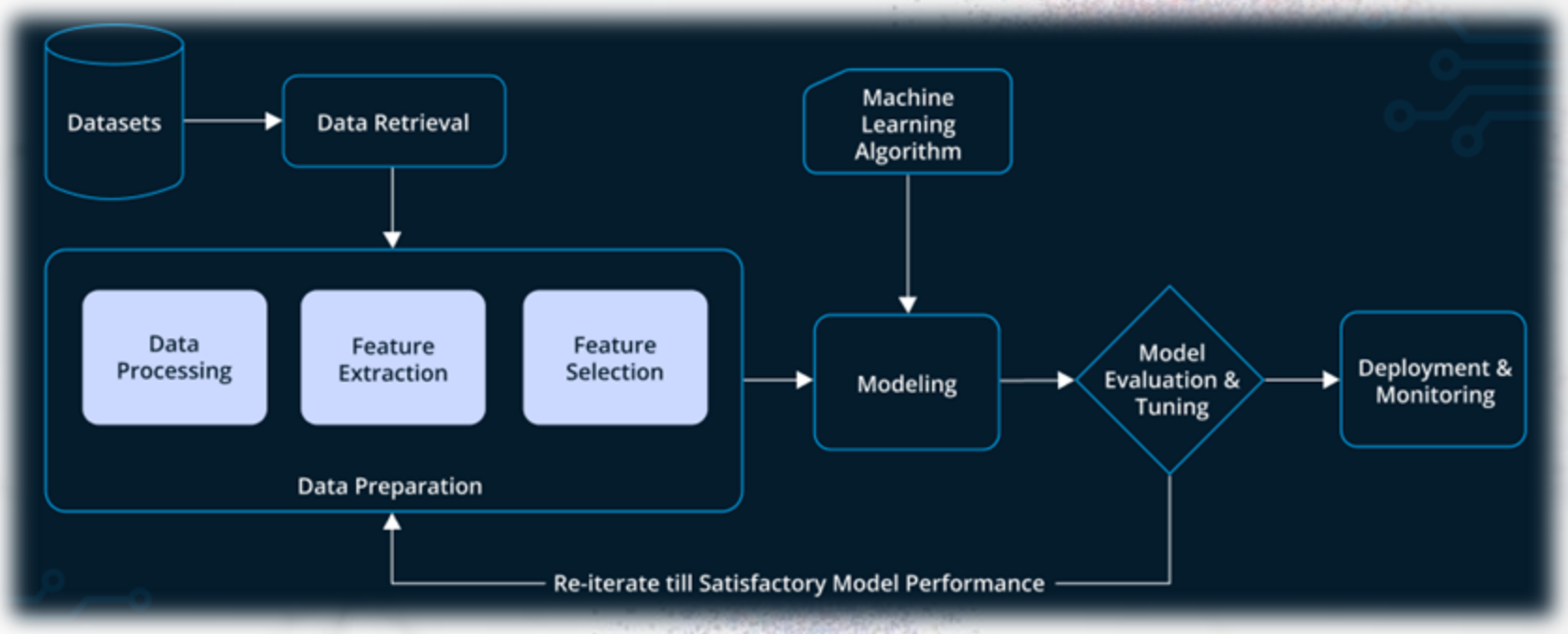
- Email → Spam or Not Spam
- Image → Cat or Dog
- House features → Price prediction

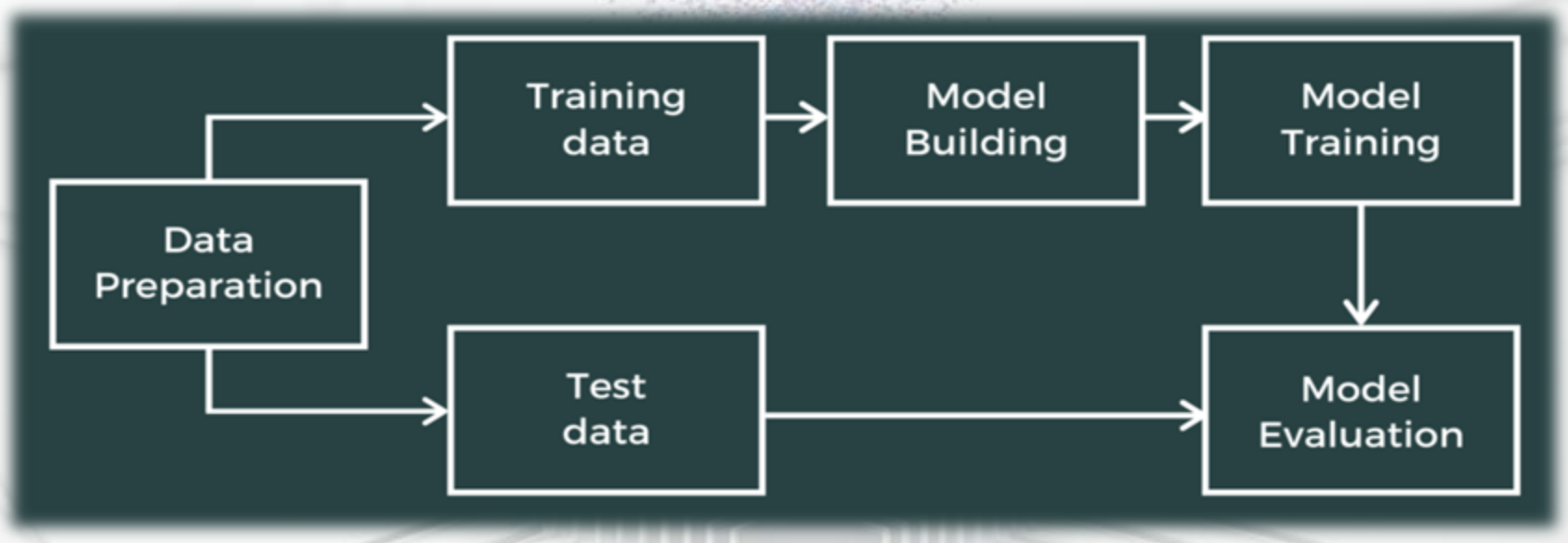


How It Works:

1. **Input Data (X):** Features or observations
2. **Output Labels (Y):** The correct answers (targets)
3. **Model Training:** The algorithm learns a mapping from X → Y
4. **Prediction:** The model uses this mapping to predict Y for new X
5. **Evaluation:** Performance is measured using metrics like accuracy, precision, etc.

Steps Involved in Supervised Learning





Steps Involved in Supervised Learning

Data Collection and Preparation:

- **Gather data:** Collect a dataset that is relevant to the problem you want to solve.
- **Clean and preprocess:** Remove noise, handle missing values, and normalize or standardize the data to ensure consistency.
- **Labeling:** Assign correct labels or outputs to each data point.

Choose a Model:

- **Select an algorithm:** Consider factors like the nature of your data (e.g., numerical, categorical), the desired output (e.g., classification, regression), and computational resources.
- **Common algorithms:** linear regression, logistic regression, decision trees, random forests, support vector machines (SVMs), neural networks,..

Split the Data:

- **Training set:** Use this portion of the data to train the model.
- **Validation set:** Evaluate the model's performance during training and fine-tune hyperparameters.
- **Test set:** Assess the model's final performance on unseen data.

Training:

- **Iterative process:** The model learns from the training data by adjusting its parameters to minimize the difference between its predicted outputs and the actual labels.
- **Optimization:** Use techniques like gradient descent to find the optimal parameter values.

Evaluation:

- **Metrics:** Use appropriate metrics to measure the model's performance on the validation and test sets.
- **Common metrics:** Examples include accuracy, precision, recall, F1-score, mean squared error (MSE), and root mean squared error (RMSE).

Deployment:

- **Integration:** Once satisfied with the model's performance, deploy it into a production environment.
- **Monitoring:** Continuously monitor the model's performance and retrain it if necessary to adapt to changing data or conditions.

Advantages and Disadvantages of Supervised Learning

ADVANTAGES

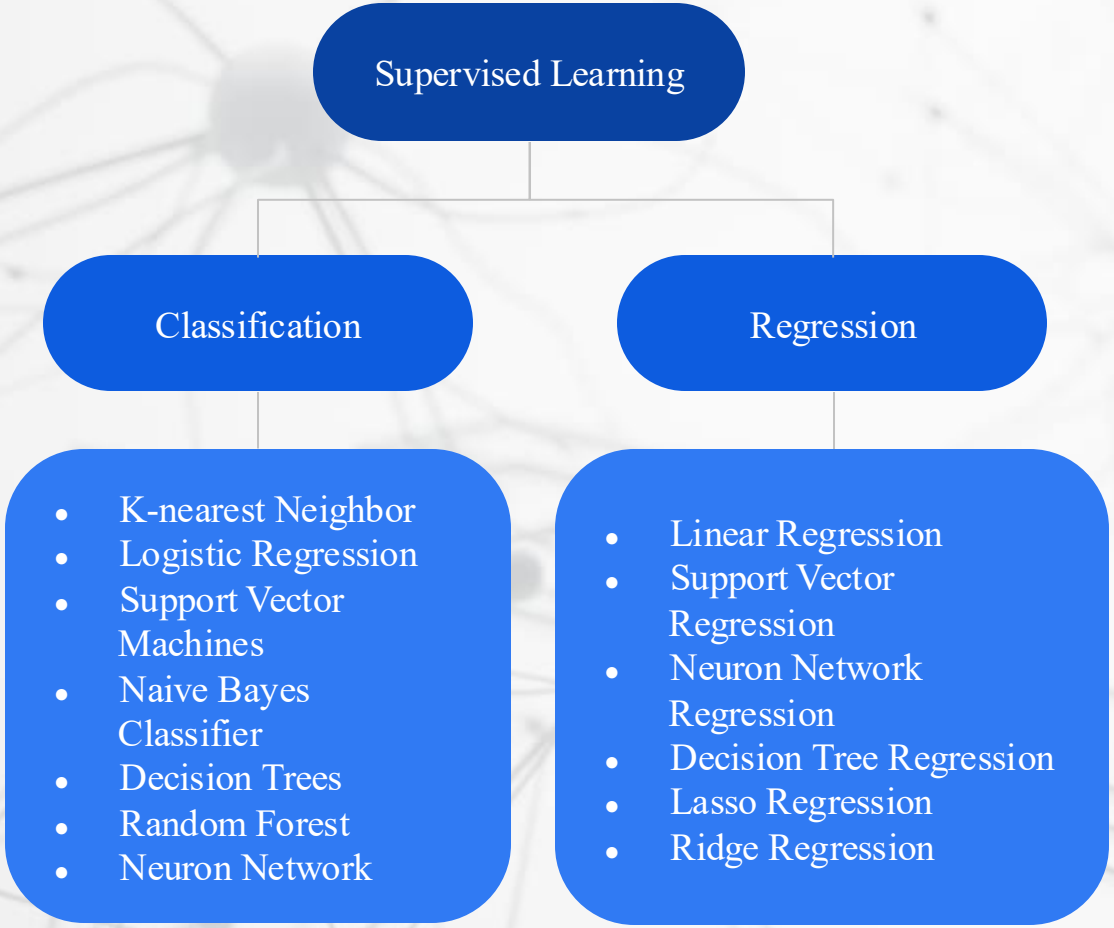
- **High Accuracy:** Produces reliable and accurate predictions with sufficient labeled data
- **Clear Objective:** Learns a direct mapping from input to output
- **Wide Applications:** Used in classification, regression, fraud detection, medical diagnosis, etc.

DISADVANTAGES

- **Needs Labeled Data:** Labeling large datasets is time-cons-uming and ad costly
- **Limited to Known Classes:** Cannot handle unseen classes or patterns well
- **Overfitting Risk:** May memorize training data and perform poorly on new data
- **Scalability Issues:** Training time can increase with complex or large datasets

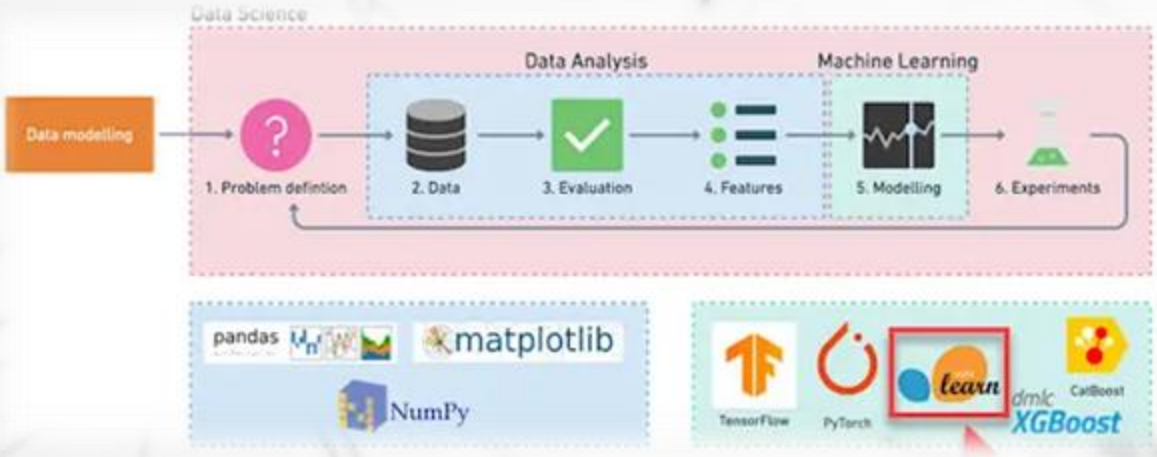
Types of Supervised Learning

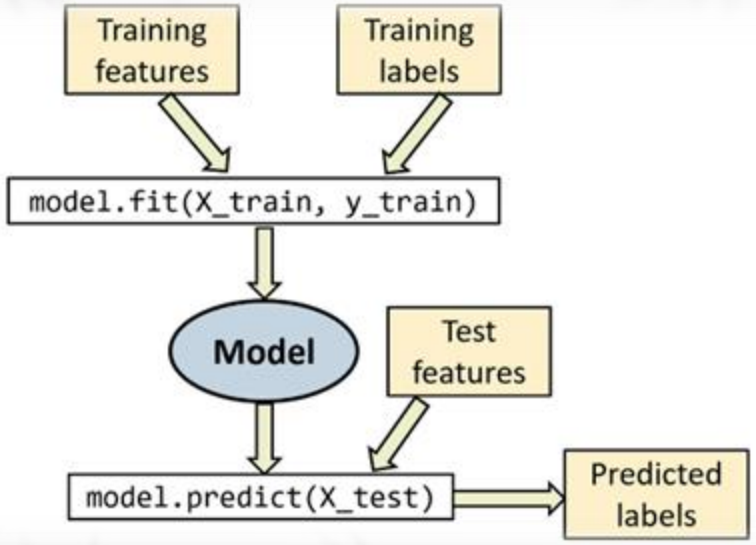
Based on the given datasets the SL problem is categorized into two types, ***Classification*** and ***Regression.***

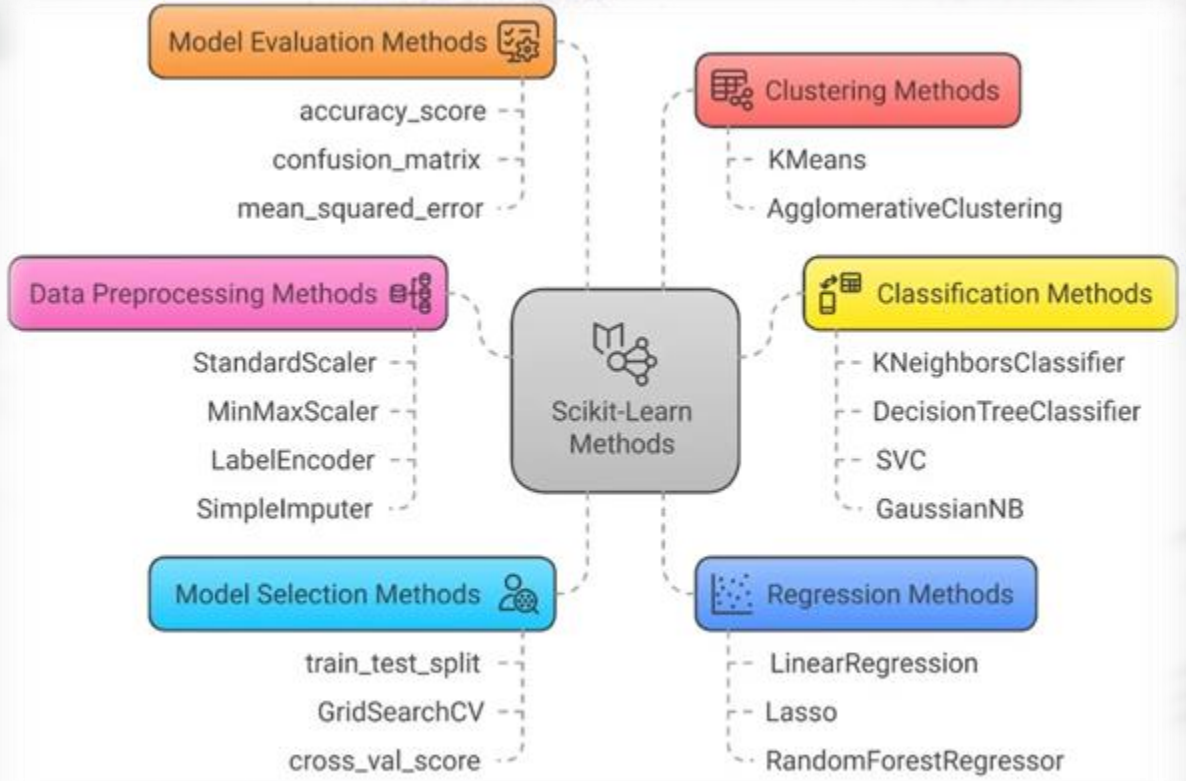


	Classification	**Regression**
Goal	To predict categorical outcomes	To predict continuous numerical values
Output type	Categorical (e.g., yes/no, class labels)	Continuous (e.g., numbers)
Evaluation metrics	Accuracy, Precision, Recall, F1-score,...	Mean Squared Error (MSE), Mean Absolute error (MAE), Root Mean Squared Error (RMSE), R-squared
Examples	Email spam detection Image recognition Customer churn prediction	House price prediction Stock price forecasting Sales prediction

Scikit-learn Methods for ML

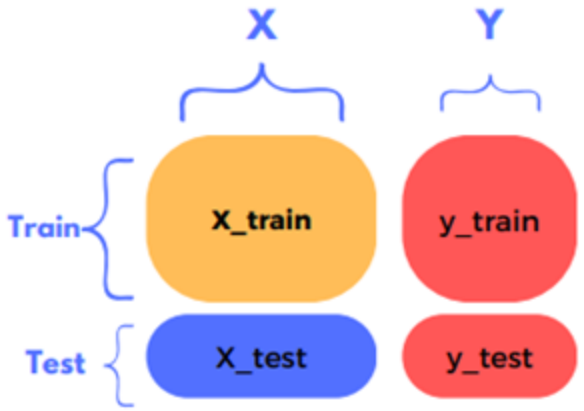






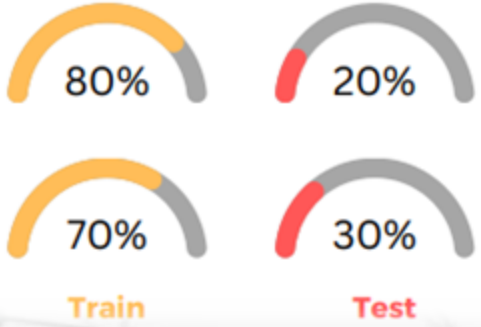
Scikit-learn Methods for Supervised Learning

Train-Test Split



- **Method of dividing the data into two part: Training and Testing**
- **Training Data : Model Training**
- **Testing Data : Model Evaluation**

Standard Ratio

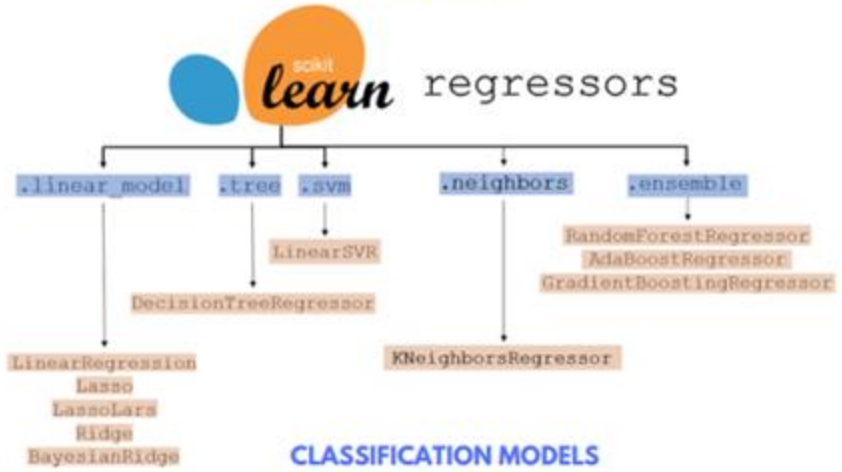



```
from sklearn.model_selection import
train_test_split
X_train,X_test,y_train,y_test =
train_test_split(X,y,test_size=0.3)
```

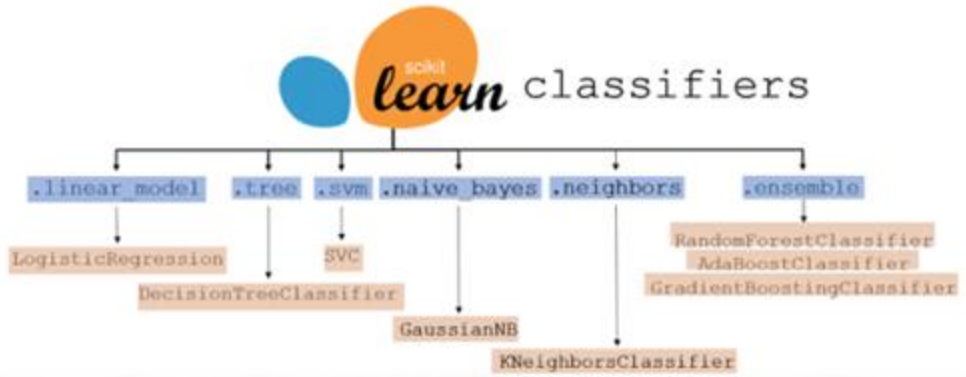
Supervised Learning Models

```
#Import
from sklearn.branch import
model_name

# Create a Instance
model = model_name()

# fit model
model.fit(X_train, y_train)
```

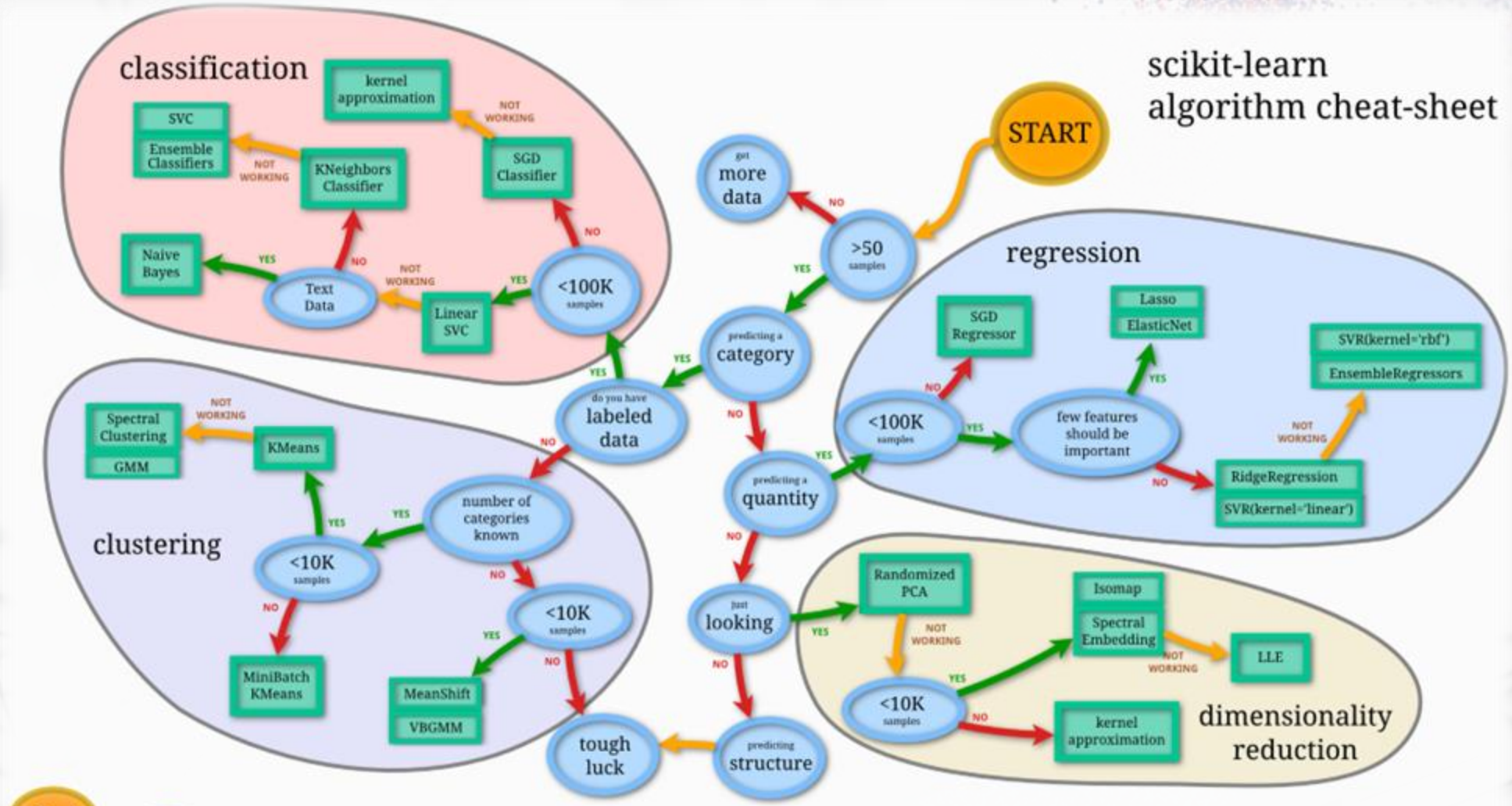
REGRESSION MODELS



CLASSIFICATION MODELS



Scikit-learn Algorithm



KDnuggets | Visit KDnuggets.com for more cheatsheets and additional learning resources.

Scikit-learn CheatSheet

Scikit-learn is an open-source Python library for all kinds of predictive data analysis. You can perform classification, regression, clustering, dimensionality reduction, model tuning, and data preprocessing tasks.

Loading the Data

Classification

```
from sklearn import datasets
X, y = datasets.load_wine(return_X_y=True)
```

Regression

```
diabetes = datasets.load_diabetes()
X, y = diabetes.data, diabetes.target
```

Training And Test Data

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, random_state=0
)
```

Preprocessing the Data

Standardization

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaled_X_train = scaler.fit_transform(X_train)
scaled_X_test = scaler.transform(X_test)
```

Normalization

```
from sklearn.preprocessing import Normalizer
norm = Normalizer()
norm_X_train = norm.fit_transform(X_train)
norm_X_test = norm.transform(X_test)
```

Binarization

```
from sklearn.preprocessing import Binarizer
binary = Binarizer(threshold=0.0)
binary_X = binary.fit_transform(X)
```

Encoding Categorical Features

```
from sklearn.preprocessing import LabelEncoder
lab_enc = LabelEncoder()
y = lab_enc.fit_transform(y)
```

Imputer

```
from sklearn.impute import SimpleImputer
imp_mean = SimpleImputer(missing_values=0,
strategy='mean')
imp_mean.fit_transform(X_train)
```

Supervised Learning Model

Linear Regression

```
from sklearn.linear_model import LinearRegression
lr = LinearRegression()
```

Support Vector Machines

```
from sklearn.svm import SVC
svm_svc = SVC(kernel='linear')
```

Naive Bayes

```
from sklearn.naive_bayes import GaussianNB
gnb = GaussianNB()
```

Unsupervised Learning Model

Principal Component Analysis

```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
```

K Means

```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=5, random_state=0)
```

Model Fitting

Supervised Learning

```
lr.fit(X_train, y_train)
svm_svc.fit(X_train, y_train)
```

Unsupervised Learning

```
model = pca.fit_transform(X_train)
kmeans.fit(X_train)
```

Prediction

Supervised Learning

```
y_pred = lr.predict_proba(X_test)
y_pred = svm_svc.predict(X_test)
```

Unsupervised Learning

```
y_pred = kmeans.predict(X_test)
```

Evaluation

Accuracy Score

```
lr.score(X_test, y_test)

from sklearn.metrics import accuracy_score
accuracy_score(y_test, y_pred)
```

Classification Report

```
from sklearn.metrics import classification_report
print(classification_report(y_test, y_pred))
```

Mean Squared Error

```
from sklearn.metrics import mean_squared_error
mean_squared_error(y_test, y_pred)
```

R2 Score

```
from sklearn.metrics import r2_score
r2_score(y_test, y_pred)
```

Adjusted Rand Index

```
from sklearn.metrics import adjusted_rand_score
adjusted_rand_score(y_test, y_pred)
```

Cross-Validation

```
from sklearn.model_selection import cross_val_score
cross_val_score( lr, X, y, cv=5, scoring='f1_macro')
```

Model Tuning

```
from sklearn.model_selection import GridSearchCV
parameters = {'kernel':('linear', 'rbf'), 'C':[1, 10]}
model = GridSearchCV(svm_svc, parameters)
model.fit(X_train, y_train)
print(model.best_score_)
print(model.best_estimator_)
```

Subscribe to KDnuggets News

Example: Iris Flower Classification with Scikit-learn

Problem: Predict the species of an Iris flower based on its sepal length, sepal width, petal length, and petal width. This is a multi-class classification problem.

Step 1: Load Data

Scikit-learn comes with several toy datasets, including the Iris dataset, which is perfect for demonstration.

- *load_iris()*: Fetches the Iris dataset.
- *iris.data*: Contains the feature measurements (sepal length, sepal width, petal length, petal width).
- *iris.feature_names*: Names of the features.
- *iris.target*: Contains the species labels (0, 1, 2 representing 'setosa', 'versicolor', 'virginica').
- *iris.target_names*: The actual names of the target classes.
- Convert *X* to a Pandas DataFrame and *y* to a *Pandas Series* for better readability and manipulation, although *NumPy* arrays work perfectly fine with Scikit-learn.

```python
# *** Step 1: Load Data ***
# Import necessary libraries
from sklearn.datasets import load_iris
import pandas as pd

# Load the Iris dataset
iris = load_iris()

# The data is stored in .data attribute
X = pd.DataFrame(iris.data, columns=iris.feature_names)

# The target variable is stored in .target attribute
y = pd.Series(iris.target)

# Display the first 5 rows of the features
print("Features (X):")
print(X.head())

# Display the first 5 target values and their corresponding names
print("\nTarget (y) and Target Names:")
print(y.head())
print("Target names:", iris.target_names)
```

Example: Iris Flower Classification with Scikit-learn

Step 2: Split Data

- *train_test_split()*: This function randomly splits the data.
- *test_size=0.30*: 30% of the data will be allocated to the testing set, and the remaining 70% to the training set.
- *random_state=42*: This ensures that the split is the same every time you run the code, which is important for reproducibility.

Step 3: Preprocess Data

For this particular dataset and Logistic Regression, scaling might not have a dramatic effect because the features are already on a similar scale. However, it's a good practice, especially for algorithms sensitive to feature scales (like SVMs, KNNs, neural networks).

- *StandardScaler()*: Standardizes features by removing the mean and scaling to unit variance. This ensures all features contribute equally to the model.
- *fit_transform(X_train)*: Calculates the mean and standard deviation from *X_train* and then applies the transformation.
- *transform(X_test)*: Applies the *same* mean and standard deviation learned from *X_train* to *X_test*. You **never** fit a scaler on the test data.

```python
# *** Step 2: Split Data ***
from sklearn.model_selection import train_test_split

# Split the data into training and testing sets
# test_size=0.30 means 30% of the data will be used for testing
# random_state for reproducibility (same split every time)
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.30,
                                                    random_state=42)

print("\nShape of X_train:", X_train.shape)
print("Shape of X_test:", X_test.shape)
print("Shape of y_train:", y_train.shape)
print("Shape of y_test:", y_test.shape)
```

```python
# *** Step 3: Preprocess Data ***
# Standardization is a common preprocessing step for many machine learning algorithms
# It scales the features to have a mean of 0 and a standard deviation of 1
from sklearn.preprocessing import StandardScaler

# Initialize the StandardScaler
scaler = StandardScaler()

# Fit the scaler on the training data and transform both training and testing data
# IMPORTANT: Fit only on training data to avoid data leakage from the test set
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

print("\nFirst 5 rows of X_train (original):")
print(X_train.head())

print("\nFirst 5 rows of X_train_scaled (after standardization):")
print(X_train_scaled[:5])
```

Example: Iris Flower Classification with Scikit-learn

Step 4: Choose a Model

For a multi-class classification problem like Iris, Logistic Regression is a good starting point due to its simplicity and interpretability.

- *LogisticRegression()*: Instantiates the Logistic Regression classifier.
- *solver='liblinear'*: Specifies the algorithm to use for optimization. Different solvers are suitable for different dataset sizes and types.
- *random_state*: Ensures the model's internal random processes (if any) are reproducible.

Step 5: Train the Model

- *model.fit(X_train_scaled, y_train)*: This is where the magic happens! The model learns the patterns and relationships between the features (*X_train_scaled*) and the target labels (*y_train*).

Step 6: Make Predictions

- *model.predict(X_test_scaled)*: The trained model takes the scaled test features as input and outputs its predicted class for each sample

```python
# *** Step 4: Choose a Model ***
# For this example, we will use Logistic Regression as a classification model
# Logistic Regression is suitable for binary and multiclass classification problems
from sklearn.linear_model import LogisticRegression

# Initialize the Logistic Regression model
# solver='liblinear' is a good choice for smaller datasets
# random_state for reproducibility of the model's internal randomness
model = LogisticRegression(solver='liblinear', random_state=42)

print("\nChosen Model:", model)
```

```python
# *** Step 5: Train the Model ***
# Fit the model on the training data
model.fit(X_train_scaled, y_train)

print("\nModel trained successfully!")
```

```python
# *** Step 6: Make Predictions ***
# Use the trained model to make predictions on the test set
y_pred = model.predict(X_test_scaled)
# print("\nPredictions on the test set:")
# print(y_pred)
print("\nFirst 10 actual labels (y_test):", y_test.values[:10])
print("First 10 predicted labels (y_pred):", y_pred[:10])
```

✓ 0.0s

Example: Iris Flower Classification with Scikit-learn

Step 7: Evaluate the Model

Evaluate how well your model performed on the test set using various metrics. For classification, common metrics include accuracy, precision, recall, and F1-score, often summarized in a classification report and a confusion matrix.

- *accuracy_score()*: Calculates the proportion of correctly classified samples.
- *classification_report()*: Provides a detailed summary of precision, recall, and F1-score for each class, as well as overall support.
- *Precision*: Of all samples predicted as a certain class, how many were actually that class?
- *Recall*: Of all samples that truly belong to a certain class, how many were correctly predicted as that class?
- *F1-Score*: The harmonic mean of precision and recall, providing a balanced measure.
- *confusion_matrix()*: A table that shows the number of correct and incorrect predictions for each class. It's excellent for understanding where your model makes mistakes.

```python
# *** Step 7: Evaluate the Model ***
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"\nAccuracy: {accuracy:.4f}")

# Generate a classification report (precision, recall, f1-score per class)
class_report = classification_report(y_test, y_pred, target_names=iris.target_names)
print("\nClassification Report:\n", class_report)

# Generate a confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print("\nConfusion Matrix:\n", conf_matrix)

# Interpretation of Confusion Matrix:
# Rows represent true labels, columns represent predicted labels.
# conf_matrix[i, j] is the number of observations known to be in group i
# and predicted to be in group j.
# For example, in a 3x3 matrix:
# [[TN, FP, FP],
#  [FN, TP, FN],
#  [FN, FN, TN]]
# (This interpretation is more for binary. For multi-class, it's just true vs predicted counts)
print("\n--- Confusion Matrix Interpretation ---")
print("Rows: True Labels")
print("Columns: Predicted Labels")
print(pd.DataFrame(conf_matrix, index=iris.target_names, columns=iris.target_names))
```

Example: Iris Flower Classification with Scikit-learn

Step 7: Evaluate the Model

Evaluate how well your model performed on the test set using various metrics. For classification, common metrics include accuracy, precision, recall, and F1-score, often summarized in a classification report and a confusion matrix.

- *accuracy_score()*: Calculates the proportion of correctly classified samples.
- *classification_report()*: Provides a detailed summary of precision, recall, and F1-score for each class, as well as overall support.
- **Precision:** Of all samples predicted as a certain class, how many were actually that class?
- **Recall:** Of all samples that truly belong to a certain class, how many were correctly predicted as that class?
- **F1-Score:** The harmonic mean of precision and recall, providing a balanced measure.
- *confusion_matrix()*: A table that shows the number of correct and incorrect predictions for each class. It's excellent for understanding where your model makes mistakes.

```python
# *** Step 7: Evaluate the Model ***
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"\nAccuracy: {accuracy:.4f}")

# Generate a classification report (precision, recall, f1-score per class)
class_report = classification_report(y_test, y_pred, target_names=iris.target_names)
print("\nClassification Report:\n", class_report)

# Generate a confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print("\nConfusion Matrix:\n", conf_matrix)

# Interpretation of Confusion Matrix:
# Rows represent true labels, columns represent predicted labels.
# conf_matrix[i, j] is the number of observations known to be in group i
# and predicted to be in group j.
# For example, in a 3x3 matrix:
# [[TN, FP, FP],
#  [FN, TP, FN],
#  [FN, FN, TN]]
# (This interpretation is more for binary. For multi-class, it's just true vs predicted counts)
print("\n--- Confusion Matrix Interpretation ---")
print("Rows: True Labels")
print("Columns: Predicted Labels")
print(pd.DataFrame(conf_matrix, index=iris.target_names, columns=iris.target_names))
```

Example: Iris Flower Classification with Scikit-learn

Step 8: Tune Hyperparameters

Hyperparameter tuning involves finding the best set of hyperparameters for your model to optimize its performance. *GridSearchCV* is a common method for this.

- *GridSearchCV*: Systematically searches through a predefined set of hyperparameter values, evaluates each combination using cross-validation, and selects the combination that yields the best performance.
- *param_grid*: A dictionary where keys are hyperparameter names and values are lists of values to try.
- *cv=5*: Performs 5-fold cross-validation, meaning the training data is split into 5 parts, and the model is trained 5 times, each time using a different part as a validation set.
- *scoring='accuracy'*: The metric used to evaluate the model during cross-validation.
- *grid_search.best_params_*: The hyperparameter combination that resulted in the best performance.
- *grid_search.best_score_*: The cross-validation score achieved with the best hyperparameters.
- *grid_search.best_estimator_*: The actual trained model with the best hyperparameters.

```python
# *** Step 8: Tune Hyperparameters ***
# Hyperparameter tuning can be done using techniques like Grid Search or Random Search.
from sklearn.model_selection import GridSearchCV

# Define the parameter grid to search
param_grid = {
    'C': [0.001, 0.01, 0.1, 1, 10, 100],  # Inverse of regularization strength
    'solver': ['liblinear', 'lbfgs']     # Algorithms to use in the optimization problem
}

# Initialize GridSearchCV
# estimator: The model to tune
# param_grid: The dictionary of hyperparameters to try
# cv: Number of cross-validation folds
# scoring: The metric to optimize (e.g., 'accuracy' for classification)
grid_search = GridSearchCV(LogisticRegression(random_state=42),
                           param_grid, cv=5, scoring='accuracy')

# Fit GridSearchCV to the scaled training data
grid_search.fit(X_train_scaled, y_train)

print("\nBest Hyperparameters found:", grid_search.best_params_)
print("Best Cross-validation Accuracy:", grid_search.best_score_)

# Get the best model
best_model = grid_search.best_estimator_

# Evaluate the best model on the test set
y_pred_tuned = best_model.predict(X_test_scaled)
accuracy_tuned = accuracy_score(y_test, y_pred_tuned)
print(f"Accuracy with Tuned Model on Test Set: {accuracy_tuned:.4f}")

# You can also get a classification report for the tuned model
print("\nClassification Report with Tuned Model:\n",
      classification_report(y_test, y_pred_tuned, target_names=iris.target_names))
```

Machine Learning for Classification

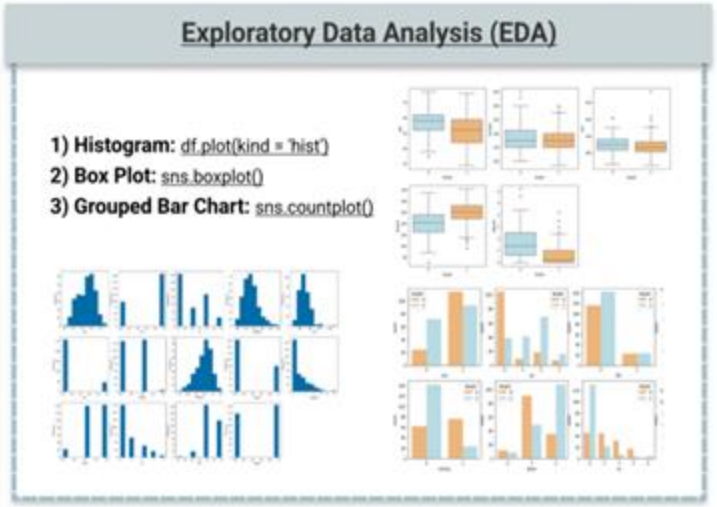
Classification is a supervised learning technique used to predict the category or class of new data points based on a set of labeled training examples. In classification, a model learns patterns from the training data to categorize unseen observations into predefined classes or groups.

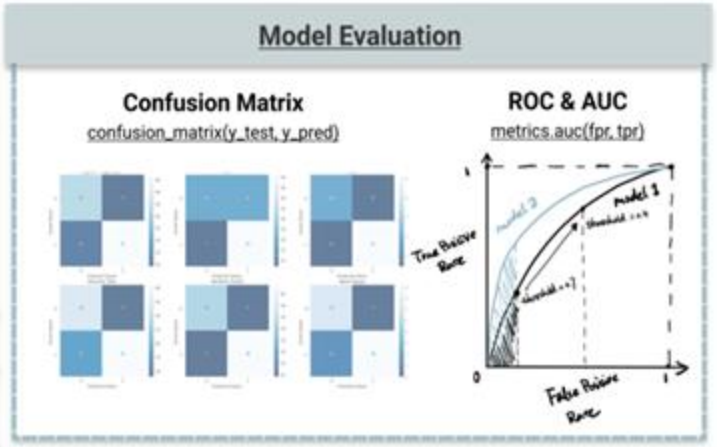
Two types of Classifications:

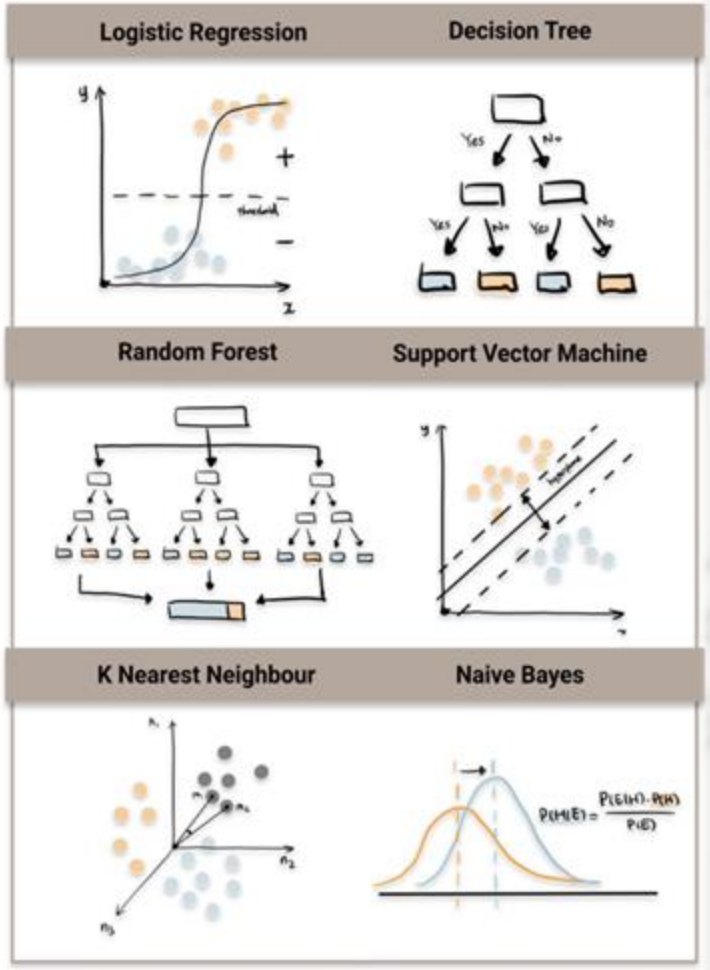
- **Binary classification:** Predicting outcomes with only two possible classes (e.g., spam or not spam, male or female, yes or no)
- **Multi-class classification:** **P**redicting outcomes with more than two possible classes (e.g., classifying types of crops, classifying types of music)

Classification algorithms are widely used in various fields, including:

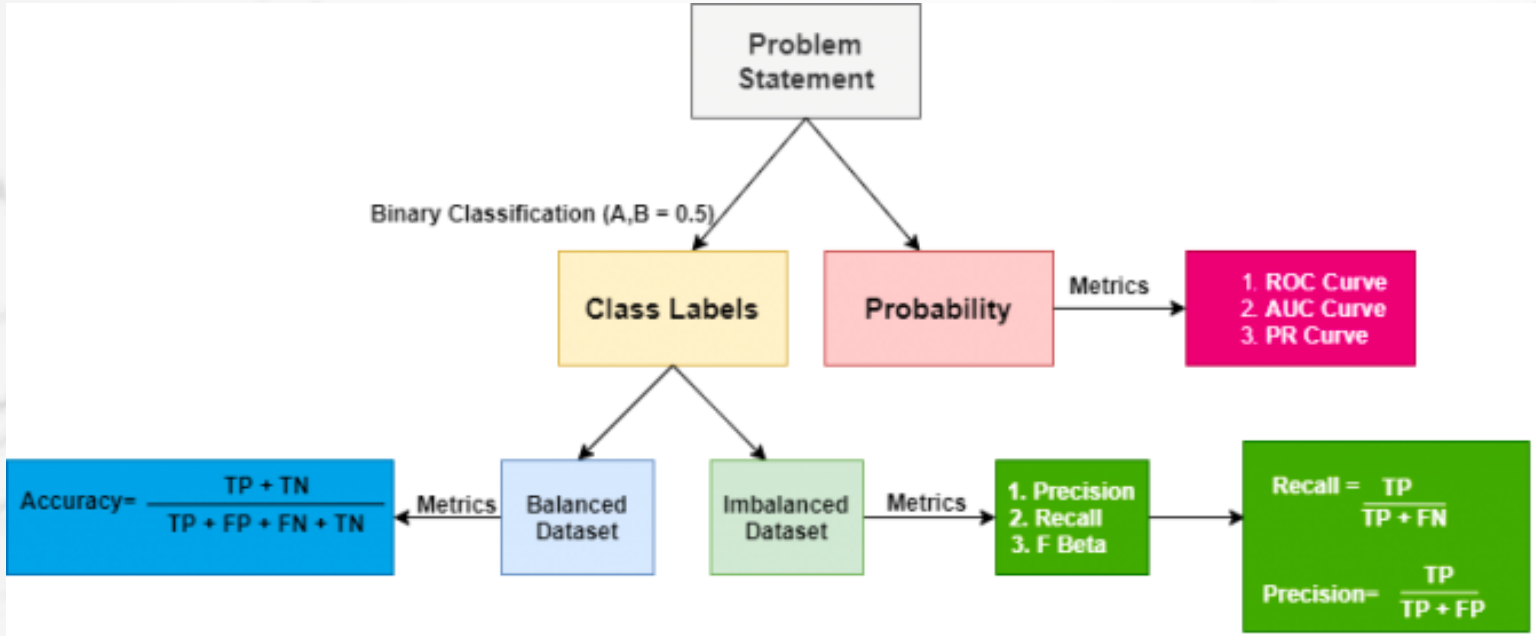
- Healthcare: Diagnosing diseases
- Finance: Fraud detection
- Marketing: Customer segmentation
- Image processing: Object recognition

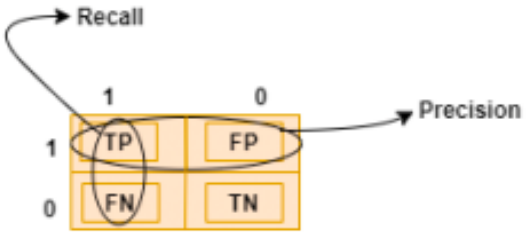




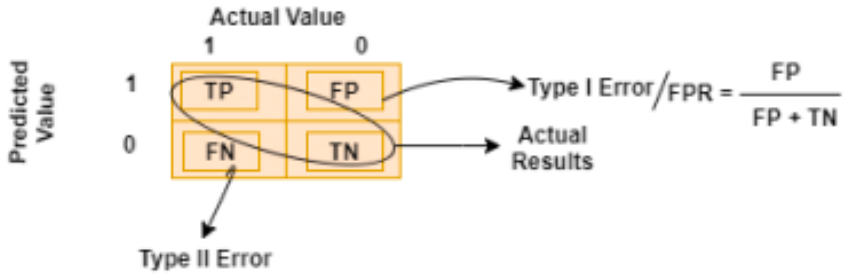


Classification Metrics in Machine Learning





Confusion Matrix



Always have to focus on reducing the Type I Error and Type II Error

Recall (TPR) : Out of the total positive actual value, how many values did we correctly predicted positively. Whenever the FN is important wrt the problem statement, use Recall.

Precision : Out of the total predicted positive results, how many results were actual positive. Whenever the FP is important wrt the problem statement, then use Precision

In some of the problem statements FP and FN both are equally important in an imbalanced dataset.In such cases we have to consider both **Recall** and **Precision.** In such cases we have to use **F-beta score**

$$\text{F-Beta} = \frac{(1+B^2)\ \text{Precision X Recall}}{B^2 \times (\text{Precision} + \text{Recall})}$$

1. If we have a problem statement where the **FP** and **FN** are both equally important, we select **Beta = 1**
2. In scenarios where the **FP** is having greater impact than the **FN**, in such case we reduce the **Beta** value between 0 to 1
3. In scenarios where the **FN** is having greater impact than the **FP**, in such case we increase the **Beta** value greater than 1 and might range between 1 to 10

Classification Metrics in Machine Learning

		Predicted class: Positive	Predicted class: Negative
Actual class	Positive	TP	FN
	Negative	FP	TN

$$\text{Precision} = \frac{True\ Positive}{True\ Positive + False\ Positive}$$

$$\text{Recall} = \frac{True\ Positive}{True\ Positive + False\ Negative}$$

$$F_1 = 2 * \frac{precision * recall}{precision + recall}$$

Confusion Matrix is a table with combinations of predicted and actual values.
True Positive (TP). We predicted positive and it's true.
True Negative (TN). We predicted negative and it's true.
False Positive: (FP). We predicted positive and it's false.
False Negative: (FN). We predicted negative and it's false.

Precision explains how many of the correctly predicted cases actually turned out to be positive.
Recall explains how many of the actual positive cases we were able to predict correctly with our model.
F1 Score is the harmonic mean of precision and recall. F1 Score could be an effective evaluation metric in the following cases:

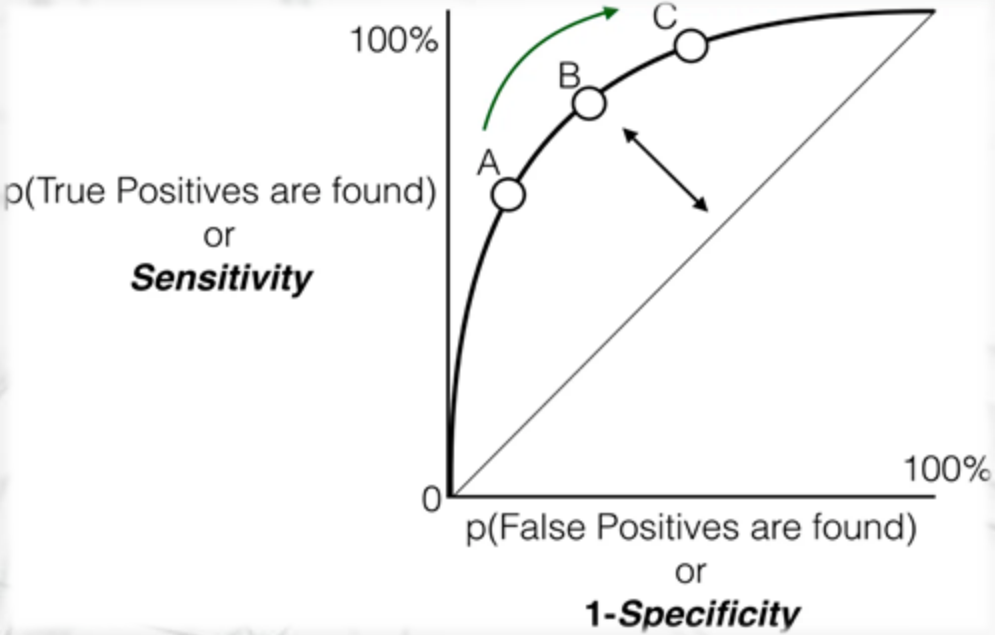
- When FP and FN are equally costly.
- Adding more data doesn't effectively change the outcome
- TN is high

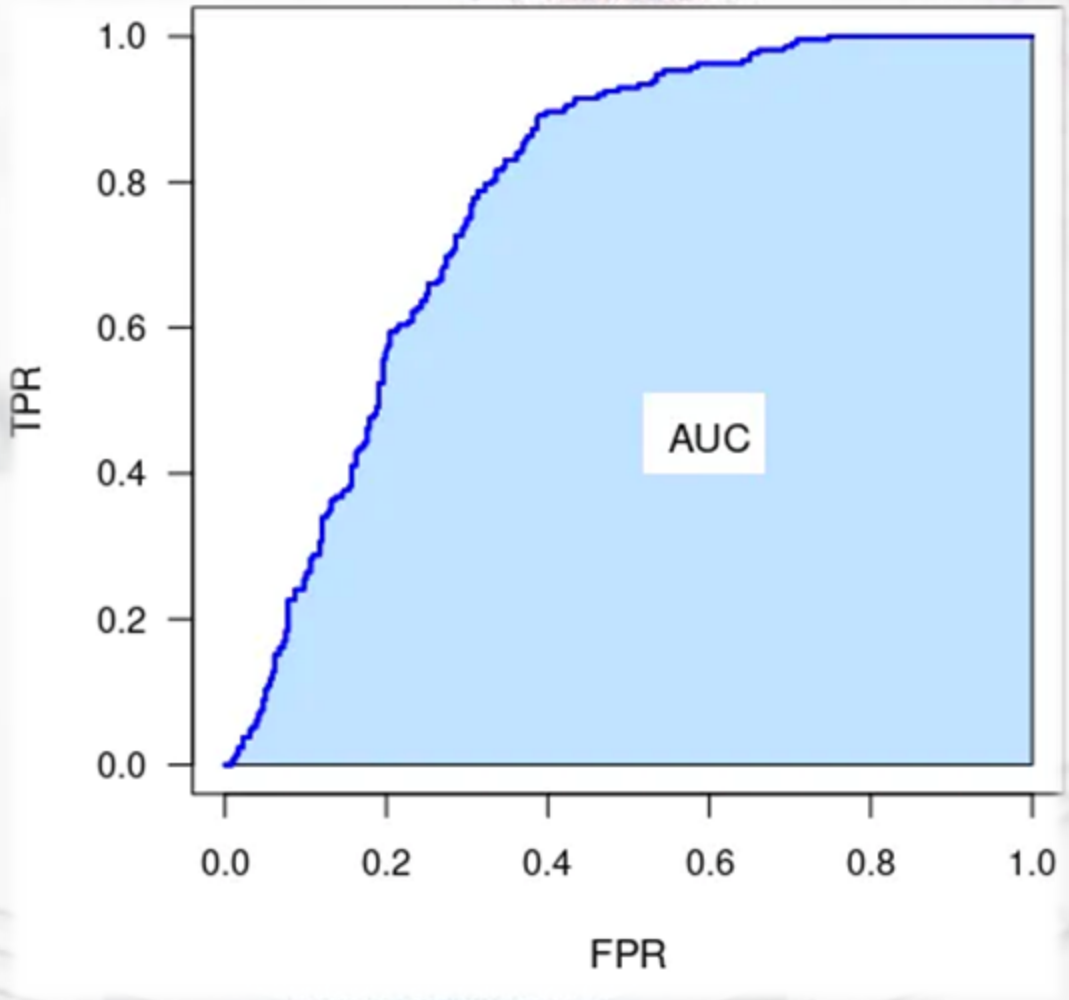
Classification Metrics in Machine Learning

ROC / AUC. The Receiver Operator Characteristic (ROC) graph provides an elegant way of presenting multiple confusion matrices produced at different thresholds. A ROC plots the relationship between the true positive rate (**TPR**) and the false positive rate (**FPR**).

TPR = Recall = Sensitivity = TP/ (TP+ FN)

***FPR* = 1 – specificity = FP/ (FP+ TN)**





Classification Algorithms

Types	Common Classification Algorithms
Linear Classifiers	• **Logistic Regression:** A statistical model that predicts the probability of a categorical outcome. • **Support Vector Machines (SVMs):** A set of supervised learning methods that create hyperplanes to separate data points into different classes.
Decision Trees and Ensembles	• **Decision Trees:** A tree-like model 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. • **Random Forests:** An ensemble of decision trees, where each tree is built on a random subset of the data and features. • **Gradient Boosting Machines (GBM):** An ensemble method that builds models sequentially, each correcting the errors of the previous model.
Naive Bayes	• **Naive Bayes:** A probabilistic classifier based on Bayes' theorem, assuming independence between features.
K-Nearest Neighbors (KNN)	• **K-Nearest Neighbors:** A non-parametric classification algorithm that assigns a class to a data point based on the majority class of its k nearest neighbors.
Neural Networks	• **Artificial Neural Networks (ANNs):** A computational model inspired by the human brain, consisting of interconnected nodes (neurons) organized in layers. • **Convolutional Neural Networks (CNNs):** Specialized ANNs for processing and analyzing image data. • **Recurrent Neural Networks (RNNs):** ANNs designed to handle sequential data, such as text or time series.

K-Nearest Neighbors

K-Nearest Neighbors (KNN) is a simple yet effective supervised machine learning algorithm that classifies or predicts data points based on their proximity to nearby examples in the training data.

The K-NN working can be explained on the basis of the below algorithm:

- Step 1: Select the number K of the neighbors
- Step 2: Calculate the Euclidean distance of K number of neighbors
- Step 3: Take the K nearest neighbors as per the calculated Euclidean distance.
- Step 4: Among these k neighbors, count the number of the data points in each category.
- Step 5: Assign the new data points to that category for which the number of the neighbor is maximum.
- Step 6: Our model is ready.

Key Parameters:

- **k-value:** The number of neighbors considered for classification or prediction. A smaller k-value can lead to overfitting, while a larger k-value can result in underfitting.
- **Distance metric:** The method used to calculate the distance between data points. Common metrics include Euclidean distance, Manhattan distance, and Minkowski distance.

Distance metrics	Formulas
Euclidean distance (p=2)	$d(x,y) = \sqrt{\sum_{i=1}^{n}(y_i - x_i)^2}$
Manhattan distance (p=1)	$\text{Manhattan Distance} = d(x,y) = \left(\sum_{i=1}^{m}\lvert x_i - y_i\rvert\right)$
Minkowski distance	$\text{Minkowski Distance} = \left(\sum_{i=1}^{n}\lvert x_i - y_i\rvert\right)^{1/p}$
Hamming distance	$\text{Hamming Distance} = D_H = \left(\sum_{i=1}^{k}\lvert x_i - y_i\rvert\right)$ $x=y \quad D=0$ $x\neq y \quad D\neq 1$

K-Nearest Neighbors

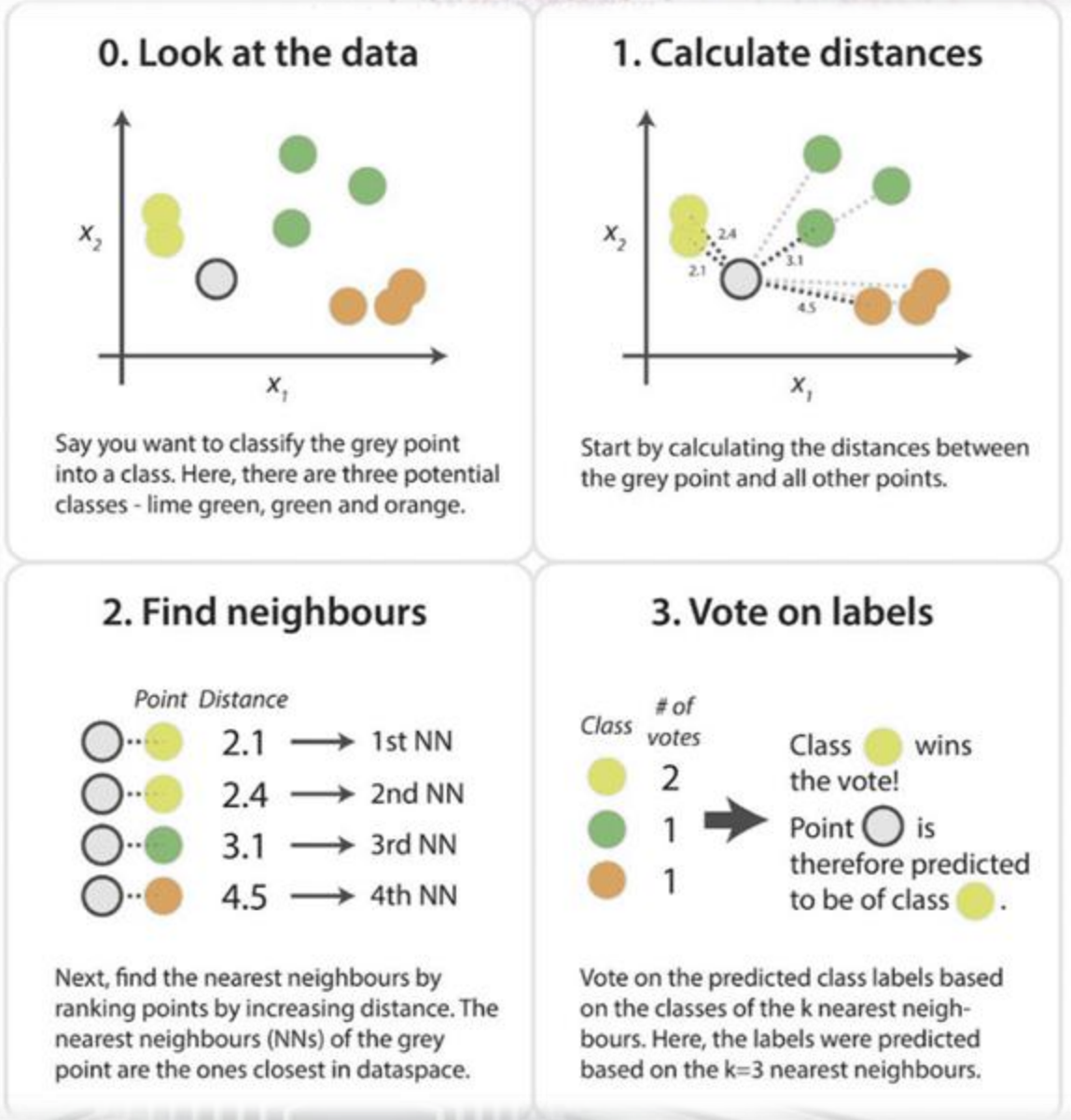
K-Nearest Neighbors (KNN) is a simple yet effective supervised machine learning algorithm that classifies or predicts data points based on their proximity to nearby examples in the training data.

The K-NN working can be explained on the basis of the below algorithm:

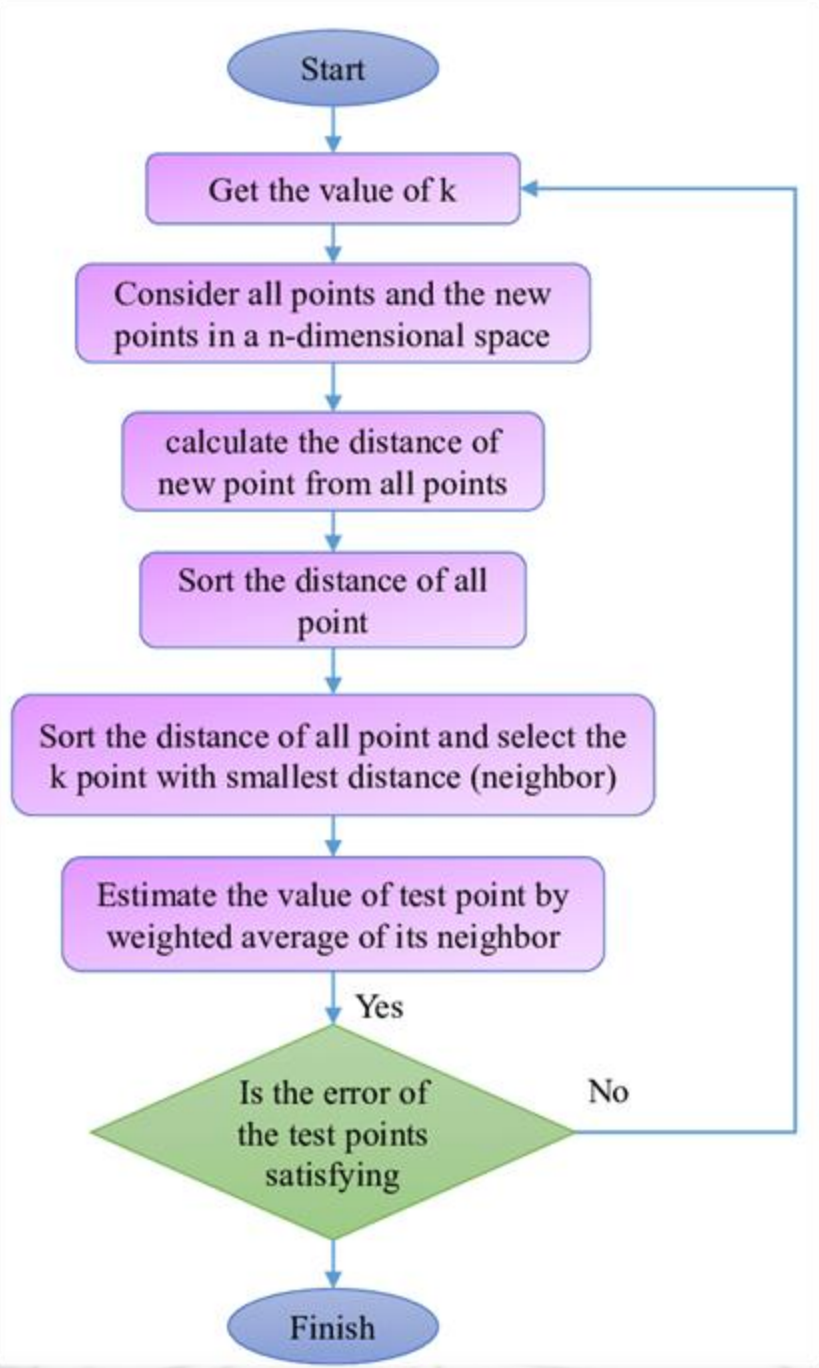
- Step 1: Select the number K of the neighbors
- Step 2: Calculate the Euclidean distance of K number of neighbors
- Step 3: Take the K nearest neighbors as per the calculated Euclidean distance.
- Step 4: Among these k neighbors, count the number of the data points in each category.
- Step 5: Assign the new data points to that category for which the number of the neighbor is maximum.
- Step 6: Our model is ready.

Key Parameters:

- **k-value:** The number of neighbors considered for classification or prediction. A smaller k-value can lead to overfitting, while a larger k-value can result in underfitting.
- **Distance metric:** The method used to calculate the distance between data points. Common metrics include Euclidean distance, Manhattan distance, and Minkowski distance.



K-Nearest Neighbors




```python
# --- 1. Define Euclidean Distance ---
def euclidean_distance(point1, point2):
    return np.sqrt(np.sum((point1 - point2)**2))

# --- 2. Implement KNN Classifier from Scratch ---
class MyKNNClassifier:
    def __init__(self, n_neighbors=5):
        self.n_neighbors = n_neighbors

    def fit(self, X, y):
        # KNN is a lazy learner, so 'fitting' just means storing the data
        self.X_train = X
        self.y_train = y

    def predict(self, X):
        predictions = [self._predict_single_sample(x) for x in X]
        return np.array(predictions)

    def _predict_single_sample(self, x):
        # Calculate distances from x to all training samples
        distances = [euclidean_distance(x, train_point) for train_point in self.X_train]

        # Get the indices of the k-nearest neighbors
        k_indices = np.argsort(distances)[:self.n_neighbors]

        # Get the labels of the k-nearest neighbors
        k_nearest_labels = [self.y_train[i] for i in k_indices]

        # Return the most common label among the k neighbors (majority vote)
        most_common = Counter(k_nearest_labels).most_common(1)
        return most_common[0][0]

```

K-Nearest Neighbors

Advantages of KNN:

- **Simplicity:** Easy to understand and implement.
- **Non-parametric:** Doesn't make assumptions about the data distribution.
- **Versatility:** Can be used for both classification and regression.
- **Accuracy:** Can achieve high accuracy for well-structured data.

Disadvantages of KNN:

- **Computational cost:** Can be slow for large datasets.
- **Sensitivity to k-value:** Choosing the right k-value is crucial.
- **Curse of dimensionality:** Can be less effective in high-dimensional spaces.
- **Sensitive to noise:** Noisy data can influence the results.

The iris dataset
→https://scikit-learn.org/stable/auto_examples/datasets/plot_iris_dataset.html#sphx-glr-auto-examples-datasets-plot-iris-dataset-py

Applications of KNN:

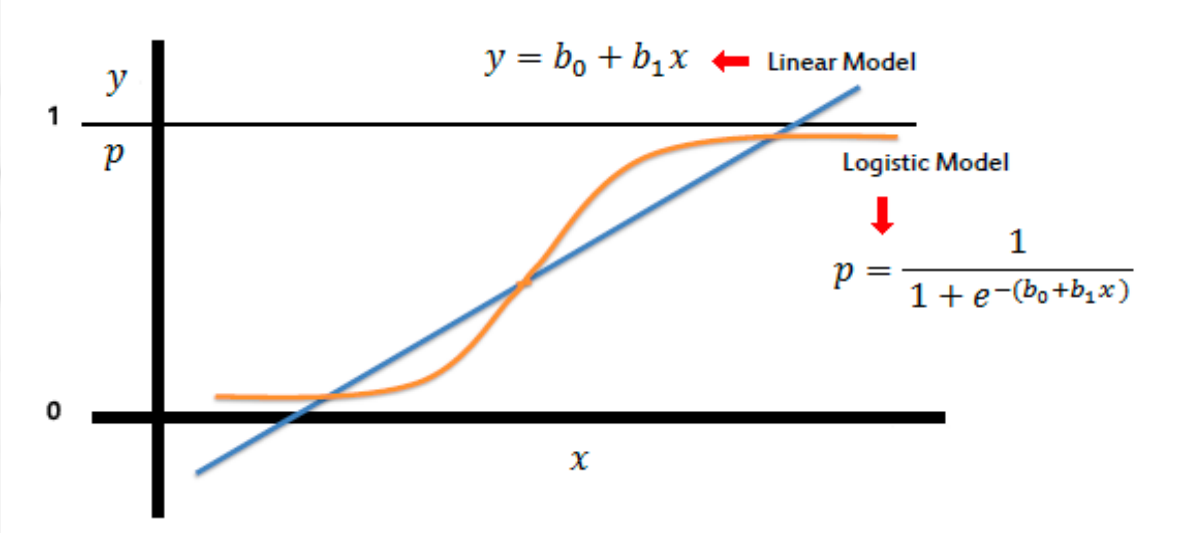
- **Image recognition:** Classifying images based on similar features.
- **Recommendation systems:** Suggesting items based on user preferences.
- **Customer segmentation:** Grouping customers based on their behavior.
- **Medical diagnosis:** Predicting diseases based on patient data.

In scikit-learn,
1- sklearn.neighbors
→ https://scikit-learn.org/stable/modules/neighbors.html#
→https://scikit-learn.org/stable/auto_examples/neighbors/plot_nearest_centroid.html

2- Example: how to use KNeighborsClassifier.
→https://scikit-learn.org/stable/auto_examples/neighbors/plot_classification.html#sphx-glr-auto-examples-neighbors-plot-classification-py

Logistic Regression

Logistic regression is a statistical model used to predict the probability of a binary outcome (e.g., yes/no, true/false, 1/0). It's a widely used technique in various fields, including machine learning, statistics, and data science.



Applications of Logistic Regression:

- Predicting customer churn: Determining whether a customer is likely to stop using a product or service.
- Credit scoring: Assessing the risk of a loan default.
- Medical diagnosis: Predicting the presence or absence of a disease.
- Email spam filtering: Identifying spam emails based on their content.
- Sentiment analysis: Determining the sentiment (positive, negative, or neutral) of a text.

In scikit-learn,

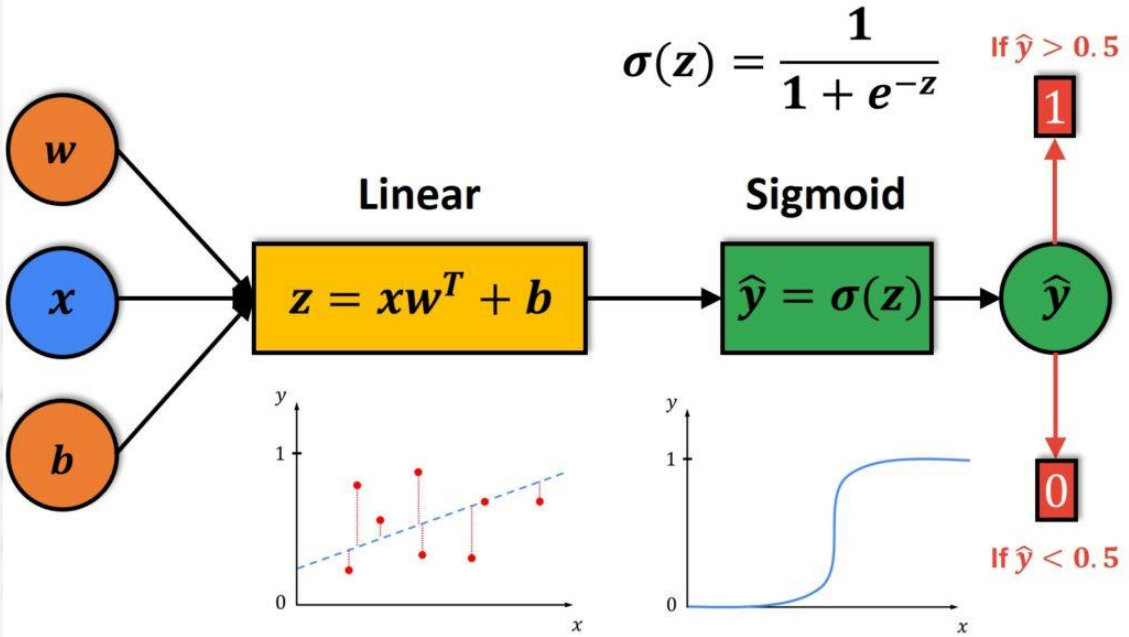
- class ***sklearn.linear_model.LogisticRegression***

→https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html#sklearn.linear_model.LogisticRegression

- **Example:** Logistic Regression 3-class Classifier

→https://scikit-learn.org/stable/auto_examples/linear_model/plot_iris_logistic.html#sphx-glr-auto-examples-linear-model-plot-iris-logistic-py

Logistic Regression




```
Input: Training data (X, y), learning rate α, number of iterations

1. Initialize weights w and bias b
2. For each iteration:
   a. Compute z = X.w + b
   b. Compute prediction y_hat = sigmoid(z)
   c. Compute loss = cross-entropy(y, y_hat)
   d. Compute gradients dw, db
   e. Update parameters: w := w - α * dw; b := b - α * db
3. Return trained weights w, b
```

How Logistic Regression Works:

- **Input:** The model takes a set of predictor variables (features) as input.
- **Linear Combination:** The model calculates a linear combination of the input features, weighted by coefficients.
- **Sigmoid Function:** The result of the linear combination is passed through a sigmoid function (also known as a logistic function). This function maps any real value to a value between 0 and 1.
- **Probability Estimation:** The output of the sigmoid function is interpreted as the probability of the positive outcome.

The Sigmoid Function:

sigmoid(x) = 1 / (1 + e^(-x))

where: x is the linear combination of the input features.
e is Euler's number (approximately 2.71828).

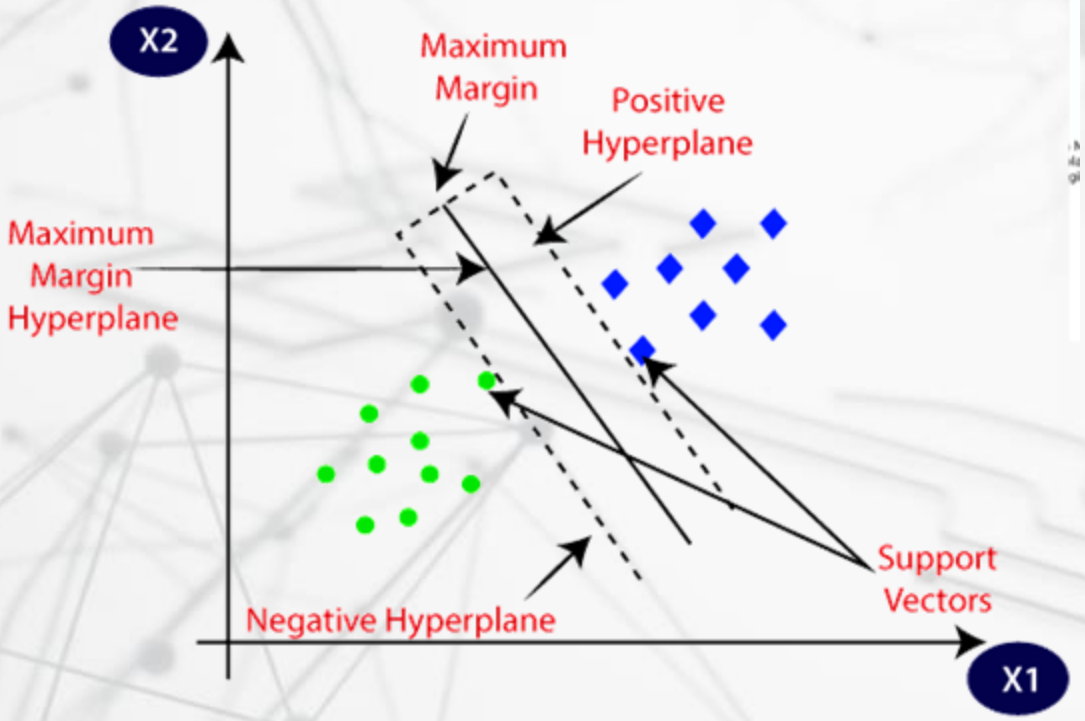
Logistic Regression

Advantages	Disadvantages
Simple and interpretable	Assumes linearity between predictors and log-odds
Computationally efficient	Limited for complex, non-linear problems
Provides probability estimates	Sensitive to outliers
Handles various data types	Requires large sample sizes
Less prone to overfitting (with regularization)	Multicollinearity issues
Good as a baseline model	Only for discrete outcomes
Widely used and understood	May underfit in high-dimensional datasets

Support Vector Machines (SVMs)

Support Vector Machines (SVMs) are a powerful machine learning algorithm used for classification and regression tasks. They are particularly effective in high-dimensional spaces and are known for their ability to handle complex decision boundaries.

SVM algorithm can be used for Face detection, image classification, text categorization, etc.



Hyperplane: In SVM, the hyperplane is the optimal decision boundary that best separates the classes. Its shape depends on the number of features: with 2 features it's a line, with 3 it's a plane, and in higher dimensions, it becomes an n-dimensional surface.

Support Vectors: The data points or vectors that are the closest to the hyperplane and which affect the position of the hyperplane are termed as Support Vector. Since these vectors support the hyperplane, hence called a Support vector.

Core Idea: SVM finds a **hyperplane** that **maximizes the margin** between two classes. The optimization goal is:

$$\min_{w,b} \frac{1}{2}||w||^2 + C\sum_{i=1}^{n} \max(0, 1 - y_i(w^T x_i + b))$$

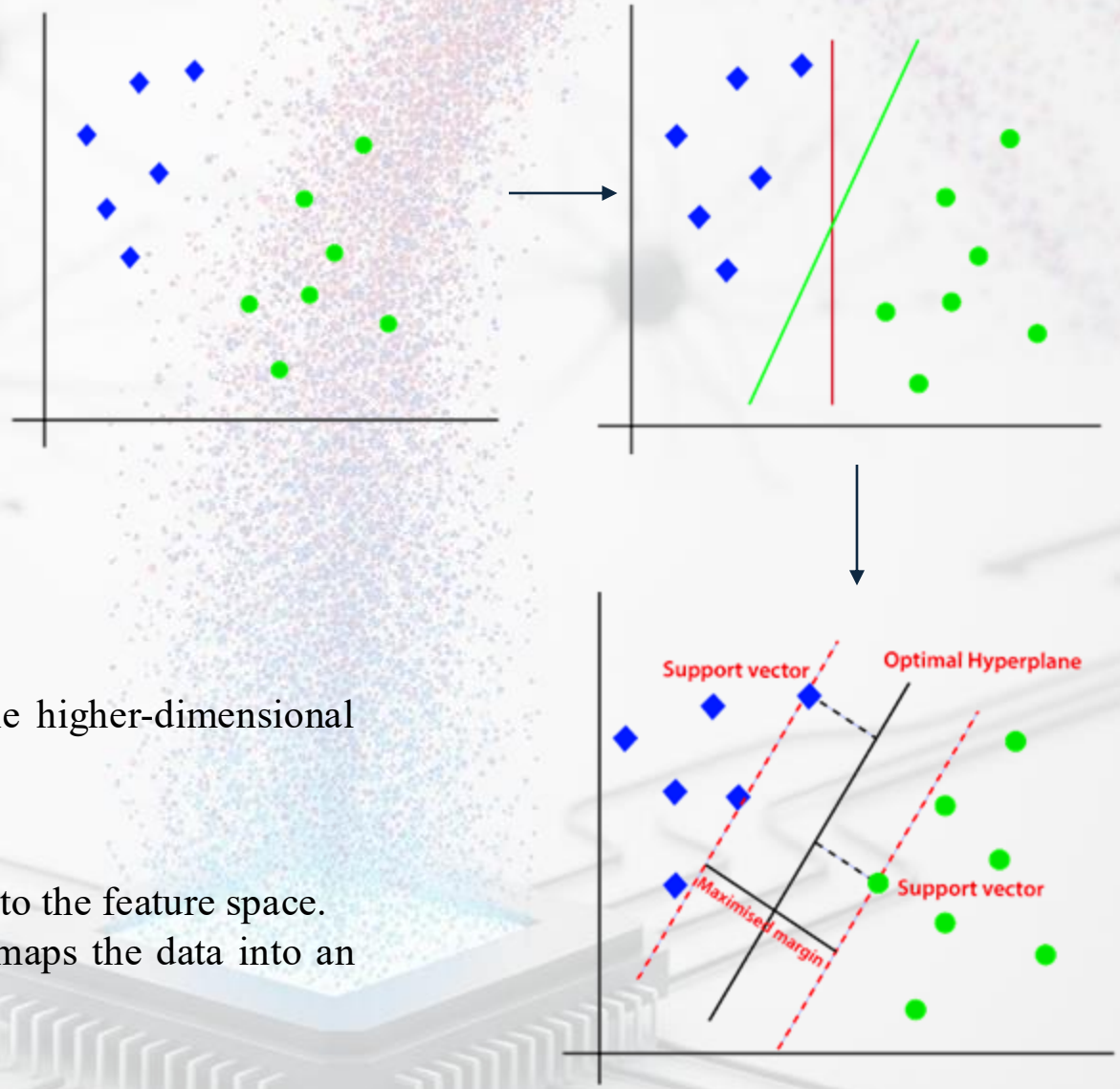
Where:

- w: weight vector
- b: bias
- C: regularization parameter (often written as $\frac{1}{\lambda}$)
- $y_i \in \{-1, +1\}$: binary class labels

Support Vector Machines (SVMs)

How SVMs Work:

1. **Feature Mapping:** The SVM maps the input data into a higher-dimensional feature space. This mapping can be linear or non-linear, depending on the kernel function used.
2. **Hyperplane Separation:** The SVM finds the optimal hyperplane (a decision boundary) that separates the data points into different classes with the maximum margin. This margin is the distance between the hyperplane and the nearest data points, known as support vectors.
3. **Classification:** New data points are classified based on which side of the hyperplane they fall on.



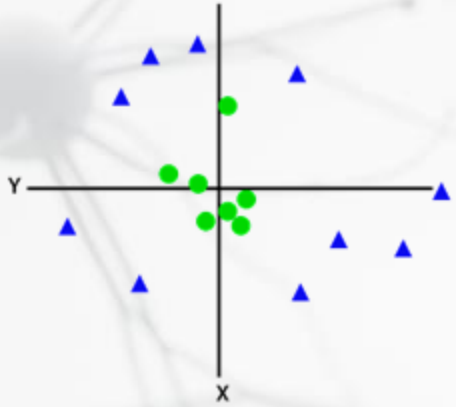
Kernel Functions:

The choice of kernel function determines the type of mapping into the higher-dimensional feature space. Common kernel functions include:

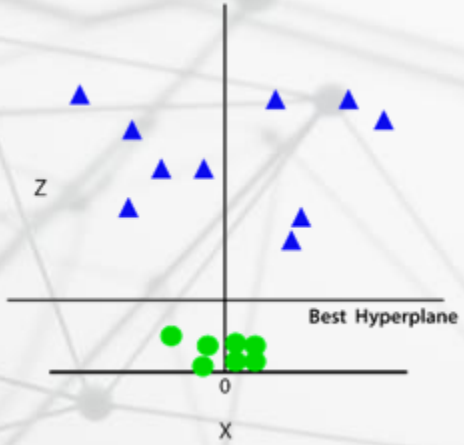
- **Linear kernel:** A simple kernel that maps the data linearly.
- **Polynomial kernel:** A kernel that introduces polynomial terms into the feature space.
- **Radial basis function (RBF) kernel:** A non-linear kernel that maps the data into an infinite-dimensional feature space.

Non-Linear SVM

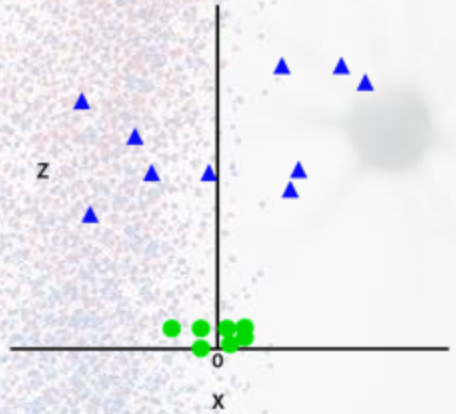
If data is linearly arranged, then we can separate it by using a straight line, but for non-linear data, we cannot draw a single straight line.



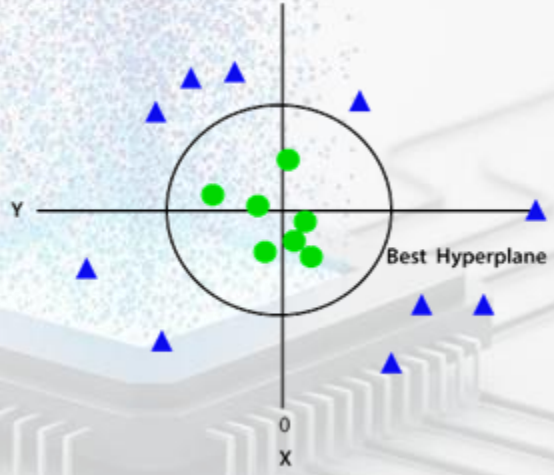
So now, SVM will divide the datasets into classes in the following way



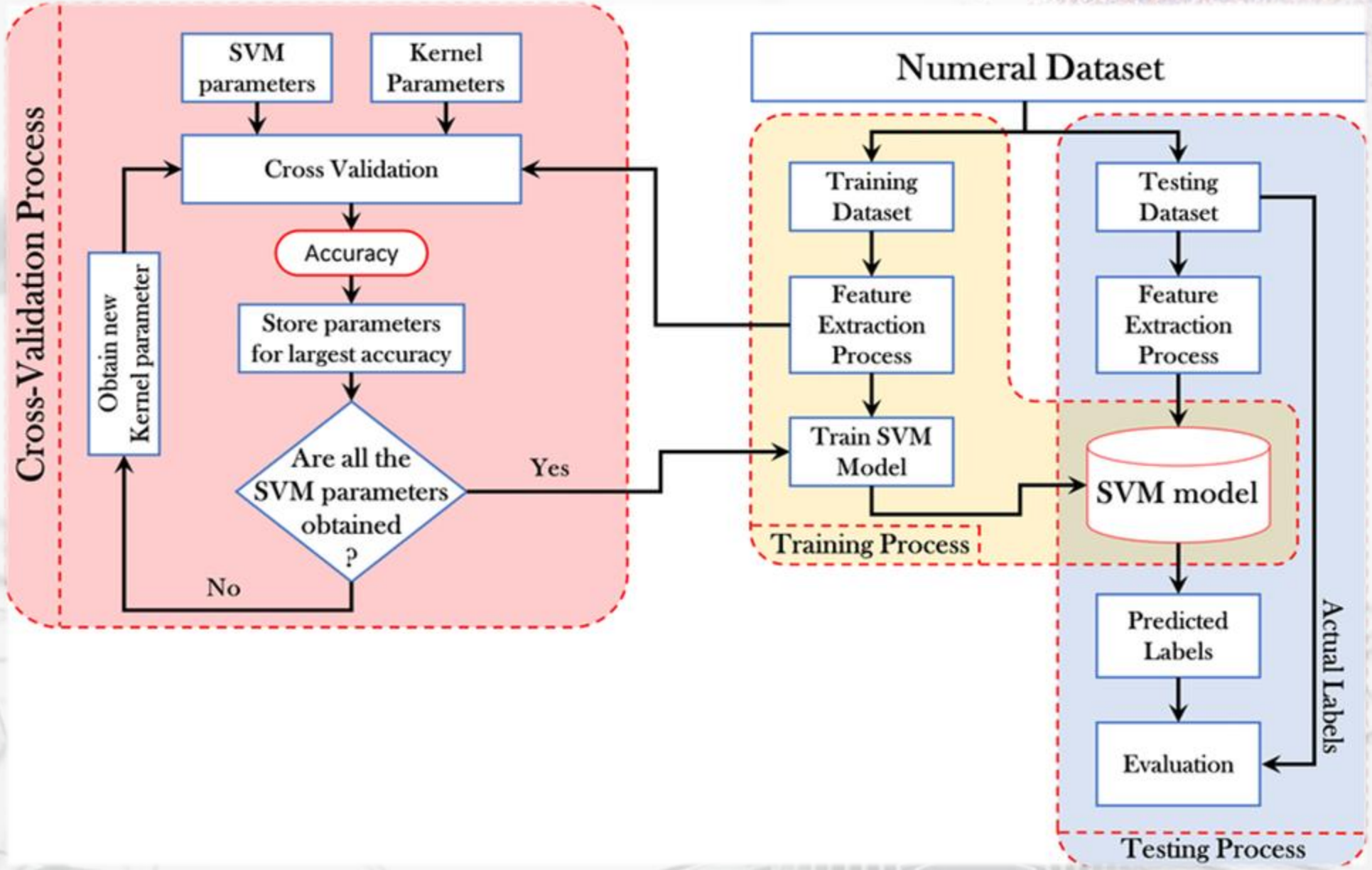
So to separate these data points, we need to add one more dimension. For linear data, we have used two dimensions x and y, so for non-linear data, we will add a third dimension z. It can be calculated as: $z=x2+y2$



Since we are in 3-d Space, hence it is looking like a plane parallel to the x-axis. If we convert it in 2d space with z=1, then it will become as



Schematic diagram for SVM



Build & Train a Linear SVM

```python
import numpy as np
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split

# Step 1: Generate simple binary data
X, y = make_blobs(n_samples=200, centers=2, random_state=42)
y = np.where(y == 0, -1, 1)  # convert to -1, +1 for SVM

# Step 2: Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Step 3: Define SVM class
class SVM:
    def __init__(self, lr=0.001, lambda_param=0.01, n_iters=1000):
        self.lr = lr
        self.lambda_param = lambda_param
        self.n_iters = n_iters
        self.w = None
        self.b = None

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.w = np.zeros(n_features)
        self.b = 0

        for _ in range(self.n_iters):
            for idx, x_i in enumerate(X):
                if y[idx] * (np.dot(x_i, self.w) + self.b) >= 1:
                    self.w -= self.lr * (2 * self.lambda_param * self.w)
                else:
                    self.w -= self.lr * (2 * self.lambda_param * self.w - y[idx] * x_i)
                    self.b -= self.lr * y[idx]

    def predict(self, X):
        return np.sign(np.dot(X, self.w) + self.b)

# Step 4: Train and evaluate
model = SVM()
model.fit(X_train, y_train)

predictions = model.predict(X_test)
accuracy = np.mean(predictions == y_test)

print(f"Test Accuracy: {accuracy:.2%}")
```

In Scikit-learn, SVMs for classification - **SVC**, **NuSVC** and **LinearSVC**

→ https://scikit-learn.org/stable/modules/svm.html#

Example:

1. Plot different SVM classifiers in the iris dataset

→https://scikit-learn.org/stable/auto_examples/svm/plot_iris_svc.html#sphx-glr-auto-examples-svm-plot-iris-svc-py

2. SVM with custom kernel

→https://scikit-learn.org/stable/auto_examples/svm/plot_custom_kernel.html#sphx-glr-auto-examples-svm-plot-custom-kernel-py

3. RBF SVM parameters

→https://scikit-learn.org/stable/auto_examples/svm/plot_rbf_parameters.html

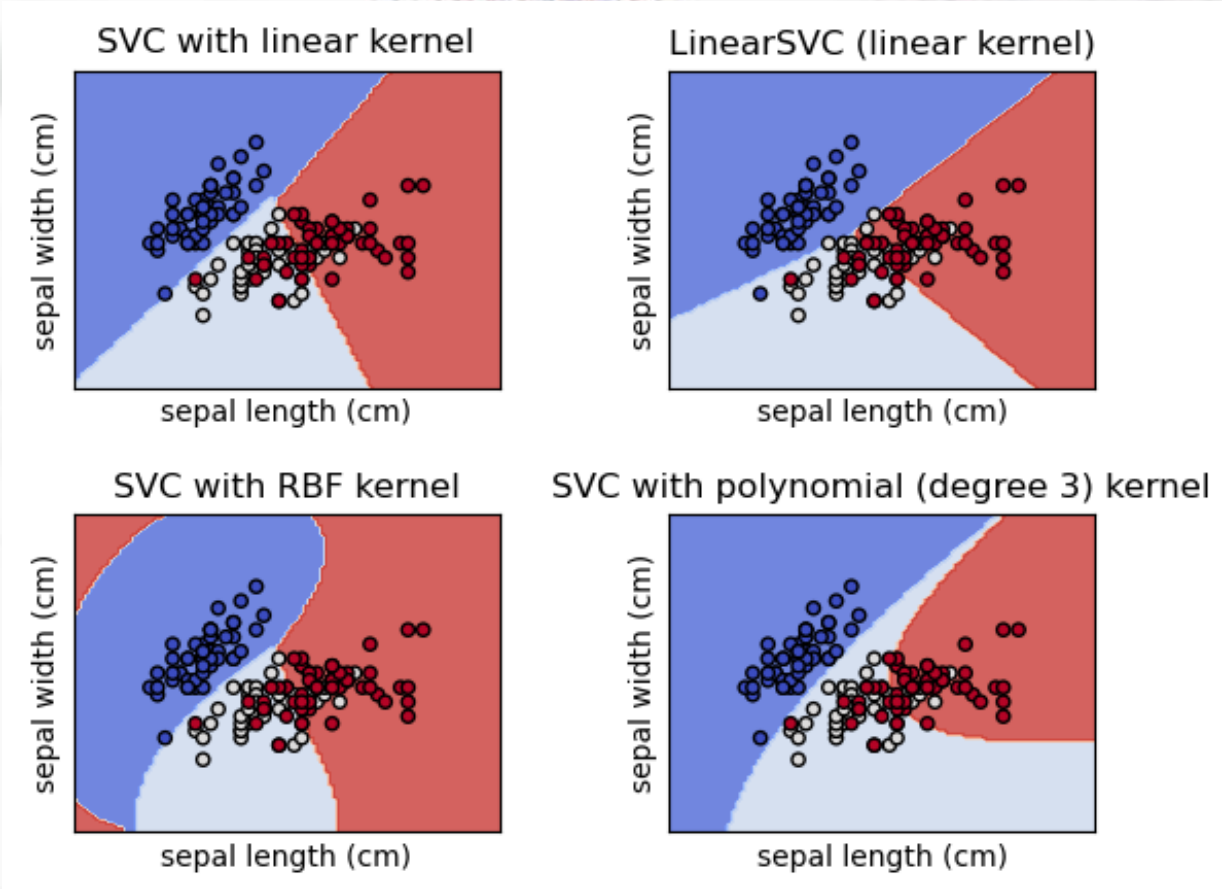
Support Vector Machines (SVMs)

Advantages of SVMs:

- **Effective in high-dimensional spaces:** SVMs can handle complex decision boundaries in high-dimensional data.
- **Robust to outliers:** SVMs are less sensitive to outliers due to their focus on the support vectors.
- **Versatile:** SVMs can be used for both classification and regression tasks.
- **Efficient:** SVMs can be efficient for large datasets, especially when using kernel tricks.

Disadvantages of SVMs:

- **Computational complexity:** Training SVMs can be computationally expensive for large datasets, especially with non-linear kernels.
- **Choice of kernel:** Selecting the appropriate kernel function can be challenging.
- **Sensitivity to hyperparameters:** SVMs have hyperparameters (like the kernel function and regularization parameter) that need to be tuned for optimal performance.



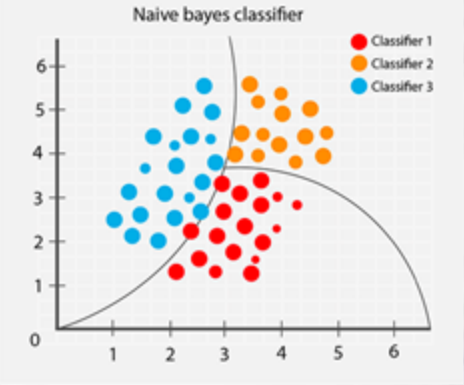
Naive Bayes Classification

Naive Bayes is a probabilistic classification algorithm based on Bayes' theorem, which assumes that features are independent given the class. While this independence assumption is often violated in real-world data, Naive Bayes can still perform surprisingly well in many cases.

$$P(A|B) = \frac{P(B|A)\,P(A)}{P(B)}$$

using Bayesian probability terminology, the above equation can be written as

$$\text{Posterior} = \frac{\text{prior x likelihood}}{\text{evidence}}$$



How Naive Bayes Works

1. **Calculate Probabilities:**
- **Prior probability:** The probability of each class occurring independently of the features.
- **Conditional probability:** The probability of a feature occurring given a particular class.
1. **Apply Bayes' Theorem:**
- Using Bayes' theorem, calculate the posterior probability of each class given the observed features.
- The class with the highest posterior probability is predicted as the most likely class.

Whether	Play
Sunny	No
Sunny	No
Overcast	Yes
Rainy	Yes
Rainy	Yes
Rainy	No
Overcast	Yes
Sunny	No
Sunny	Yes
Rainy	Yes
Sunny	Yes
Overcast	Yes
Overcast	Yes
Rainy	No

Frequency Table

Whether	No	Yes
Overcast		4
Sunny	2	3
Rainy	3	2
Total	5	9

Likelihood Table 1

Whether	No	Yes		
Overcast		4	=4/14	0.29
Sunny	2	3	=5/14	0.36
Rainy	3	2	=5/14	0.36
Total	5	9		
	=5/14	=9/14		
	0.36	0.64		

Likelihood Table 2

Whether	No	Yes	Posterior Probability for No	Posterior Probability for Yes
Overcast		4	0/5=0	4/9=0.44
Sunny	2	3	2/5=0.4	3/9=0.33
Rainy	3	2	3/5=0.6	2/9=0.22
Total	5	9		

The primary types of Naive Bayes classifiers

01	Gaussian Naive Bayes	• Assumes features are normally distributed. • Suitable for continuous numerical data. • Calculates conditional probabilities based on the mean and standard deviation of each feature within each class.
02	Multinomial Naive Bayes	• Designed for count data (e.g., word frequencies in text documents). • Assumes features follow a multinomial distribution. • Calculates conditional probabilities based on the frequency of each feature within each class.
03	Bernoulli Naive Bayes	• Suitable for binary features (e.g., presence or absence of a word in a document). • Assumes features follow a Bernoulli distribution. • Calculates conditional probabilities based on the probability of a feature being present or absent within each class.

Naive Bayes Classification

Advantages of Naive Bayes

- **Simplicity:** Easy to implement and understand.
- **Efficiency:** Can handle large datasets efficiently.
- **Robustness:** Can perform well even with noisy or missing data.

Disadvantages of Naive Bayes

- **Independence Assumption:** The assumption of feature independence can be violated in many real-world scenarios.
- **Sensitivity to Zero Counts:** If a feature-class combination has zero occurrences in the training data, the conditional probability becomes zero, leading to an incorrect prediction.

Applications of Naive Bayes

- **Text classification:** Spam filtering, sentiment analysis, topic modeling
- **Recommendation systems:** Suggesting items based on user preferences
- **Medical diagnosis:** Predicting diseases based on symptoms
- **Weather prediction:** Forecasting weather conditions

In scikit-learn,

→ https://scikit-learn.org/1.5/modules/naive_bayes.html

→ **example:** https://scikit-learn.org/1.5/auto_examples/classification/plot_classifier_comparison.html

Decision Tree Classification

Decision Trees are a popular machine learning algorithm often used for both classification and regression tasks.

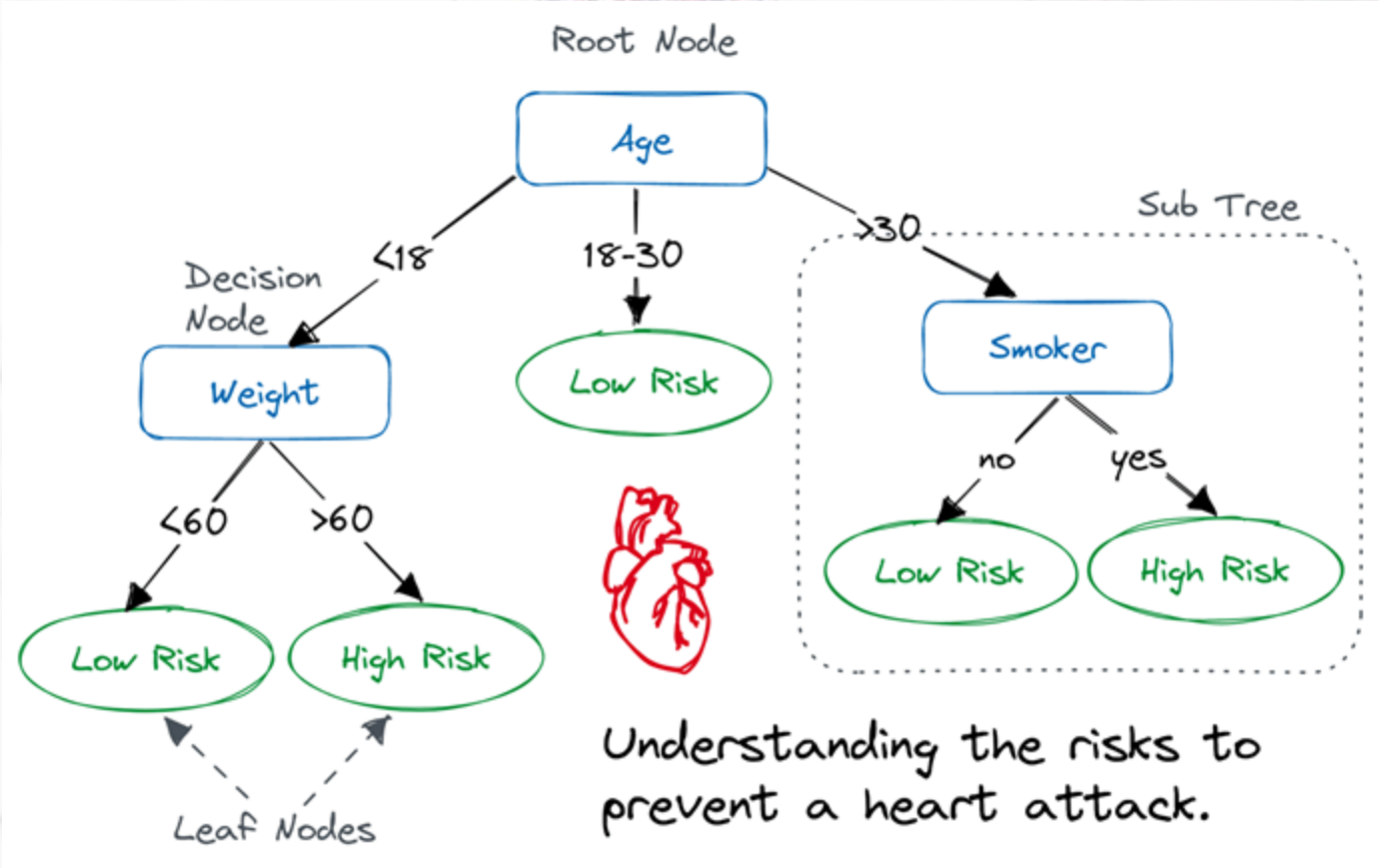
In the context of classification, they create a tree-like model where each internal node represents a test on an attribute (e.g., "Is age greater than 30?"), each branch represents the possible outcomes of the test, and each leaf node represents a class label.

The decision tree is a distribution-free or non-parametric method which does not depend upon probability distribution assumptions.

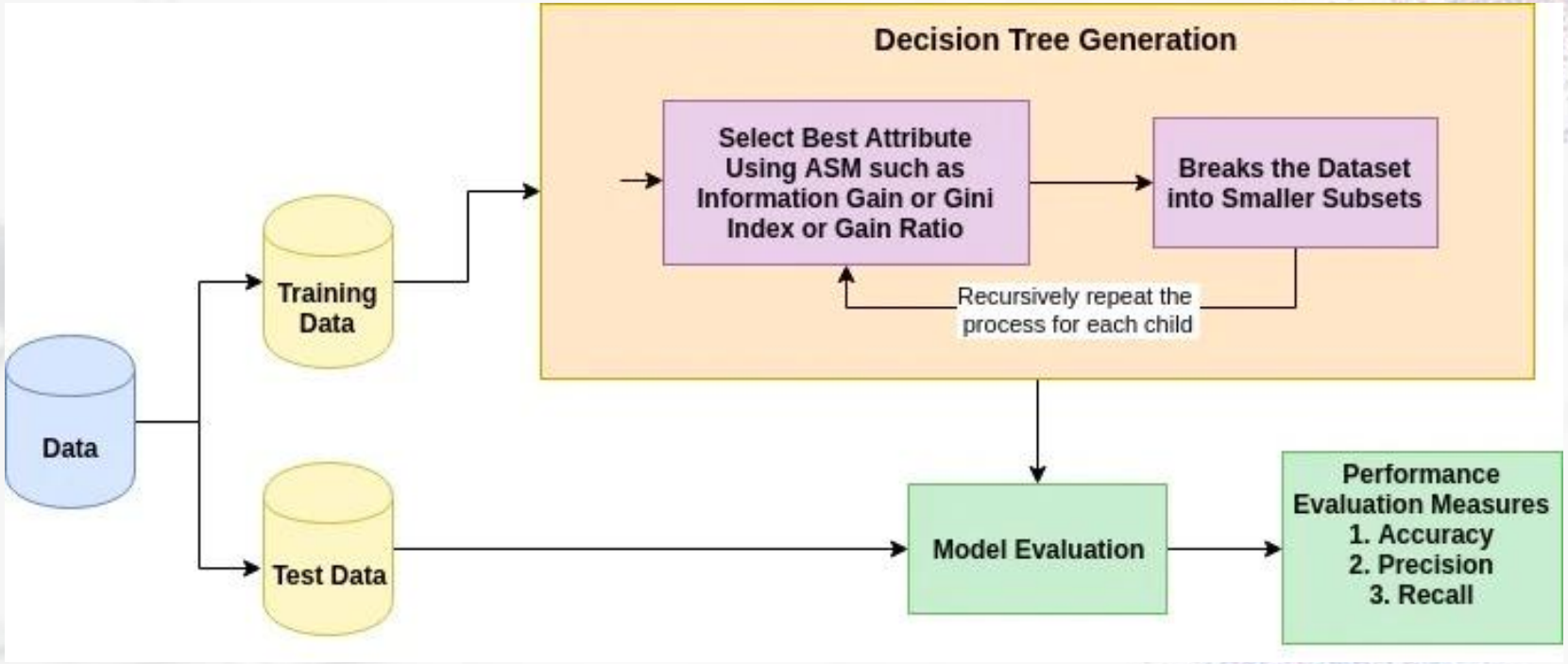
Decision trees can handle high-dimensional data with good accuracy.

Practice:

- → https://www.datacamp.com/tutorial/decision-tree-classification-python



Decision Tree Algorithm



The basic idea behind any decision tree algorithm is as follows:

1. Select the best attribute using Attribute Selection Measures (ASM) to split the records.
2. Make that attribute a decision node and breaks the dataset into smaller subsets.
3. Start tree building by repeating this process recursively for each child until one of the conditions will match:
    - All the tuples belong to the same attribute value.
    - There are no more remaining attributes.
    - There are no more instances.

Random Forest Classification

Random Forest is a powerful ensemble learning method that combines multiple decision trees to make predictions. It's particularly effective for classification tasks due to its ability to handle large datasets, reduce overfitting, and provide feature importance.

How Does Random Forest Work?

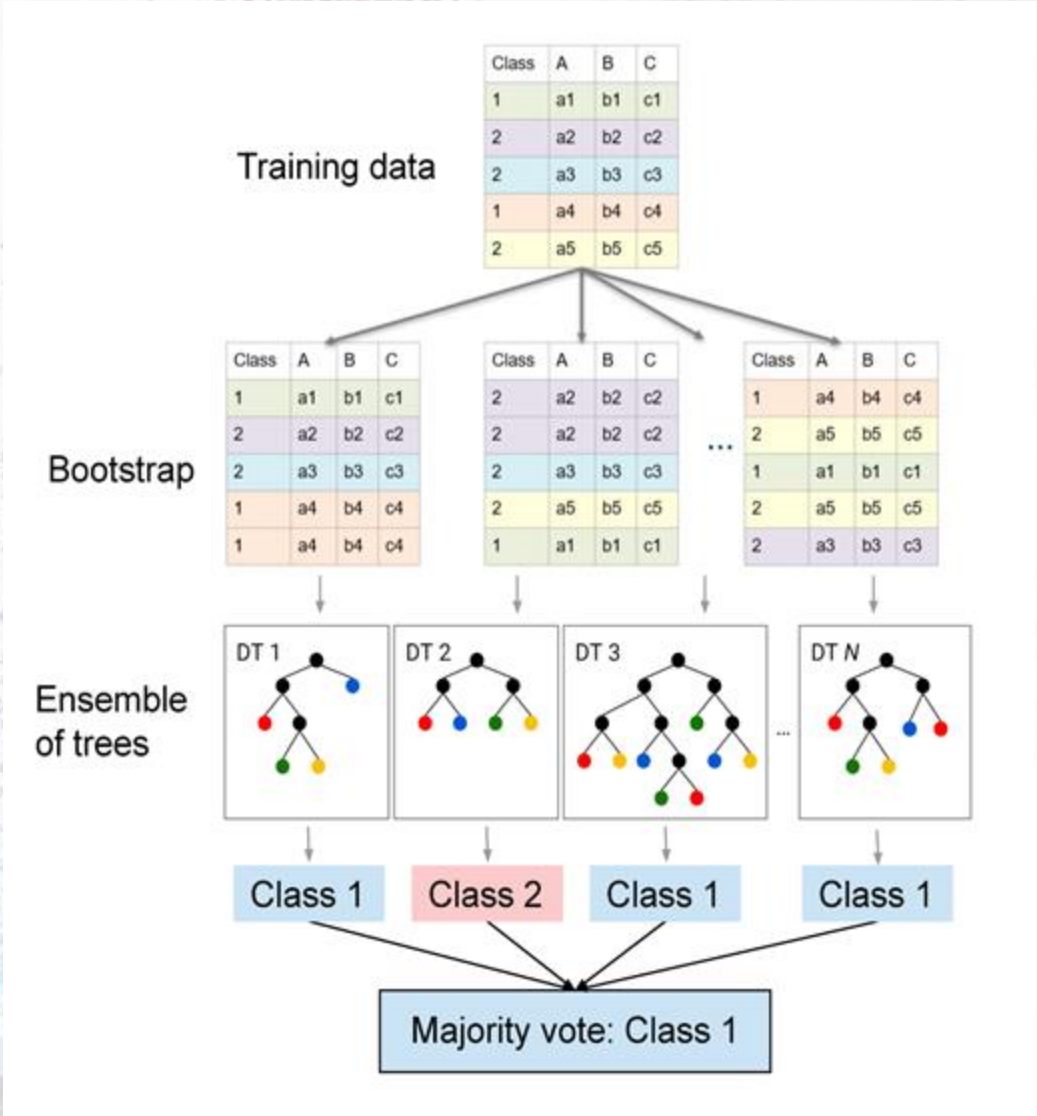
- **Bootstrap Sampling:** The algorithm randomly selects subsets of data from the original dataset with replacement. This creates multiple "bootstrap samples."
- **Decision Tree Growth:** Each bootstrap sample is used to grow a decision tree. The trees are grown to their maximum size without pruning.
- **Prediction:** To make a prediction for a new data point, the algorithm passes it through each decision tree and collects the predicted class from each.
- **Voting:** The most frequent class among all the predictions from the individual trees becomes the final prediction.

Key Parameters in Random Forest

- **Number of Trees:** The number of decision trees in the forest.
- **Maximum Depth:** The maximum depth of each decision tree.
- **Minimum Samples Split:** The minimum number of samples required to split an internal node.
- **Minimum Samples Leaf:** The minimum number of samples required to be at a leaf node.
- **Bootstrap:** Whether bootstrap sampling is used.

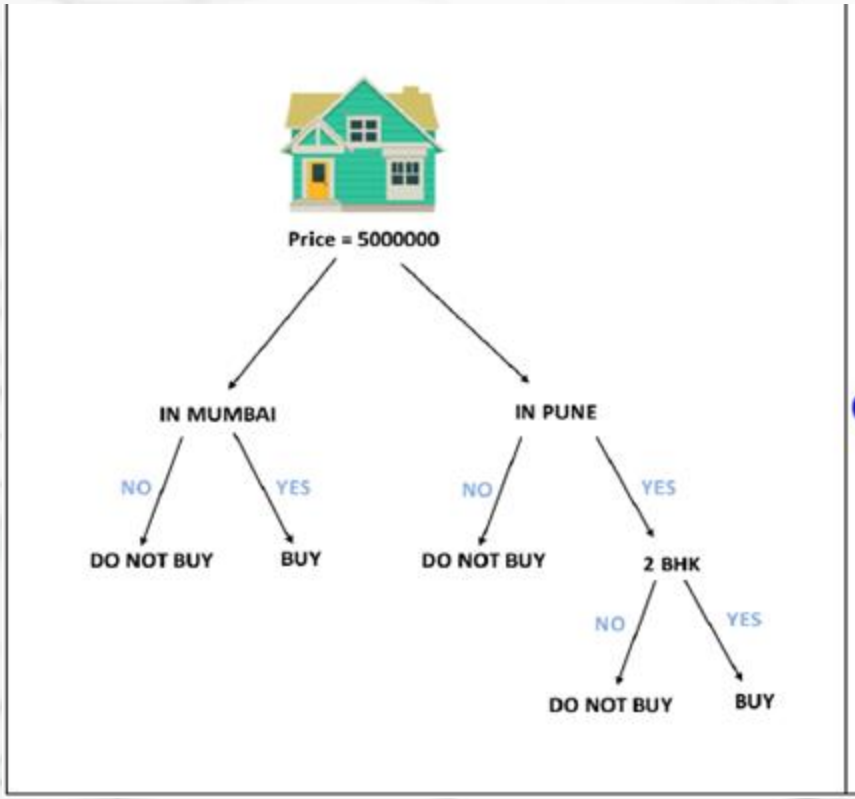
Practice:

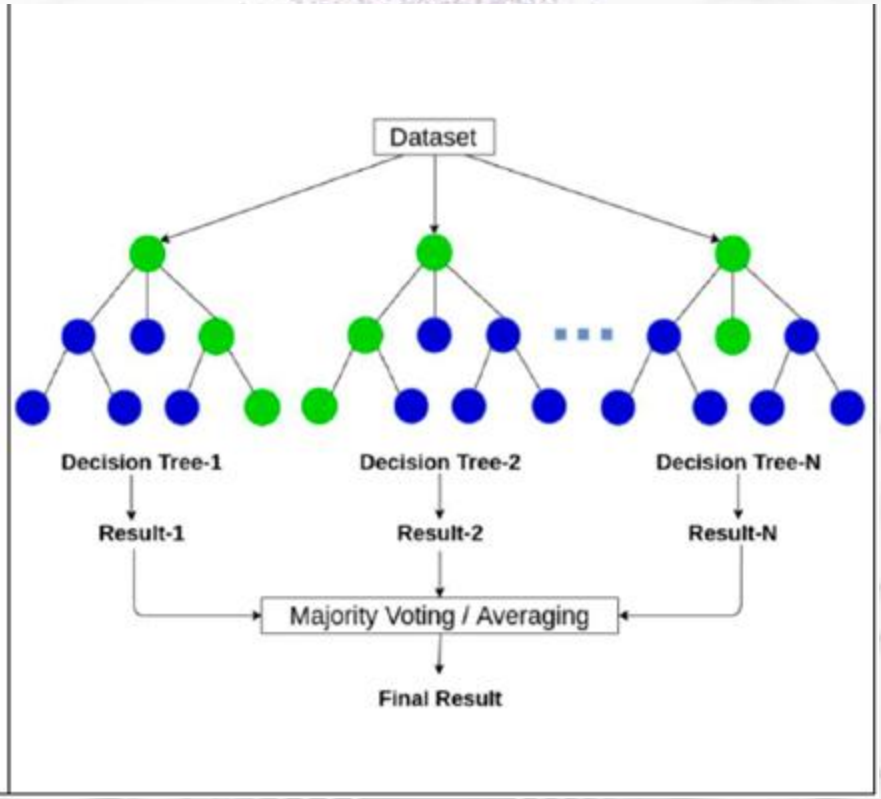
→ https://www.datacamp.com/tutorial/random-forests-classifier-python



Difference Between Random Forest and Decision Tree

Feature	Decision Tree	Random Forest
Structure	Single tree	Ensemble of trees
Overfitting	Prone to overfitting	Less prone to overfitting
Accuracy	Generally less accurate	Generally more accurate
Interpretability	Highly interpretable	Less interpretable
Computational cost	Relatively low	Can be computationally expensive



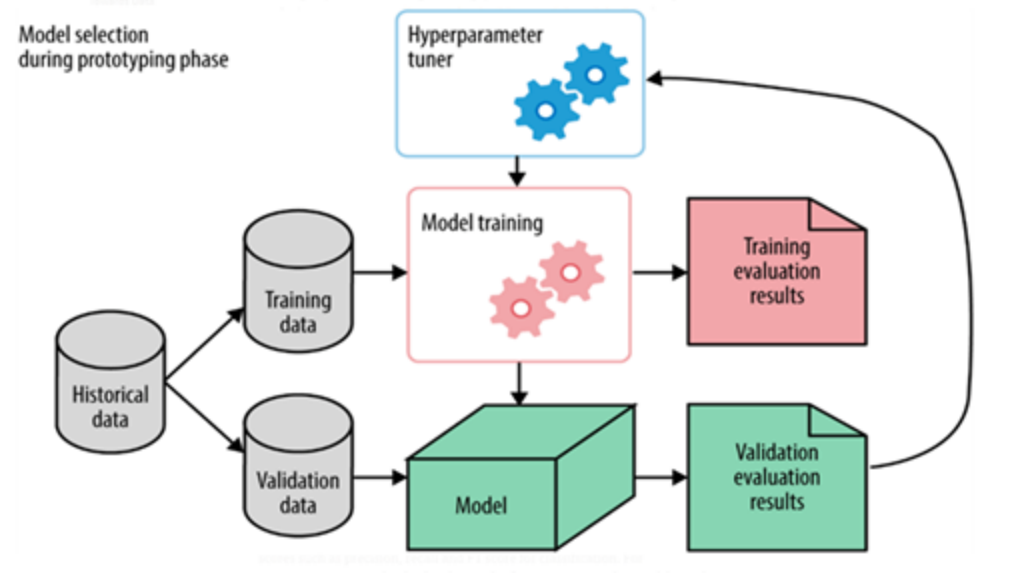


Hyper-parameter Tuning

Hyperparameters are settings that are not learned from the data but are set before training a machine learning model. They control the behavior of the learning algorithm. Tuning these hyperparameters can significantly impact a model's performance.

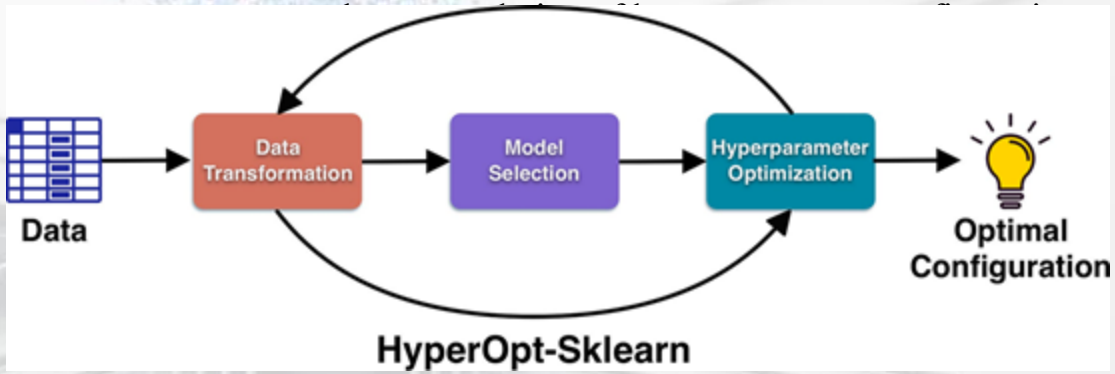
Common Hyperparameters:

- **Learning rate:** Controls how quickly the model adjusts its parameters during training.
- **Regularization strength:** Controls the amount of regularization applied to the model (e.g., L1 or L2 regularization).
- **Number of hidden layers and neurons:** Determines the complexity of a neural network.
- **Batch size:** The number of samples used in each training iteration.
- **Epochs:** The number of times the entire dataset is passed through the model during training.



Hyperparameter Tuning Techniques:

1. **Grid Search:**
   - Defines a grid of hyperparameter values and trains a model for each combination.
   - Time-consuming for large search spaces.
2. **Random Search:**
   - Randomly samples hyperparameter values from a specified distribution.
   - Often more efficient than grid search for large search spaces.
3. **Bayesian Optimization:**
   - Uses a probabilistic model to build a surrogate function of the objective function.
   - Explores the search space more intelligently by focusing on promising regions.
4. **Evolutionary Algorithms:**
   - Inspired by biological evolution, these algorithms use concepts like selection, crossover, and mutation to



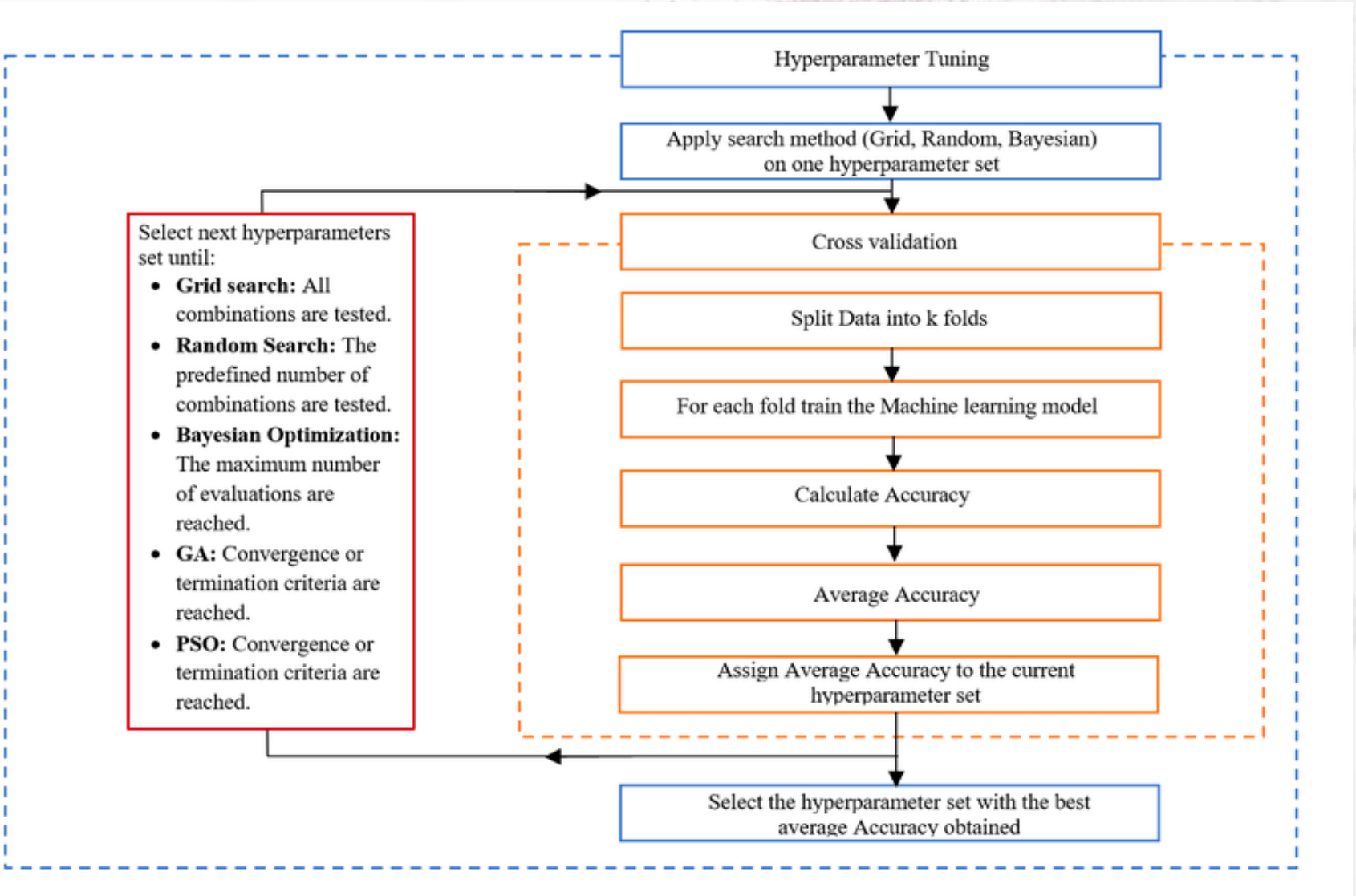
Hyperparameter Tuning Techniques

Hyperparameter Tuning Techniques with Scikit-learn

1. Grid Search
2. Random Search
3. Bayesian Optimization
4. Evolutionary Algorithms

Reference:

1) https://scikit-learn.org/1.5/model_selection.html
2) https://www.geeksforgeeks.org/sklearn-model-hyper-parameters-tuning/



Objective Function in Regression

The objective function in regression is a mathematical expression that quantifies the "error" or "distance" between the predicted values and the actual values. It serves as a target that the regression model aims to minimize.

The choice of objective function depends on the specific characteristics of the regression problem and the desired properties of the model. Consider the following factors:

- **Sensitivity to outliers:** If your data contains outliers, MAE or Huber Loss might be more suitable than MSE.
- **Interpretability:** RMSE is often preferred for interpretability as it is in the same units as the target variable.
- **Optimization:** MSE is generally easier to optimize due to its differentiability.

Common Objective Functions:

1. **Mean Squared Error (MSE):**

- Calculates the average squared difference between predicted and actual values.
- Formula: MSE = 1/n * Σ(yi - ŷi)^2
- Advantages: Easy to compute, differentiable, and widely used.
- Disadvantages: Sensitive to outliers due to squaring.

2. Mean Absolute Error (MAE):

- Calculates the average absolute difference between predicted and actual values.
- Formula: MAE = 1/n * Σ|yi - ŷi|
- Advantages: Less sensitive to outliers than MSE.
- Disadvantages: Not differentiable at zero, making optimization more challenging.

3. Root Mean Squared Error (RMSE):

- The square root of the MSE.
- Formula: RMSE = √(1/n * Σ(yi - ŷi)^2)
- Advantages: Interpretable in the same units as the target variable.
- Disadvantages: Same as MSE regarding sensitivity to outliers.

4. Huber Loss:

- Combines the advantages of MSE and MAE by using a quadratic loss for small errors and a linear loss for large errors.
- Formula:
  Huber Loss = 1/n * Σ(δ^2 * (yi - ŷi)^2 / 2 , if |yi - ŷi| ≤ δ,
  Huber Loss = |yi - ŷi| - δ^2 / 2, otherwise
- Advantages: Robust to outliers while maintaining differentiability.

Overfitting and Underfitting in Machine Learning

Overfitting

Overfitting occurs when a model learns the training data too well, capturing noise and irrelevant patterns. This leads to a model that performs exceptionally well on the training data but poorly on new, unseen data.

Characteristics of Overfitting:

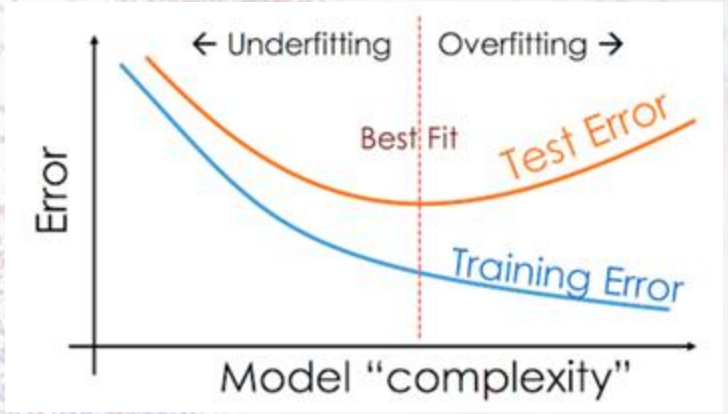
- High performance on training data: The model achieves very high accuracy or low error on the training set.
- Poor performance on validation/test data: The model's performance significantly drops when evaluated on unseen data.
- Complex model: The model may have too many parameters or features, making it prone to memorizing the training data instead of learning underlying patterns.

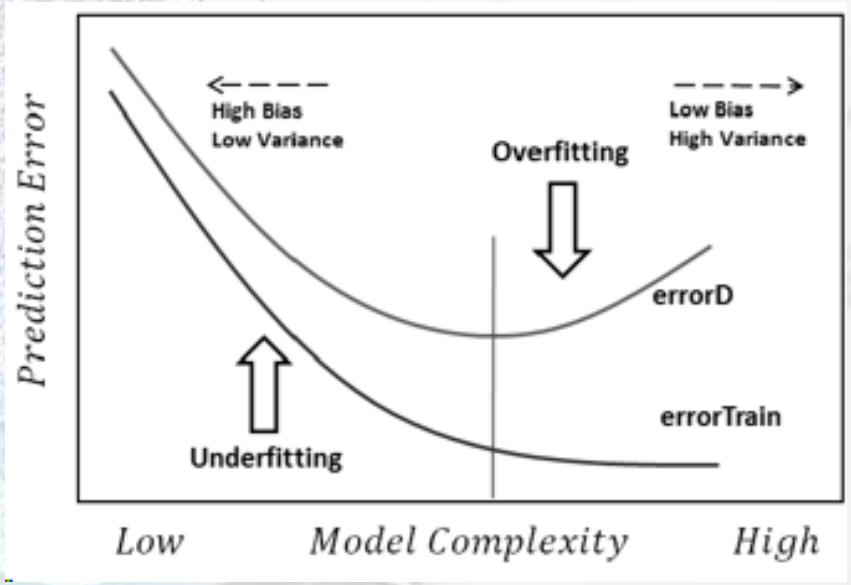
Underfitting

Underfitting happens when a model is too simple to capture the underlying patterns in the data. This results in a model that performs poorly on both the training and validation/test data.

Characteristics of Underfitting:

- Poor performance on both training and validation/test data: The model consistently achieves low accuracy or high error on both sets.
- Simple model: The model may have too few parameters or features, limiting its ability to learn complex relationships.





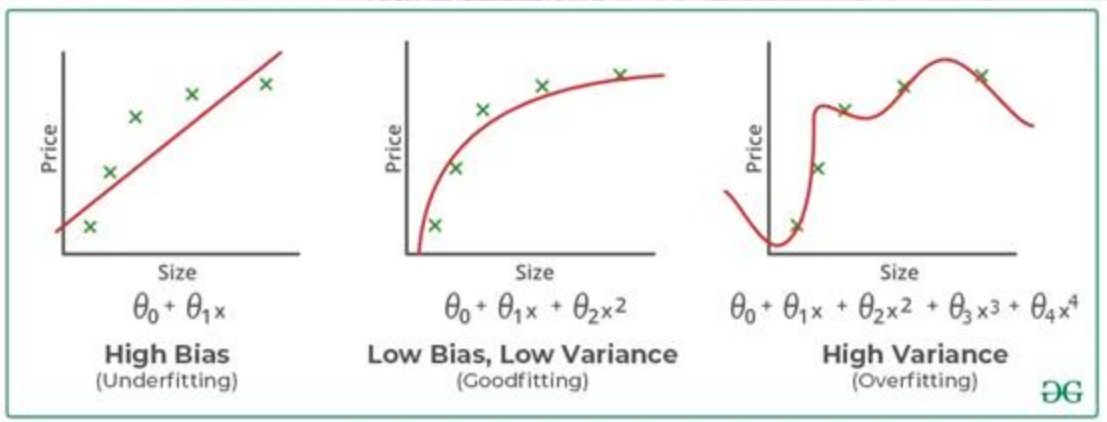
Addressing Overfitting and Underfitting

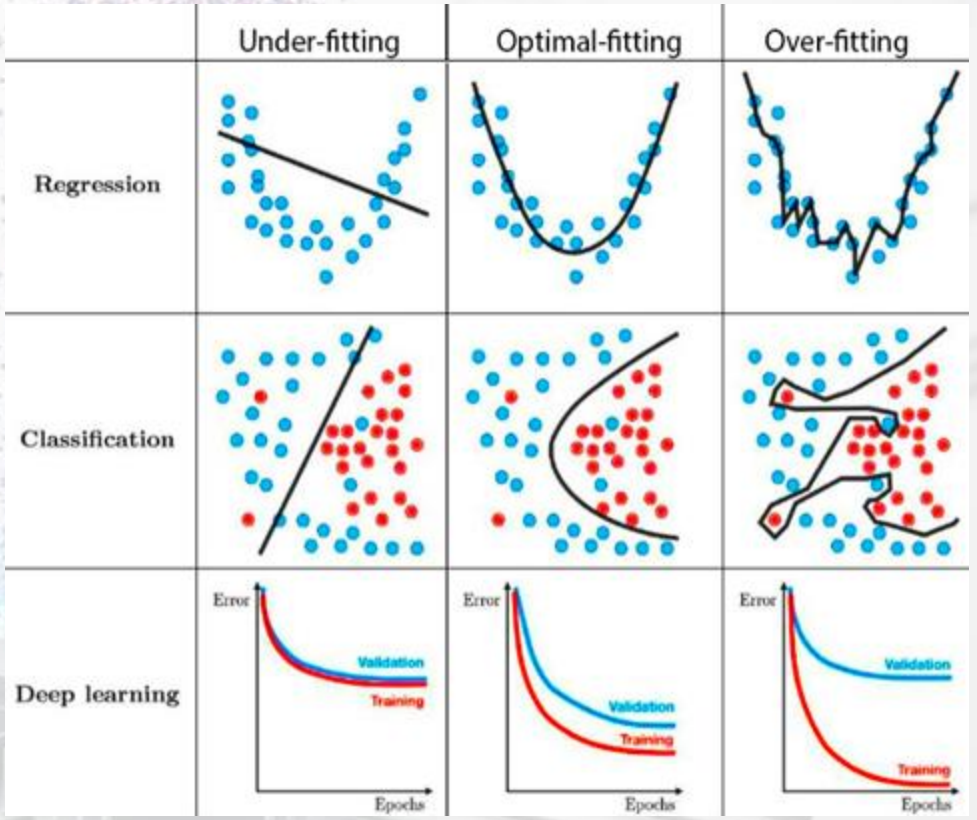
Addressing Overfitting and Underfitting

1. **Regularization:**

- **L1 regularization (Lasso):** Introduces a penalty term that encourages sparsity, meaning many model parameters become zero. This can help prevent overfitting by reducing the complexity of the model.
- **L2 regularization (Ridge):** Adds a penalty term that discourages large parameter values, preventing the model from becoming too sensitive to small changes in the input data.

1. **Cross-validation:** Splits the data into multiple folds and trains the model on different subsets to evaluate its performance on unseen data. This helps identify overfitting or underfitting early in the modeling process.
2. **Early stopping:** Monitors the model's performance on a validation set during training and stops the training process when performance starts to deteriorate, preventing overfitting.
3. **Feature engineering:** Creating new features or transforming existing ones can improve the model's ability to capture relevant patterns and reduce overfitting.
4. **Simpler model:** If the model is overfitting, consider using a simpler model with fewer parameters.



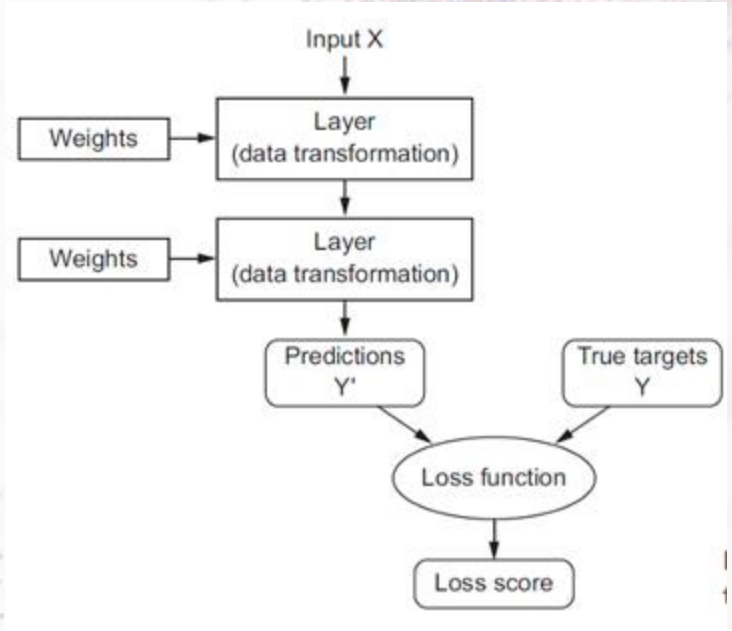


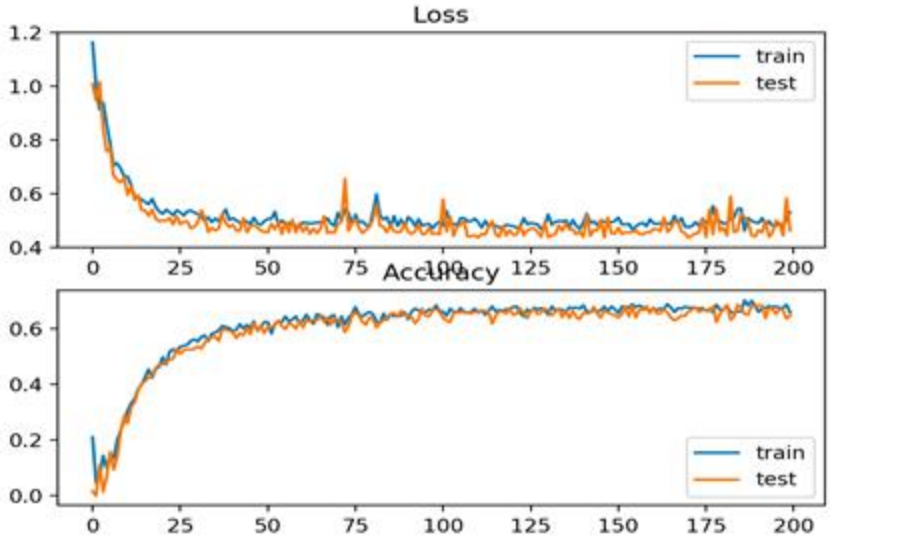
Loss Function in Machine Learning

A **loss function**, also known as a cost function or objective function, is a mathematical function that quantifies the "error" or "distance" between the predicted values and the actual values in a machine learning model. It serves as a target that the model aims to minimize during the training process.

Commonly-used loss functions in machine learning

Task	Error type	Loss function	Note
Regression	Mean-squared error	$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$	Easy to learn but sensitive to outliers (MSE, L2 loss)
	Mean absolute error	$\frac{1}{n}\sum_{i=1}^{n}\lvert y_i - \hat{y}_i\rvert$	Robust to outliers but not differentiable (MAE, L1 loss)
Classification	Cross entropy = Log loss	$-\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)] =$	Quantify the difference between two probability





Choosing the Right Loss Function:

The choice of loss function depends on the specific characteristics of the problem and the desired properties of the model. Consider the following factors:

- Type of problem: Regression or classification?
- Sensitivity to outliers: Is the data noisy or prone to outliers?
- Interpretability: Do you need the error to be in the same units as the target variable?

Regularized Loss Minimization

Regularized loss minimization is a technique used in machine learning to prevent overfitting by adding a penalty term to the loss function. This penalty term is designed to discourage overly complex models, which are more prone to overfitting.

Loss Function

A loss function quantifies the "error" or "distance" between the predicted values and the actual values. Common loss functions include:

- Mean Squared Error (MSE)
- Mean Absolute Error (MAE)
- Cross-Entropy Loss

Regularization Term

The regularization term is added to the loss function to penalize large parameter values. The two most common types of regularization are:

1. **L1 Regularization (Lasso):**
   - Adds the absolute value of the parameters to the loss function.
   - Encourages sparsity, meaning many parameters become zero.
   - Can be useful for feature selection.
2. **L2 Regularization (Ridge):**
   - Adds the square of the parameters to the loss function.
   - Discourages large parameter values.
   - Can help prevent overfitting by reducing the variance of the model.

Overall Objective. The goal of regularized loss minimization is to minimize the following objective function:

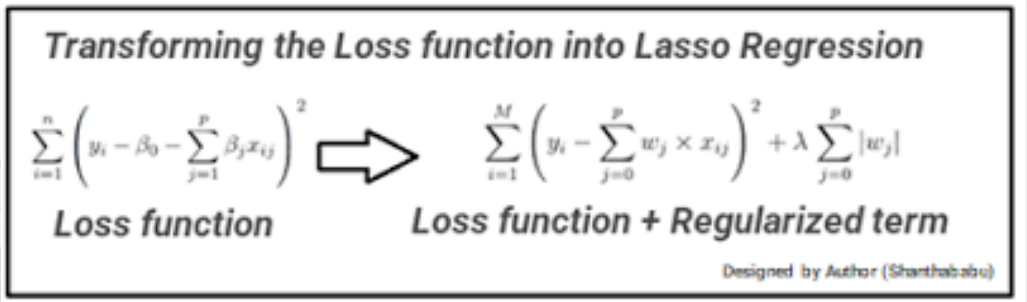
$$\text{Loss}(w) + \lambda * R(w)$$

where:

- $\text{Loss}(w)$ is the loss function.
- λ is the regularization parameter, which controls the strength of the regularization.
- $R(w)$ is the regularization term (L1 or L2).

Benefits of Regularized Loss Minimization

- **Prevents overfitting:** By penalizing large parameter values, regularization can help prevent models from becoming too complex and memorizing the training data.
- **Improves generalization:** Regularized models tend to generalize better to unseen data.
- **Feature selection:** L1 regularization can be used for feature selection by setting many parameters to zero.



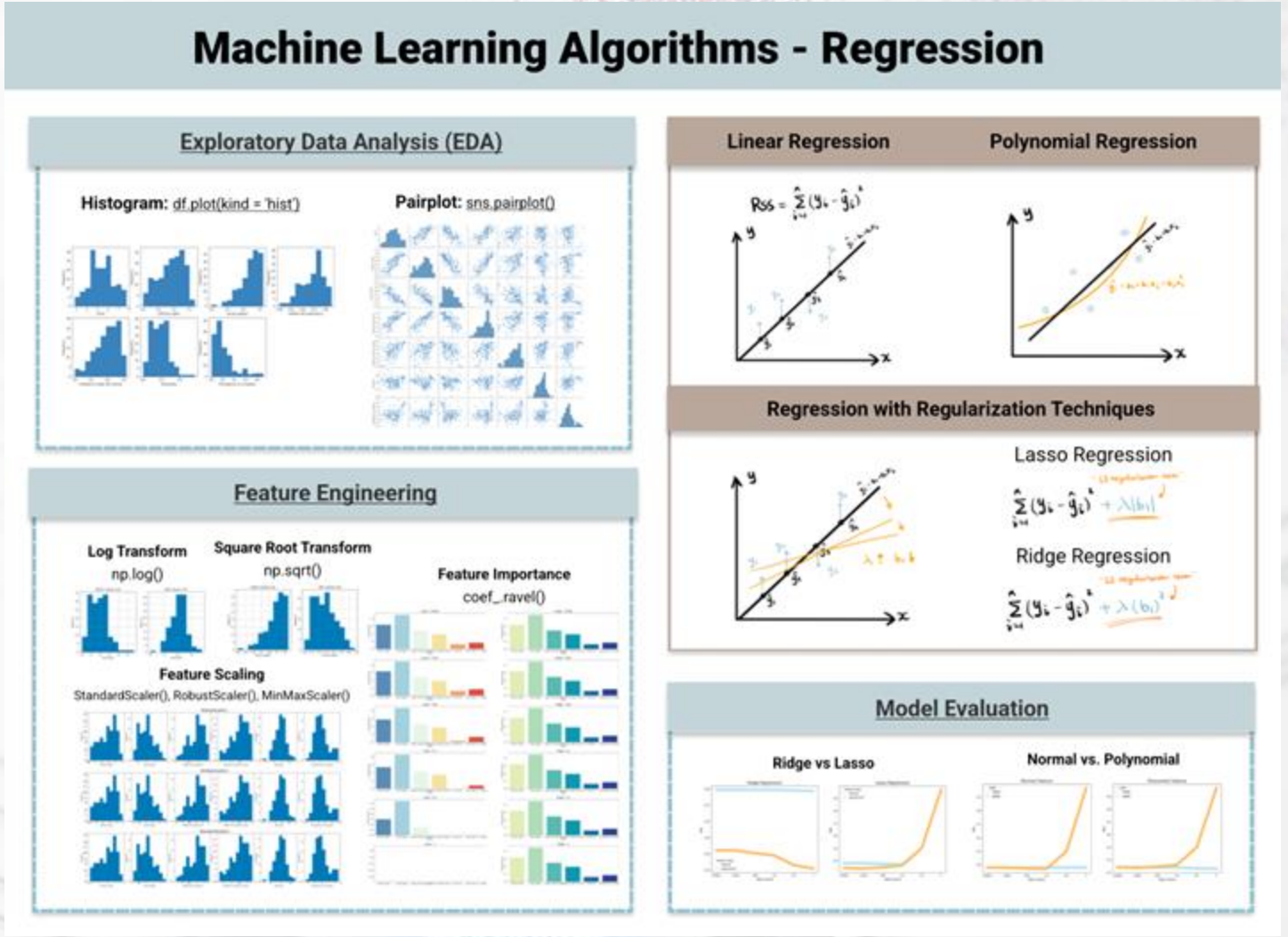
Machine learning for Regression

Regression is a statistical method used to model the relationship between a dependent variable (the outcome) and one or more independent variables (predictors). In machine learning, regression algorithms are employed to predict continuous numerical values.

Regression evaluation metrics are used to evaluate the performance of regression models, such as MSE,RMSE, MAE, MAPE,... They quantify how well a model's predictions align with the actual values.

Regression Applications

- Predicting Sales: Forecasting future sales based on historical data.
- Stock Price Prediction: Predicting future stock prices.
- Demand Forecasting: Predicting the demand for a product or service.
- Real Estate Price Prediction: Estimating the price of a property based on its features.
- Customer Lifetime Value Prediction: Predicting the total revenue a customer will generate over their lifetime.



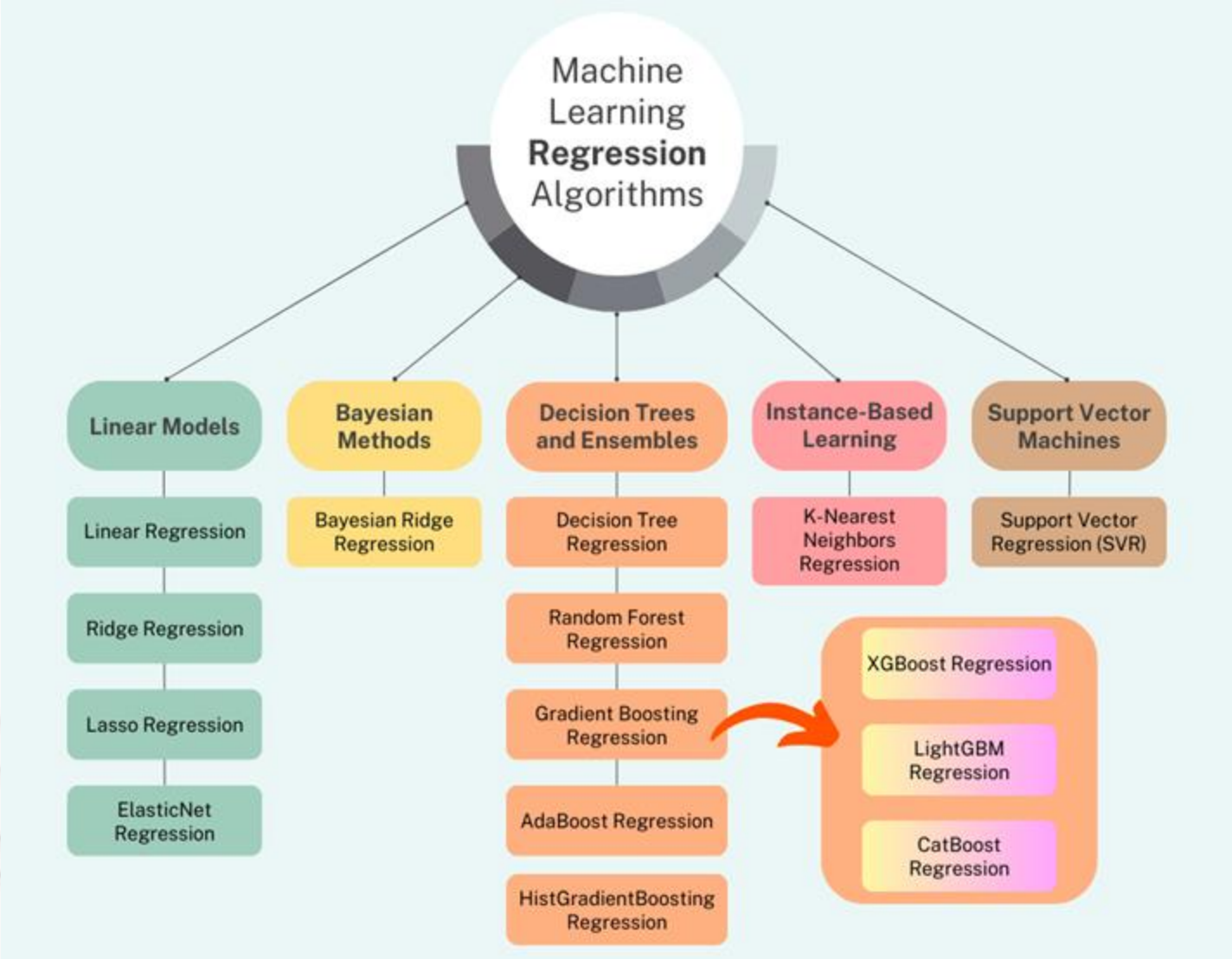
Regression evaluation metrics

Mean Squared Error (MSE)	**$MSE = 1/n * \Sigma(y_i - \hat{y}_i)^2$** Measures the average squared difference between predicted and actual values. Lower MSE indicates better performance.
Root Mean Squared Error (RMSE)	**$RMSE = \sqrt{(MSE)}$** The square root of MSE, which is often preferred because it is in the same units as the target variable.
Mean Absolute Error (MAE)	**$MAE = 1/n * \Sigma\lvert y_i - \hat{y}_i\rvert$** Measures the average absolute difference between predicted and actual values. Less sensitive to outliers than MSE.
R-squared (R^2)	**$R^2 = 1 - SSres / SStot$** Measures the proportion of variance in the dependent variable explained by the independent variables. Higher R^2 indicates better fit.
Adjusted R-squared	**Adjusted $R^2 = 1 - (1 - R^2) * (n - 1) / (n - p - 1)$** Penalizes R^2 for adding unnecessary predictors.
Mean Absolute Percentage Error (MAPE)	**$MAPE = 100 * 1/n * \Sigma\lvert y_i - \hat{y}_i\rvert / \lvert y_i\rvert$** Measures the average percentage error between predicted and actual values. Useful for comparing models on different scales.
Weighted Mean Squared Error (WMSE)	**$WMSE = 1/n * \Sigma w_i * (y_i - \hat{y}_i)^2$** Assigns different weights to errors based on the importance of the data points.

Machine Learning Regression Algorithm

Common Regression Algorithms

- **Linear Regression:** A simple model that assumes a linear relationship between the dependent and independent variables.
- **Polynomial Regression:** A more flexible model that can capture non-linear relationships by fitting a polynomial curve to the data.
- **Ridge Regression:** A regularization technique that adds a penalty term to the loss function to prevent overfitting.
- **Lasso Regression:** Another regularization technique that uses a different penalty term to encourage sparsity in the model, meaning it can select the most important features.
- **Elastic Net:** A combination of Ridge and Lasso regression, offering the benefits of both.
- **Decision Trees:** Can be used for regression by making predictions based on the average values of the target variable in the leaf nodes.
- **Random Forest:** An ensemble method that combines multiple decision trees to improve accuracy and reduce overfitting.
- **Support Vector Machines (SVM):** Can be used for regression by finding a hyperplane that maximizes the margin between the data points.
- **Neural Networks:** Deep learning models that can learn complex non-linear relationships between the features and the target variable.



Linear Regression: A Foundation in Machine Learning

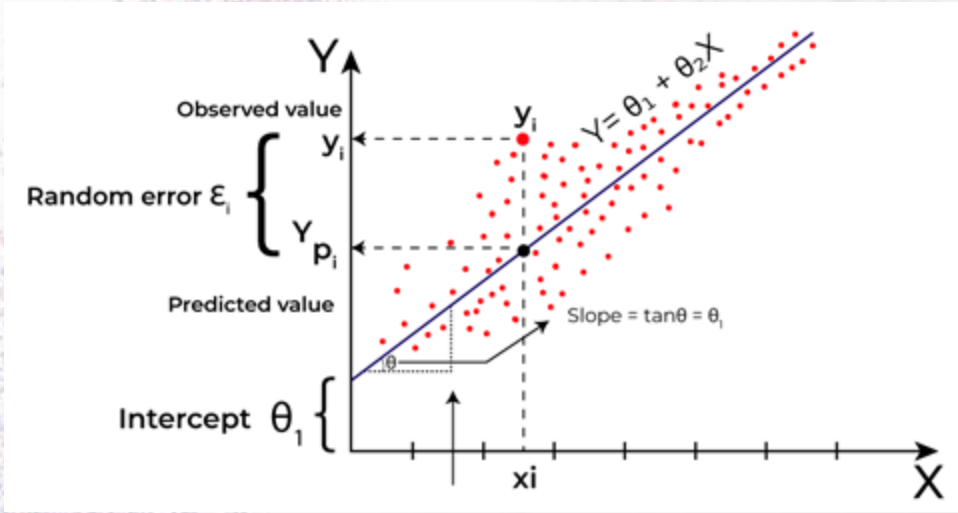
Linear Regression is a statistical method used to model the relationship between a dependent variable (the outcome) and one or more independent variables (predictors). It assumes a linear relationship between the variables and aims to find the best-fitting line to represent this relationship.

The equation of a linear regression model is:

$$y = mx + b$$

where:

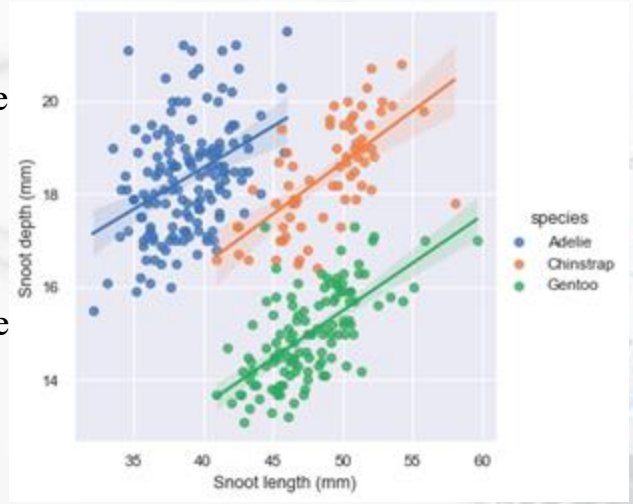
- y is the dependent variable (outcome)
- x is the independent variable (predictor)
- m is the slope of the line
- b is the y-intercept

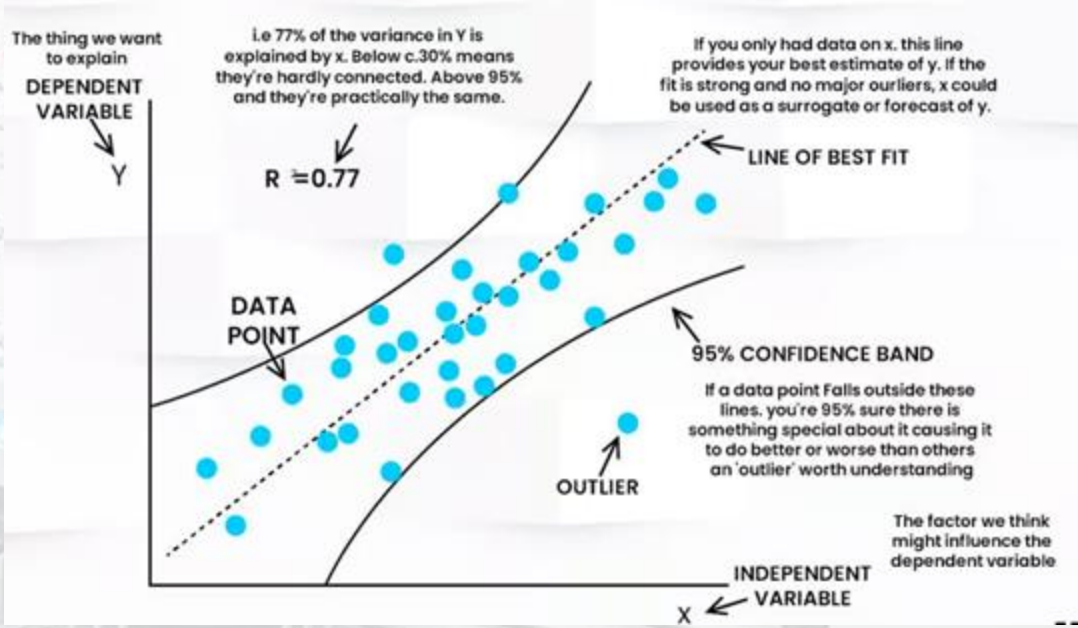


Finding the Best-Fit Line. The goal of linear regression is to find the values of m and b that minimize the error between the predicted values and the actual values. This is often done using the least squares method, which calculates the line that minimizes the sum of the squared differences between the predicted and actual values.

Types of Linear Regression:

- Simple Linear Regression: Involves a single independent variable.
  - Equation: **y = mx + b**
  - Example: Predicting house prices based on the size of the house.
- Multiple Linear Regression: Involves multiple independent variables.
  - Equation: **y = b0 + b1x1 + b2x2 + ... + bn*xn**
  - Example: Predicting car prices based on factors like mileage, age, brand, and features.





Ordinary Least Squares (OLS)

Ordinary Least Squares (OLS) is the most common and simplest method of linear regression. It aims to find the best-fitting line that minimizes the sum of the squared differences between the predicted and actual values.

In OLS, the goal is to minimize the sum of squared residuals (SSR):

$$\mathbf{SSR = \Sigma(y_i - \hat{y}_i)^2}$$

where:

y_i - the observed value of the dependent variable.

$\hat{y}_i$ - the predicted value of the dependent variable.

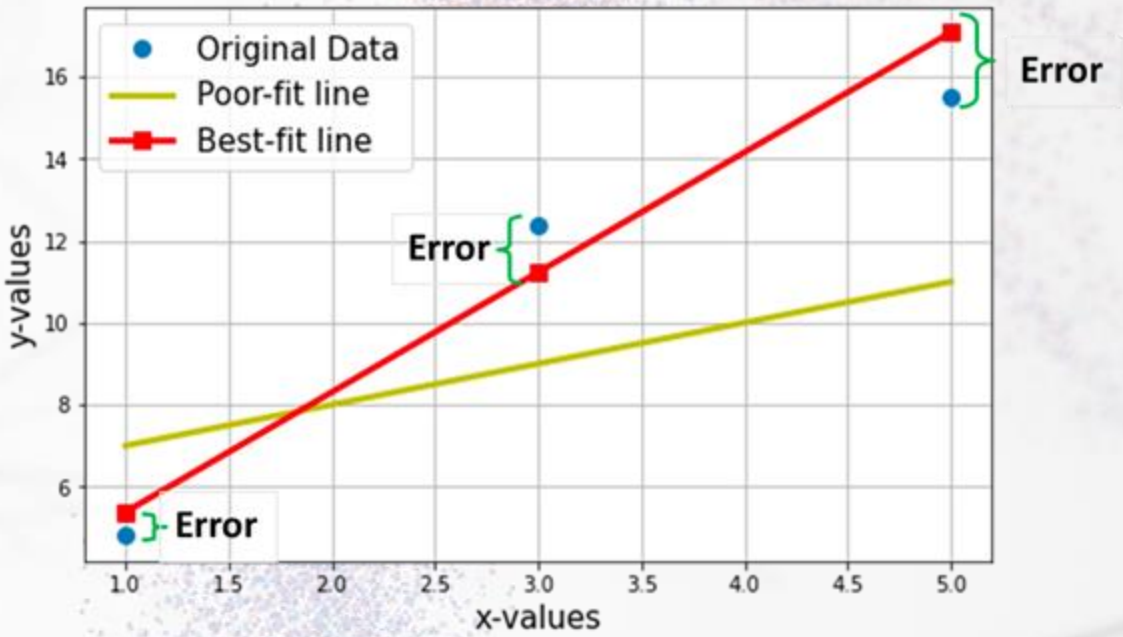
The OLS estimates of the regression coefficients (slope and intercept) are obtained by solving the following normal equations: $\min_w ||Xw - y||_2^2$

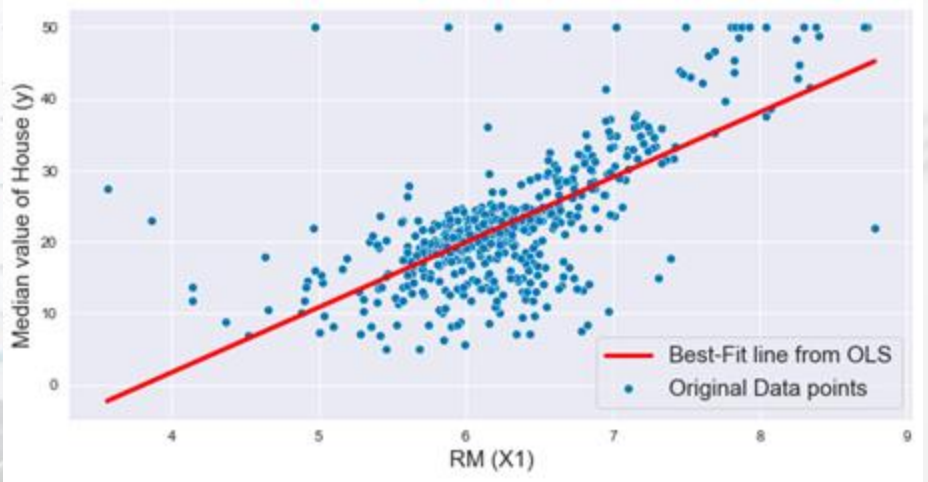
Where, predicted value : $\hat{y}(w, x) = w_0 + w_1 x_1 + \ldots + w_p x_p$

vector $w = (w_1, \ldots, w_p)$ as `coef_` and w_0 as `intercept_`.

Using scikit-learn,

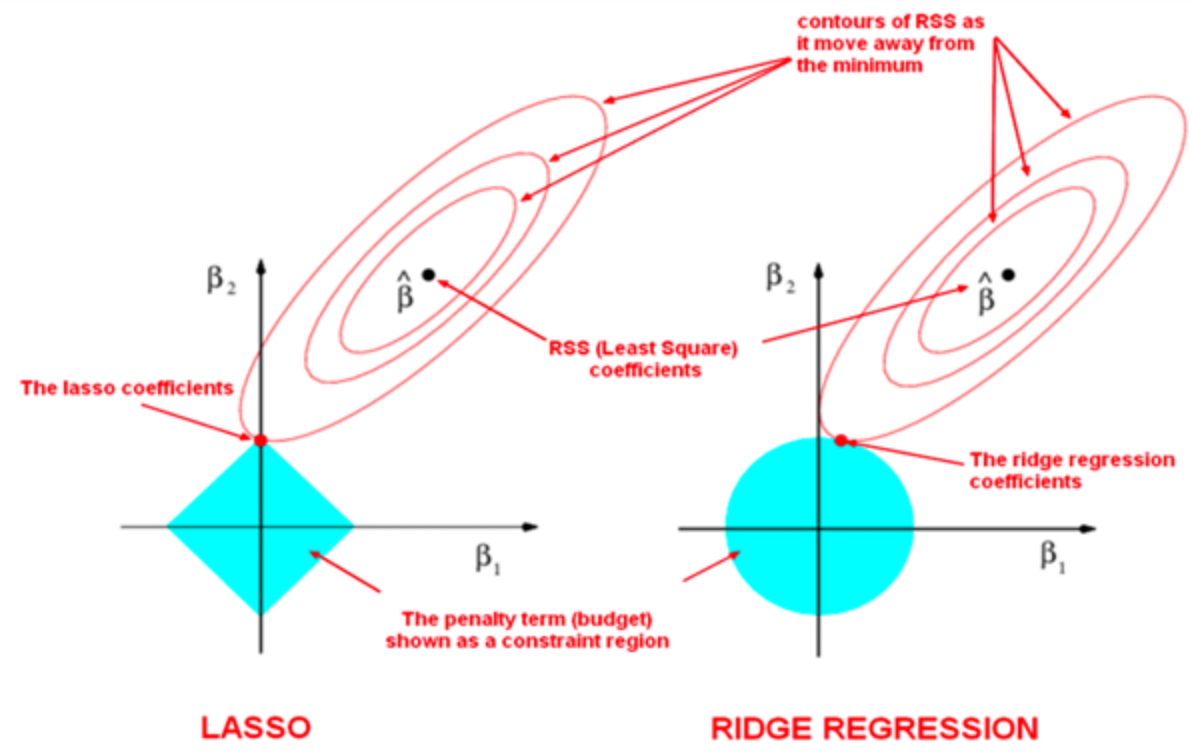
- LinearRegression
https://scikit-learn.org/stable/modules/linear_model.html#ordinary-least-squares
- Linear Regression example https://scikit-learn.org/stable/auto_examples/linear_model/plot_ols.html#sphx-glr-auto-examples-linear-model-plot-ols-py
- A Comprehensive Guide to OLS Regression
https://www.analyticsvidhya.com/blog/2023/01/a-comprehensive-guide-to-ols-regression-part-1/





Lasso Regression vs Ridge Regression

LASSO (Least Absolute Shrinkage and Selection Operator) and Ridge Regression are both regularization techniques used to prevent overfitting in linear regression models. While they share similarities, they have distinct characteristics and applications.



Feature	LASSO	Ridge Regression
Penalty Term	$\Sigma\|wj\|$	$\Sigma(wj^2)$
Feature Selection	Yes	No
Coefficient Shrinkage	Sets some coefficients to zero	Shrinks all coefficients towards zero
Bias-Variance Trade-off	Higher bias, lower variance	Lower bias, higher variance

When to Use Which

- **LASSO:**
  - When feature selection is desired.
  - When the true model is believed to be sparse (has many zero coefficients).
- **Ridge Regression:**
  - When all features are believed to be relevant.
  - When overfitting is a concern and feature selection is not necessary.

Hybrid Approaches

- **Elastic Net**: Combines the L1 and L2 penalties, offering a balance between feature selection and shrinkage.
- **Adaptive LASSO**: Adapts the regularization parameter for each feature based on their initial estimates.

Using scikit-learn,

→ Laso Regression

https://scikit-learn.org/1.5/modules/linear_model.html#lasso

→ Rigdge Regression

https://scikit-learn.org/1.5/modules/linear_model.html#regression

→ Practice

https://www.datacamp.com/tutorial/tutorial-lasso-ridge-regression

Polynomial Regression

Polynomial Regression is a form of regression analysis in which the relationship between the independent variables and dependent variables are modeled in the nth degree polynomial.
For Multiple variable the matrix calculation is done by

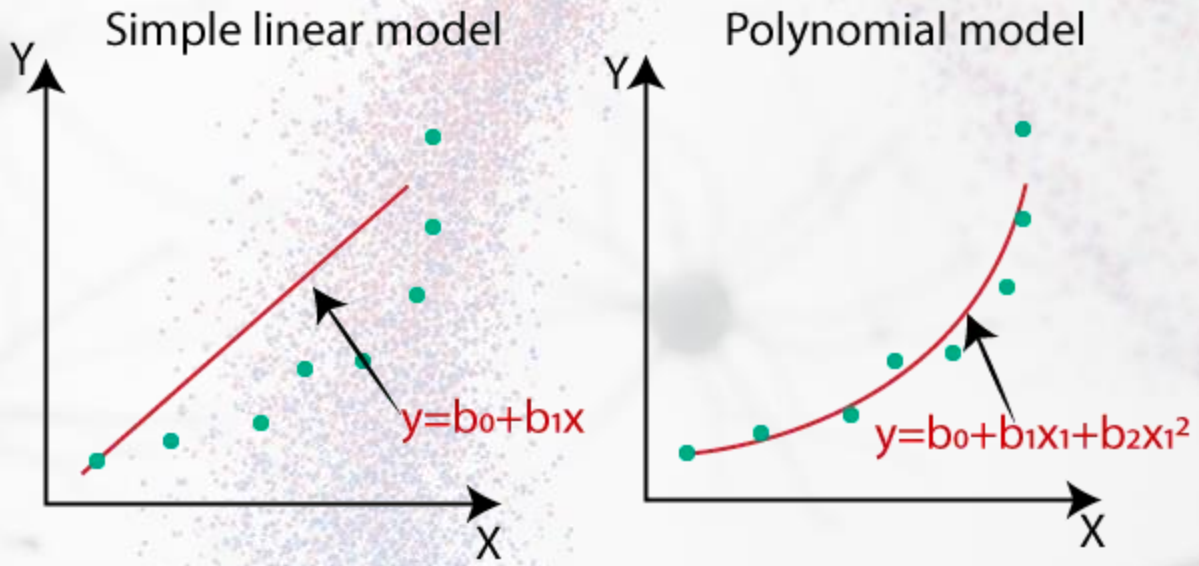
$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^m \\ 1 & x_2 & x_2^2 & \dots & x_2^m \\ 1 & x_3 & x_3^2 & \dots & x_3^m \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^m \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_m \end{bmatrix} + \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \varepsilon_3 \\ \vdots \\ \varepsilon_n \end{bmatrix},$$

Cost Function of Polynomial Regression.
Cost function measures a model's performance by calculating the average error between its predictions and actual values (loss function for each data point). Think of cost function as the overall "grade" and loss function as individual "scores."

$$J = \frac{1}{n}\sum_{i=1}^{n}(pred_i - y_i)^2$$

Polynomial regression can improve model fit by creating a curved line that better matches your data, potentially reducing the cost function's value. Higher-order polynomials can achieve even more precise fits.

To minimize the cost function and improve model performance, we can use gradient descent. This algorithm iteratively adjusts the model's weights to reduce the error between predicted and actual values.



Simple Linear Regression

$$y = b_0 + b_1 x_1$$

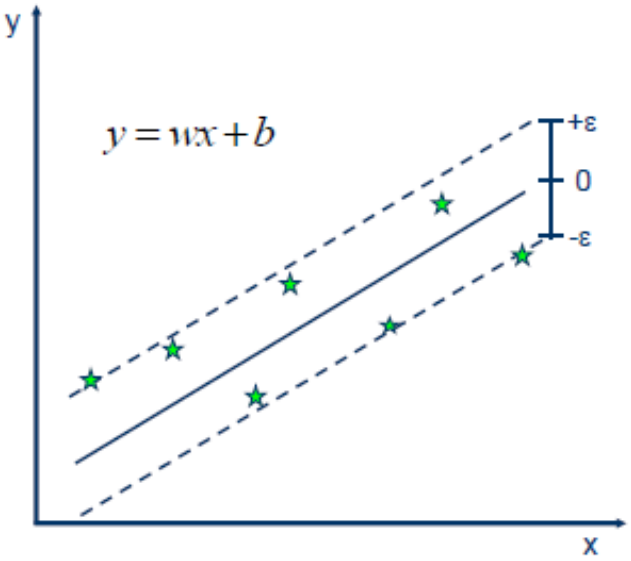
Multiple Linear Regression

$$y = b_0 + b_1 x_1 + b_2 x_2 + \ldots + b_n x_n$$

Polynomial Linear Regression

$$y = b_0 + b_1 x_1 + b_2 x_1^2 + \ldots + b_n x_1^n$$

SVM for Regression (SVR)

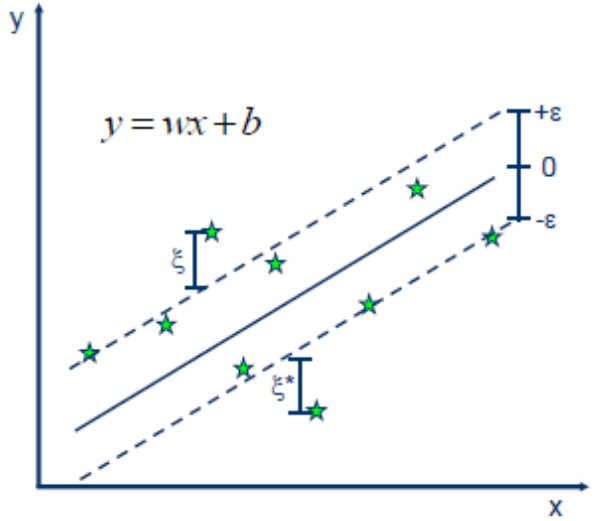


- Solution:

$$\min \frac{1}{2}\|w\|^2$$

- Constraints:

$$y_i - wx_i - b \le \varepsilon$$
$$wx_i + b - y_i \le \varepsilon$$



- Minimize:

$$\frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N}\left(\xi_i + \xi_i^*\right)$$

- Constraints:

$$y_i - wx_i - b \le \varepsilon + \xi_i$$
$$wx_i + b - y_i \le \varepsilon + \xi_i^*$$
$$\xi_i, \xi_i^* \ge 0$$

Linear SVR

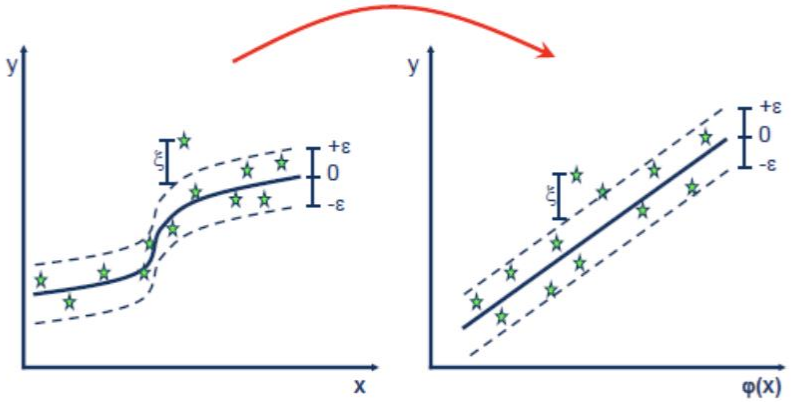
$$y = \sum_{i=1}^{N}\left(\alpha_i - \alpha_i^*\right)\cdot\langle x_i, x\rangle + b$$

Non-linear SVR

The kernel functions transform the data into a higher dimensional feature space to make it possible to perform the linear separation.

$$y = \sum_{i=1}^{N}\left(\alpha_i - \alpha_i^*\right)\cdot\langle\varphi(x_i), \varphi(x)\rangle + b$$

$$y = \sum_{i=1}^{N}\left(\alpha_i - \alpha_i^*\right)\cdot K(x_i, x) + b$$



Kernel functions

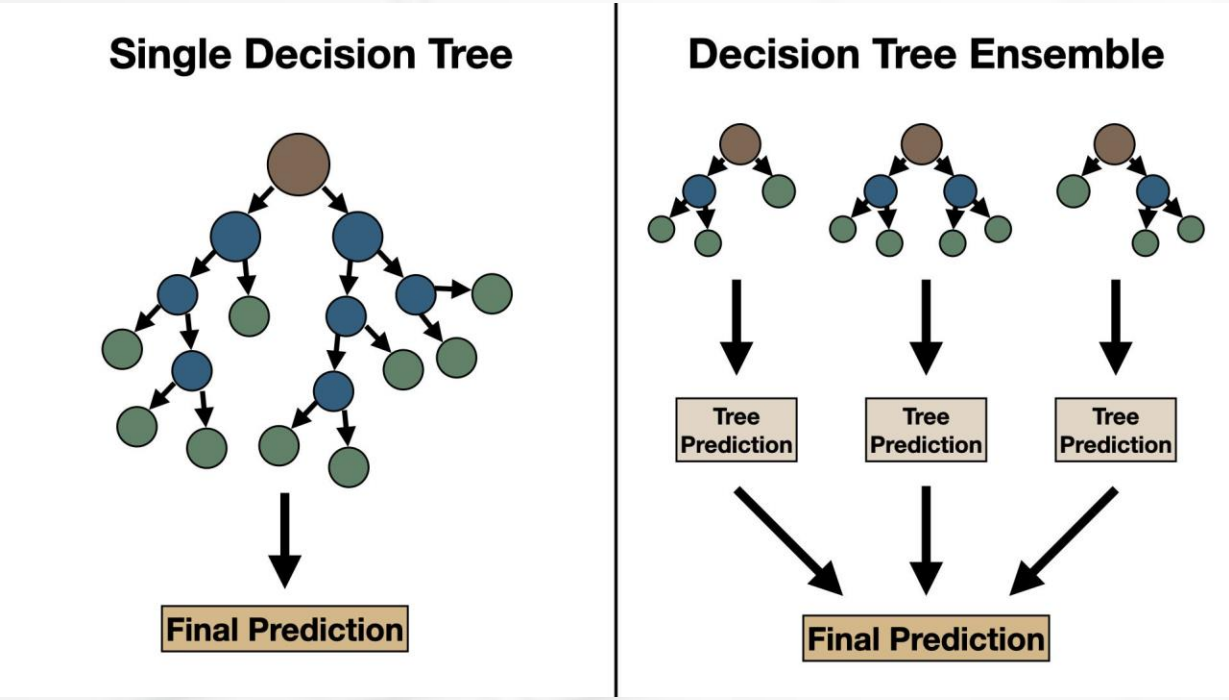
Polynomial

$$k(\mathrm{x}_i, \mathrm{x}_j) = (\mathrm{x}_i.\mathrm{x}_j)^d$$

Gaussian Radial Basis function

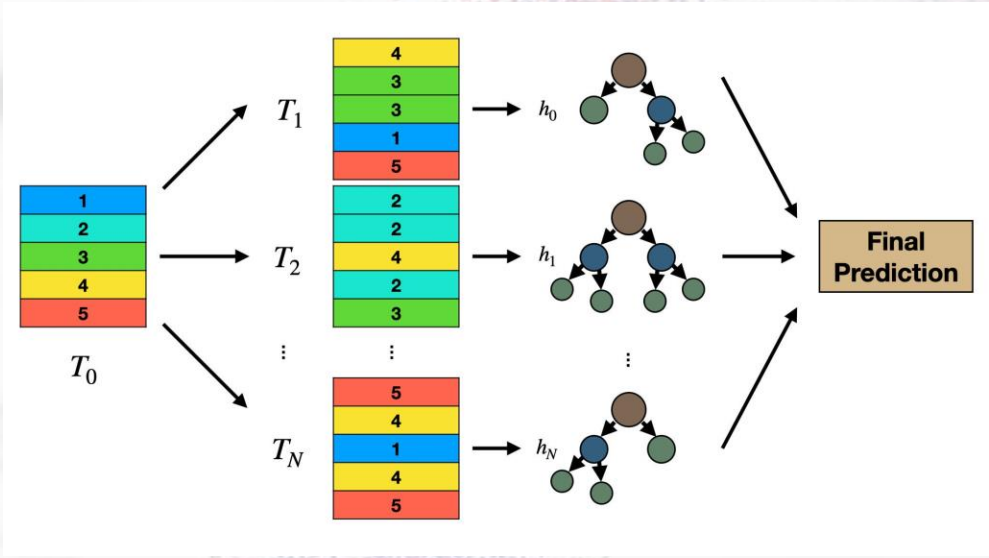
$$k(\mathrm{x}_i, \mathrm{x}_j) = \exp\left(-\frac{\|\mathrm{x}_i - \mathrm{x}_j\|^2}{2\sigma^2}\right)$$

Decision tree and Random forest for Regression

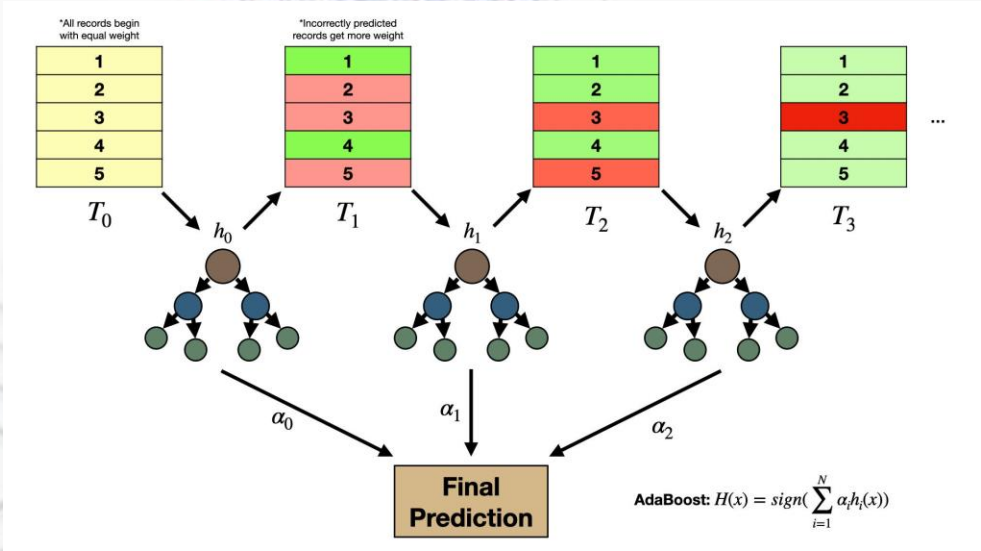


https://towardsdatascience.com/10-decision-trees-are-better-than-1-719406680564/

1) Bagging (Random Forest)



2) Boosting (AdaBoost, Gradient Boosting, XGBoost)



DECISION TREE	RANDOM FOREST
Less accurate result.	More accurate result.
Simple to understand.	Little bit complex to understand.
A decision tree is a decision support tool that uses a tree-like model of decisions and their possible consequences, including chance event outcomes, resource costs, and utility.	Random forests or random decision forests are an ensemble learning method for classification, regression and other tasks that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes or mean prediction of the individual trees.
There is a chance of overfitting.	The risk of overfitting is very minimum.