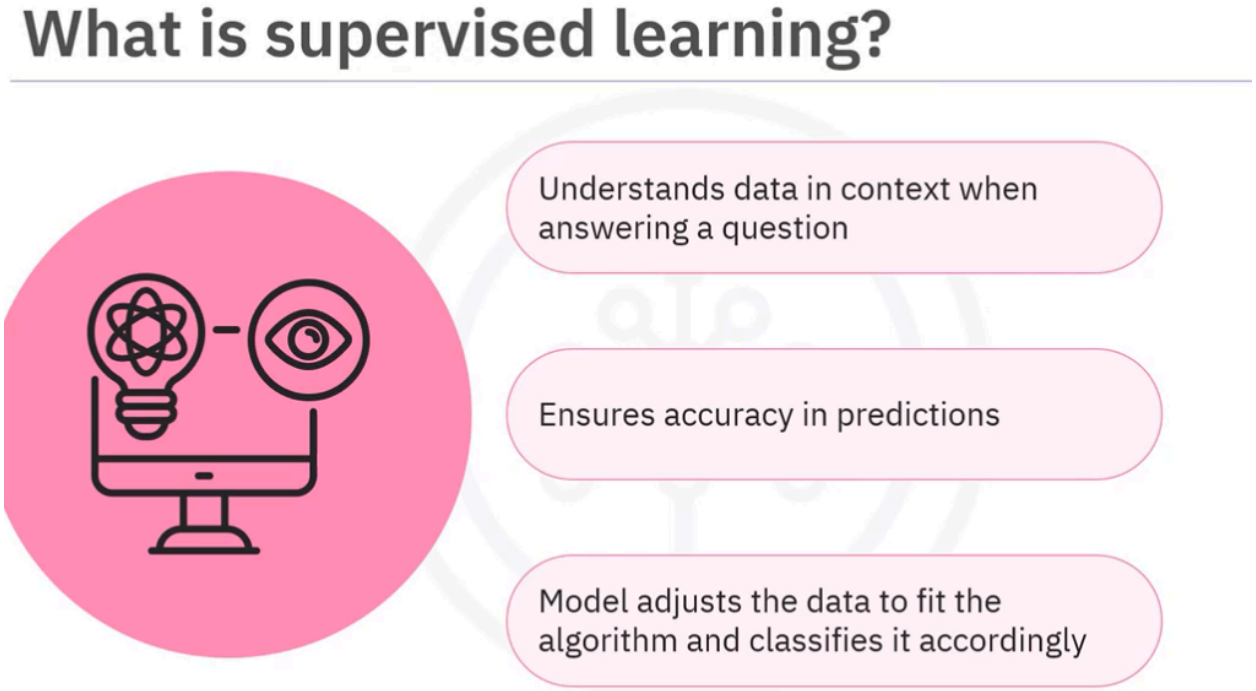
In this module, you will build and evaluate a range of supervised machine learning models to solve both classification and regression problems. You'll start by describing how classification models predict categorical outcomes, and implement multi-class classification strategies using real-world data. You'll then explore how decision trees make predictions and apply them to both classification and regression tasks. The module also covers using support vector machines (SVM) for fraud detection, applying K-Nearest Neighbors (KNN) for customer classification, and training ensemble models like Random Forest and XGBoost to improve accuracy and efficiency. You'll differentiate bias and variance in model performance and explore how ensemble methods help balance this tradeoff. To support your learning, you'll receive a Cheat Sheet: Building Supervised Learning Models with key terms, model types, and evaluation tips.

Learning Objectives

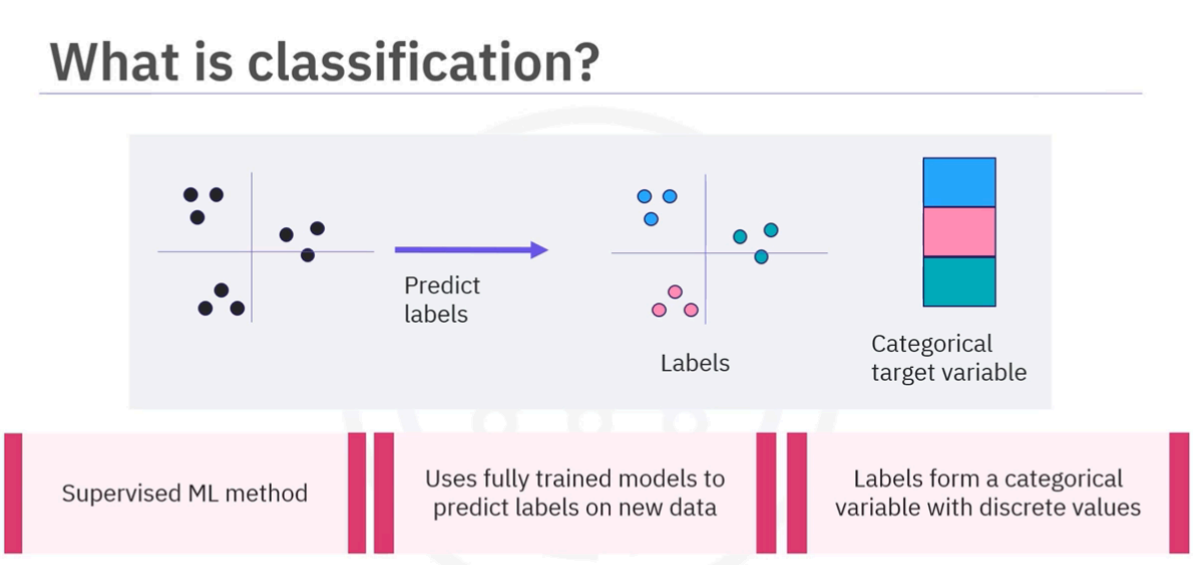
- Describe how classification models are used to predict categorical outcomes
- Implement multi-class classification strategies on real-world datasets
- Explain how decision tree models make data-driven predictions through a sequence of decisions
- Build and evaluate decision tree classifiers using patient health data
- Apply regression trees to predict continuous values using real-world data
- Compare the use of decision trees and support vector machines for fraud detection tasks
- Use the K-Nearest Neighbors algorithm to classify customer data
- Differentiate between bias and variance, and explain how ensemble models address these issues
- Train and evaluate Random Forest and XGBoost regression models for performance comparison

---

📘 Classification in Machine Learning



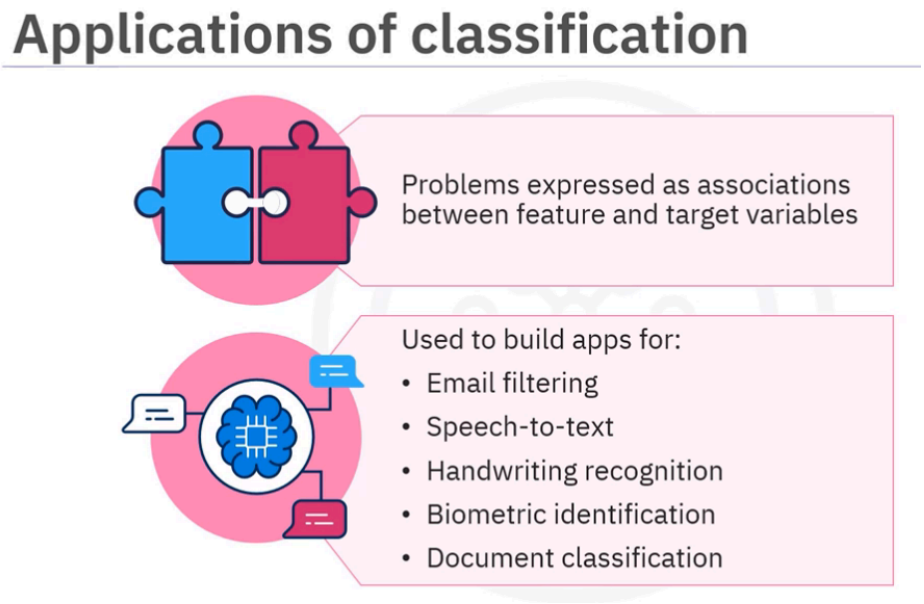
🎯 What is Classification?



- A **supervised ML method** that uses **labelled data**.
- Goal: predict **categorical/discrete labels** for new data.

- Input → processed by model → predicted class/label.

⚡ Applications & Use Cases



- **Email filtering** (spam / not spam).
- **Speech-to-text**.
- **Handwriting recognition**.
- **Biometric identification**.
- **Document classification**.
- **Churn prediction** – predict if customer leaves service.
- **Customer segmentation** – group customers into categories.
- **Advertising responsiveness** – predict if customer will respond.
- **Loan default prediction** – bank predicts if applicant will default.
- **Drug prescription** – predict which medication suits a patient.

Applications of classification

	tenure	age	address	income	ed	employ	equip	callcard	wireless	churn
0	11.0	33.0	7.0	136.0	5.0	5.0	0.0	1.0	1.0	Yes
1	33.0	33.0	12.0	33.0	2.0	0.0	0.0	0.0	0.0	Yes
2	23.0	30.0	9.0	30.0	1.0	2.0	0.0	0.0	0.0	No
3	38.0	35.0	5.0	76.0	2.0	10.0	1.0	1.0	1.0	No
4	7.0	35.0	14.0	80.0	2.0	15.0	0.0	1.0	0.0	?

Churn prediction: If customer will discontinue service

Customer segmentation: Predict the category of a customer

Advertising: Predict if a customer will respond to a campaign

-

🔑 Classification Types

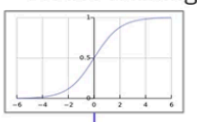
1. **Binary Classification**

Use cases of classification

Loan default prediction

age	ed	employ	address	income	debtinc	creddebt	othdebt	default
41	3	17	12	176	9.3	11.359	5.009	1
27	1	10	6	31	17.3	1.362	4.001	0
40	1	15	14	55	5.5	0.856	2.169	0
41	1	15	14	120	2.9	2.659	0.821	0
24	2	2	0	28	17.3	1.787	3.057	1
41	2	5	5	25	10.2	0.393	2.157	0
39	1	20	9	67	30.6	3.834	16.668	0
43	1	12	11	38	3.6	0.129	1.239	0
24	1	3	4	19	24.4	1.358	3.278	1
36	1	0	13	25	19.7	2.778	2.147	0

Model Training



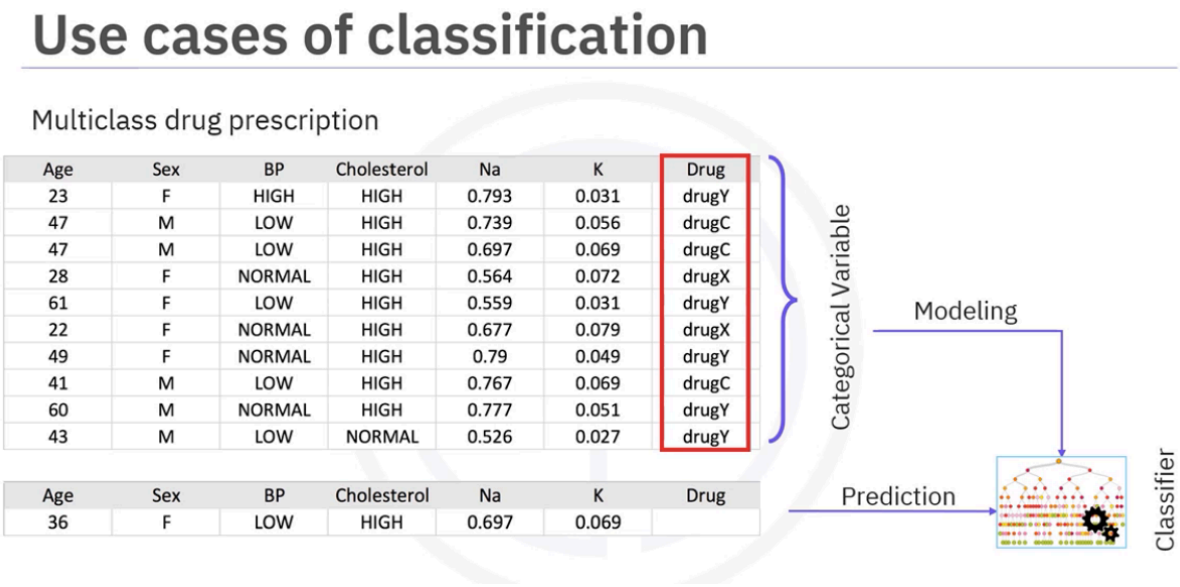
age	ed	employ	address	income	debtinc	creddebt	othdebt	default
37	2	16	10	130	9.3	10.23	3.21	0

Trained Classifier

Prediction

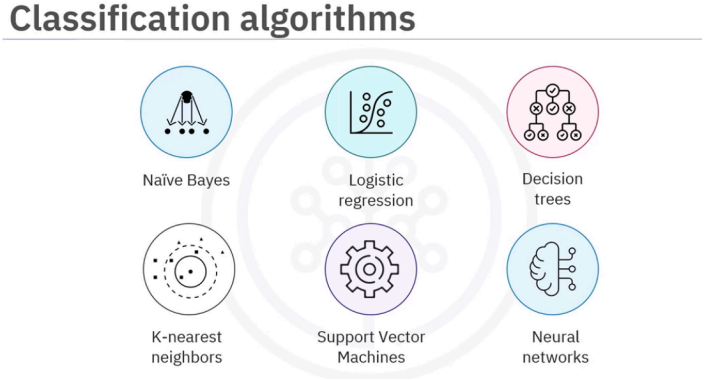
   - Two possible outcomes.
   - Example: Loan default (Yes/No), Churn (Stay/Leave).

2. **Multi-class Classification**



- More than two classes.
- Example: Predict which drug (Drug A, B, or C).

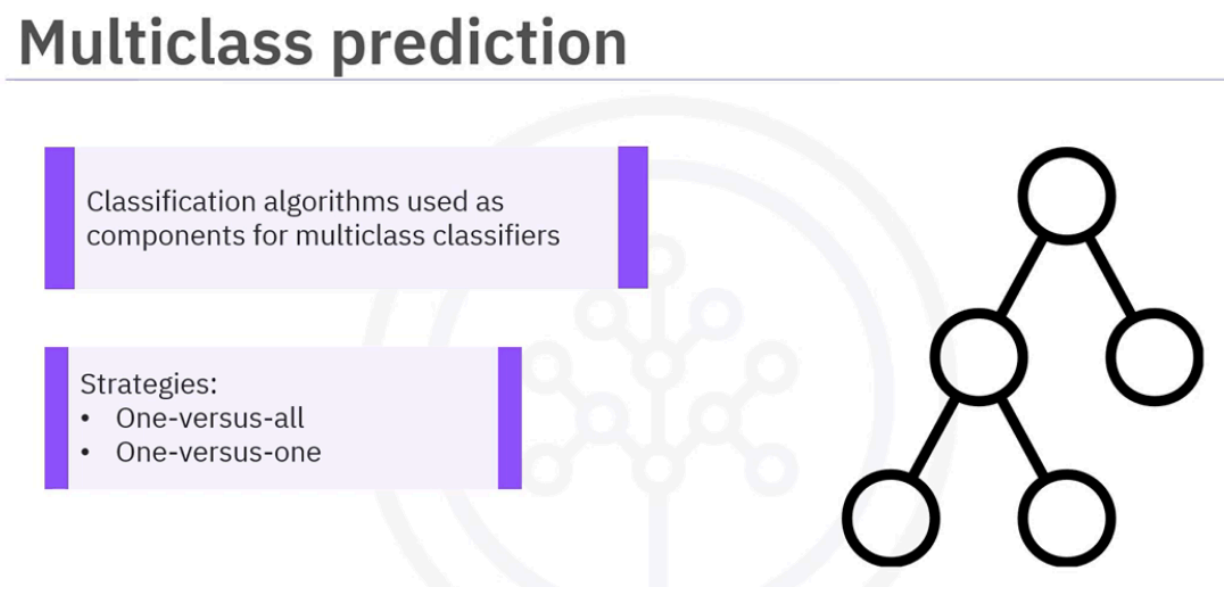
🛠️ Common Classification Algorithms



- **Naive Bayes**
- **Logistic Regression**
- **Decision Trees**

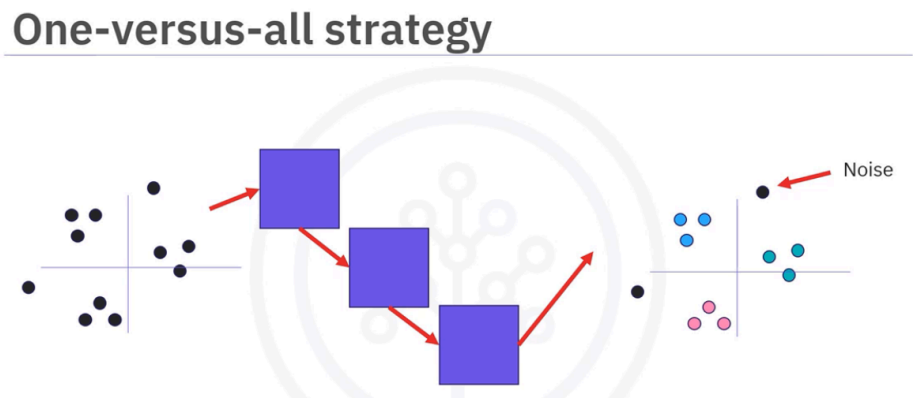
- **K-Nearest Neighbours (KNN)**
- **Support Vector Machines (SVMs)**
- **Neural Networks**

🎲 Multi-class Prediction Strategies

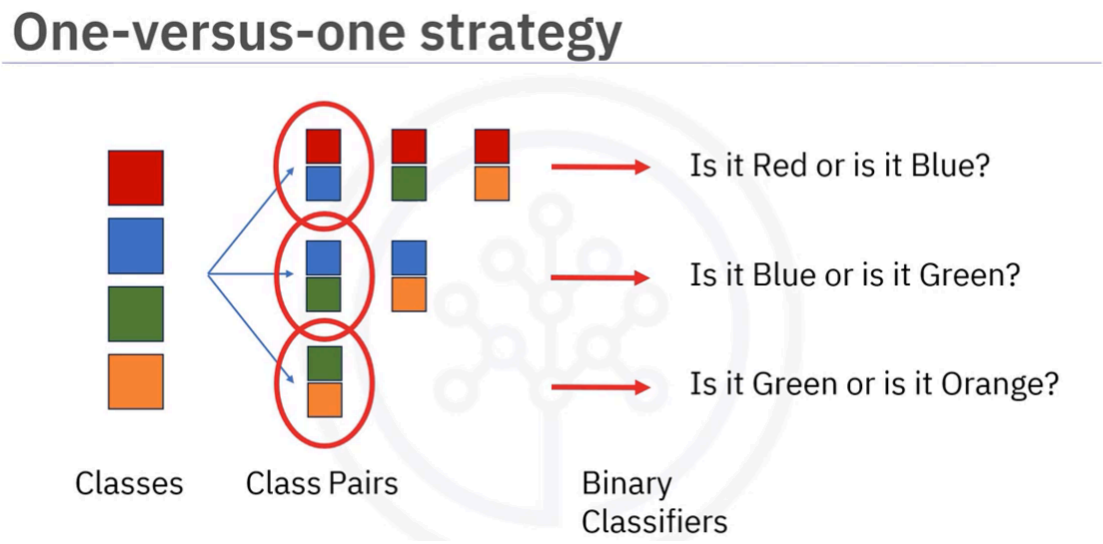


1. One-vs-All (OvA)

- Train **k binary classifiers** (one per class).
- Each classifier: "Is it this class (1) or not (0)?"
- Total classifiers = number of classes (k).
- Useful for outlier/noise detection (unclassified points).



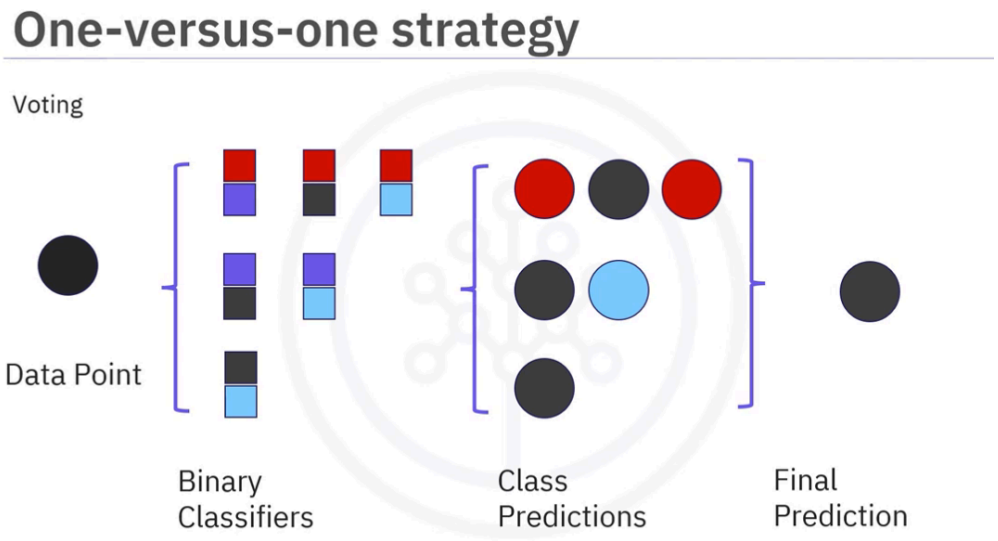
2. One-vs-One (OvO)

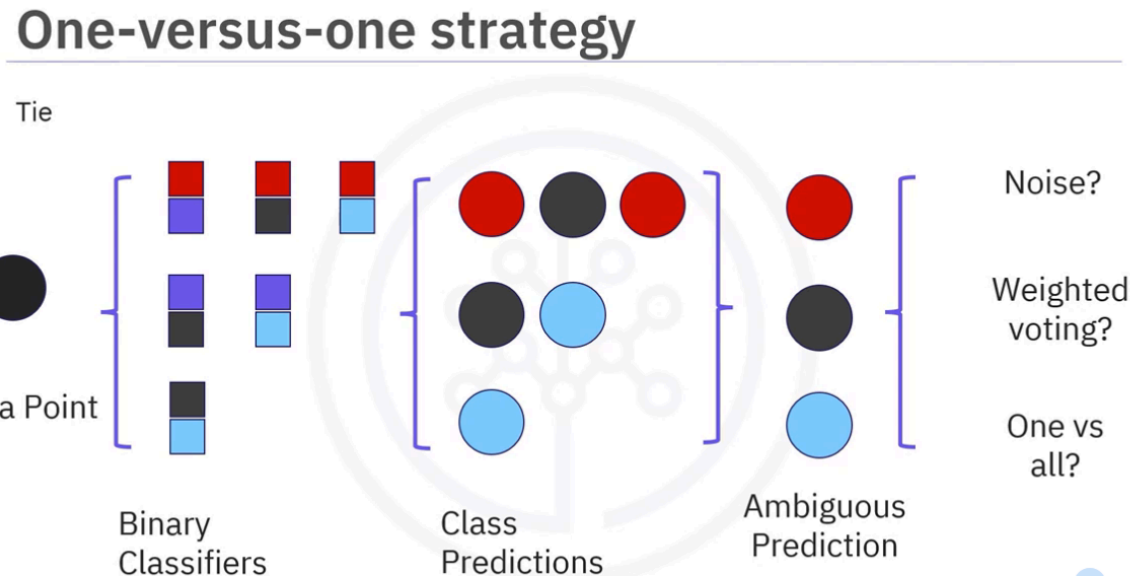


- Train classifiers for **every pair of classes**.
- Each classifier answers: "Is it this class OR that class?"
- Final label = chosen by **voting**.
    - **Simple voting**: majority wins.

    - **Weighted voting**: confidence/probability considered.

- Better when classes are balanced.





📌 Key Takeaways

- Classification = **supervised ML for categorical outcomes**.
- Wide applications: finance, healthcare, marketing, security.

- Algorithms: Logistic Regression, Decision Trees, KNN, SVM, Neural Nets.
- Binary classifiers → extended to multi-class using **OvA** or **OvO**.

Got it ✅
Here are **well-structured notes with explanations + code snippets** from your *Multi-class Classification* text. I've broken it into learning sections with the most important concepts, workflows, and code.

📘 Multi-class Classification — Notes

Date: September 22, 2025
Dataset: Obesity Risk Prediction (UCI)
Samples: 2,111 | **Features:** 17 | **Target:** Obesity level (`NObeyesdad`)

🎯 Objectives

After this lab, you should be able to:

1. Use **one-hot encoding** for categorical variables.
2. Train **logistic regression models** using:
    - **One-vs-All (OvA)**
    - **One-vs-One (OvO)**
3. Evaluate performance with metrics like **accuracy**.

---

1. Setup

Install and Import Libraries

```
!pip install numpy==2.2.0 pandas==2.2.3 scikit-learn==1.6.0 matplotlib==3.9.3 seaborn==0.13.2
```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.multiclass import OneVsOneClassifier
from sklearn.metrics import accuracy_score

import warnings
warnings.filterwarnings("ignore")
```

---

2. Dataset

- **Continuous Features:** Age, Height, Weight, FCVC, NCP, CH2O, FAF, TUE
- **Categorical Features:** Gender, CAEC, CALC, MTRANS, etc.
- **Target:** `NObeyesdad` (Obesity level classification)

Load the Dataset

```
file_path =
"https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/GkDzb7bWrtvGXdPOfk6CIg/Obesity-level-prediction-dataset.csv"

data = pd.read_csv(file_path)

data.head()
```

---

3. Exploratory Data Analysis

Target Distribution

```
sns.countplot(y='NObeyesdad', data=data)

plt.title('Distribution of Obesity Levels')

plt.show()
```

👉 Dataset is fairly **balanced**.

Check for Nulls & Summary

```
print(data.isnull().sum())   # No nulls

print(data.info())           # dtypes & memory

print(data.describe())       # statistics
```

---

4. Preprocessing

🔹 Feature Scaling

```
continuous_columns = data.select_dtypes(include=['float64']).columns.tolist()

scaler = StandardScaler()

scaled_features = scaler.fit_transform(data[continuous_columns])
```

```
scaled_df = pd.DataFrame(scaled_features,
columns=scaler.get_feature_names_out(continuous_columns))

scaled_data = pd.concat([data.drop(columns=continuous_columns), scaled_df], axis=1)
```

🔹 One-Hot Encoding (categorical vars)

```
categorical_columns = scaled_data.select_dtypes(include=['object']).columns.tolist()

categorical_columns.remove('NObeyesdad')


encoder = OneHotEncoder(sparse_output=False, drop='first')

encoded_features = encoder.fit_transform(scaled_data[categorical_columns])

encoded_df = pd.DataFrame(encoded_features,
columns=encoder.get_feature_names_out(categorical_columns))


prepped_data = pd.concat([scaled_data.drop(columns=categorical_columns), encoded_df],
axis=1)
```

🔹 Encode Target

```
prepped_data['NObeyesdad'] = prepped_data['NObeyesdad'].astype('category').cat.codes
```

🔹 Final Split

```
X = prepped_data.drop('NObeyesdad', axis=1)

y = prepped_data['NObeyesdad']


X_train, X_test, y_train, y_test = train_test_split(

    X, y, test_size=0.2, random_state=42, stratify=y
```

)

---

5. Model Training & Evaluation

5.1 Logistic Regression — One-vs-All (OvA)

- Trains **k classifiers** (one for each class vs all others).

```
model_ova = LogisticRegression(multi_class='ovr', max_iter=1000)
model_ova.fit(X_train, y_train)

y_pred_ova = model_ova.predict(X_test)
print("OvA Accuracy:", round(100*accuracy_score(y_test, y_pred_ova), 2), "%")
```

📊 **Result:** ~76.1% accuracy

5.2 Logistic Regression — One-vs-One (OvO)

- Trains **k(k-1)/2 classifiers** (pairwise comparisons).

```
model_ovo = OneVsOneClassifier(LogisticRegression(max_iter=1000))
model_ovo.fit(X_train, y_train)

y_pred_ovo = model_ovo.predict(X_test)
print("OvO Accuracy:", round(100*accuracy_score(y_test, y_pred_ovo), 2), "%")
```

📊 **Result:** ~92.2% accuracy

---

6. Exercises

Q1. Impact of Different Test Sizes

```
for test_size in [0.1, 0.3]:
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=test_size, random_state=42, stratify=y
    )
    model_ova.fit(X_train, y_train)
    y_pred = model_ova.predict(X_test)
    print(f"Test size: {test_size} | Accuracy: {accuracy_score(y_test, y_pred):.4f}")
```

Q2. Feature Importance

For OvA model:

```
feature_importance = np.mean(np.abs(model_ova.coef_), axis=0)
plt.barh(X.columns, feature_importance)
plt.title("Feature Importance (OvA)")
plt.xlabel("Importance")
plt.show()
```

For OvO model:

```
coefs = np.array([est.coef_[0] for est in model_ovo.estimators_])
feature_importance = np.mean(np.abs(coefs), axis=0)
```

```
plt.barh(X.columns, feature_importance)

plt.title("Feature Importance (OvO)")

plt.xlabel("Importance")

plt.show()
```

Q3. Automating with a Pipeline

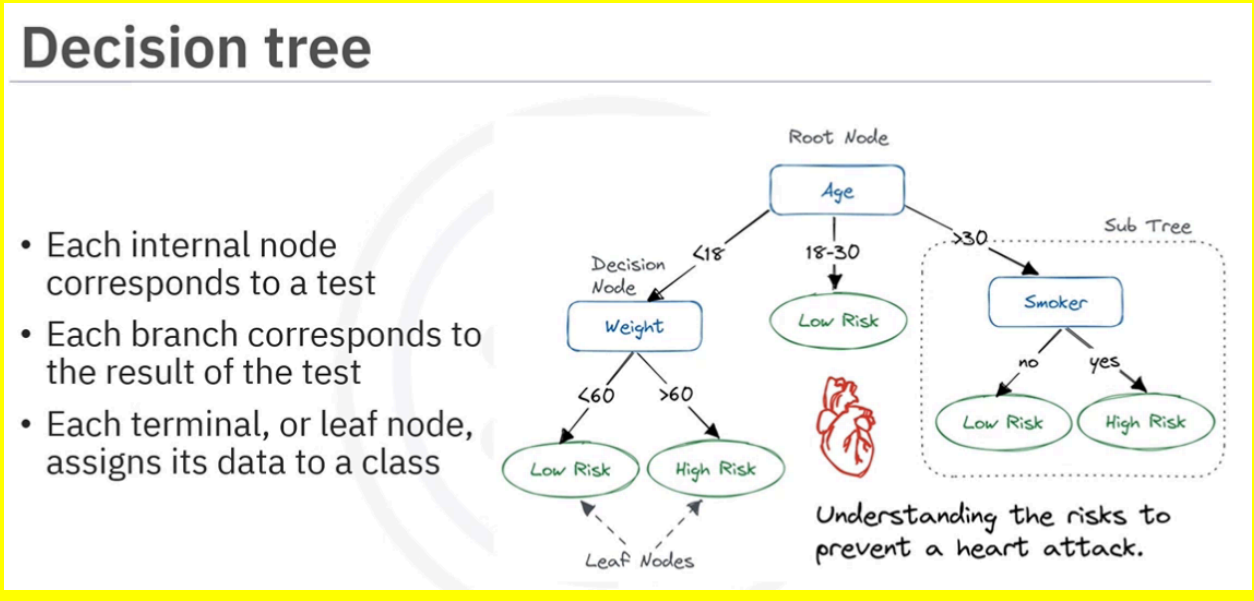
```
def obesity_risk_pipeline(data_path, test_size=0.2):

    # Load

    data = pd.read_csv(data_path)


    # Scale

    continuous_columns = data.select_dtypes(include=['float64']).columns.tolist()

    scaler = StandardScaler()

    scaled_features = scaler.fit_transform(data[continuous_columns])

    scaled_df = pd.DataFrame(scaled_features,
columns=scaler.get_feature_names_out(continuous_columns))

    scaled_data = pd.concat([data.drop(columns=continuous_columns), scaled_df], axis=1)


    # One-hot encode

    categorical_columns = scaled_data.select_dtypes(include=['object']).columns.tolist()

    categorical_columns.remove('NObeyesdad')

    encoder = OneHotEncoder(sparse_output=False, drop='first')

    encoded_features = encoder.fit_transform(scaled_data[categorical_columns])
```

```
    encoded_df = pd.DataFrame(encoded_features, columns=encoder.get_feature_names_out(categorical_columns))

    prepped_data = pd.concat([scaled_data.drop(columns=categorical_columns), encoded_df], axis=1)


    # Encode target

    prepped_data['NObeyesdad'] = prepped_data['NObeyesdad'].astype('category').cat.codes


    # Split

    X = prepped_data.drop('NObeyesdad', axis=1)

    y = prepped_data['NObeyesdad']

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size, random_state=42, stratify=y)


    # Train multinomial logistic regression

    model = LogisticRegression(multi_class='multinomial', max_iter=1000)

    model.fit(X_train, y_train)


    # Evaluate

    y_pred = model.predict(X_test)

    print(f"Pipeline Accuracy (test_size={test_size}): {accuracy_score(y_test, y_pred):.4f}")


# Run pipeline

obesity_risk_pipeline(file_path, test_size=0.2)
```

📝 Key Takeaways

- **OvA** is simpler but less accurate (~76%).
- **OvO** achieves much higher accuracy (~92%) but is computationally heavier.
- **Preprocessing** (scaling + encoding) is crucial for logistic regression.
- **Feature importance** can be derived from logistic regression coefficients.
- A **pipeline function** helps automate preprocessing → training → evaluation.

🌳 Decision Trees in Machine Learning – Notes



📖 What is a Decision Tree?

- A **flowchart-like algorithm** for classification.
- **Structure**:
    - Internal node → a **test** on a feature.

- Branch → outcome of the test.
- Leaf/terminal node → final **class assignment**.

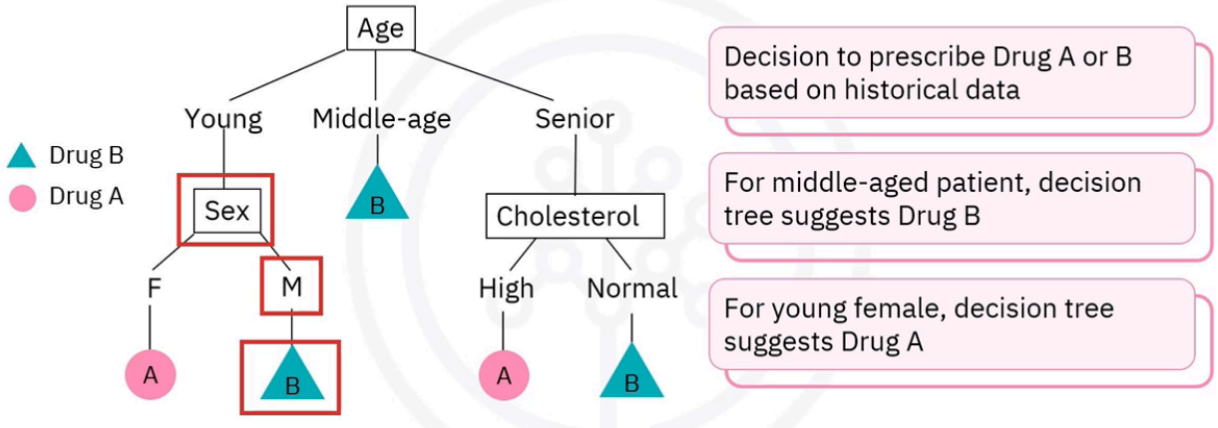
How to build a decision tree?

Patient ID	Age	Sex	BP	Cholesterol	Drug
p1	Young	F	High	Normal	Drug A
p2	Young	F	High	High	Drug A
p3	Middle-age	F	High	Normal	Drug B
p4	Senior	F	Normal	Normal	Drug B
p5	Senior	M	Low	Normal	Drug B
p6	Senior	M	Low	High	Drug A
p7	Middle-age	M	Low	High	Drug B
p8	Young	F	Normal	Normal	Drug A
p9	Young	M	Low	Normal	Drug B
p10	Senior	M	Normal	Normal	Drug B
p11	Young	M	Normal	High	Drug B
p12	Middle-age	F	Normal	High	Drug B
p13	Middle-age	M	High	Normal	Drug B
p14	Senior	F	Normal	High	Drug A
p15	Middle-age	F	Low	Normal	?

- Consider features of a data set
- Example in medical study: Age, sex, blood pressure, and cholesterol
- Use training part of data set to build decision tree
- Use decision tree to predict class of unknown patient

Example (Drug Prescription)

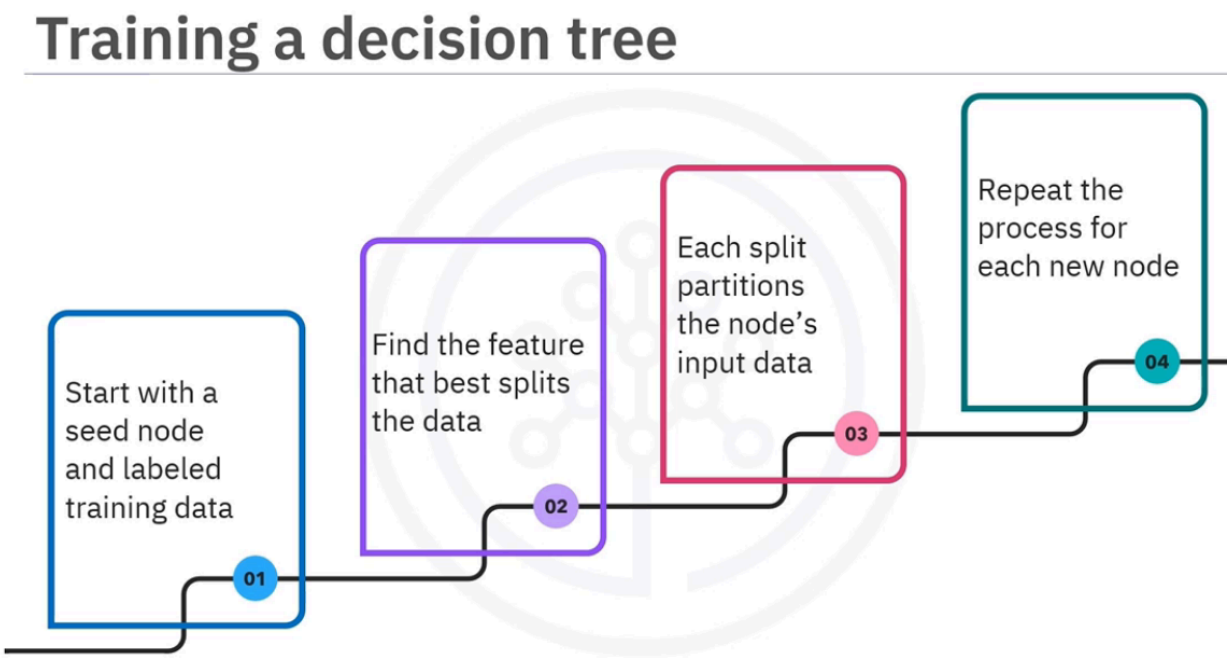
Patient classifier example



- Dataset: Age, Gender, Blood Pressure, Cholesterol.
- Target: Drug (A or B).

- Tree Example:
    - Middle-aged → Drug B.
    - Young male → Drug B.
    - Young female → Drug A.
    - Senior with normal cholesterol → Drug B.
    - Senior with high cholesterol → Drug A.

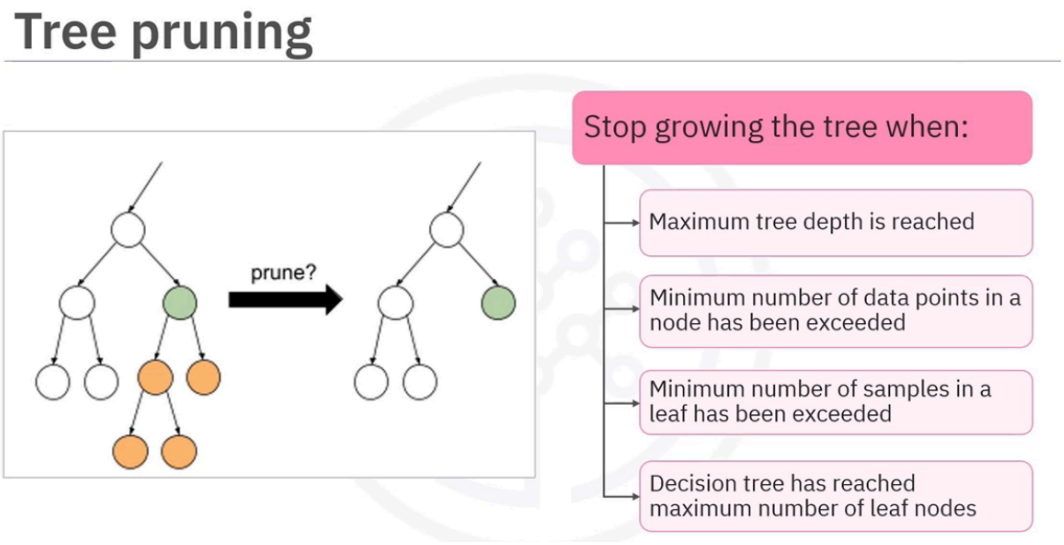
⚙️ How a Decision Tree is Built



1. Start with root node + labeled data.
2. Choose the **feature that best splits** data (splitting criterion).
3. Partition data → pass subsets to branches.
4. Repeat recursively → new nodes.

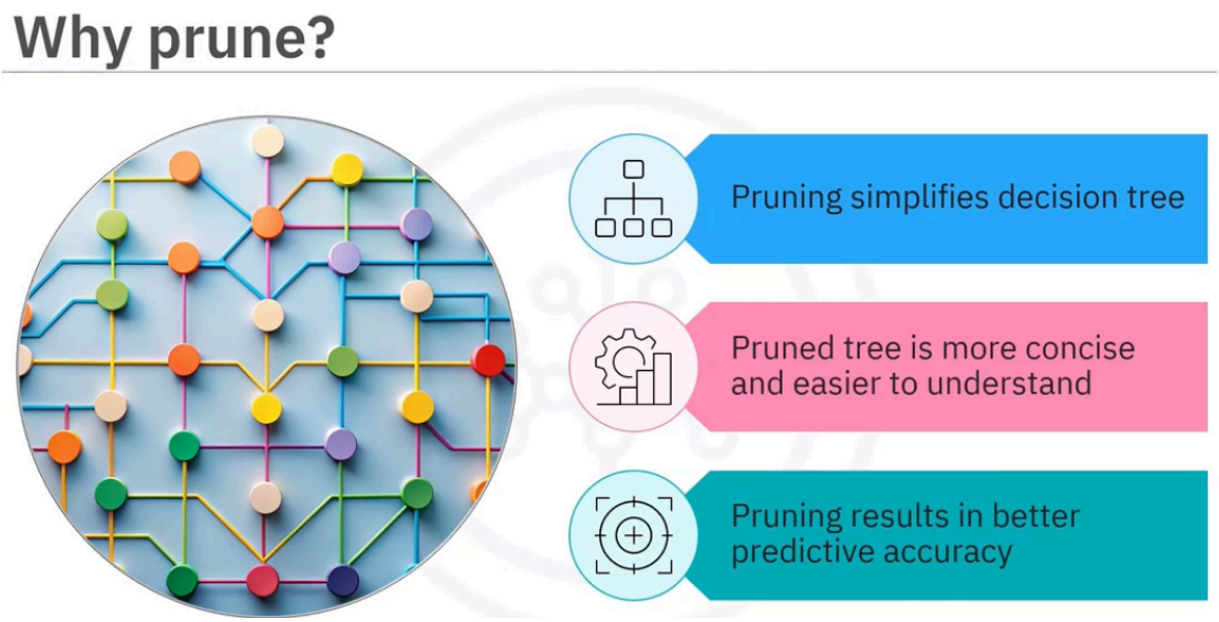
5. Stop when:
    - Nodes contain only one class.
    - No features left.
    - Stopping criterion reached (**pre-pruning**).

✂️ Pruning (Stopping Growth)

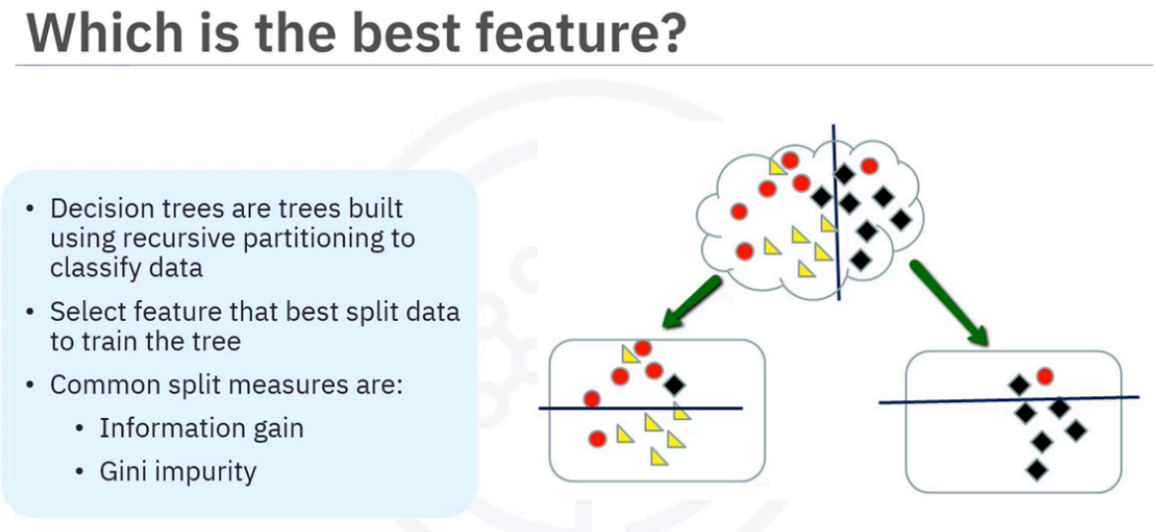


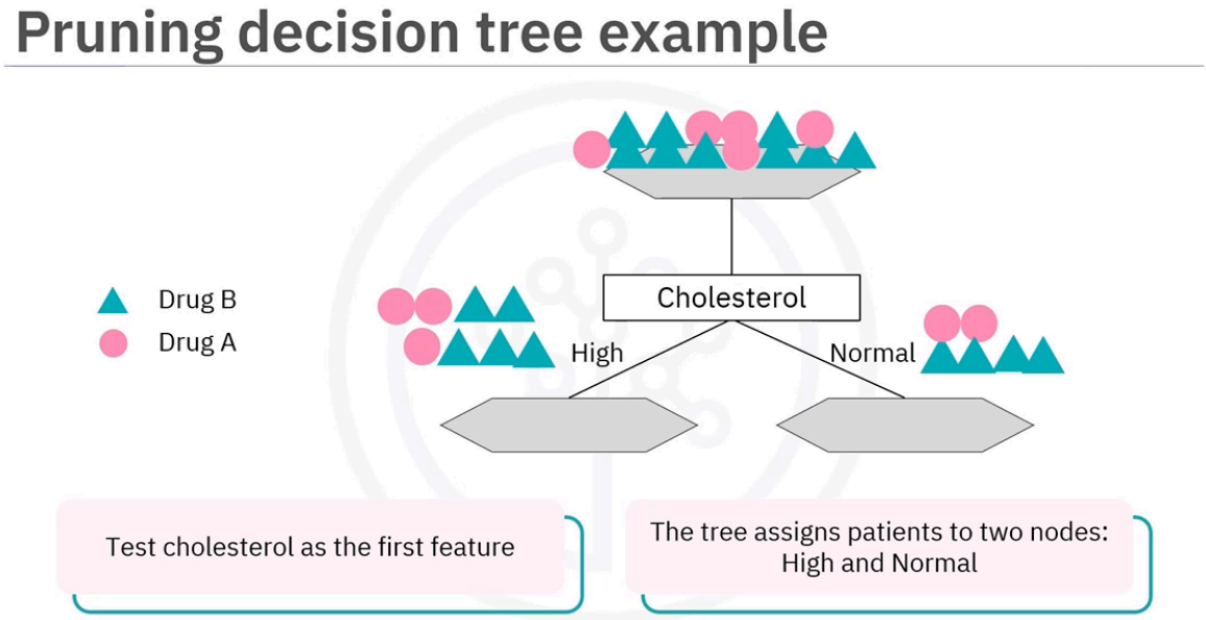
- **Stopping criteria**:
    - Max depth reached.
    - Minimum samples per node/leaf.
    - Max number of leaf nodes.

- **Why prune?**

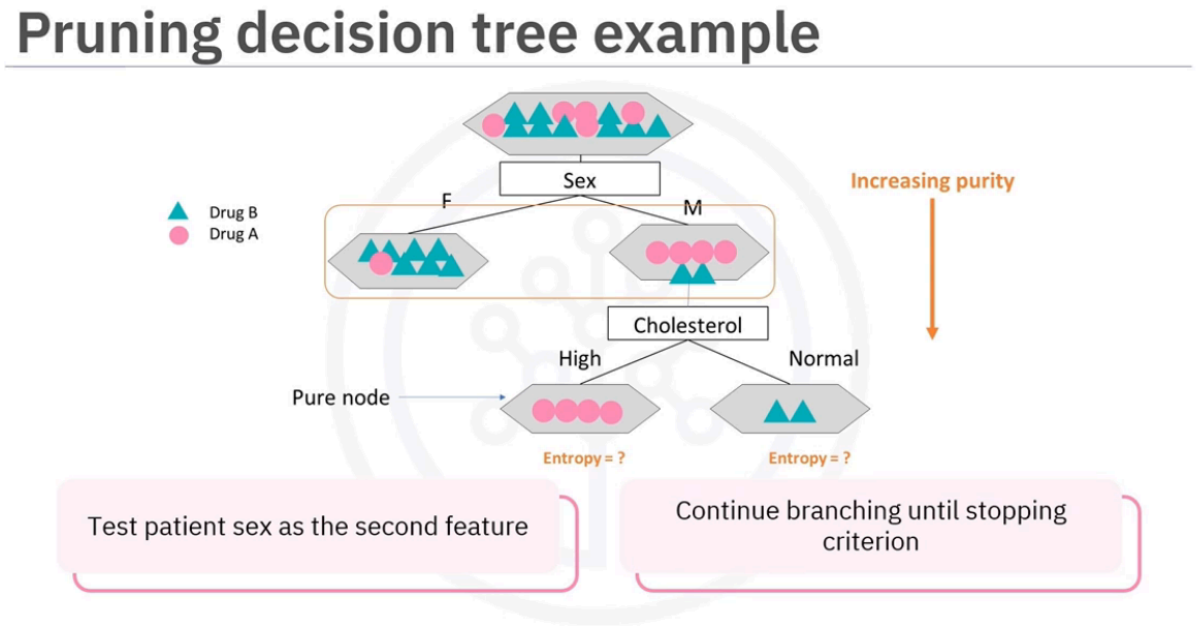


  - Avoid **overfitting**.
  - Remove noise/irrelevant splits.
  - Simpler & more interpretable model.
  - Improves **generalization** & accuracy.

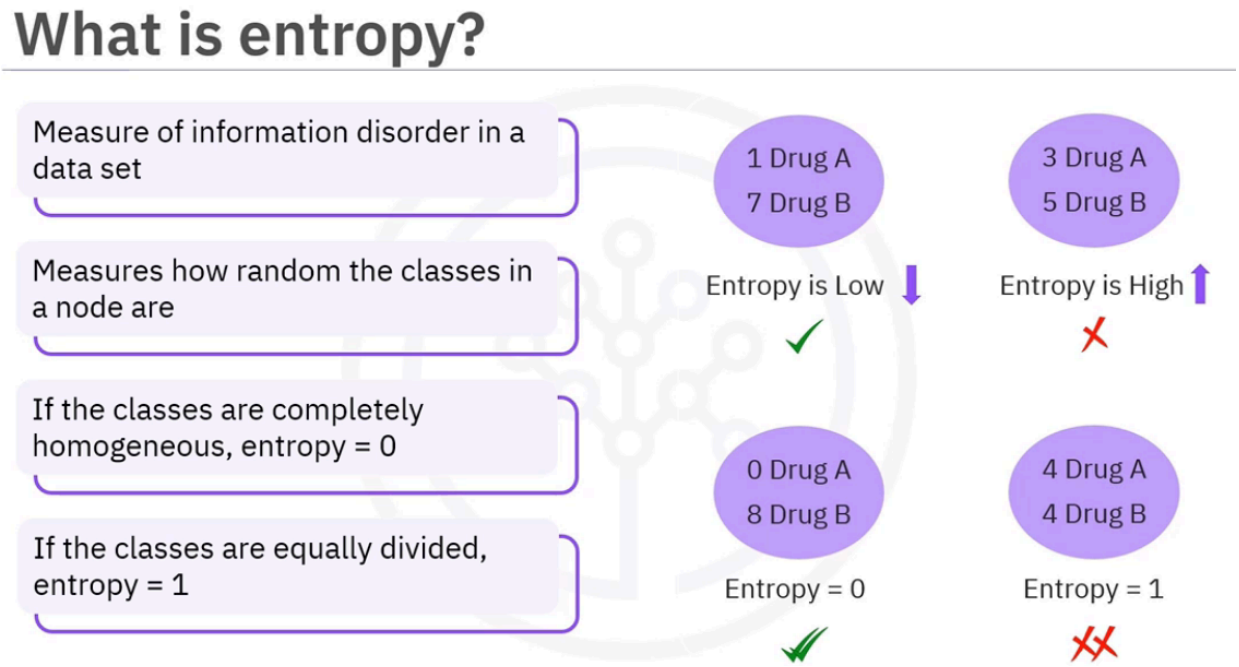




📊 Splitting Criteria

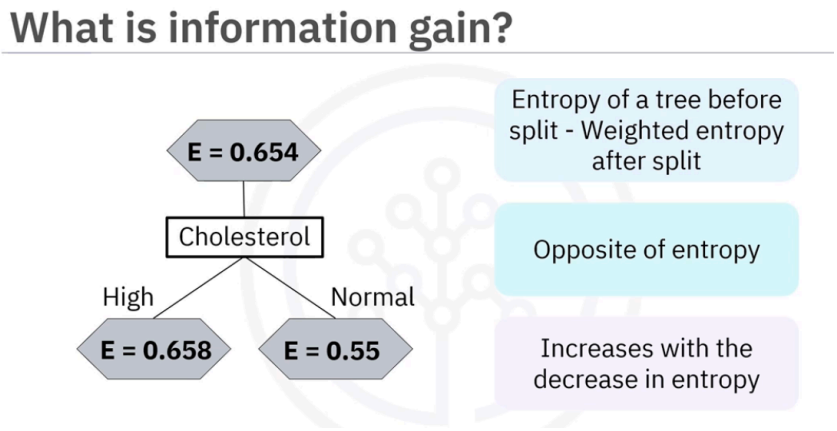


1. Entropy



- Measures **uncertainty/disorder** in a node.
- Formula:
  Entropy=−∑pilog2(pi)Entropy = - \sum p_i \log_2(p_i)
- **Entropy = 0** → pure node (all same class).
- **Entropy = 1** → maximum randomness (equal classes).

2. Information Gain



- IG=Entropybefore−EntropyafterIG = Entropy_{before} - Entropy_{after}

- High IG = better split (reduces uncertainty).

3. Gini Impurity (alternative metric)

- Measures probability of misclassification.
- Formula:
  Gini=1−∑pi2Gini = 1 - \sum p_i^2

✅ Advantages of Decision Trees

- Easy to **visualize** → highly interpretable.
- Shows **feature importance** via splits.
- Handles both **numerical and categorical data**.
- Non-linear decision boundaries possible.

📌 Key Takeaways

- Decision Tree = **recursive partitioning** of data.
- Built using **best splits** (Entropy, Info Gain, Gini).
- **Pruning** prevents overfitting, improves generalization.
- Widely used for classification tasks due to **clarity + interpretability**.

Here are **well-structured notes with clean code snippets** from your provided text on **Decision Trees**.
I've organized them for quick learning and reference.

📘 Decision Trees – Notes & Code

🎯 Objectives

After completing this lab, you will be able to:

- Develop a classification model using Decision Tree Algorithm.
- Apply Decision Tree classification on a real-world dataset.

🔍 Introduction

- Decision Trees are powerful for **classification** and **decision-making**.
- Dataset: **Drug prediction** based on patient health parameters.
- Features: `Age, Sex, BP, Cholesterol, Na_to_K`
- Target: `Drug (DrugA, DrugB, DrugC, DrugX, DrugY)`

📦 Importing Libraries

```
# Install required packages (if needed)

!pip install numpy==2.2.0 pandas==2.2.3 scikit-learn==1.6.0 matplotlib==3.9.3


# Import libraries

import numpy as np

import pandas as pd

from matplotlib import pyplot as plt
```

```
from sklearn.preprocessing import LabelEncoder

from sklearn.model_selection import train_test_split

from sklearn.tree import DecisionTreeClassifier, plot_tree

from sklearn import metrics

import warnings

warnings.filterwarnings('ignore')

%matplotlib inline
```

📂 Load Dataset

```
# Load dataset from IBM Cloud storage

path = 'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/' \

       'IBMDeveloperSkillsNetwork-ML0101EN-SkillsNetwork/labs/Module%203/data/drug200.csv'

my_data = pd.read_csv(path)

my_data.head()
```

🛠️ Data Analysis & Preprocessing

Dataset Info

```
my_data.info()
```

➡ 200 rows, 6 columns, with **categorical features** (`Sex`, `BP`, `Cholesterol`).

Label Encoding

```
label_encoder = LabelEncoder()

my_data['Sex'] = label_encoder.fit_transform(my_data['Sex'])

my_data['BP'] = label_encoder.fit_transform(my_data['BP'])

my_data['Cholesterol'] = label_encoder.fit_transform(my_data['Cholesterol'])

my_data.head()
```

Mappings

- `Sex`: M → 1, F → 0
- `BP`: High → 0, Low → 1, Normal → 2
- `Cholesterol`: High → 0, Normal → 1

Missing Values Check

```
my_data.isnull().sum()
```

➡ No missing values ✅

Encode Target Variable

```
custom_map = {'drugA':0,'drugB':1,'drugC':2,'drugX':3,'drugY':4}

my_data['Drug_num'] = my_data['Drug'].map(custom_map)
```

Correlation with Target

```
my_data.drop("Drug", axis=1).corr()['Drug_num']
```

➡ Strongest correlation:

- `Na_to_K` → 0.589
- `BP` → 0.373

Class Distribution

```
category_counts = my_data['Drug'].value_counts()

plt.bar(category_counts.index, category_counts.values, color='blue')

plt.xlabel('Drug')

plt.ylabel('Count')

plt.title('Category Distribution')

plt.xticks(rotation=45)

plt.show()
```

➡ `DrugX` and `DrugY` dominate.

🌳 Modeling

Train-Test Split

```
X = my_data.drop(['Drug','Drug_num'], axis=1)

y = my_data['Drug']


X_trainset, X_testset, y_trainset, y_testset = train_test_split(
```

```
    X, y, test_size=0.3, random_state=32
)
```

Train Decision Tree

```
drugTree = DecisionTreeClassifier(criterion="entropy", max_depth=4)
drugTree.fit(X_trainset, y_trainset)
```

📊 Evaluation

Predictions & Accuracy

```
tree_predictions = drugTree.predict(X_testset)
print("Decision Tree's Accuracy:", metrics.accuracy_score(y_testset, tree_predictions))
```

✅ Accuracy: **98.33%**

🌲 Visualize Decision Tree

```
plot_tree(drugTree, filled=True, feature_names=X.columns, class_names=drugTree.classes_)
plt.show()
```

🔑 Decision Rules (from Tree)

- **Drug Y**: `Na_to_K > 14.627`
- **Drug A**: `Na_to_K <= 14.627, BP = High, Age <= 50.5`
- **Drug B**: `Na_to_K <= 14.627, BP = High, Age > 50.5`
- **Drug C**: `Na_to_K <= 14.627, BP = Low, Cholesterol <= High`
- **Drug X**: `Na_to_K <= 14.627, BP = Normal, Cholesterol = High`

🔄 Effect of Tree Depth

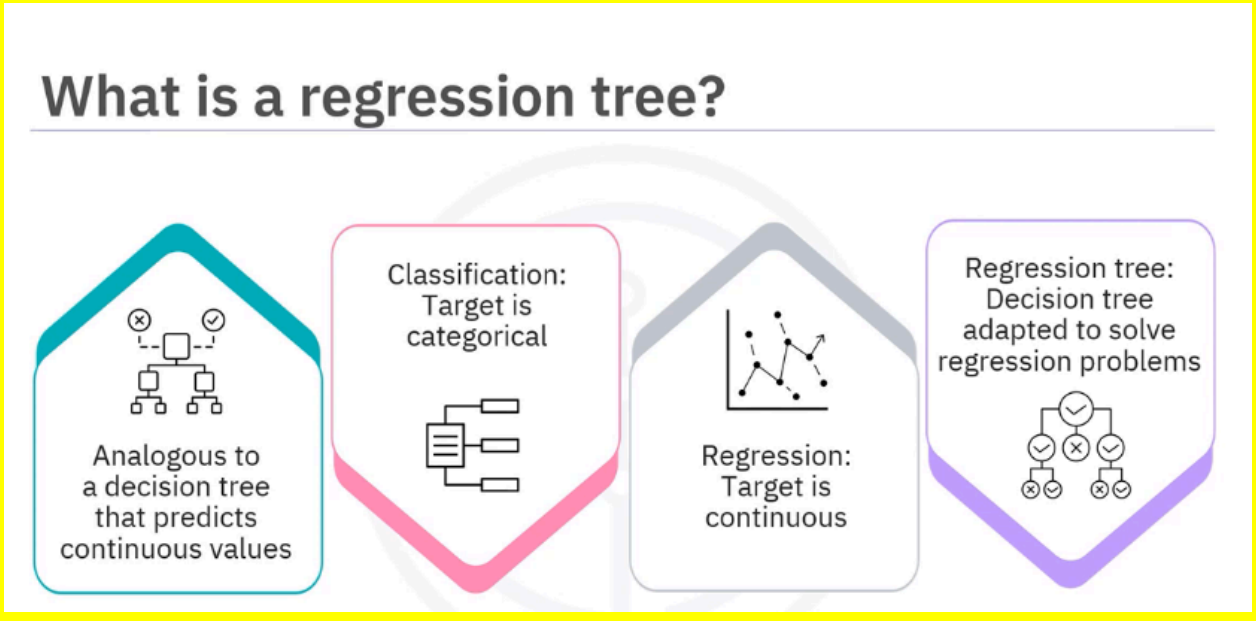
```
drugTree = DecisionTreeClassifier(criterion="entropy", max_depth=3)
drugTree.fit(X_trainset, y_trainset)
tree_predictions = drugTree.predict(X_testset)
print("Decision Tree's Accuracy:", metrics.accuracy_score(y_testset, tree_predictions))
```

➡ Accuracy remains **98.33%** (same as depth=4).

✅ Summary

- Decision Trees classify patients into 5 drug categories.
- **Na_to_K** and **BP** are the most influential features.
- Achieved **98.3% accuracy** with entropy-based Decision Tree.
- Even reducing tree depth maintained accuracy → simpler tree, same performance.

📌 Regression Trees – Notes



🔹 What is a Regression Tree?

- A **regression tree** is similar to a decision tree but predicts **continuous values** instead of categorical classes.
- **Key difference**:
  - **Classification Tree** → Target = categorical (e.g., spam/not spam, drug A/B).
  - **Regression Tree** → Target = continuous (e.g., salary, temperature, revenue).

🔹 **Comparison: Classification vs Regression Trees**

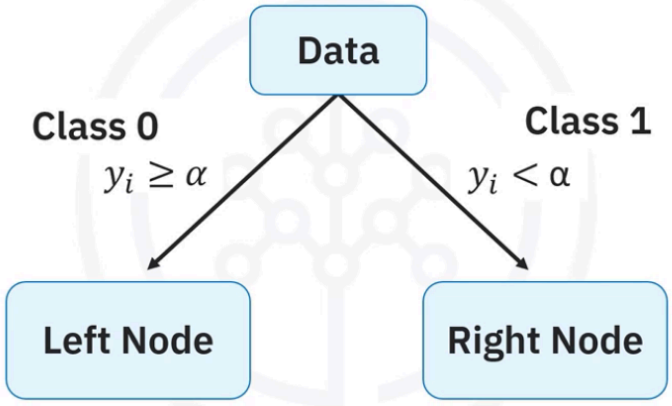
Classification versus regression trees

	Classification Trees	Regression Trees
Objective	Classify data into discrete sets	Predict continuous target variable
Target Variable	Categorical	Float
Splitting Criterion	Gini impurity or entropy	Variance reduction
Prediction at Leaf Nodes	Class label majority vote	Average value of target values
Example Use Cases	Spam detection, image classification, medical diagnosis	Predicting revenue, temperatures, wildfire risk

---

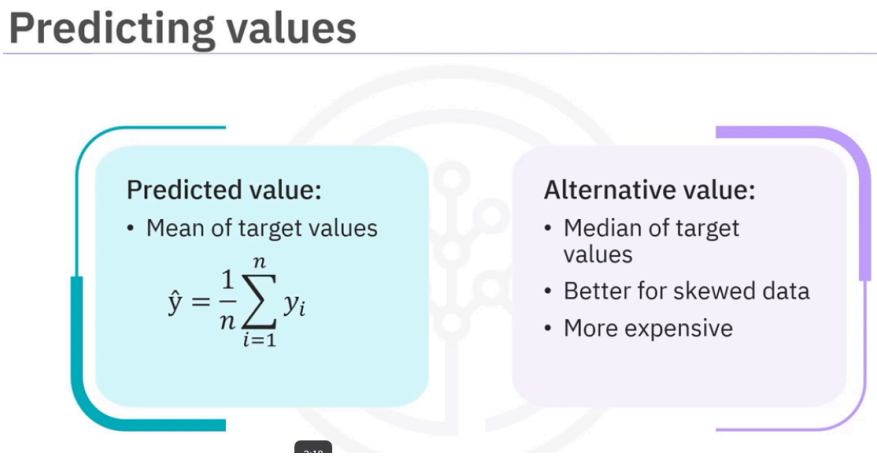
🔹 **How Regression Trees Work**

Creating regression trees

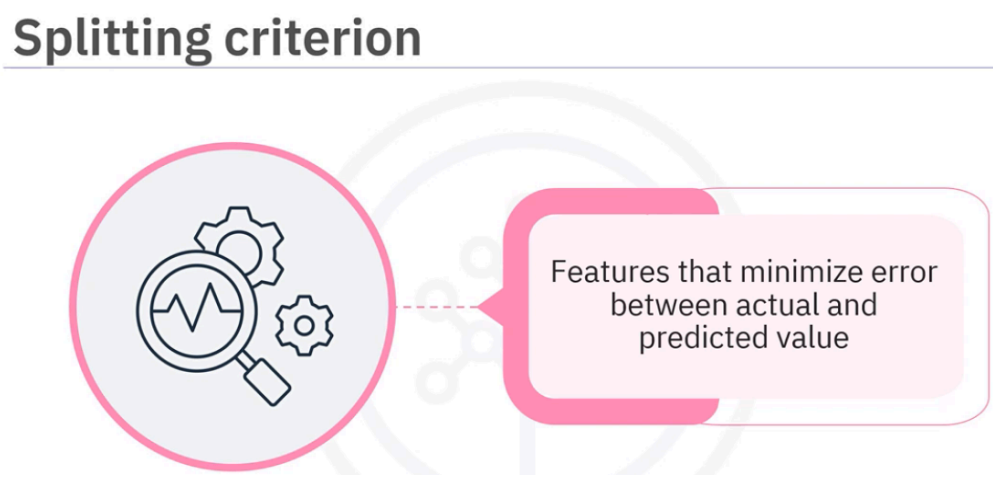


1. **Recursive Splitting** – Split data into subsets repeatedly.
2. **Prediction at Leaf Node** – Average (or median) of target values.
3. **Split Quality Measure** – Instead of entropy/info gain, use **Mean Squared Error (MSE)**.

- MSE = variance of target values in the node.
- Smaller variance → better split.

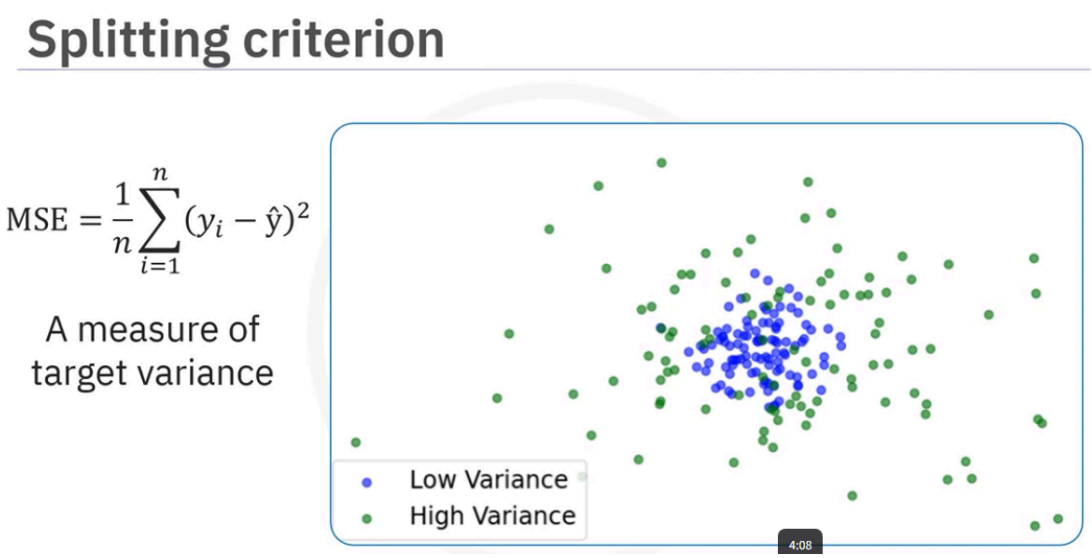


🔹 **Splitting Criteria**

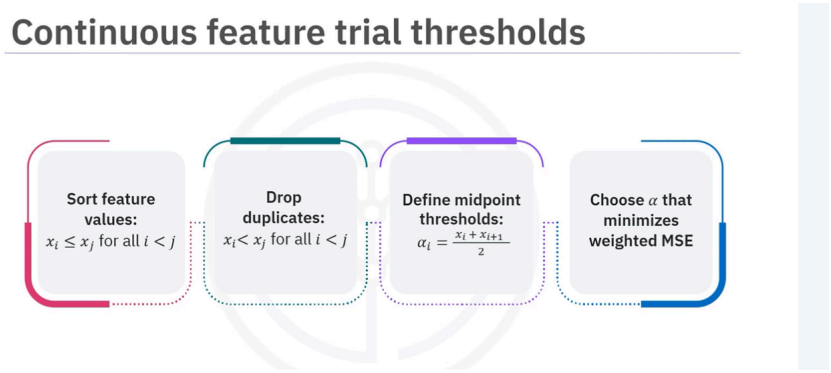


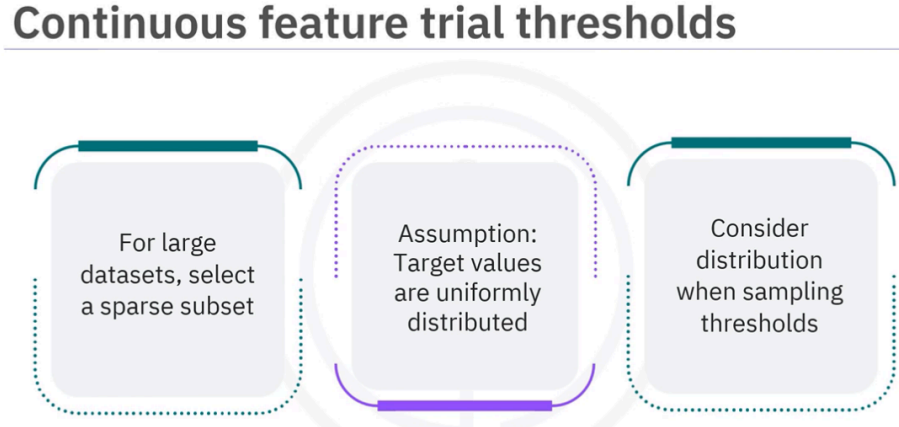
- **Weighted MSE** = average of left & right node MSEs (weighted by sample size).
- Best split = lowest weighted MSE.
- For **binary features**: simple split into 2 classes.

- For **multi-class features**: strategies like **one-vs-one** or **one-vs-all** are used.

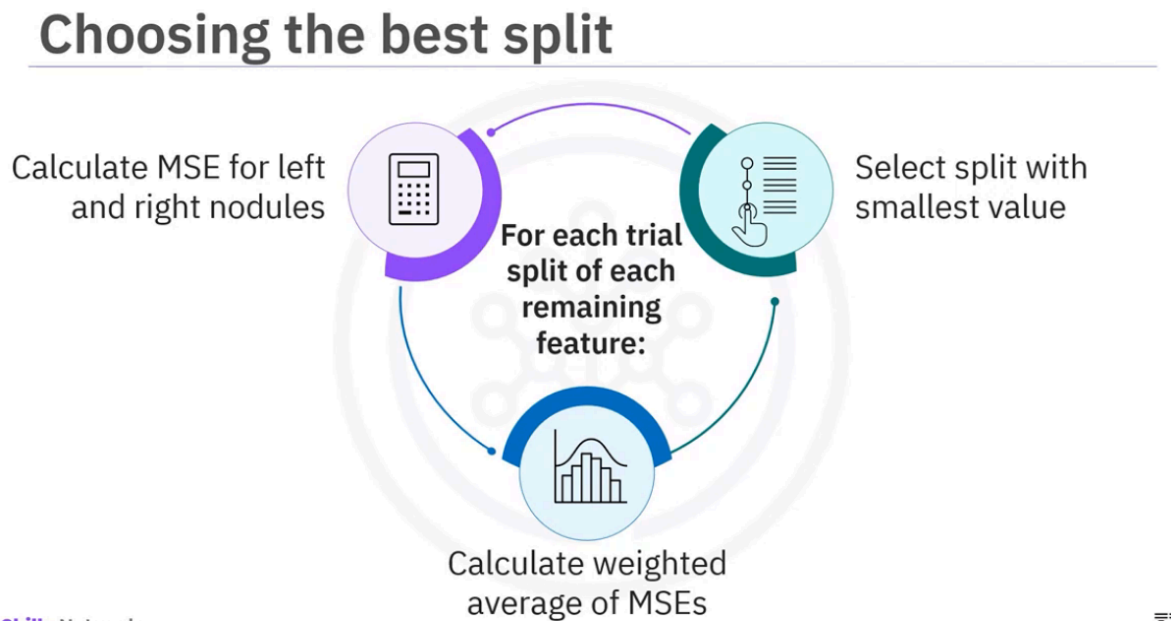


🔹 Threshold Selection (for Continuous Features)





1. Sort values in ascending order.
2. Remove duplicates.
3. Candidate thresholds (α_i) = midpoints between consecutive values.
4. Select threshold that minimizes **weighted MSE**.
    - Works well for small datasets.
    - For large datasets → sample thresholds for efficiency (trade-off: less accuracy).

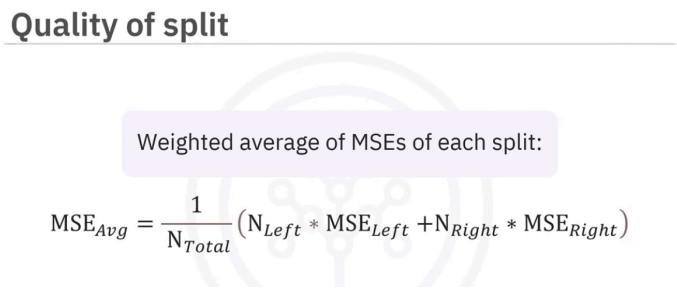


🔹 Advantages of Regression Trees

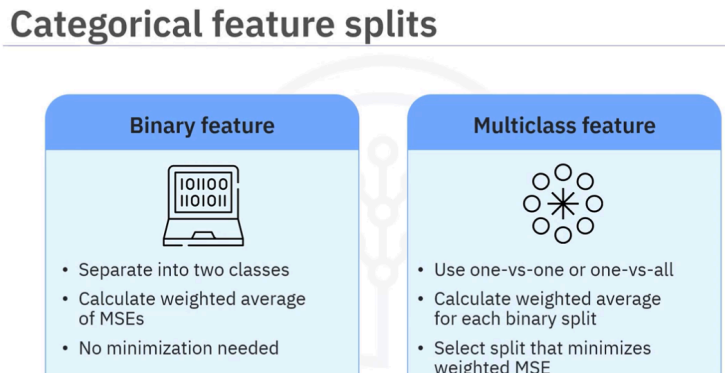
✅ Easy to interpret (tree structure).
✅ Works with both categorical & continuous features.
✅ Handles nonlinear relationships.
✅ Useful for real-world prediction problems.

🔹 Limitations

⚠️ Computationally expensive for very large datasets (threshold search).
⚠️ Risk of **overfitting** (pruning often required).



- **Key Formula**



Prediction at Node (ŷ):

y^=1N∑i=1Nyi\hat{y} = \frac{1}{N} \sum_{i=1}^{N} y_i

Split Quality (Weighted MSE):

MSE=nLNMSEL+nRNMSERMSE = \frac{n_L}{N} MSE_L + \frac{n_R}{N} MSE_R

✅ **In summary**: Regression Trees are decision-tree-based models that predict continuous values using **MSE** to evaluate splits. They split recursively until a stopping criterion is met, with predictions based on mean (or median) of target values in each leaf.

📘 Regression Trees – Taxi Tip Prediction

📅 **Date: September 22, 2025**

🎯 Objectives

By the end of this session, you should be able to:

- Perform **data preprocessing** using Scikit-Learn
- Model a **regression task** with Scikit-Learn
- Train a **Decision Tree Regressor**
- Evaluate model performance with **MSE** and **R² score**

📂 Dataset

- Source: [NYC TLC Trip Record Data]
- Task: Predict **tip_amount** (target) from trip-related features.

Each row = one taxi trip, with **13 features** including:

- VendorID, passenger_count, trip_distance, RatecodeID,
- PULocationID, DOLocationID, payment_type, fare_amount, tolls_amount, etc.

🛠️ Step 1: Import Libraries

```
!pip install numpy pandas matplotlib scikit-learn
```

```
from __future__ import print_function

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

%matplotlib inline


from sklearn.model_selection import train_test_split

from sklearn.preprocessing import normalize

from sklearn.metrics import mean_squared_error

from sklearn.tree import DecisionTreeRegressor


import warnings

warnings.filterwarnings('ignore')
```

📊 Step 2: Load Dataset

```
url = 'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/pu9kbeSaAtRZ7RxdJKX9_A/yellow-tripdata.csv'

raw_data = pd.read_csv(url)

raw_data.head()
```

👉 Target variable: **`tip_amount`**

📈 Step 3: Correlation Analysis

```
correlation_values = raw_data.corr()['tip_amount'].drop('tip_amount')

correlation_values.plot(kind='barh', figsize=(10, 6))
```

🔑 Observations:

- **High correlation**: `fare_amount`, `tolls_amount`, `trip_distance`
- **Low/no correlation**: `payment_type`, `VendorID`, `store_and_fwd_flag`, `improvement_surcharge`

🧹 Step 4: Preprocessing

```
# Extract target

y = raw_data[['tip_amount']].values.astype('float32')


# Drop target from features

proc_data = raw_data.drop(['tip_amount'], axis=1)


# Feature matrix

X = proc_data.values


# Normalize features

X = normalize(X, axis=1, norm='l1', copy=False)
```

🔀 Step 5: Train/Test Split

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)
```

🌳 Step 6: Decision Tree Regressor

```
dt_reg = DecisionTreeRegressor(
    criterion='squared_error',
    max_depth=8,
    random_state=35
)

dt_reg.fit(X_train, y_train)
```

📏 Step 7: Model Evaluation

```
y_pred = dt_reg.predict(X_test)

# MSE
mse_score = mean_squared_error(y_test, y_pred)
print(f"MSE score : {mse_score:.3f}")
```

R² score

r2_score = dt_reg.score(X_test, y_test)

print(f"R^2 score : {r2_score:.3f}")

📌 Example Output:

MSE score : 24.555

R^2 score : 0.028

📝 Practice Questions

Q1. Effect of increasing `max_depth` to 12?

- **MSE increases**, **R² becomes negative** → model **overfits**.

Q2. Top 3 features affecting `tip_amount`?

abs(correlation_values).sort_values(ascending=False)[:3]

✅ Output:

- `fare_amount` (0.20)
- `tolls_amount` (0.11)
- `trip_distance` (0.10)

Q3. Remove weak features & retrain

```
raw_data = raw_data.drop(
    ['payment_type', 'VendorID', 'store_and_fwd_flag', 'improvement_surcharge'],
    axis=1
)
```

- MSE and R^2 remain almost the same → these features add little value.

Q4. Effect of decreasing `max_depth` to 4?

- **MSE decreases**, **R^2 increases** → model generalizes better.

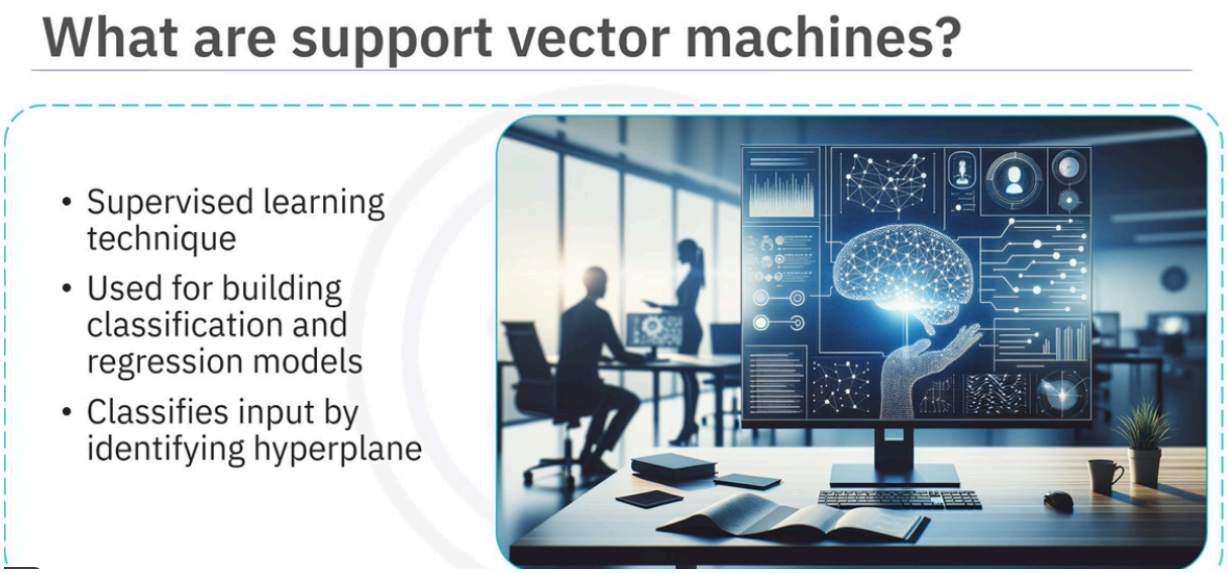
✅ Key Takeaways

- Regression trees can capture non-linear relationships.
- Too deep → **overfitting**, Too shallow → **underfitting**.
- Feature selection improves efficiency but not always accuracy.
- **Top predictors**: `fare_amount`, `tolls_amount`, `trip_distance`.

Here are **well-structured notes** on **Support Vector Machines (SVMs)** from your transcript 👇

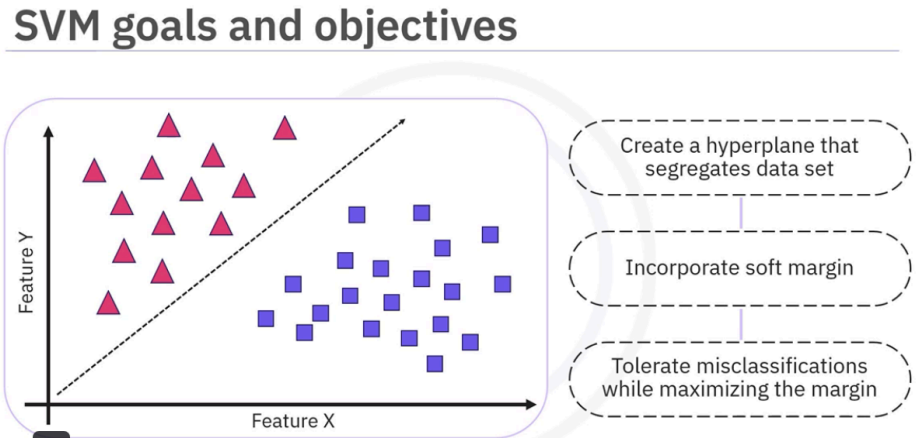
📌 Support Vector Machines (SVMs) – Notes

🔹 What is SVM?

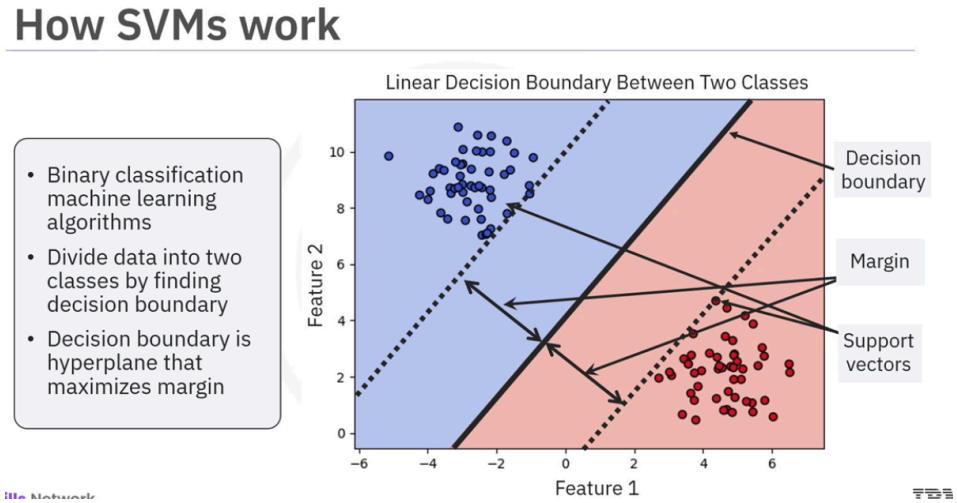


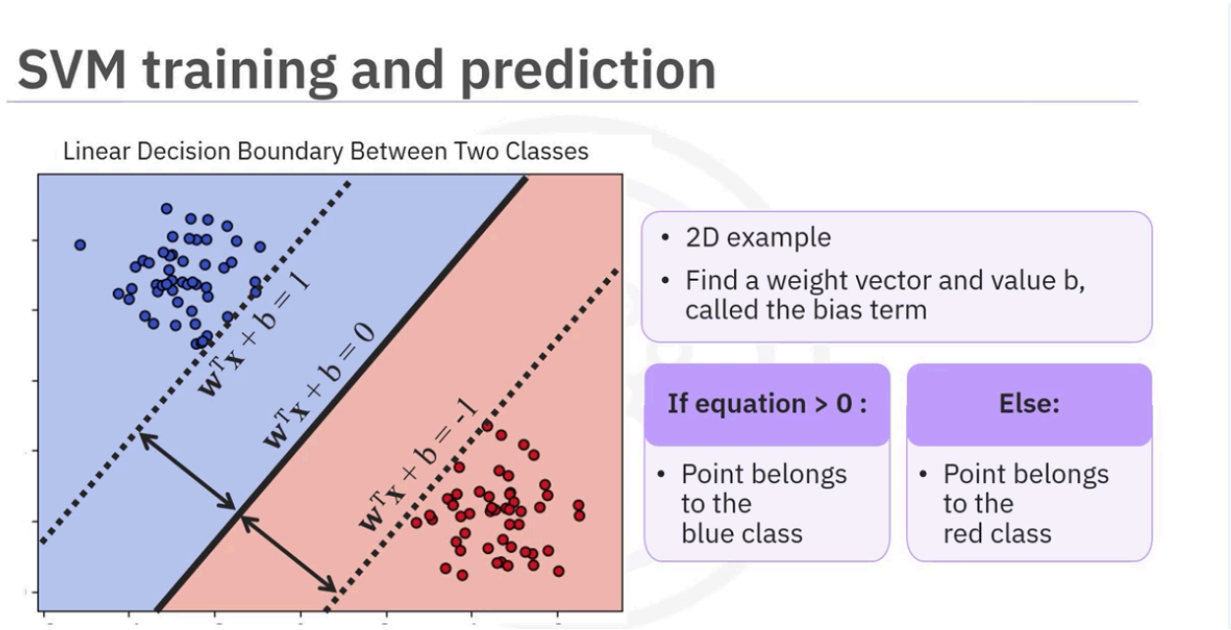
- **SVM (Support Vector Machine)** is a **supervised learning algorithm** used for:
    - **Classification (main use)**
    - **Regression (SVR – Support Vector Regression)**
- Works by mapping data into a **multidimensional space** and finding a **hyperplane** that separates classes with the **maximum margin**.

🔹 Key Concepts



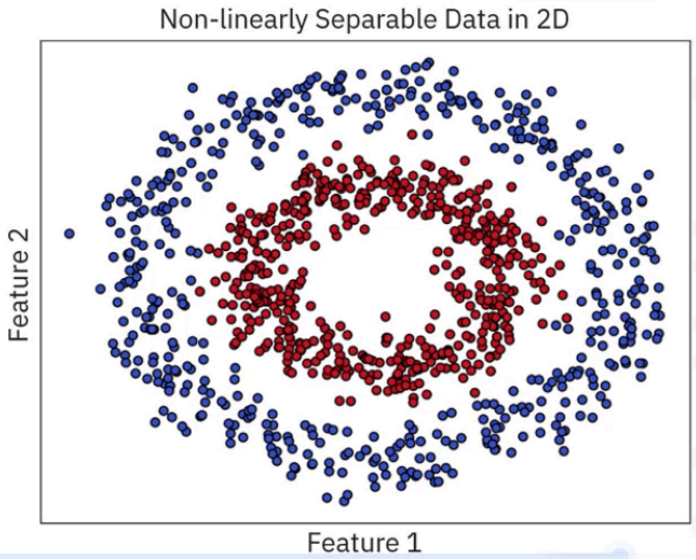
1. **Hyperplane** – Decision boundary that separates classes.
    - In 2D → a line
    - In 3D → a plane
    - In higher dimensions → a hyperplane
2. **Margin** – Distance between the hyperplane and the **nearest data points** from each class.
    - Larger margin → better generalization.
3. **Support Vectors** – The closest data points to the hyperplane; they define the margin.





4.

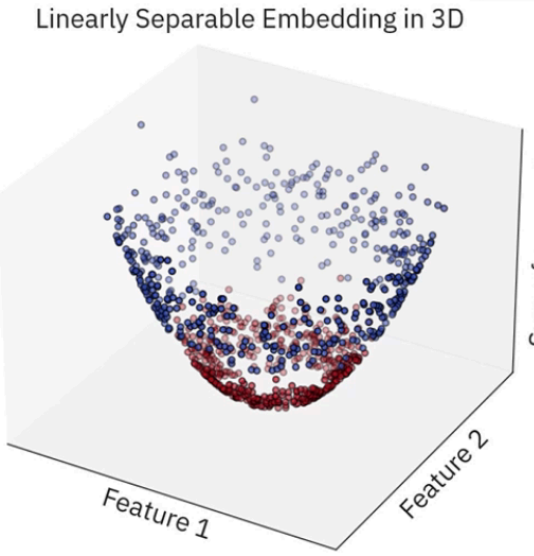
Nonlinearly separable classes



- 2D object
- Two non-overlapping classes
- Imagine these points as map contours

x_1 = Feature 1
x_2 = Feature 2

Parabolic 3D embedding



Sum of squared features

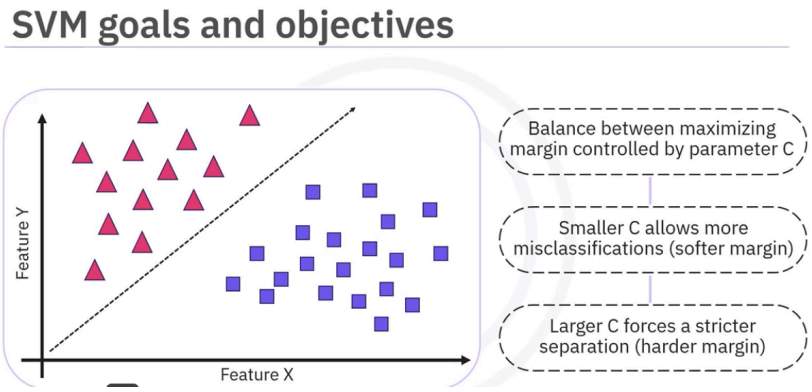
- Create 3D object to classify new cases
- Map data into a higher-dimensional space

$x_3 = x_1^2 + x_2^2$

Kernelling
$(x_1, x_2) \rightarrow (x_1, x_2, x_3)$

5. **Soft Margin (parameter C):**
   - Small **C** → softer margin (allows misclassifications, higher tolerance for noise).

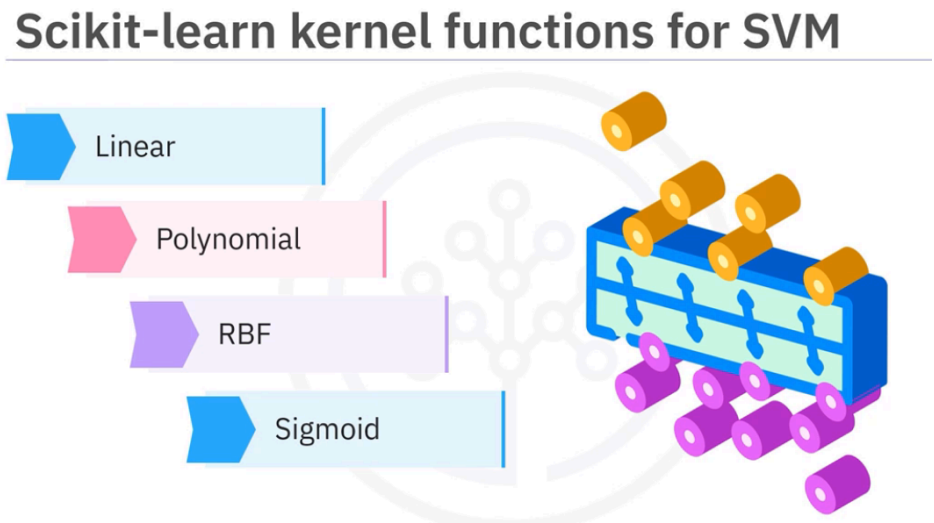
- Large **C** → harder margin (strict separation, fewer misclassifications).



🔹 SVM for Classification

- **Binary Classification:** Separates two classes by maximizing margin.
- Handles **linearly separable** data easily.
- For **non-linear data**, uses **kernel trick** (maps data to higher dimension).

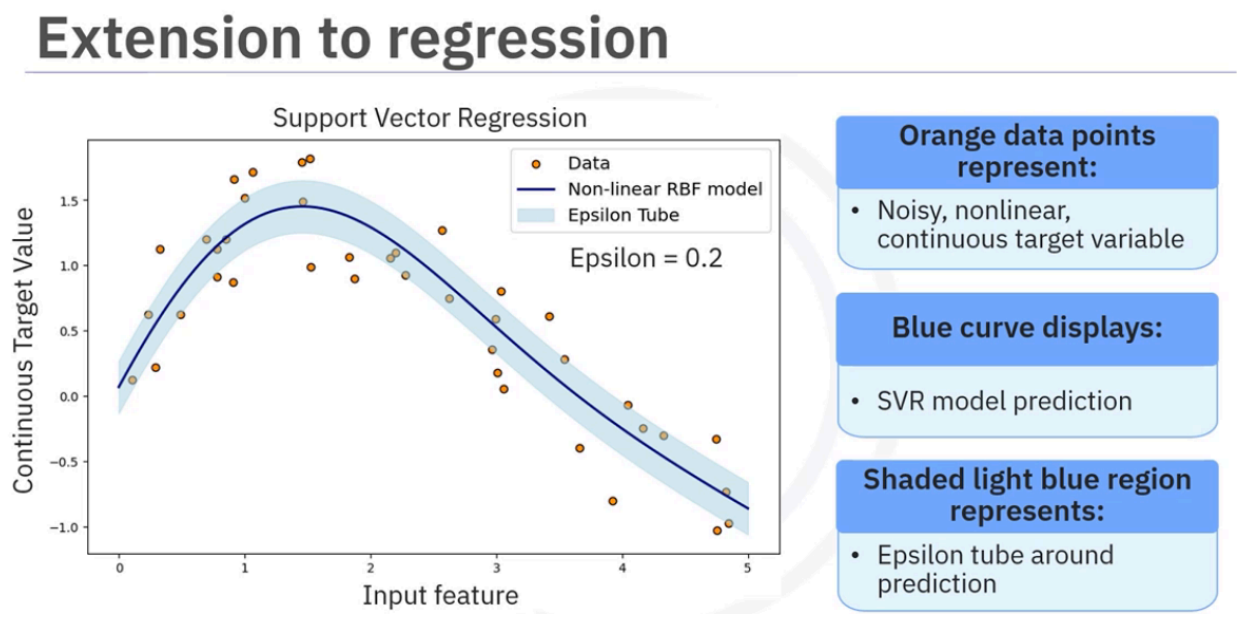
🔹 Kernel Functions in SVM



Used to transform non-linear data into higher dimensions for linear separation.

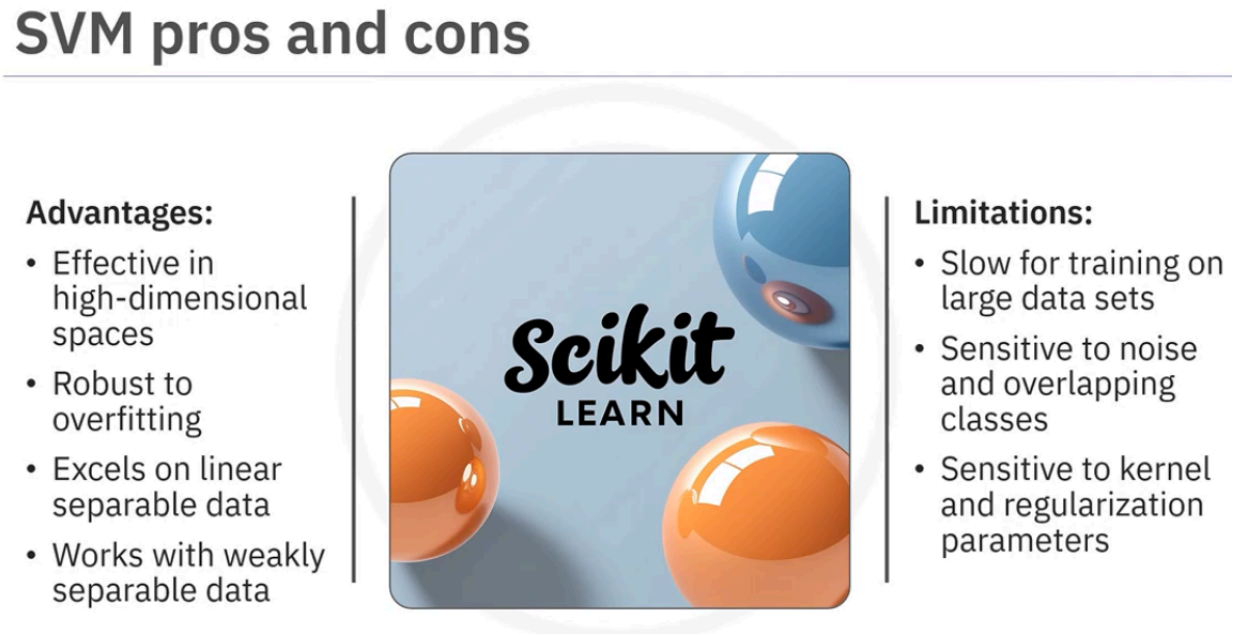
1. **Linear Kernel** – Default, works when data is linearly separable.
2. **Polynomial Kernel** – Maps data using polynomial functions.
3. **RBF (Radial Basis Function)** – Popular, measures similarity (high for close points, decreases exponentially with distance).
4. **Sigmoid Kernel** – Same function as logistic regression activation.

🔹 SVM for Regression (SVR)



- Predicts **continuous values** with a margin of tolerance (epsilon tube).
- **Epsilon (ε):** Defines the margin around the regression line.
    - Points inside tube → considered acceptable predictions (signal).
    - Points outside → treated as noise.

🔹 Advantages of SVM

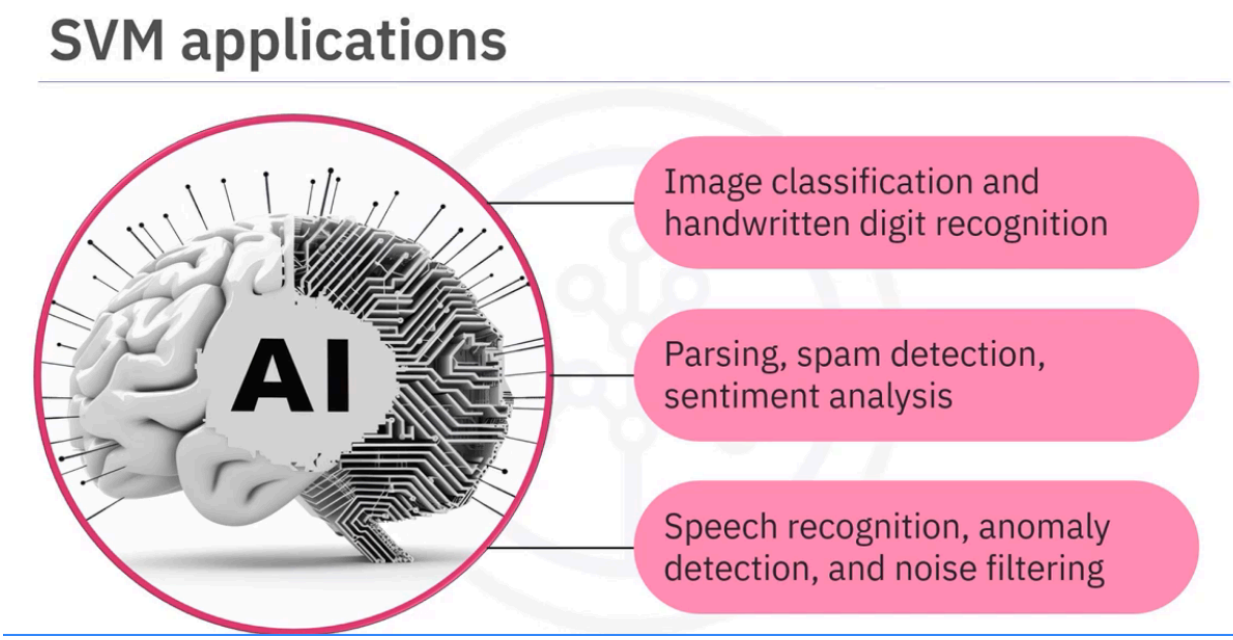


✅ Works well in **high-dimensional spaces**.
✅ Effective for **linear & weakly separable data**.
✅ Robust to **overfitting** (especially with proper kernel & C).
✅ Suitable for small/medium datasets.

🔹 Limitations of SVM

⚠️ **Slow** for training on large datasets.
⚠️ **Sensitive to noise** and overlapping classes.
⚠️ Performance depends heavily on **kernel choice** & parameter tuning (C, epsilon, gamma).

🔹 **Applications of SVM**



- **Image classification** (face recognition, handwritten digit recognition).
- **Text classification** (spam detection, sentiment analysis).
- **Speech recognition & NLP tasks**.
- **Anomaly detection** (fraud, intrusion).
- **Noise filtering** in data.

✅ **In summary**:
SVM builds classification and regression models by finding a **hyperplane** that maximizes margin. With **kernel tricks**, it handles non-linear data. It's powerful in **high-dimensional problems** like image & text classification but requires careful tuning and struggles with **large, noisy datasets**.

Here's a **concise and well-structured set of notes** with code from your provided text on **Credit Card Fraud Detection using Decision Trees and SVM**:

Credit Card Fraud Detection with Decision Trees and SVM

Date: September 22, 2025
Estimated Time: 30 minutes

1. Objectives

After completing this lab, you will be able to:

- Preprocess data in Python
- Model a classification problem using Scikit-Learn
- Train Decision Tree (DT) and Support Vector Machine (SVM) models
- Evaluate model performance using ROC-AUC

---

2. Introduction

- Binary classification:
    - `Class = 1` → Fraudulent transaction
    - `Class = 0` → Legitimate transaction
- Dataset is highly imbalanced:
    - 492 frauds out of 284,807 transactions (~0.172%)
- Data source: Kaggle Credit Card Fraud Detection

---

3. Libraries

Install necessary libraries (if needed)

```
!pip install pandas==2.2.3 scikit-learn==1.6.0 matplotlib==3.9.3


# Import libraries

import pandas as pd

import matplotlib.pyplot as plt

%matplotlib inline

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import normalize, StandardScaler

from sklearn.utils.class_weight import compute_sample_weight

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import roc_auc_score

from sklearn.svm import LinearSVC

import warnings

warnings.filterwarnings('ignore')
```

---

4. Load Dataset

```
url =
"https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSkillsNetwork-ML0101EN-SkillsNetwork/labs/Module%203/data/creditcard.csv"

raw_data = pd.read_csv(url)

raw_data.head()
```

---

5. Dataset Analysis

- Target variable: `Class`
- Highly imbalanced dataset

```
labels = raw_data.Class.unique()
sizes = raw_data.Class.value_counts().values


fig, ax = plt.subplots()
ax.pie(sizes, labels=labels, autopct='%1.3f%%')
ax.set_title('Target Variable Value Counts')
plt.show()
```

- Feature correlation with `Class`:

```
correlation_values = raw_data.corr()['Class'].drop('Class')
correlation_values.plot(kind='barh', figsize=(10, 6))
```

---

6. Data Preprocessing

- Standard scaling + L1 normalization
- Exclude `Time` variable (feature 0)

```
raw_data.iloc[:, 1:30] = StandardScaler().fit_transform(raw_data.iloc[:, 1:30])
data_matrix = raw_data.values
```

```
X = data_matrix[:, 1:30]  # Features

y = data_matrix[:, 30]    # Target


X = normalize(X, norm="l1")
```

---

7. Train/Test Split

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

---

8. Decision Tree Classifier

- Handle class imbalance using `sample_weight`

```
w_train = compute_sample_weight('balanced', y_train)


dt = DecisionTreeClassifier(max_depth=4, random_state=35)

dt.fit(X_train, y_train, sample_weight=w_train)
```

---

9. Support Vector Machine

- Handle class imbalance with `class_weight='balanced'`

```
svm = LinearSVC(class_weight='balanced', random_state=31, loss="hinge", fit_intercept=False)
svm.fit(X_train, y_train)
```

---

10. Model Evaluation

Decision Tree ROC-AUC

```
y_pred_dt = dt.predict_proba(X_test)[:,1]
roc_auc_dt = roc_auc_score(y_test, y_pred_dt)
print(f'Decision Tree ROC-AUC score : {roc_auc_dt:.3f}')
```

SVM ROC-AUC

```
y_pred_svm = svm.decision_function(X_test)
roc_auc_svm = roc_auc_score(y_test, y_pred_svm)
print(f'SVM ROC-AUC score: {roc_auc_svm:.3f}')
```

- Example results:
    - Decision Tree: 0.939
    - SVM: 0.986

---

11. Feature Selection Exercise

- Top 6 correlated features with `Class`:

```
top_features = abs(raw_data.corr()['Class']).drop('Class').sort_values(ascending=False)[:6]
```

print(top_features)

['V17','V14','V12','V10','V16','V3']

- Use these features for training:

X = data_matrix[:, [3, 10, 12, 14, 16, 17]]

- Observations:
    - Decision Tree improves ROC-AUC after feature selection
    - SVM may decrease in ROC-AUC because it benefits from more features

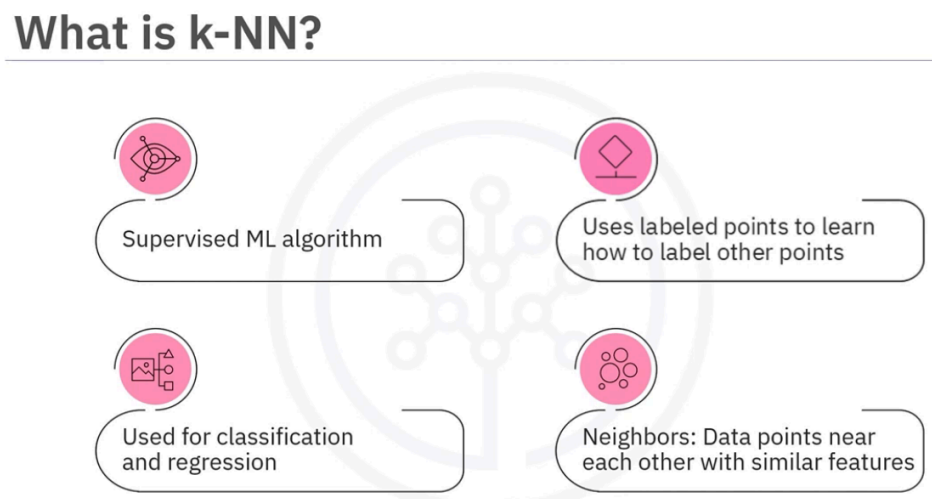
---

12. Key Takeaways

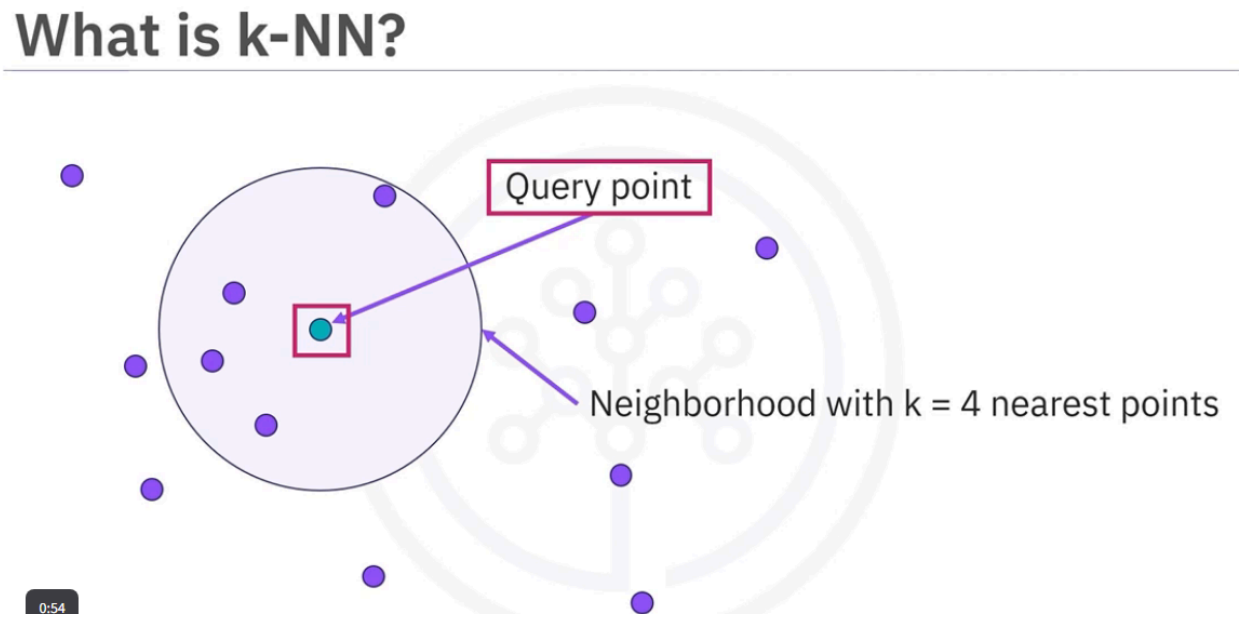
- SVM performs better with higher feature dimensionality
- Decision Trees benefit from feature selection
- ROC-AUC is a suitable metric for imbalanced datasets

Author: Abishek Gagneja
Other Contributors: Jeff Grossman

📌 K-Nearest Neighbors (KNN) – Notes



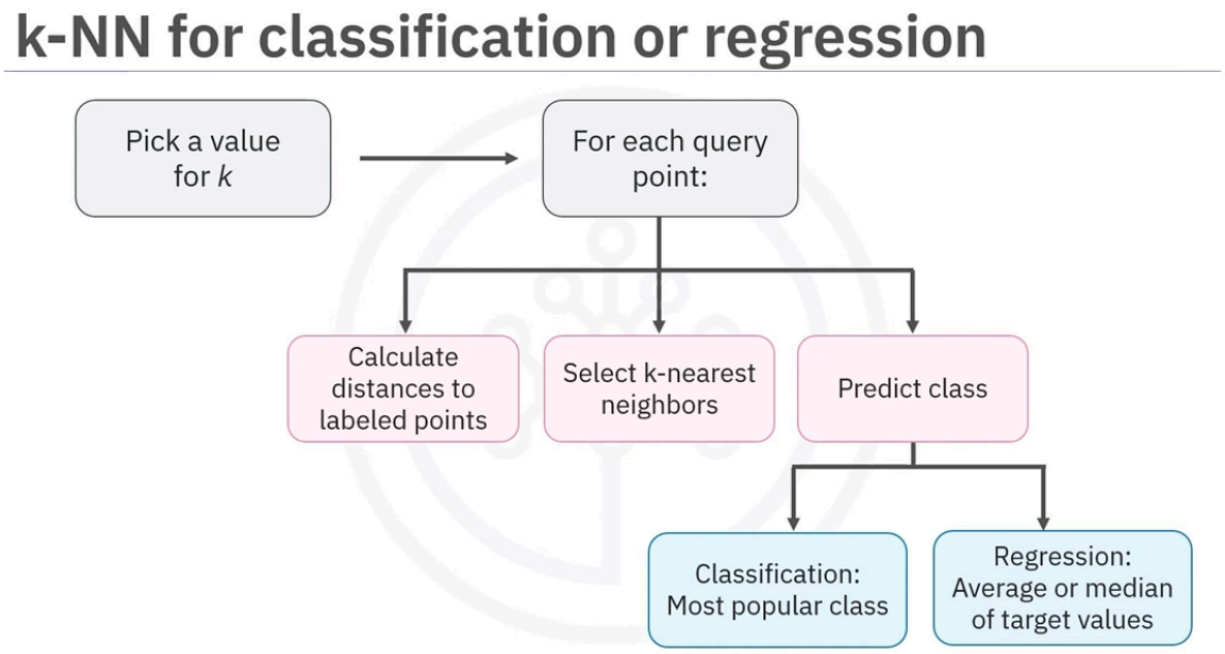
- 🔹 **What is KNN?**



- **KNN (K-Nearest Neighbors)** is a **supervised learning algorithm** used for:
  - **Classification**
  - **Regression**
- Based on the principle: **"Similar points are near each other."**

- Predictions are made by looking at the **K closest neighbors** of a query point.

🔹 How KNN Works



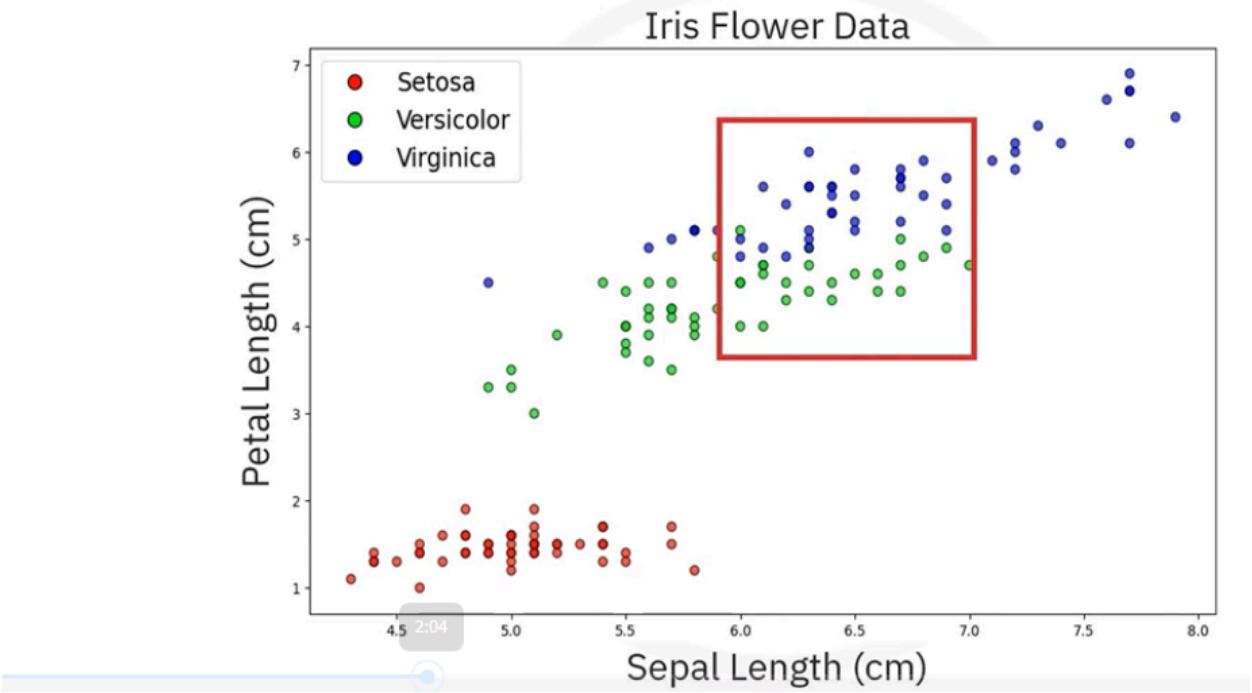
1. Choose a value of **K**.
2. Compute **distance** from query point to all training points (commonly Euclidean distance).
3. Select the **K nearest neighbors**.
4. Predict:
    - **Classification:** Majority vote among neighbors.
    - **Regression:** Mean or median of neighbors' values.

k-NN for classification

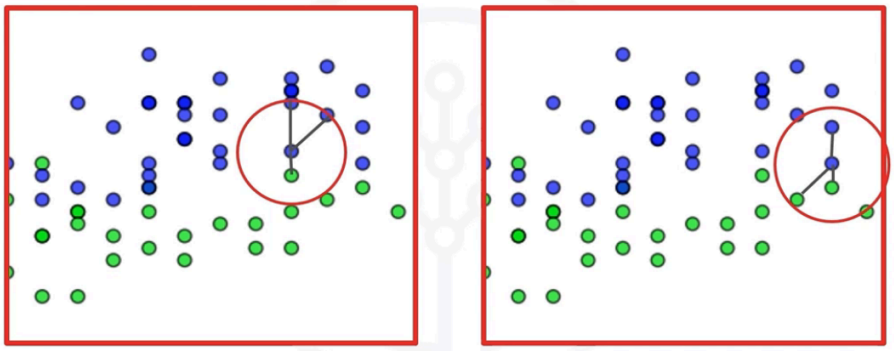
	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target	iris_name
0	5.1	3.5	1.4	0.2	0	setosa
1	4.9	3.0	1.4	0.2	0	setosa
2	4.7	3.2	1.3	0.2	0	setosa
3	4.6	3.1	1.5	0.2	0	setosa
4	5.0	3.6	1.4	0.2	0	setosa
...	...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	2	virginica
146	6.3	2.5	5.0	1.9	2	virginica
147	6.5	3.0	5.2	2.0	2	virginica

- **Effect of K**

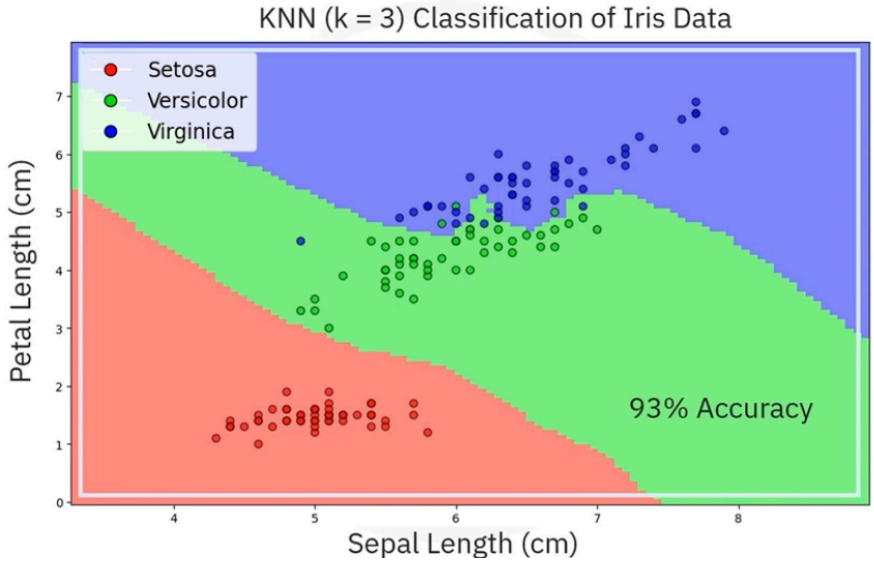
Determining classes with k = 3



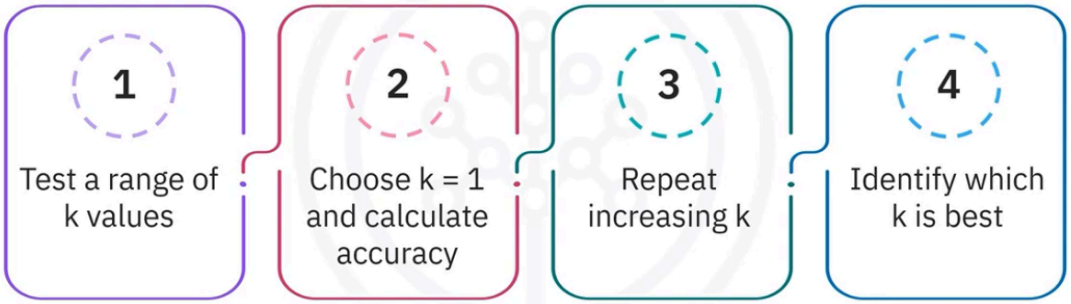
Determining classes with k = 3



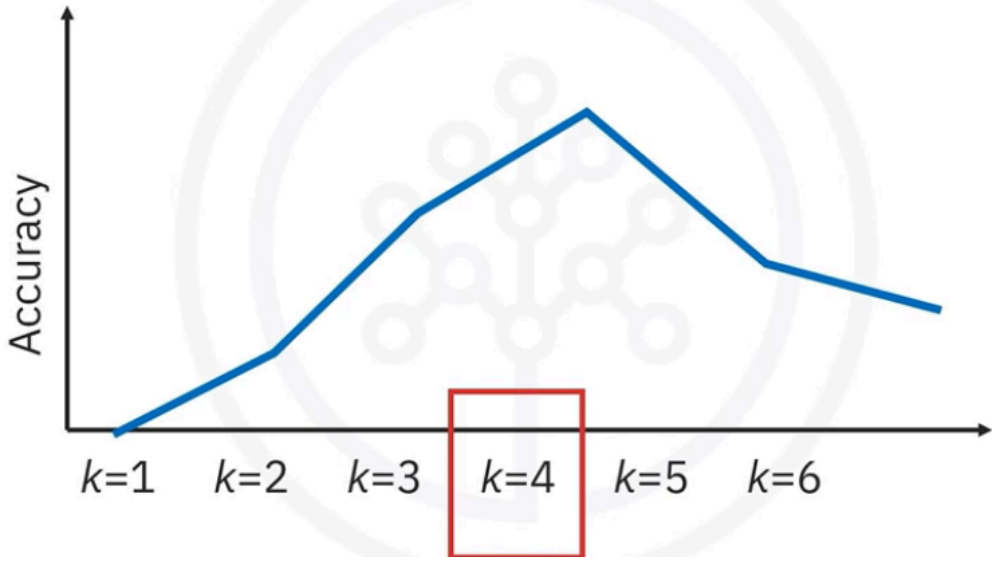
k-NN decision boundary



Finding the optimal k



Finding the optimal k

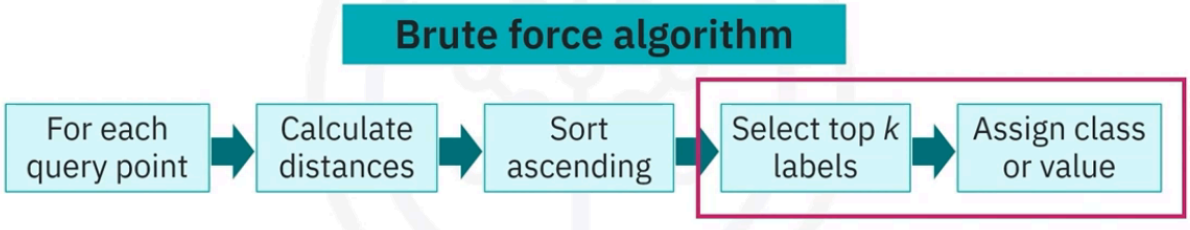


- **Small K** → Overfitting (model too sensitive, fluctuates with noise).
- **Large K** → Underfitting (too smooth, loses finer detail).
- **Optimal K** → Found by testing multiple K values using a validation/test set.

🔹 **Important Concepts**

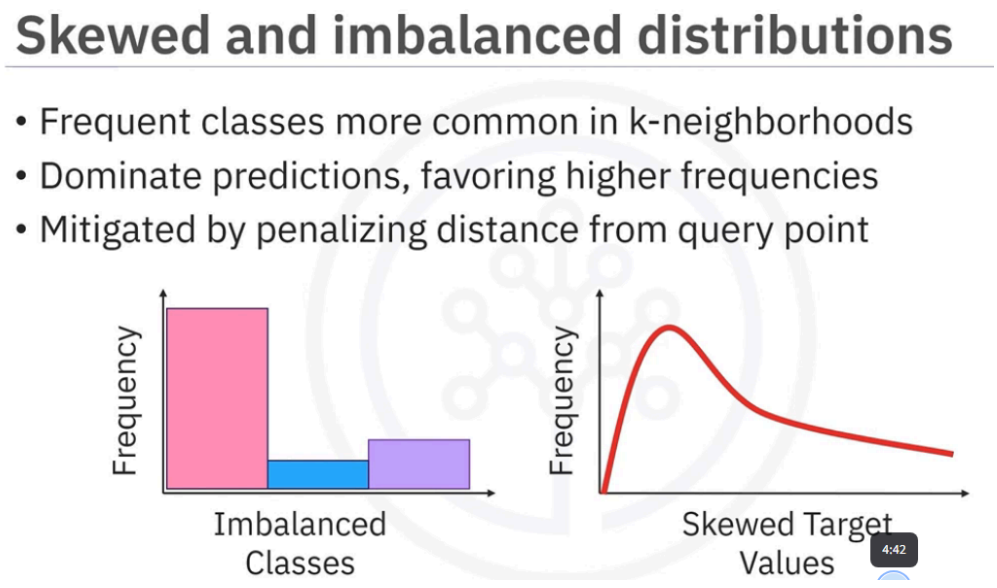
K-NN is a lazy learner

- Memorizes training data
- Makes predictions based on distance to training data points



- **Lazy learner:** KNN does not build an explicit model; it stores training data and computes distances at query time.
- **Decision boundary:** Shapes depend on value of K; e.g., with K=3 on iris dataset → 93% accuracy.
- **Distance weighting:** Closer neighbors can be given higher weight to reduce skewed predictions.

🔹 **Challenges & Solutions**

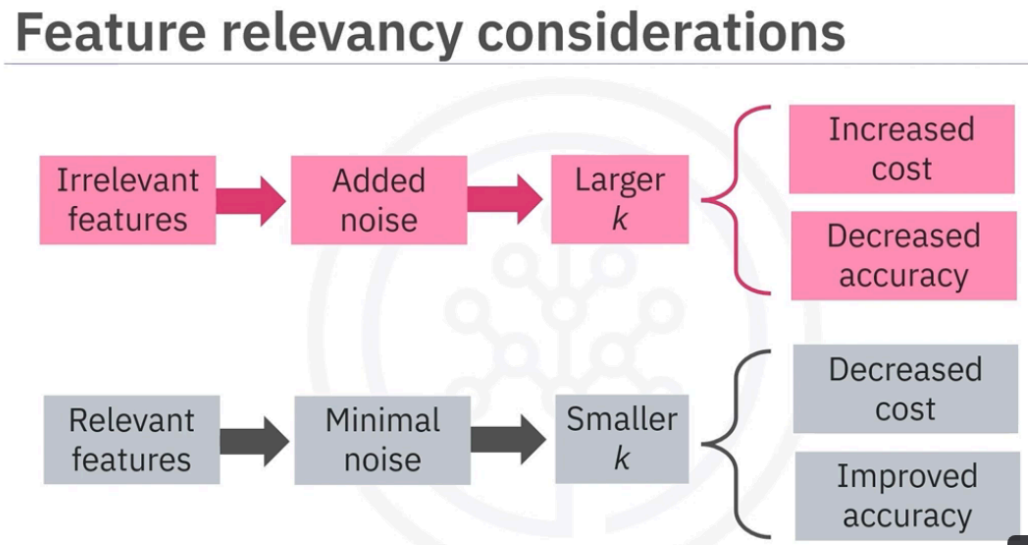


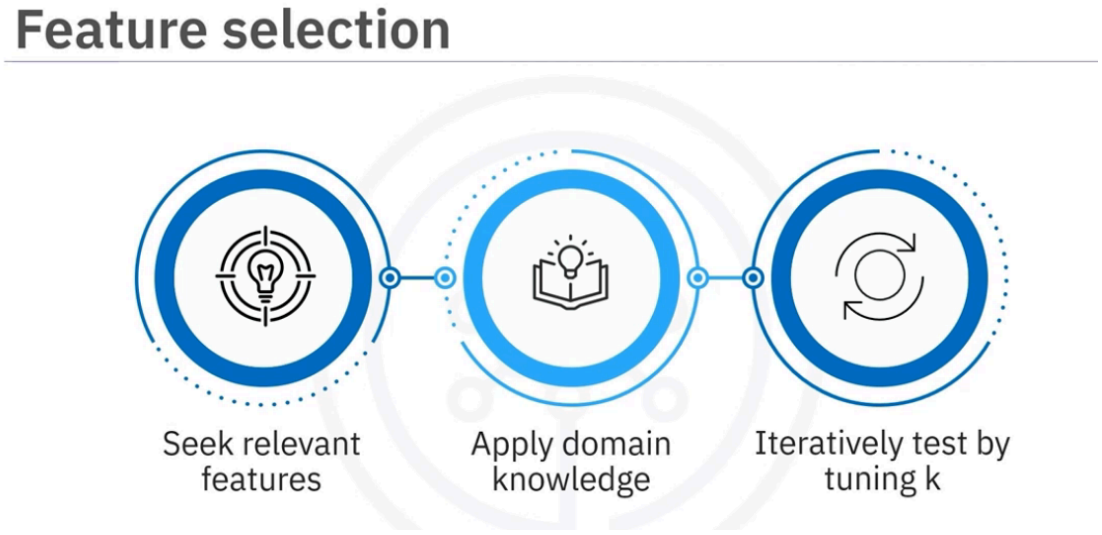
1. **Skewed class distribution** – Frequent classes dominate → use **distance-weighted voting**.

2. **Feature scaling** – Features with larger ranges dominate distance → use **standardization/normalization**.

3. **Irrelevant/noisy features** – Add noise, increase computation → remove redundant features.

4. **Computational cost** – Prediction requires computing distances for all points → not efficient for very large datasets.





🔹 Advantages of KNN

✅ Simple and intuitive.
✅ Works for both **classification & regression**.
✅ No training phase (useful for small datasets).
✅ Flexible (choice of distance metrics, weighting, etc.).

🔹 Limitations of KNN

⚠️ **Slow at prediction** (must compare with all training data).
⚠️ Requires careful **choice of K**.
⚠️ Sensitive to **irrelevant features & scaling**.
⚠️ Struggles with **large datasets** and **high-dimensional data (curse of dimensionality)**.

🔹 Applications of KNN

- **Iris flower classification** (classic dataset).
- **Image recognition** (e.g., digit classification).
- **Recommender systems** (finding similar users/items).
- **Medical diagnosis** (disease classification).

✅ **In summary:**
KNN is a **non-parametric, instance-based algorithm**. It works by storing training data and predicting labels of new points using the majority class (classification) or average (regression) of the **K nearest neighbors**. Its performance depends heavily on **K selection, feature scaling, and data relevance**.

Got it ✅ I'll create clean, well-structured **notes with explanations and runnable code** from your provided KNN_Classification lab. These will be revision-friendly and easy to reuse for projects or exams.

📘 Notes: K-Nearest Neighbors (KNN) Classification

🔹 Objectives

After this lab, you'll be able to:

- Use **K-Nearest Neighbors (KNN)** to classify data.
- Apply KNN classifier on a **real-world dataset**.

We will load customer demographic data, train a KNN model, and predict service categories.

🔹 Step 1: Import Required Libraries

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns


from sklearn.preprocessing import StandardScaler

from sklearn.model_selection import train_test_split

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import accuracy_score


%matplotlib inline
```

🔹 Step 2: Load Dataset

Dataset: Telecom customers segmented into 4 groups (`custcat` target).
Service categories:

1. Basic Service
2. E-Service
3. Plus Service
4. Total Service

```
# Load data
df = pd.read_csv(
    'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/'
    'IBMDeveloperSkillsNetwork-ML0101EN-SkillsNetwork/labs/Module%203/data/'
    'teleCust1000t.csv'
)

# Preview first 5 rows
df.head()
```

🔹 Step 3: Explore Data

Class distribution

```
df['custcat'].value_counts()
```

📊 Output:

- Plus Service → 281
- Basic Service → 266
- Total Service → 236
- E-Service → 217

Dataset is **fairly balanced**.

Correlation Heatmap

```
correlation_matrix = df.corr()
plt.figure(figsize=(10,8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt='.2f', linewidths=0.5)
plt.show()
```

Feature importance (by correlation with `custcat`)

```
correlation_values = abs(df.corr()['custcat'].drop('custcat')).sort_values(ascending=False)
print(correlation_values)
```

📌 Insight:

- `ed` and `tenure` have highest influence.
- `retire` and `gender` least influence.

🔹 Step 4: Prepare Data

Separate features and target

```
X = df.drop('custcat', axis=1)  # input features
```

y = df['custcat'] # target variable

Normalize features

KNN uses distance → normalization is critical.

X_norm = StandardScaler().fit_transform(X)

Train-Test Split

X_train, X_test, y_train, y_test = train_test_split(X_norm, y, test_size=0.2, random_state=4)

🔹 Step 5: KNN Classification

Train with k=3

k = 3

knn_model = KNeighborsClassifier(n_neighbors=k).fit(X_train, y_train)

Prediction

yhat = knn_model.predict(X_test)

Accuracy

print("Test set Accuracy:", accuracy_score(y_test, yhat))

✅ Example Output: `0.315`

Exercise: Train with k=6

```
k = 6

knn_model_6 = KNeighborsClassifier(n_neighbors=k).fit(X_train, y_train)

yhat6 = knn_model_6.predict(X_test)


print("Test set Accuracy:", accuracy_score(y_test, yhat6))
```

✅ Example Output: `0.31`

🔹 Step 6: Find Best k

We try multiple values of k and track accuracy.

```
Ks = 10

acc = np.zeros((Ks))

std_acc = np.zeros((Ks))


for n in range(1, Ks+1):

    knn_model_n = KNeighborsClassifier(n_neighbors=n).fit(X_train, y_train)

    yhat = knn_model_n.predict(X_test)

    acc[n-1] = accuracy_score(y_test, yhat)

    std_acc[n-1] = np.std(yhat == y_test) / np.sqrt(yhat.shape[0])
```

Plot Accuracy vs k

```
plt.plot(range(1, Ks+1), acc, 'g')

plt.fill_between(range(1, Ks+1), acc - 1*std_acc, acc + 1*std_acc, alpha=0.1)
```

```
plt.legend(('Accuracy value', 'Standard Deviation'))

plt.ylabel('Model Accuracy')

plt.xlabel('Number of Neighbors (K)')

plt.tight_layout()

plt.show()


print("Best accuracy:", acc.max(), "with k =", acc.argmax()+1)
```

✅ Example Output:
Best accuracy = **0.34** with k = **9**

🔹 Key Takeaways

- KNN works by finding **nearest neighbors** based on distance.
- **Normalization** ensures fair contribution of features.
- Accuracy depends on the **choice of k**.
- For this dataset, **k=9** gave the best results.

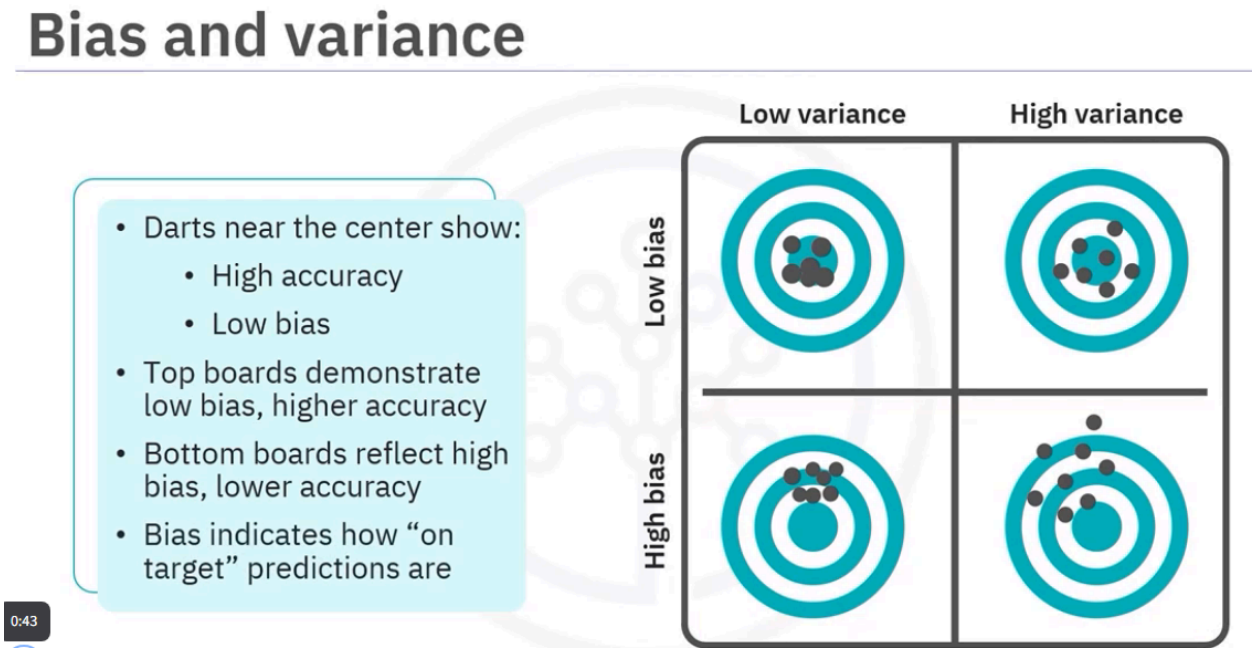
Here are **clear, structured notes** on **Bias, Variance, and Ensemble Models** from your transcript:

📌 Bias, Variance, and Ensemble Models

🎯 Key Learning Objectives

- Analyze the impact of **bias** and **variance** on accuracy and precision.
- Explain the **bias-variance tradeoff** in model complexity.
- Evaluate techniques to mitigate bias and variance.
- Understand **bagging** and **boosting** methods.

🔹 Bias and Variance Basics

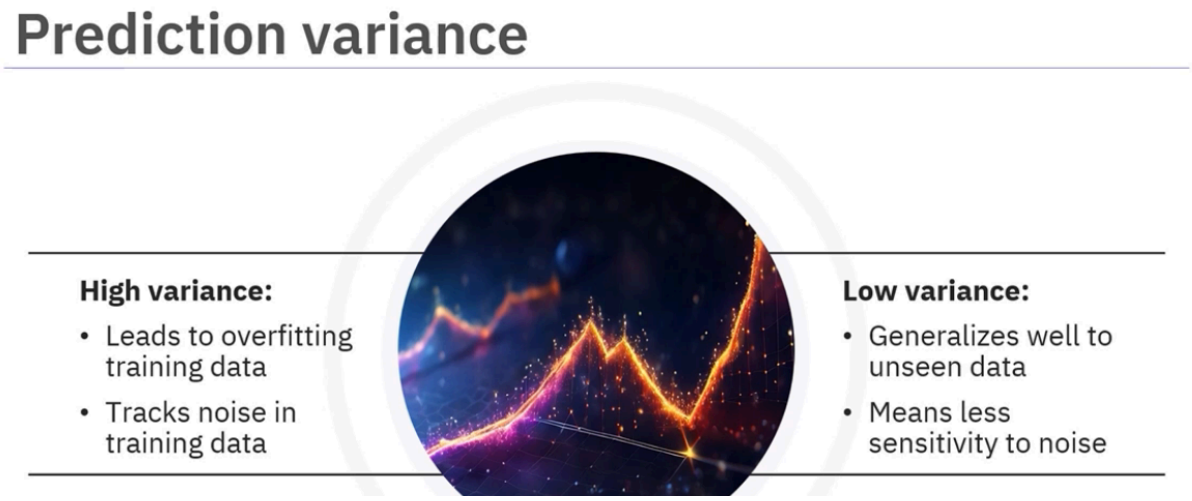


🎯 Bias

- **Definition**: Error due to oversimplification of the model (how far predictions are from actual values).
- **High Bias** → Model is too simple → **Underfitting**.

- **Low Bias** → Predictions closer to reality → More accurate.

🎯 Variance

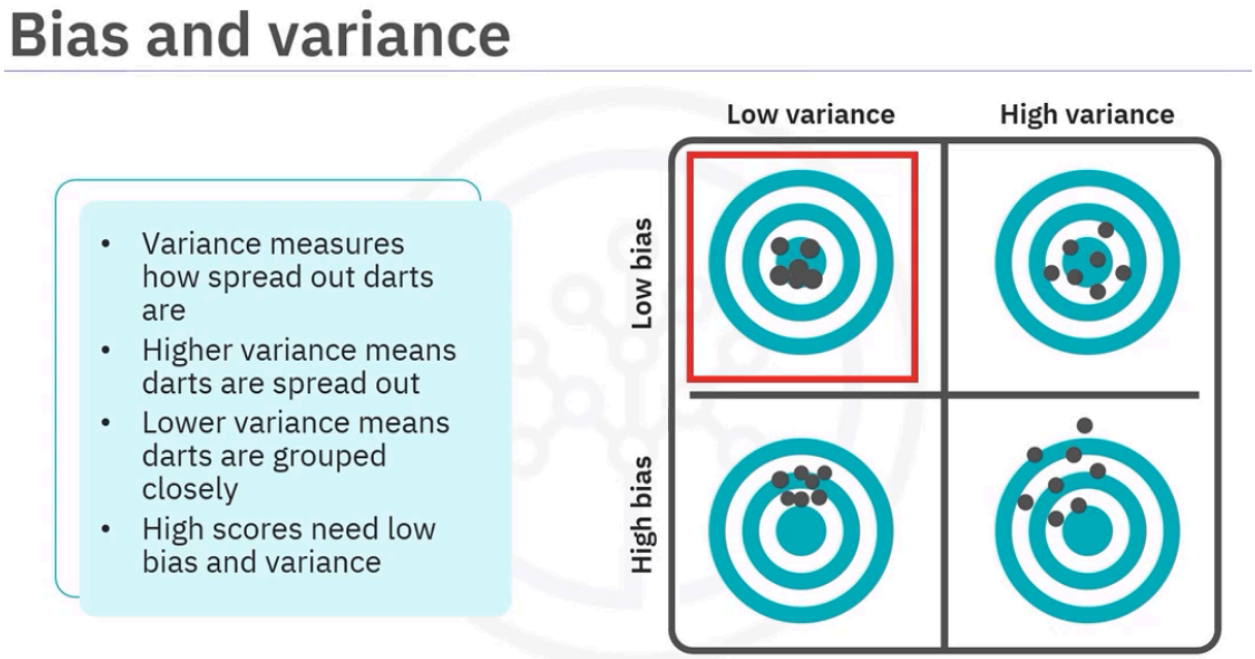


- **Definition**: Sensitivity of the model to training data (spread of predictions).
- **High Variance** → Model is too complex → **Overfitting**.
- **Low Variance** → Predictions are consistent/stable.

📌 Analogy:

- **Bias** → How close darts are to the center.
- **Variance** → How tightly darts are grouped together.

- ✅ Ideal: **Low Bias + Low Variance**.



📉 Bias-Variance Tradeoff

Prediction bias

Measures the accuracy of predictions

$$\text{Bias} = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i) = \frac{1}{N}\sum_{i=1}^{N}\hat{y}_i - \frac{1}{N}\sum_{i=1}^{N}y_i$$

Reflects differences from target values

Average prediction – average of actuals

Is zero for perfect predictors

$\hat{y}_i = y_i$ for all $i \implies$ Bias = 0

1:18

Prediction bias

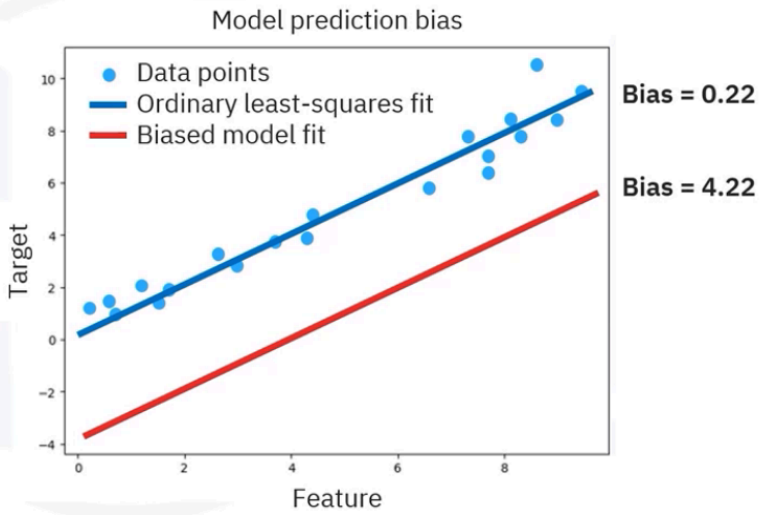
Blue line:

- Shows least-squares fit
- Bias is 0.22

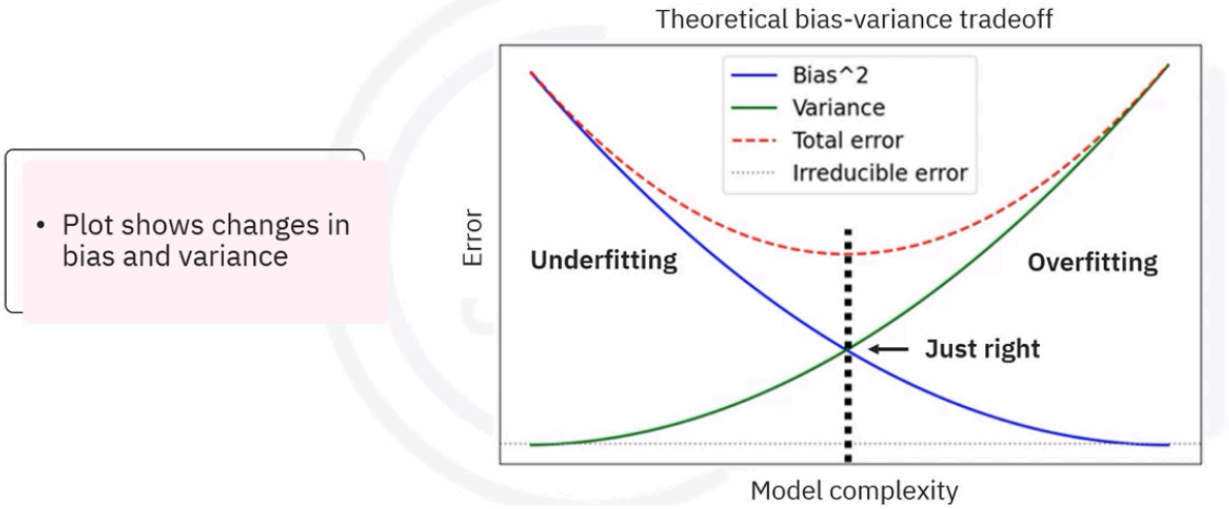
Red line:

- Shifts model down by 4
- Bias is 4.22

1:32



Bias-variance tradeoff

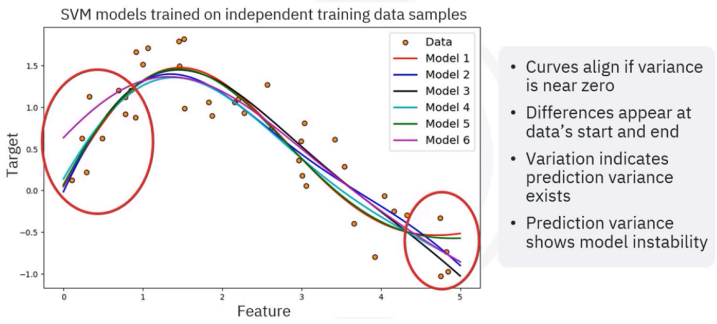


- Plot shows changes in bias and variance

- **Low model complexity** → High Bias, Low Variance → Underfitting.
- **High model complexity** → Low Bias, High Variance → Overfitting.
- **Sweet spot**: Balance where **generalization error** is minimized.
- Some error is unavoidable due to **noise in data**.

🔹 Prediction Error Components

Prediction variance

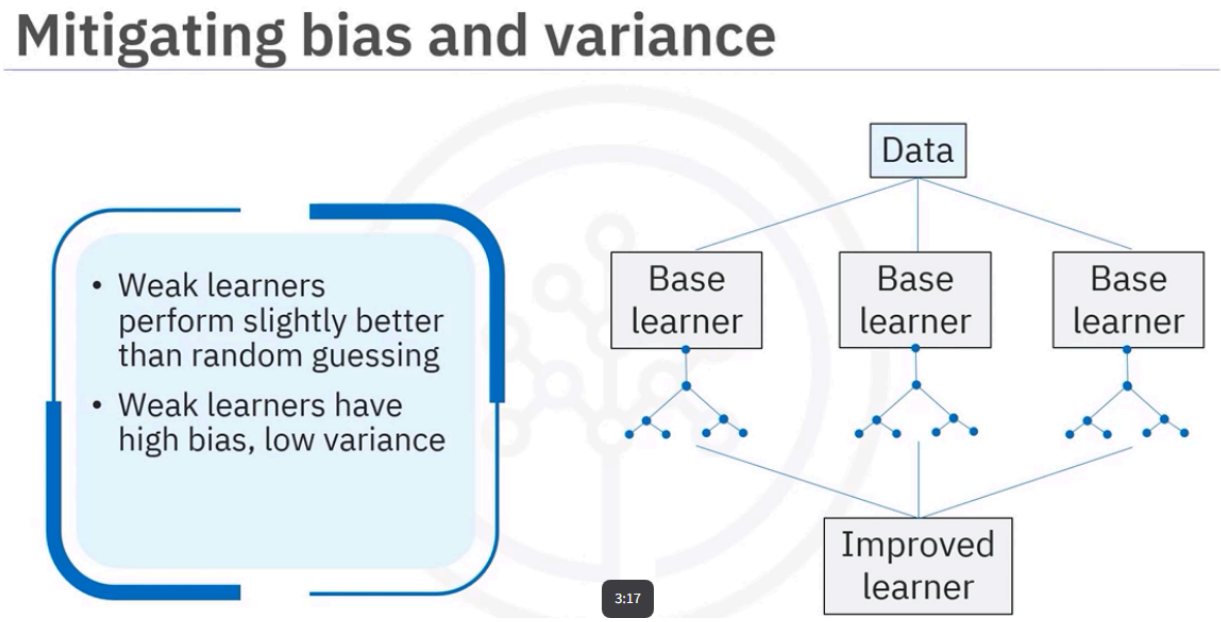


- Curves align if variance is near zero
- Differences appear at data's start and end
- Variation indicates prediction variance exists
- Prediction variance shows model instability

- **Prediction Bias**: Avg. difference between predicted & actual values.

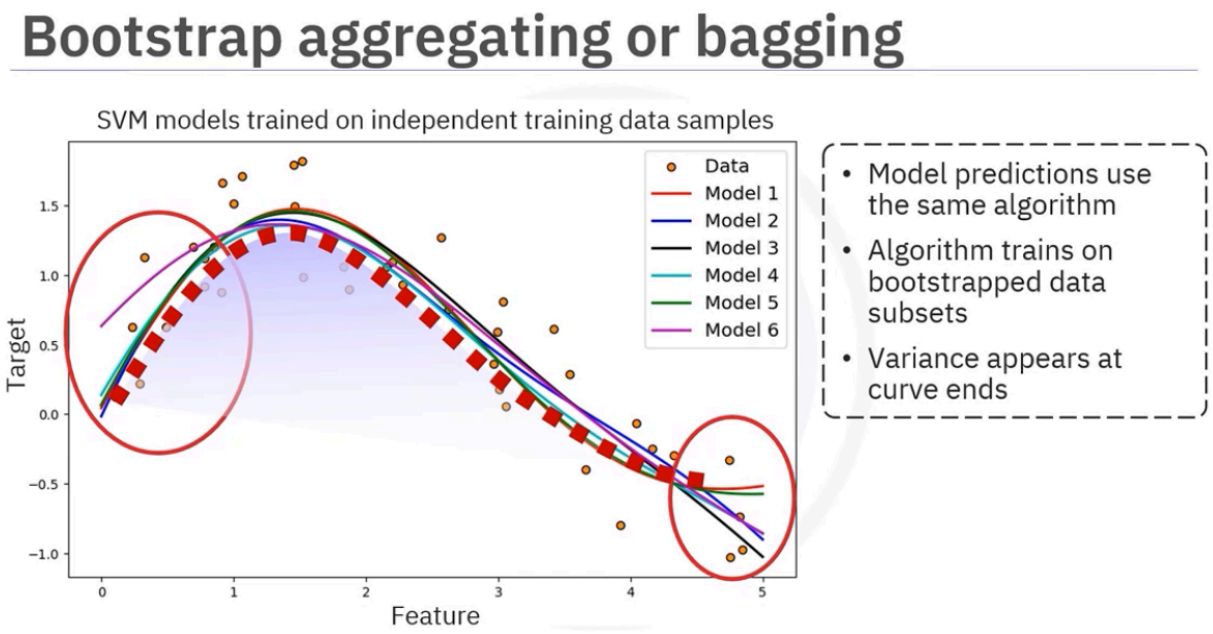
- **Prediction Variance**: Fluctuation of predictions across different training subsets.

🔹 Weak vs. Strong Learners



- **Weak Learners**:
    - Perform slightly better than random guessing.
    - High Bias, Low Variance → Underfitting.
- **Strong Learners**:
    - More accurate, complex models.
    - Low Bias, High Variance → Risk of Overfitting.

🔹 Ensemble Methods



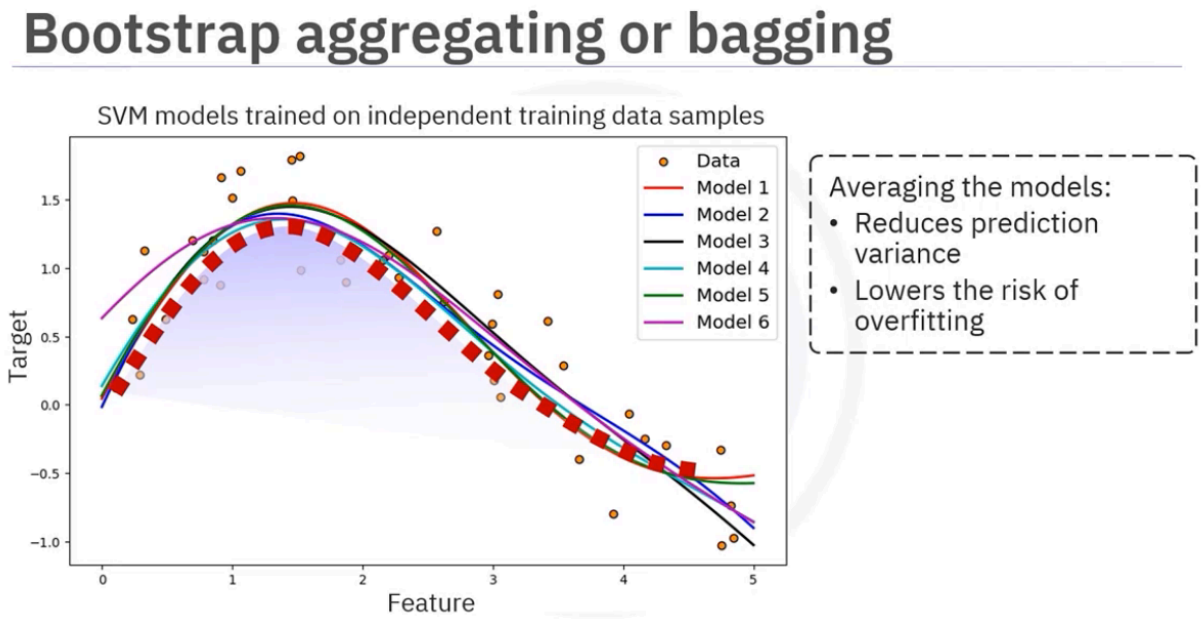
1. Bagging (Bootstrap Aggregating)

- Train multiple models on **bootstrapped (resampled) datasets**.
- Predictions are averaged (regression) or majority-voted (classification).
- **Effect**:
    - ✅ Reduces **variance**.
    - Slightly increases bias.
- **Example**: Random Forests.

2. Boosting

- Train models **sequentially**, each correcting errors of the previous one.
- Misclassified points get **higher weight** in the next round.
- Final prediction is a **weighted sum** of weak learners.

- **Effect**:
    - ✅ Reduces **bias**.
    - Increases model complexity.
- **Examples**: AdaBoost, Gradient Boosting, XGBoost.



📊 Summary Table

Method	Trains On	Goal	Fixes	Example
Bagging	Parallel, bootstrapped subsets	Reduce Variance	Overfitting	Random Forest
Boosting	Sequential, weighted subsets	Reduce Bias	Underfitting	AdaBoost, XGBoost

---

✅ **In short:**

- **Bias** = Accuracy problem (systematic error).
- **Variance** = Precision problem (sensitivity to data).
- **Bagging** lowers variance, **Boosting** lowers bias.
- Best models balance both for generalization.

Got it ✅ I'll turn your provided text into **clean, structured notes with code** so you can revise quickly.

📘 Notes: Random Forest vs XGBoost (Regression)

🎯 Objectives

- Implement **Random Forest** and **XGBoost** regression models using scikit-learn and xgboost.
- Compare their performance in terms of **accuracy** and **speed**.

Dataset: **California Housing Dataset** (predicting house prices).

🛠️ Installation

```
!pip install numpy==2.2.0
```

```
!pip install scikit-learn==1.6.0
```

```
!pip install matplotlib==3.9.3
```

```
!pip install xgboost==2.1.3
```

📚 Import Libraries

```
import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import fetch_california_housing

from sklearn.model_selection import train_test_split

from sklearn.ensemble import RandomForestRegressor

from xgboost import XGBRegressor

from sklearn.metrics import mean_squared_error, r2_score

import time
```

📂 Load Dataset

```
# Load the dataset

data = fetch_california_housing()

X, y = data.data, data.target


# Train-test split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)


# Dataset shape

N_observations, N_features = X.shape
```

```
print(f"Observations: {N_observations}, Features: {N_features}")
```

Output:

Observations: 20640

Features: 8

🌲 Initialize Models

```
n_estimators = 100

rf = RandomForestRegressor(n_estimators=n_estimators, random_state=42)

xgb = XGBRegressor(n_estimators=n_estimators, random_state=42)
```

⏱️ Training

```
# Random Forest

start_time_rf = time.time()

rf.fit(X_train, y_train)

rf_train_time = time.time() - start_time_rf


# XGBoost

start_time_xgb = time.time()

xgb.fit(X_train, y_train)

xgb_train_time = time.time() - start_time_xgb
```

🔮 Predictions

```
# Random Forest

start_time_rf = time.time()

y_pred_rf = rf.predict(X_test)

rf_pred_time = time.time() - start_time_rf


# XGBoost

start_time_xgb = time.time()

y_pred_xgb = xgb.predict(X_test)

xgb_pred_time = time.time() - start_time_xgb
```

📏 Evaluation Metrics

```
mse_rf = mean_squared_error(y_test, y_pred_rf)

mse_xgb = mean_squared_error(y_test, y_pred_xgb)


r2_rf = r2_score(y_test, y_pred_rf)

r2_xgb = r2_score(y_test, y_pred_xgb)


print(f"Random Forest: MSE = {mse_rf:.4f}, R² = {r2_rf:.4f}")

print(f"XGBoost:       MSE = {mse_xgb:.4f}, R² = {r2_xgb:.4f}")
```

```
print(f"Random Forest: Training = {rf_train_time:.3f}s, Prediction = {rf_pred_time:.3f}s")

print(f"XGBoost:       Training = {xgb_train_time:.3f}s, Prediction = {xgb_pred_time:.3f}s")
```

Example Output:

Random Forest: MSE = 0.2556, R² = 0.8050

XGBoost: MSE = 0.2226, R² = 0.8301

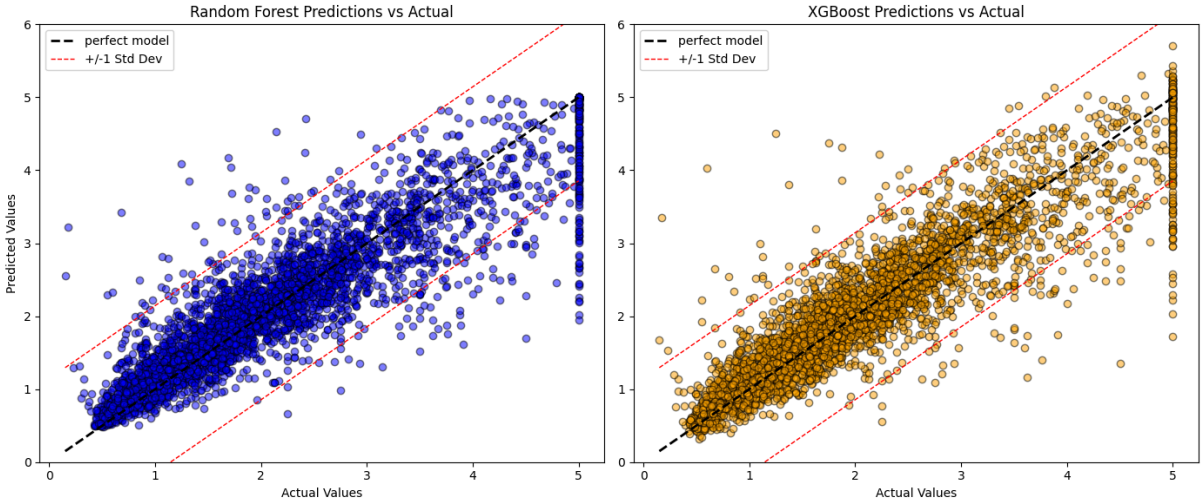
Random Forest: Training = 15.703s, Prediction = 0.174s

XGBoost: Training = 0.339s, Prediction = 0.012s

📊 Visualization

```
# Standard deviation of test values

std_y = np.std(y_test)


plt.figure(figsize=(14,6))


# Random Forest

plt.subplot(1,2,1)

plt.scatter(y_test, y_pred_rf, alpha=0.5, color="blue", ec="k")

plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'k--', lw=2, label="Perfect Model")

plt.plot([y_test.min(), y_test.max()], [y_test.min()+std_y, y_test.max()+std_y], 'r--', lw=1, label="+/-1 Std Dev")
```

```
plt.plot([y_test.min(), y_test.max()], [y_test.min()-std_y, y_test.max()-std_y], 'r--', lw=1)

plt.title("Random Forest Predictions vs Actual")

plt.xlabel("Actual Values")

plt.ylabel("Predicted Values")

plt.ylim(0,6)

plt.legend()


# XGBoost

plt.subplot(1,2,2)

plt.scatter(y_test, y_pred_xgb, alpha=0.5, color="orange", ec="k")

plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'k--', lw=2, label="Perfect Model")

plt.plot([y_test.min(), y_test.max()], [y_test.min()+std_y, y_test.max()+std_y], 'r--', lw=1, label="+/-1 Std Dev")

plt.plot([y_test.min(), y_test.max()], [y_test.min()-std_y, y_test.max()-std_y], 'r--', lw=1)

plt.title("XGBoost Predictions vs Actual")

plt.xlabel("Actual Values")

plt.ylim(0,6)

plt.legend()


plt.tight_layout()

plt.show()
```



📝 Key Takeaways

- **Accuracy**: XGBoost generally performed better (lower MSE, higher R^2).
- **Speed**: XGBoost was **much faster** in both training and prediction compared to Random Forest.
- **Prediction Behavior**:
    - Random Forest stayed within target value range.
    - XGBoost sometimes **overshot upper limits**.

Module 3 Summary and Highlights

You have completed this lesson. At this point in the course, you know:

- Classification is a supervised machine learning method used to predict labels on new data with applications in churn prediction, customer segmentation, loan default prediction, and multiclass drug prescriptions.
- Binary classifiers can be extended to multiclass classification using one-versus-all or one-versus-one strategies.
- A decision tree classifies data by testing features at each node, branching based on test results, and assigning classes at leaf nodes.
- Decision tree training involves selecting features that best split the data and pruning the tree to avoid overfitting.
- Information gain and Gini impurity are used to measure the quality of splits in decision trees.
- Regression trees are similar to decision trees but predict continuous values by recursively splitting data to maximize information gain.
- Mean Squared Error (MSE) is used to measure split quality in regression trees.
- K-Nearest Neighbors (k-NN) is a supervised algorithm used for classification and regression by assigning labels based on the closest labeled data points.
- To optimize k-NN, test various k values and measure accuracy, considering class distribution and feature relevance.
- Support Vector Machines (SVM) build classifiers by finding a hyperplane that maximizes the margin between two classes, effective in high-dimensional spaces but sensitive to noise and large datasets.
- The bias-variance tradeoff affects model accuracy, and methods such as bagging, boosting, and random forests help manage bias and variance to improve model performance.
- Random forests use bagging to train multiple decision trees on bootstrapped data, improving accuracy by reducing variance.

Cheat Sheet: Building Supervised Learning Models

Perfect, you've already gathered a **solid cheat sheet** for supervised learning models 👌
Here's a **cleaner, structured version** of it so you can use it for quick revision:

📌 Cheat Sheet: Building Supervised Learning Models

🔹 Common Models

Model	Process	Key Hyperparameters	Pros	Cons	Example Code
One vs One Classifier (OvO)	Trains 1 classifier for **each pair** of classes	`estimator`	Works well for small datasets	Expensive for many classes	`python\nfrom sklearn.multiclass import OneVsOneClassifier\nfrom sklearn.linear_model import LogisticRegression\nmodel = OneVsOneClassifier(LogisticRegression())`
One vs Rest Classifi	Trains 1 classifier per class vs	`estimator`, `multi_class='ovr'`	Simpler, scalable	Weak on imbalanced data	`python\nfrom sklearn.linear_model import LogisticRegression\nmodel`

er (OvR)	all others				`= LogisticRegression(multi_class='ovr')`
Decision Tree Classifier	Splits data recursively based on features	`max_depth`	Easy to interpret, visualize	Overfits without pruning	`python\nfrom sklearn.tree import DecisionTreeClassifier\nmodel = DecisionTreeClassifier(max_depth=5)`
Decision Tree Regressor	Predicts continuous values via splits	`max_depth`	Handles non-linear data	Poor on noisy data	`python\nfrom sklearn.tree import DecisionTreeRegressor\nmodel = DecisionTreeRegressor(max_depth=5)`
Linear SVM Classifier	Finds best hyperplane with max margin	`C`, `kernel`, `gamma`	Great for high-dim spaces	Struggles with non-linear data (w/o kernels)	`python\nfrom sklearn.svm import SVC\nmodel = SVC(kernel='linear', C=1.0)`
K-Nearest Neighbors (KNN)	Classifies based on **majority vote** of neighbors	`n_neighbors`, `weights`, `algorithm`	Simple, no training cost	Slow for large datasets	`python\nfrom sklearn.neighbors import KNeighborsClassifier\nmodel = KNeighborsClassifier(n_neighbors=5, weights='uniform')`

Random Forest Regressor	Ensemble of decision trees (bagging)	`n_estimators`, `max_depth`	Reduces overfitting	Complex as trees ↑	`python\nfrom sklearn.ensemble import RandomForestRegressor\nmodel = RandomForestRegressor(n_estimators=100, max_depth=5)`
XGBoost Regressor	Gradient boosting trees	`n_estimators`, `learning_rate`, `max_depth`	Very accurate, handles big data	Complex to tune	`python\nimport xgboost as xgb\nmodel = xgb.XGBRegressor(n_estimators=100, learning_rate=0.1, max_depth=5)`

🔹 Useful Functions

Function	Purpose	Example Code
OneHotEncoder	Convert categorical → binary matrix	`python\nfrom sklearn.preprocessing import OneHotEncoder\nencoder = OneHotEncoder(sparse=False)\nencoded = encoder.fit_transform(data)`
accuracy_score	Model accuracy check	`python\nfrom sklearn.metrics import accuracy_score\naccuracy = accuracy_score(y_true, y_pred)`

LabelEncoder	Encode labels as integers	`python\nfrom sklearn.preprocessing import LabelEncoder\nencoder = LabelEncoder()\ny_encoded = encoder.fit_transform(y)`
plot_tree	Visualize decision tree	`python\nfrom sklearn.tree import plot_tree\nplot_tree(model, filled=True)`
normalize	Standardize data	`python\nfrom sklearn.preprocessing import normalize\nX_norm = normalize(X, norm='l2')`
compute_sample_weight	Handle class imbalance	`python\nfrom sklearn.utils.class_weight import compute_sample_weight\nweights = compute_sample_weight('balanced', y)`
roc_auc_score	Evaluate binary classification (AUC-ROC)	`python\nfrom sklearn.metrics import roc_auc_score\nauc = roc_auc_score(y_true, y_score)`

✅ This cheat sheet gives you:

- Core **supervised models** with pros, cons, and syntax.
- Essential **utility functions** for preprocessing & evaluation.