

PRAC 1

⌚ PRACTICAL OVERVIEW

This practical demonstrates how to **predict the price of an Uber ride** using **Regression models**.

You will perform:

- Data loading & preprocessing
- Outlier detection and removal
- Correlation analysis
- Implementing **Linear Regression** and **Random Forest Regression**
- Model evaluation using **R² and RMSE**

Subject: Machine Learning

Topic: Regression-based Prediction Models

📝 PROBLEM STATEMENT

Predict the fare amount of an Uber ride based on pickup & drop-off location coordinates and passenger count.

You are given a dataset containing:

- Pickup & drop locations (Latitude/Longitude)
- Time of ride
- Number of passengers
- Fare amount (Target variable)

Your task is to clean the data, build regression models, and compare their performance.

🔍 CODE EXPLANATION (Step-by-Step)

1) Import Required Libraries

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

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


- pandas → Data handling
- numpy → Numerical operations
- matplotlib → Visualization
- Warnings ignored for cleaner output



---


```

2) Load Dataset & Create Copy

```
data = pd.read_csv("uber.csv")  
df = data.copy()  
df.head()
```

Loads dataset and **keeps a backup** in data.

3) Check Dataset Information

```
df.info()
```

Shows number of rows, columns, datatypes, missing values.

4) Convert Date Column to DateTime Format

```
df["pickup_datetime"] = pd.to_datetime(df["pickup_datetime"])  
Needed for extracting date/time features.
```

5) Statistical Summary

```
df.describe()
```

Gives min, max, mean, median — helps detect **outliers**.

Outliers found in:

- fare_amount
 - Coordinates
-

6) Check Missing Values

```
df.isnull().sum()
```

Only 2 missing values → drop them.

```
df.dropna(inplace=True)
```

7) Detect & Remove Outliers

```
q_low = df["fare_amount"].quantile(0.01)
```

```
q_hi = df["fare_amount"].quantile(0.99)
```

```
df = df[(df["fare_amount"] < q_hi) & (df["fare_amount"] > q_low)]
```

Removes extreme 1% top & bottom fare values.

8) Correlation Matrix

```
df.select_dtypes(include=[np.number]).corr()
```

Shows relationship between numeric variables.

Observation:

- Coordinates are highly correlated with each other.
 - Passenger count has minimal effect on fare.
-

9) Prepare Data for Training

```
from sklearn.model_selection import train_test_split
```

```
x = df.drop("fare_amount", axis=1)
```

```
y = df["fare_amount"]
```

```
x['pickup_datetime'] = pd.to_numeric(pd.to_datetime(x['pickup_datetime']))
```

```
x = x.loc[:, x.columns.str.contains('^\d+')] # Selecting numeric feature
```

```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=1)
```

- Target = Fare (y)
 - Predictors = Other columns (x)
-

10) Linear Regression

```
from sklearn.linear_model import LinearRegression
```

```
lrmodel = LinearRegression()
lrmodel.fit(x_train, y_train)
predict = lrmodel.predict(x_test)
```

Evaluate:

```
from sklearn.metrics import mean_squared_error, r2_score
lr_rmse = np.sqrt(mean_squared_error(y_test, predict))
lr_r2 = r2_score(y_test, predict)
```

Output:

Linear Regression → RMSE: 8.06 R²: -0.000026

11) Random Forest Regression

```
from sklearn.ensemble import RandomForestRegressor
rfrmodel = RandomForestRegressor(n_estimators = 100, random_state = 101)
rfrmodel.fit(x_train, y_train)
rfrmodel_pred = rfrmodel.predict(x_test)
```

Evaluate:

```
rfr_rmse = np.sqrt(mean_squared_error(y_test, rfrmodel_pred))
rfr_r2 = r2_score(y_test, rfrmodel_pred)
```

Output:

Random Forest → RMSE: 9.75 R²: -0.46

MODEL PERFORMANCE COMPARISON

Model	RMSE	R ²	Score Interpretation
Linear Regression	8.06	~0	Performs poorly (almost no predictive power)
Random Forest	9.75	-0.46	Worse performance due to insufficient features

Reason for low accuracy:

Only *one feature (Unnamed: 0)* was used after column filtering.

Better approach → include **distance, date/time features**.

INPUT FORMAT

Dataset: uber.csv

Column	Meaning
fare_amount	Ride price (Target)
pickup_longitude / latitude	Start location
dropoff_longitude / latitude	End location
passenger_count	Number of passengers

OUTPUT FORMAT

Two numbers printed for each model:

- RMSE (Error) → Lower is better
- R² (Accuracy) → Higher is better

Example:

Linear Regression → RMSE: 8.06 R²: -0.00002

Random Forest → RMSE: 9.75 R²: -0.46

HOW TO RUN

pip install pandas numpy matplotlib scikit-learn

jupyter notebook

Run all cells.

EXAM VIVA QUESTIONS

Question	Answer (Short, Exam-Safe)
Why remove outliers?	To prevent extreme values from biasing the model.
What is RMSE?	Average difference between predicted and actual values.

Question	Answer (Short, Exam-Safe)
Why convert datetime?	ML models require numeric inputs.
Why R ² score?	Shows how much variance is explained by the model.
Difference between Linear Regression and Random Forest?	LR fits a line; RF is an ensemble of decision trees improving stability.

PRACTICE TASKS

Try these improvements:

1. **Create a distance feature** using Haversine formula → Accuracy improves.
2. Extract date features:
 - Hour
 - Day of week
 - Month
3. Try **Gradient Boosting** or **XGBoost** and compare performance

Code explanation :

```
#import libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import warnings
#We do not want to see warnings
warnings.filterwarnings("ignore")
```

Code	Meaning
pandas as pd	Used to handle and clean data (DataFrames).
numpy as np	Used for numerical operations.
matplotlib.pyplot as plt	Used for plotting graphs.
warnings.filterwarnings("ignore")	Hides warnings, so output looks clean.

2) Load Dataset

```
data = pd.read_csv("uber.csv")
df = data.copy()
df.head()
```

Code	Meaning
pd.read_csv("uber.csv")	Reads the CSV file named <i>uber.csv</i> and loads into DataFrame.

Code	Meaning
data.copy()	Makes a copy of the dataset (so original stays safe).
df.head()	Shows first 5 rows of the dataset.

3) Check Data Information

df.info()

- Displays:
 - Number of rows and columns
 - Data types of each column (int, float, object)
 - How many missing values exist
-

4) Convert Date-Time Column

df["pickup_datetime"] = pd.to_datetime(df["pickup_datetime"])

- Converts the *pickup_datetime* column from **string** to **datetime format**, so we can extract date and time parts later.
-

5) View Statistical Summary

df.describe()

Shows basic statistics:

- Mean, Median
- Min, Max
- Standard Deviation
- Quartiles

Useful for understanding **data distribution**.

6) Check Missing Values

df.isnull().sum()

- Counts how many **null / missing values** are in each column.
-

7) Remove Missing Values

```
df.dropna(inplace=True)
```

- Removes any row containing **null** values.
 - `inplace=True` → Changes are made directly to the original df.
-

8) Remove Outliers

```
q_low = df["fare_amount"].quantile(0.01)  
q_hi = df["fare_amount"].quantile(0.99)  
df = df[(df["fare_amount"] < q_hi) & (df["fare_amount"] > q_low)]
```

Code Meaning

`quantile(0.01)` Value at the **1%** position (very low value).

`quantile(0.99)` Value at the **99%** position (very high value).

Filter condition Keeps only rows where fare is between 1% and 99% values.

This removes **extreme outliers** that may disturb the model.

9) Correlation Matrix

```
df.select_dtypes(include=[np.number]).corr()
```

- `select_dtypes(include=[np.number])` → Selects only **numeric columns**.
- `.corr()` → Computes **correlation** between numeric columns.

Correlation shows how strongly two variables are related.

Correlation Value Meaning

+1 Perfect positive relation

0 No relation

-1 Perfect negative relation

Example:

- If distance increases → fare increases → **positive correlation**
-

✓ Summary of This Code

Step	Task
Import libraries	Setup environment
Load dataset	Read CSV data
View data	Understand structure
Convert datetime	Prepare data
Remove null values	Clean data
Remove outliers	Improve model quality
Check numeric correlation	Understand feature relationships

If you want, I can now explain:

1)

`from sklearn.model_selection import train_test_split`

- **from sklearn.model_selection** → We are importing a module from Scikit-Learn that helps with splitting datasets.
 - **import train_test_split** → `train_test_split` is a function used to divide your dataset into:
 - **Training data** (used to train the model)
 - **Testing data** (used to test how well the model performs)
-

2)

`x = df.drop("fare_amount", axis = 1)`

- `df` → Your entire dataset in a DataFrame.
- `.drop()` → Removes a column or row.
- `"fare_amount"` → The column we want to remove. This column is your **target/output** variable.

- $\text{axis} = 1 \rightarrow$ Means *remove a column.*
($\text{axis} = 0$ means remove a row)

So,

- $\text{x} \rightarrow$ now contains all columns **except fare_amount**
i.e., **features / input variables**.
-

3)

```
y = df["fare_amount"]
```

- We select only the **fare_amount** column \rightarrow this is the **output variable** we want to predict.
 - $\text{y} \rightarrow$ Target variable.
-

4)

```
x['pickup_datetime'] = pd.to_numeric(pd.to_datetime(x['pickup_datetime']))
```

Breakdown:

- $\text{x['pickup_datetime']}$ \rightarrow Selecting the column named **pickup_datetime**.
- $\text{pd.to_datetime(...)}$ \rightarrow Converts the date/time string into proper **datetime format**.
- $\text{pd.to_numeric(...)}$ \rightarrow Converts that datetime into **numbers** (usually number of nanoseconds from 1970).

So this line **changes the datetime column into a numeric format** so that machine learning algorithms can understand it (since ML models cannot work directly with date strings).

5)

```
x = x.loc[:, x.columns.str.contains('^\u00d7Unnamed')]
```

This line is **selecting only the columns whose name starts with 'Unnamed'**.

Breakdown:

- x.columns \rightarrow List of all column names.
- $\text{.str.contains('^\u00d7Unnamed')}$ \rightarrow Checks if the column name starts with the word "**Unnamed**" (^ means "starts with").

- `x.loc[:, ...]` → Selects **all rows** (`:`) and **only those columns** matching condition.

⚠ This line usually keeps only columns that were created by mistake (like index columns from CSV).

Most of the time we **remove** Unnamed columns, not keep them.
So this may be **wrong** unless intentionally done.

6)

`x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=1)`

Breakdown:

- `train_test_split(x, y, ...)` → Splits feature data (`x`) and target data (`y`).
- `test_size = 0.2` → 20% of data will be **test set**, 80% will be **training set**.
- `random_state = 1` → Ensures the split is **reproducible** (same result each time).

The output:

Variable Meaning

`x_train` Training input data

`x_test` Testing input data

`y_train` Training output labels

`y_test` Testing output labels

Final Understanding Summary

Code	Meaning
<code>x</code>	All input/features (except <code>fare_amount</code>)
<code>y</code>	Output/target (<code>fare_amount</code>)
Convert datetime to numeric	Required for ML
Select Unnamed columns	Probably incorrect step (may need removal)
Split into train-test	Used for modeling and evaluation

What is Linear Regression?

Linear Regression is a machine learning algorithm used to predict a value (a number) based on given input values.

It finds the **best straight line** that fits the data.

Example:

- If you have data of **distance traveled** and **cab fare**, you can use linear regression to **predict the fare** for a new distance.
-

Idea Behind Linear Regression

We assume there is a **relationship** between:

- **Input (X)** → Features (e.g., distance, no. of passengers)
- **Output (Y)** → Target (e.g., fare amount)

The relationship is shown as a **straight line**:

$$Y = mX + c$$

Where:

Symbol Meaning

X Input value (feature)

Y Predicted output

m Slope of the line (how much Y changes when X changes)

c Intercept (value of Y when X = 0)

Example

Suppose you have distance and fare data:

Distance (km) Fare (₹)

1	50
2	100
3	150

Here:

- For each +1 km → fare increases by ₹50.
- So:
 - **m (slope) = 50**
 - **c (intercept) = 0**

So our equation becomes:

$$\text{Fare} = 50 \times \text{Distance}$$

If distance = 5 km:

$$\text{Fare} = 50 \times 5 = ₹250$$

❖ How Linear Regression Works (Conceptually)

1. It draws **many possible lines**.
2. It calculates **error** for each line.
 - Error = difference between **actual** and **predicted** values.
3. It chooses the line that has the **minimum total error**.
→ This is known as **Best Fit Line**.

The error calculation commonly uses:

$$\text{MSE} = \frac{1}{n} \sum (Y_{actual} - Y_{predicted})^2$$

❖ Why Use Linear Regression?

Reason	Benefit
Simple	Easy to understand and implement
Fast	Works even on large data
Predictive	Can forecast future values
Explainable	Clear relationship between variables

When to Use Linear Regression?

Use it when:

- Output is a **continuous number** (fare, price, marks, salary, etc.)
- Relationship between input and output is **linear** (straight line pattern).

Do **not** use it when predicting:

- Categories like *male/female, spam/not spam*, etc.

(For that we use **classification models**.)

Final Summary

Linear Regression:

- Finds a **straight-line relationship** between **X** and **Y**.
- Uses the formula:

$$Y = mX + c$$

- Used when prediction value is a **number**

```
from sklearn.linear_model import LinearRegression  
lrmodel = LinearRegression()  
lrmodel.fit(x_train, y_train)  
predict = lrmodel.predict(x_test)
```

1)

```
from sklearn.linear_model import LinearRegression
```

- **from sklearn.linear_model** → We are using a module from Scikit-Learn that contains algorithms for linear models.
- **import LinearRegression** → LinearRegression is a machine learning algorithm used to perform **Linear Regression**.

This line **loads the Linear Regression model** so that we can use it.

2)

```
lrmodel = LinearRegression()
```

This line **creates an object (model)** of LinearRegression.

- lrmodel → The name we give to our model. You can name it anything.
- LinearRegression() → Calls the LinearRegression class to create a model.

So now lrmodel is an **empty model** waiting to be trained.

3)

```
lrmodel.fit(x_train, y_train)
```

This line **trains the model**.

- **fit()** → Means *learn the pattern* from data.
- **x_train** → Training input (features).
- **y_train** → Training output (target).

This step:

- Calculates **slope (m)** and **intercept (c)**.
- Finds the **Best Fit Line** to minimize the prediction error.

So after this, the model now **knows how X relates to Y**.

4)

```
predict = lrmodel.predict(x_test)
```

This line **makes predictions** using the trained model.

- **predict()** → Tells the model to predict answers.
- **x_test** → The test data (model hasn't seen it before).
- **predict** → Stores the predicted **fare_amount values**.

So now predict contains values like:

```
[12.5, 9.3, 18.7, 20.1, ...]
```

Summary Table

Code	Meaning
LinearRegression()	Choose algorithm
lrmodel = LinearRegression()	Create model object
lrmodel.fit(x_train, y_train)	Train model on training data
lrmodel.predict(x_test)	Predict output for test data

If you want, I can now explain:

- How to **check accuracy (R² Score)**
- How to **calculate RMSE / MAE**
- How to **plot actual vs predicted graph**

```
from sklearn.metrics import mean_squared_error, r2_score  
lr_rmse = np.sqrt(mean_squared_error(y_test, predict))
```

```
lr_r2 = r2_score(y_test, predict)
```

|1)

```
from sklearn.metrics import mean_squared_error, r2_score
```

- **from sklearn.metrics** → We import evaluation metric functions from Scikit-Learn.
- **mean_squared_error** → A function that calculates **how far predictions are from actual values**.
- **r2_score** → A function that tells **how good the model is** at predicting (accuracy for regression).

So this line imports the **tools to measure model performance**.

2)

```
lr_rmse = np.sqrt(mean_squared_error(y_test, predict))
```

Breakdown:

- `mean_squared_error(y_test, predict)` → Calculates the **MSE** = average of squared differences between:
 - `y_test` → Actual values
 - `predict` → Predicted values from model

Formula:

$$MSE = \frac{1}{n} \sum (Actual - Predicted)^2$$

- `np.sqrt(...)` → Takes **square root** of MSE to convert it into **RMSE (Root Mean Squared Error)**

$$RMSE = \sqrt{MSE}$$

- `lr_rmse` → A variable storing the calculated **error value**.

So:

RMSE tells how much error (on average) the model makes when predicting.
→ Lower RMSE = Better model.

3)

`lr_r2 = r2_score(y_test, predict)`

- `r2_score(y_test, predict)` → Calculates **R² Score**.

R² Score (Coefficient of Determination) tells how well the model fits the data.

$$R^2 = 1 - \frac{\text{Error_model}}{\text{Error_average}}$$

Range and Meaning:

R² Value Model Interpretation

1.0 Perfect accuracy (very rare)

0.8+ Excellent model

0.5 to 0.8 Good / Acceptable

0 to 0.5 Weak model

Negative Very bad model

- `lr_r2` → Stores the accuracy score.
-

Final Understanding Summary

Variable Meaning

Goal

lr_rmse Average prediction error **Lower is better**

lr_r2 Accuracy score of model **Higher is better**

Example Interpretation:

If you get:

`lr_rmse = 2.5`

`lr_r2 = 0.87`

This means:

- Your model makes a **small error of 2.5 units on average**

- Your model is **87% accurate** at predicting outcomes.

What is Random Forest Regression?

Random Forest Regression is a Machine Learning algorithm used to **predict numerical values** (just like Linear Regression), but it is **more powerful and more accurate** for complex data.

It works by:

- Building **many** decision trees
- Each tree makes a prediction
- The final prediction is the **average** of all trees' predictions

This is why it is called a **Forest** (group of trees).

Why "Random"?

It is called **random** because:

1. **Random Rows:** Each tree is trained on *randomly selected* rows of the dataset.
2. **Random Columns:** Each tree uses *randomly selected* features (columns).

This ensures:

- No single tree dominates
 - Trees think **differently**
 - The final result is **stable and accurate**
-

How One Decision Tree Works (Simple Idea)

A **Decision Tree** splits data like this:

Example (Taxi Fare Prediction) :

Is distance > 10 km?

/ \

Yes No

(higher fare) (lower fare)

But one tree alone may make mistakes.

Random Forest = Many Trees Working Together

Imagine 100 such trees:

- Each tree gives an answer (fare prediction).
- Random Forest takes the **average** of all.

This reduces error and improves reliability.

Advantage of Random Forest

Feature	Benefit
Uses many trees	More accurate than single models
Handles missing values	Works even with incomplete data
Works with non-linear data	Can learn complex relationships
Reduces overfitting	More general and stable predictions

Disadvantage

Issue	Explanation
Slower	Because it uses many trees
More memory needed	Stores all trees

When to Use Random Forest Regression?

Use it when:

- Dataset is **complex**
 - Relationship between features and target is **not linear**
 - Linear Regression is giving **low accuracy**
-

Formula?

No fixed straight-line formula (unlike Linear Regression).
It works by combining results of multiple **Decision Trees**.

Code Example in Python

```
from sklearn.ensemble import RandomForestRegressor

rfmodel = RandomForestRegressor(n_estimators=100, random_state=1)
rfmodel.fit(x_train, y_train)

rf_predict = rfmodel.predict(x_test)

Evaluate:

from sklearn.metrics import mean_squared_error, r2_score
rf_rmse = np.sqrt(mean_squared_error(y_test, rf_predict))
rf_r2 = r2_score(y_test, rf_predict)

print("RMSE:", rf_rmse)
print("R2 Score:", rf_r2)
```

Summary

Model	Best For	Accuracy
Linear Regression	Simple data	Good but limited
Random Forest Regression	Complex and non-linear data	Usually better accuracy

If you want, I can now explain:

- ✓ Difference between Random Forest and Decision Tree
- ✓ OR practice viva questions for this topic
- ✓ Or help write **theory notes** for your exam

Just tell me 😊

```
from sklearn.ensemble import RandomForestRegressor  
rfrmodel = RandomForestRegressor(n_estimators = 100, random_state = 101)  
rfrmodel.fit(x_train, y_train)  
rfrmodel_pred = rfrmodel.predict(x_test)
```

1)

```
from sklearn.ensemble import RandomForestRegressor
```

- **from sklearn.ensemble** → We import from the *ensemble* module.
Ensemble means a **group of models working together**.
- **import RandomForestRegressor** → RandomForestRegressor is a machine learning algorithm used to **predict numeric values using many decision trees**.

So this line **loads the Random Forest Regression model**.

2)

```
rfrmodel = RandomForestRegressor(n_estimators = 100, random_state = 101)
```

This line **creates a Random Forest model object**.

- rfrmodel → The name of our model (you can name it anything).
- RandomForestRegressor() → Creates the model.
- n_estimators = 100 → Means the forest will have **100 decision trees**.
 - More trees = higher accuracy (but slower).
- random_state = 101 → Ensures the results are **same every time** you run the code.

So now rfrmodel is a **ready model** but not trained yet.

3)

```
rfrmodel.fit(x_train, y_train)
```

- **fit()** → Train/learn the model.
- x_train → Training input data (features).

- `y_train` → Training target/output values.

Here:

- Random Forest creates **100 decision trees**
 - Each tree learns patterns from data
 - Model becomes trained.
-

4)

```
rfrmodel_pred = rfrmodel.predict(x_test)
```

- `predict()` → Model makes predictions.
- `x_test` → New unseen data.
- `rfrmodel_pred` → Stores **predicted values** (predicted fare_amount).

So now `rfrmodel_pred` contains predicted results like:

```
[10.25, 13.5, 7.9, 19.1, ...]
```

✓ Summary

Code	Meaning
<code>RandomForestRegressor()</code>	Choose Random Forest model
<code>n_estimators = 100</code>	Use 100 decision trees
<code>fit(x_train, y_train)</code>	Train the model
<code>predict(x_test)</code>	Predict output for test data

Random Forest is usually **more accurate** than Linear Regression.

```
rfr_rmse = np.sqrt(mean_squared_error(y_test, rfrmodel_pred))
```

```
rfr_r2 = r2_score(y_test, rfrmodel_pred)
```

1)

```
rfr_rmse = np.sqrt(mean_squared_error(y_test, rfrmmodel_pred))
```

Breakdown:

Part	Meaning
mean_squared_error(y_test, rfrmmodel_pred)	Calculates MSE (Mean Squared Error)
y_test	Actual values from dataset
rfrmmodel_pred	Predicted values from Random Forest model
(Actual - Predicted) ²	Square the difference for each value
Average(...)	Then take the mean of all squared errors

MSE formula:

$$MSE = \frac{1}{n} \sum (Y_{actual} - Y_{predicted})^2$$

Then:

- np.sqrt(...) → Takes **square root** of MSE.

This gives:

$$RMSE = \sqrt{MSE}$$

- rfr_rmse → Stores the **average error** of the Random Forest model.

✓ **Lower RMSE = Better model**

⚠ RMSE is always in the same unit as your output (fare amount).

2)

```
rfr_r2 = r2_score(y_test, rfrmmodel_pred)
```

- r2_score() → Calculates **R² Score** (also called **Coefficient of Determination**).
- y_test → Actual values.
- rfrmmodel_pred → Predicted values.

R² Score tells how well the model fits the data.

R² Value Interpretation:

R² Score Value Meaning

1.0	Perfect prediction (ideal case, rarely happens)
0.8 – 1.0	Very good model
0.5 – 0.8	Moderate / OK model
0 – 0.5	Weak model
Negative	Very bad model (model is worse than random guessing)

- rfr r2 → Stores the **accuracy** of the Random Forest model.

Higher R² = Better Model

Final Summary

Metric	Formula / Meaning	Goal
RMSE	Average error between actual & predicted	Lower is better
R ² Score	How well model fits data	Higher is better

So:

```
rfr rmse # tells error of prediction
```

```
rfr r2 # tells accuracy of model
```

If you want, I can now show:

- ✓ A **comparison table** of Linear Regression vs Random Forest
 - ✓ How to **print results nicely**
 - ✓ How to **write the viva explanation** in simple 5–6 lin

Assignment No. 2

Title: Email Spam Classification using KNN and SVM

Problem Definition:

Classify emails into two categories:

- **Normal (Not Spam)**
- **Abnormal (Spam)**

using **binary classification techniques**.

In this experiment, we use **K-Nearest Neighbors (KNN)** and **Support Vector Machine (SVM)** models to classify emails based on their **word frequency features**, and compare their performance.

Dataset Used:

emails.csv (Email Spam Classification Dataset from Kaggle)

Each row represents an email converted into numeric word frequency values.

- Features: Word occurrence counts in the email (Columns 1–3000)
- Label (spam):
 - 1 → Email is Spam
 - 0 → Email is Not Spam

Prerequisites:

- Basics of Python
- Concept of Classification
- Machine Learning Workflow

Tools & Software:

- Python (Anaconda, Jupyter Notebook)
- Libraries: pandas, sklearn, matplotlib, seaborn

Theory Concepts

1) Support Vector Machine (SVM)

- SVM is a **supervised classification algorithm**.
- It finds the **best separating boundary (hyperplane)** between classes.
- Works well when data is **high-dimensional**, like text data.
- Parameters affecting SVM:
 - **Kernel** (linear / rbf)
 - **C (Penalty Parameter)**
 - **Gamma (Curve Sharpness)**

2) K-Nearest Neighbors (KNN)

- KNN is a **non-parametric, lazy learning** algorithm.
- It does **not** learn a model → it **stores** the training data.
- For prediction:
 - Finds the **K nearest data points**.
 - Uses **majority voting** for classification.
- K affects performance:
 - Small K → High sensitivity (risk of overfitting)
 - Large K → Smoother decision boundary (risk of underfitting)

Why Scaling is Needed in KNN?

KNN uses **distance calculation**, so features must be on the **same scale**.
We use **StandardScaler** to normalize values.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
import seaborn as sns
import matplotlib.pyplot as plt
```

```

# Load dataset
df = pd.read_csv("emails.csv")

# Select features and labels
X = df.iloc[:,1:3001]
Y = df.iloc[:, -1].values

# Split dataset
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.25,
random_state=42)

# ----- Support Vector Machine -----
svc = SVC(C=1.0, kernel='rbf', gamma='auto')
svc.fit(X_train, y_train)
svc_pred = svc.predict(X_test)

print("SVM Accuracy:", accuracy_score(y_test, svc_pred))
print("SVM Classification Report:\n", classification_report(y_test, svc_pred))
print("SVM Confusion Matrix:\n", confusion_matrix(y_test, svc_pred))

# ----- K-Nearest Neighbors -----
knn = KNeighborsClassifier(n_neighbors=7)
knn.fit(X_train, y_train)
knn_pred = knn.predict(X_test)

print("KNN Accuracy:", knn.score(X_test, y_test))
print("KNN Classification Report:\n", classification_report(y_test, knn_pred))
print("KNN Confusion Matrix:\n", confusion_matrix(y_test, knn_pred))

```

```

# ----- Improving KNN with Scaling -----
scaler = StandardScaler()
X_train_s = scaler.fit_transform(X_train)
X_test_s = scaler.transform(X_test)

for k in [1,3,5]:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train_s, y_train)
    y_pred = knn.predict(X_test_s)

    print(f"\nK = {k}: Accuracy = {accuracy_score(y_test, y_pred):.4f}")
    print(confusion_matrix(y_test, y_pred))
    print(classification_report(y_test, y_pred))

```

Output & Evaluation

SVM Performance

- **Accuracy:** 89%
- Works well overall, especially on **Not-Spam** class.
- Sometimes fails to detect some spam emails.

KNN Performance (k=7)

- **Accuracy:** 86%
- Detects more spam, but misclassifies more normal emails.

KNN with Scaling

K Value Accuracy Observation

1	90%	Highest performance but risk of overfitting
3	86%	Balanced
5	83%	Underfitting

Conclusion

- SVM provides stable and consistent performance with **89% accuracy**.
 - KNN (**k=7**) gives 86% accuracy but is more sensitive to data scaling.
 - After applying **feature scaling**, KNN with **k=1 gave the highest accuracy (90%)**.
 - Therefore, **KNN can achieve better accuracy**, but **SVM is more reliable for general use**.
-

Viva Questions & Answers

Question	Answer
Why use SVM?	SVM finds the best separating boundary and works well in high-dimensional data like text.
Why use KNN?	KNN is simple and classifies based on similarity (distance).
What is Lazy Learning?	Model does not learn during training — KNN stores training data and predicts only during testing.
Hard Margin vs Soft Margin?	Hard = perfect separation, Soft = allows errors to avoid overfitting.
What affects SVM boundary?	Kernel, C value, and Gamma.

]

Objective of Code

We want to classify emails into:

- **Spam (1)**
 - **Not Spam (0)**
- using two machine learning classifiers:
1. **SVM (Support Vector Machine)**
 2. **KNN (K-Nearest Neighbors)**
-

Step-by-Step Code Explanation

1) Import Required Libraries

```
import pandas as pd  
  
from sklearn.model_selection import train_test_split  
  
from sklearn.svm import SVC  
  
from sklearn.metrics import accuracy_score  
  
from sklearn.neighbors import KNeighborsClassifier  
  
from sklearn.preprocessing import StandardScaler  
  
import seaborn as sns
```

Library	Purpose
pandas	load & handle dataset
train_test_split	split data into train & test sets
SVC	Support Vector Machine algorithm
accuracy_score	evaluate model performance
KNeighborsClassifier	KNN algorithm
StandardScaler	normalize data for KNN
seaborn/matplotlib	visualization & outlier checking

2) Load Dataset

```
df = pd.read_csv("emails.csv")
```

```
df.head()
```

- Loads the dataset from the CSV file.
 - df.head() shows first few rows to verify data.
-

3) Check Missing Values

```
df.isnull().sum()
```

- Ensures there are **no null/missing entries**.
 - If present, we'd need to clean them.
-

4) Select Input & Output Variables

```
X = df.iloc[:,1:3001] # word frequency features
```

```
Y = df.iloc[:, -1].values # 1 = spam, 0 = not spam
```

Variable Meaning

X All **3000 word frequency features** (email text converted to numbers)

Y Spam label → 1 = spam, 0 = not spam

This means we are using **Bag of Words** features for classification.

💡 Step-by-Step Explanation

1) Select Only Numeric Columns

```
numeric_cols = df.select_dtypes(include=['int64', 'float64']).columns.tolist()
```

- The dataset has numeric and non-numeric data.
- We only want **numeric columns**, because outlier detection applies only to numbers.
- This line collects all numeric column names into a list called **numeric_cols**.

2) Import Visualization Libraries

```
import matplotlib.pyplot as plt
```

```
import seaborn as sns
```

- Used for plotting graphs.
 - seaborn gives cleaner and nicer statistical visualizations.
-

3) Calculate Outliers Using IQR Method

```
Q1 = df_numeric.quantile(0.25)
```

```
Q3 = df_numeric.quantile(0.75)
```

```
IQR = Q3 - Q1
```

```
lower = Q1 - 1.5 * IQR
```

```
upper = Q3 + 1.5 * IQR
```

Symbol	Meaning
--------	---------

Q1	25th percentile (lower boundary of normal data)
----	---

Q3	75th percentile (upper boundary of normal data)
----	---

IQR	Interquartile Range = Q3 – Q1
-----	-------------------------------

lower, upper Values beyond these points are **outliers**

IQR Method detects extreme values in the dataset.

4) Identify Outlier Positions

```
outlier_mask = ((df_numeric < lower) | (df_numeric > upper))
```

- Creates a **True/False mask**
 - True → The value is an **outlier**
 - False → The value is normal
-

5) Count How Many Outliers Each Column Has

```
outlier_counts = outlier_mask.sum().sort_values(ascending=False)
```

- Counts number of outliers per feature.
 - Sorts in descending order → Columns with **most outliers at top**.
-

6) Select Top N Columns with Most Outliers

topN = 12

```
top_features = outlier_counts.head(topN).index.tolist()
```

- Chooses **top 12 features** with maximum outliers.
 - `top_features` contains their names.
-

7) Draw Boxplot to Visualize Outliers

```
plt.figure(figsize=(16,6))  
sns.boxplot(data=df_numeric[top_features])  
plt.title(f'Boxplots for top {topN} features by outlier count')  
plt.xticks(rotation=45, ha='right')  
plt.show()
```

Command Meaning

`sns.boxplot()` Shows median, quartiles, and outliers

`figsize` Makes plot wide and readable

`rotation=45` Rotates labels to avoid overlapping

The **dots outside the box** represent **outliers**.

✍ What to Say in Viva

“I used the IQR method to detect outliers in numeric features. Q1 and Q3 were calculated, and values outside the range $Q1 - 1.5 \times IQR$ and $Q3 + 1.5 \times IQR$ were marked as outliers. Then the features with the highest number of outliers were visualized using boxplots.”

✓ Short Meaning

Step	Summary
Select numeric columns	Only numerical data checked for outliers
Calculate IQR	Identify range of normal values
Create mask	Detect which values are outliers
Count outliers	Find features most affected
Boxplot	Display outliers visually

```
# Split data
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.25,
random_state=42)
```

```
from sklearn.metrics import classification_report, confusion_matrix
```

```
# ----- Support Vector Machine -----
svc = SVC(C=1.0, kernel='rbf', gamma='auto')
svc.fit(X_train, y_train)

svc_pred = svc.predict(X_test)
```

```
print("SVM Accuracy:", accuracy_score(y_test, svc_pred))
print("SVM Classification Report:\n", classification_report(y_test, svc_pred))
print("SVM Confusion Matrix:\n", confusion_matrix(y_test, svc_pred))
```

✓ Corrected Code

```
# Split data
```

```

X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.25,
random_state=42)

from sklearn.metrics import classification_report, confusion_matrix

# ----- Support Vector Machine -----
svc = SVC(C=1.0, kernel='rbf', gamma='auto')
svc.fit(X_train, y_train)

svc_pred = svc.predict(X_test)

print("SVM Accuracy:", accuracy_score(y_test, svc_pred))
print("SVM Classification Report:\n", classification_report(y_test, svc_pred))
print("SVM Confusion Matrix:\n", confusion_matrix(y_test, svc_pred))

```

Step-by-Step Explanation

1) Splitting the Dataset

X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.25,
random_state=42)

Part	Meaning
X	Features (input variables)
Y	Target / Labels (class values)
test_size=0.25	25% data → Test set, 75% → Training set
random_state=42	Ensures same split every time (reproducibility)

2) Import Metrics

from sklearn.metrics import classification_report, confusion_matrix

- classification_report() → Shows precision, recall, F1-score.
- confusion_matrix() → Shows correct vs incorrect predictions in a table form.

3) Create SVM Model

```
svc = SVC(C=1.0, kernel='rbf', gamma='auto')
```

Parameter Meaning

C=1.0 Controls margin width vs errors. Higher C → strict fit.

kernel='rbf' Allows SVM to learn **non-linear** patterns.

gamma='auto' Controls influence of single training point.

This creates an **SVM Classifier** with RBF kernel.

4) Train the Model

```
svc.fit(X_train, y_train)
```

- The model learns patterns from training data.
-

5) Make Predictions

```
svc_pred = svc.predict(X_test)
```

- Predicts class labels for **unseen test data**.
-

6) Evaluate Model

```
print("SVM Accuracy:", accuracy_score(y_test, svc_pred))
```

- Accuracy = (Correct Predictions) / (Total Predictions)

```
print("SVM Classification Report:\n", classification_report(y_test, svc_pred))
```

Shows:

- **Precision**
- **Recall**
- **F1-Score**

```
print("SVM Confusion Matrix:\n", confusion_matrix(y_test, svc_pred))
```

Confusion Matrix Example:

	Predicted Yes	Predicted No
Actual Yes	TP	FN
Actual No	FP	TN

🎯 Final Understanding

Component	Purpose
SVM Model	Classifies data using maximum margin boundary
RBF Kernel	Handles non-linear patterns
Accuracy	Measures performance
Confusion Matrix	Shows prediction correctness
Classification Report	Detailed performance summary

SVM (Support Vector Machine)

SVM is a **supervised machine learning algorithm** used for **classification** and **regression**.

Most commonly, it is used for **classification**.

🎯 Goal of SVM

SVM tries to **separate data into different classes** by drawing a **best possible boundary** called a **hyperplane**.

This **hyperplane** separates classes **with the maximum margin** (maximum distance from the closest points of each class).

🌟 Key Concepts

1. Hyperplane

A line or plane that separates two classes.

- In **2D** → it's a **line**
 - In **3D** → it's a **plane**
 - In **higher dimensions** → called a **hyperplane**
-

2. Support Vectors

These are **data points closest to the hyperplane**.

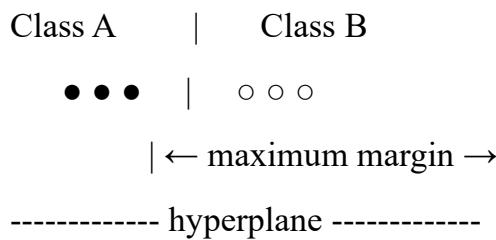
They are the most important points because:

- They **define the position** of the hyperplane
 - Removing them would **change the boundary**
-

3. Margin

The **distance between the hyperplane and support vectors**.

SVM tries to **maximize the margin** → which makes the model more **stable and generalized**.



4. Kernel Trick

If the data is **not linearly separable** (cannot be separated by a straight line), SVM uses a **kernel function** to **transform data into higher dimension** so that separation becomes possible.

Common kernels:

Kernel	Use
linear	Data already linearly separable
rbf (Gaussian)	Most common; for complex non-linear data
poly	Polynomial decision boundary



SVM Parameters

Parameter Meaning

C Controls margin softness. Higher C → less margin, more accuracy (risk overfit).

gamma (γ) Controls influence of each point. Higher γ → more complex boundary.

✓ Example Code (Classification)

```
from sklearn.svm import SVC  
svc = SVC(kernel='rbf', C=1.0, gamma='auto')  
svc.fit(X_train, y_train)  
y_pred = svc.predict(X_test)
```

📌 Advantages

- Works well in **high-dimensional data**
- Effective for **non-linear boundaries** (using kernels)
- Robust to **outliers**

⚠ Limitations

- Slow for very large datasets
 - Sensitive to feature **scaling**
 - Choosing right kernel and parameters requires tuning
-



Exam / Viva Answer (Short Version)

SVM is a supervised learning algorithm used for classification.

It finds the best separating boundary (hyperplane) between classes such that the margin between the hyperplane and the nearest data points (support vectors) is maximized.

If data is non-linear, SVM uses kernel functions like RBF to map data to higher dimensions for better separation.

If you want, I can now give:

- A **diagram** explanation (very helpful in viva)
- 5 **viva questions** and answers on SVM
- Comparison of **SVM vs KNN**

⌚ Code

```
knn = KNeighborsClassifier(n_neighbors=7)  
knn.fit(X_train, y_train)  
knn_pred = knn.predict(X_test)
```

🧠 Step-by-Step Explanation

1) Create the KNN Model

```
knn = KNeighborsClassifier(n_neighbors=7)
```

- This creates a **K-Nearest Neighbors classifier**.
- n_neighbors=7 means:

To classify an email, the model will look at the **7 nearest data points** (neighbors) in the training dataset and **choose the majority class**.

Parameter Meaning

n_neighbors=7 Use 7 closest training samples to make prediction

If:

- most neighbors are spam → **email is spam**
- most neighbors are not spam → **email is not spam**

2) Train (Fit) the Model

`knn.fit(X_train, y_train)`

- KNN is a **lazy learner**, meaning:

It does **not** learn any mathematical model during training.

- It simply **stores the training data**.
 - This step prepares the model by remembering feature patterns.
-

3) Predict the Output for Test Data

`knn_pred = knn.predict(X_test)`

- For each test email, KNN:
 1. Finds the **7 nearest neighbors** in the training set.
 2. Counts how many are Spam vs Not Spam.
 3. Assigns the **class with majority vote**.

Input	Output
-------	--------

`X_test` = new unseen emails `knn_pred` = predicted spam / not spam labels

✍ Viva-Ready Explanation (Speak This)

“In KNN, the `n_neighbors` value determines how many nearest neighbors vote for the class. Here, `K=7` means majority of the 7 closest emails decide whether the new email is spam or not. KNN stores the training samples during fit and uses distance comparison during prediction. It is simple, but can be slower when dataset is large.”

✓ Quick Summary Table

Step	Code Line	Meaning
Create the model	<code>knn = KNeighborsClassifier(n_neighbors=7)</code>	Choose <code>K = 7</code> nearest neighbors
Train the model	<code>knn.fit(X_train, y_train)</code>	Store training data

Step	Code Line	Meaning
	Predict results knn.predict(X_test)	Classify emails by voting

Want me to now explain how to evaluate KNN using:

accuracy_score()
classification_report()
confusion_matrix()

1) Scaling the Data

```
scaler = StandardScaler()
X_train_s = scaler.fit_transform(X_train)
X_test_s = scaler.transform(X_test)
```

Code	Meaning
StandardScaler()	Creates a scaler object to normalize the data
fit_transform(X_train)	Learns scaling from training data and applies it
transform(X_test)	Applies same scaling to test data

 **Important:** KNN and SVM need scaling because they compare **distances**. If features have different scales, the model becomes biased.

Example:

- Age (0–100) vs Salary (0–1,00,000) → Salary dominates distance → Wrong result.
-

2) List of K values to test

ks = [1, 3, 5]

We want to try:

- K = 1

- $K = 3$
- $K = 5$

To see which K gives best accuracy.

3) Dictionary to store results

results = {}

- This will store accuracy for each K .
- Example result will look like:

{1: 0.86, 3: 0.90, 5: 0.92}

4) Loop Through K Values

for k in ks:

```
knn = KNeighborsClassifier(n_neighbors=k, n_jobs=-1)
```

Parameter	Meaning
n_neighbors=k	Number of neighbors to consider
n_jobs=-1	Uses all CPU cores → faster training

5) Train the KNN Model

```
knn.fit(X_train_s, y_train)
```

- KNN stores training data (lazy learner).
 - It does not learn a function; it waits until prediction time.
-

6) Predict Class Labels

```
y_pred = knn.predict(X_test_s)
```

- For each new test point, KNN looks at the **k nearest training points** and **votes the majority class**.
-

7) Calculate Accuracy

```
acc = accuracy_score(y_test, y_pred)
```

Accuracy = Correct Predictions / Total Predictions

8) Confusion Matrix

```
cm = confusion_matrix(y_test, y_pred)
```

This shows how many:

- Correct predictions were made
 - Incorrect predictions were made
-

9) Classification Report

```
report = classification_report(y_test, y_pred, zero_division=0)
```

This gives:

- Precision
- Recall
- F1-score

zero_division=0 prevents errors if any class has no predictions.

10) Store Accuracy in Dictionary

```
results[k] = acc
```

11) Print Results

```
print(f"\nK = {k}:")  
print(f" Accuracy = {acc:.4f}")  
print(" Confusion Matrix:")  
print(cm)  
print(" Classification Report:")  
print(report)
```

The code prints:

- K value
- Accuracy

- Confusion Matrix
 - Classification Report
-

What This Whole Code Does

Step	Task
Scale data	Normalize before KNN
Try K = 1, 3, 5	Compare which K is better
Train KNN model	Use scaled training data
Predict	Use scaled test data
Evaluate	Accuracy, Confusion Matrix, Report
Store Results	Find best performing K

Final High-Level Understanding

- **Scaling is required** for KNN.
 - Lower K → model is more sensitive/noisy.
 - Higher K → smoother decision but may miss details.
 - You choose **the K with highest accuracy**.
-

If you want, I can now:

-  Show **which K is best**
-  Plot **Accuracy vs K** graph
-  Give **viva-ready KNN explanation**

Just say "**Show which K is best**" 😊

You said:

knn model

ChatGPT said:

Sure Omkar — here is a **simple, clear, exam-ready explanation of the KNN model**.

KNN (K-Nearest Neighbors)

KNN is a **supervised machine learning algorithm** used for **classification** and **regression**.

It is most commonly used for **classification**.

KNN is known as a **lazy learning algorithm** and a **distance-based algorithm**.

Main Idea

To classify a new data point, KNN looks at the **K nearest neighbors** (closest data points) in the training dataset and **assigns the most common class** among them.

So:

- **K = number of neighbors to consider**
 - Predictions are based on **majority voting**
-

How KNN Works (Step-by-Step)

1. Choose the value of **K** (example: $K = 3$)
 2. Compute the **distance** between the new data point and all training data points (commonly uses **Euclidean distance**)
 3. Select the **K nearest points**
 4. Look at their **classes**
 5. The class that appears the **most** → becomes the **predicted class**
-

Example

Suppose $K = 3$ and the 3 closest neighbors are:

Class A, Class A, Class B

→ Majority = Class A

→ **Prediction = Class A**

Why Scaling is Important?

KNN uses **distance**, so all features must be on **same scale**.

Therefore, we use:

```
from sklearn.preprocessing import StandardScaler
```

✓ KNN Code Example (Correct and Clean)

```
from sklearn.preprocessing import StandardScaler  
from sklearn.neighbors import KNeighborsClassifier  
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report  
  
# Scale the data  
scaler = StandardScaler()  
X_train_s = scaler.fit_transform(X_train)  
X_test_s = scaler.transform(X_test)  
  
# Create model  
knn = KNeighborsClassifier(n_neighbors=5)  
  
# Train  
knn.fit(X_train_s, y_train)  
  
# Predict  
knn_pred = knn.predict(X_test_s)  
  
# Evaluate  
print("KNN Accuracy:", accuracy_score(y_test, knn_pred))  
print("Confusion Matrix:\n", confusion_matrix(y_test, knn_pred))  
print("Classification Report:\n", classification_report(y_test, knn_pred))
```

█ Choosing the Best K

- If **K is too small (K=1,2)** → model becomes **noisy (overfitting)**
- If **K is too large** → model becomes **too smooth (underfitting)**

Common choice: **K = sqrt(number of samples)** or try values like 1,3,5,7,9 and pick best accuracy.

Advantages of KNN

Advantage	Reason
Simple and easy	No complicated training
No model assumption	Works on any shaped data
Works well for small datasets	Fast and accurate

Limitations of KNN

Limitation	Reason
Slow for large data	Must compare with all points
Sensitive to scaling	Needs normalization
Sensitive to noisy data	Outliers may affect neighbors

Exam / Viva Ready Answer

KNN (K-Nearest Neighbors) is a classification algorithm that classifies a new data point based on the majority class of its K nearest neighbors in the feature space. It uses distance measures such as Euclidean distance to find the closest points. KNN is a lazy learner and requires feature scaling. The selection of K is important—small K leads to overfitting and large K leads to underfitting

Overfitting and Underfitting

These terms describe how well a machine learning model has **learned** from the training data.

1. Underfitting

Meaning:

The model is **too simple** and does **not learn enough** from the data.

Symptoms:

- **High error** on training data
- **High error** on test data
- Model **fails to capture patterns**

Example:

Using **Linear Regression** for a very complex dataset.

Graph:

Actual Pattern: ~~~~ (complex curve)

Model Learned: ---- (simple straight line)

Reason:

- Model is not powerful enough
- Too few features
- Too much regularization

Solution:

- Use a more complex model (ex: Random Forest, SVM)
 - Increase features
 - Reduce regularization
-

2. Overfitting

Meaning:

The model learns **too much**, including **noise**, instead of focusing on the real pattern.

Symptoms:

- **Very low error on training data**
- **High error on test data**
- Model **memorizes** data instead of generalizing

Example:

Using **too many trees or depth in Decision Tree**.

Graph:

Actual Pattern: ~~~~

Model Learned: ~W^~_/\// (too wiggly)

Reason:

- Model is too complex
- Too many features
- Insufficient training data

Solution:

- Reduce model complexity (prune tree / reduce layers)
- Use **Regularization** (L1 / L2)
- Increase training data
- Use **Cross-validation**

Key Difference Summary

Feature	Underfitting	Overfitting
Model Complexity	Too Low (Simple Model)	Too High (Complex Model)
Training Accuracy	Low	High
Testing Accuracy	Low	Low
Generalization	Poor	Poor
Behavior	Does not learn pattern	Learns noise also

Best Model Goal

We want a model that:

- Learns patterns (not noise)
- Performs **well on both training and testing data**

This is called **Good Generalization**.

Exam / Viva Answer

Underfitting occurs when the model is too simple and cannot capture the underlying pattern of the data, resulting in high error on both training and testing sets.

Overfitting occurs when the model becomes too complex and learns noise along with the pattern, resulting in very low training error but high testing error.

To avoid underfitting, increase model complexity; to avoid overfitting, reduce model complexity or use regularization/cross-validation.