ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

INTRODUCTION TO TIME SERIES ANALYSIS

CHAPTER 1: WHAT IS TIME SERIES ANALYSIS?

1.1 Definition

Time series analysis is a statistical technique used to analyze **sequential data points** collected over time. It helps in identifying trends, seasonal patterns, and making predictions based on historical data.

1.2 Characteristics of Time Series Data

- **Time-Dependent**: Observations are collected at different time intervals.
- **Ordered Data**: Unlike regular datasets, time series data maintains a chronological order.
- **Patterns and Trends**: Data may show upward/downward trends or seasonal effects.

1.3 Examples of Time Series Data

- **Finance**: Stock prices, exchange rates, inflation rates.
- **Weather**: Temperature, rainfall, wind speed.

- **Economics**: GDP growth, unemployment rates.
- **Healthcare**: Patient counts, disease outbreak trends.
- **Retail**: Sales data, website traffic.

🔹 Chapter 2: Components of Time Series Data

2.1 Trend

A long-term increase or decrease in the data over time. ✓ **Example**: The increase in global temperatures over decades.

2.2 Seasonality

Recurring patterns at fixed intervals (daily, weekly, monthly, yearly). ✓ **Example**: Higher ice cream sales in summer.

2.3 Cyclical Patterns

Fluctuations that occur over long time periods due to economic cycles. ✓ **Example**: Business cycles affecting market trends.

2.4 Irregular (Residual) Variations

Unpredictable fluctuations due to unforeseen events (e.g., natural disasters, pandemics).

🔹 Chapter 3: Types of Time Series Models

3.1 Deterministic vs. Stochastic Models

- **Deterministic Models**: Predictable changes (e.g., linear trends).
- **Stochastic Models**: Include random variations (e.g., stock prices).

3.2 Additive vs. Multiplicative Models

- **Additive Model:**

$$Y(t) = Trend + Seasonality + Noise$$

✔ Used when variations are constant over time.

- **Multiplicative Model:**

$$Y(t) = Trend \times Seasonality \times Noise$$

✔ Used when variations grow over time.

🔹 CHAPTER 4: DATA PREPROCESSING FOR TIME SERIES ANALYSIS

4.1 Handling Missing Data

Missing values can disrupt time series models. Techniques include: ✓ **Forward Fill**: Use the previous value.
✓ **Interpolation**: Estimate missing values.
✓ **Dropping Missing Values**: If few values are missing.

4.2 Resampling Data

Changing the time frequency (e.g., daily to weekly) for better analysis.

df.resample('M').mean() # Resample daily data into monthly averages

4.3 Smoothing Techniques

Used to reduce noise: ✓ **Moving Averages**
✓ **Exponential Smoothing**

- CHAPTER 5: TIME SERIES VISUALIZATION

5.1 Line Plot

A basic method to visualize trends.

```
import pandas as pd

import matplotlib.pyplot as plt


df = pd.read_csv("timeseries_data.csv", parse_dates=["Date"], index_col="Date")

df.plot(figsize=(10, 5), title="Time Series Data")

plt.show()
```

5.2 Rolling Mean (Moving Average)

Used to smooth out short-term fluctuations.

```
df["Rolling_Mean"] = df["Value"].rolling(window=7).mean()

df[["Value", "Rolling_Mean"]].plot(figsize=(10, 5))

plt.show()
```

5.3 Seasonal Decomposition

Breaks down a time series into trend, seasonality, and residuals.

```
from statsmodels.tsa.seasonal import seasonal_decompose


result = seasonal_decompose(df["Value"], model="additive")

result.plot()

plt.show()
```

CHAPTER 6: FORECASTING TECHNIQUES

6.1 Naïve Forecasting

Uses the last observed value as the next prediction. ✓ **Example**: Tomorrow's stock price is assumed to be today's stock price.

6.2 Moving Averages

Predicts using the average of past observations.

```
df["Forecast"] = df["Value"].rolling(window=5).mean()

df.plot(figsize=(10,5))

plt.show()
```

6.3 Exponential Smoothing

Gives more weight to recent observations.

```
from statsmodels.tsa.holtwinters import SimpleExpSmoothing


model = SimpleExpSmoothing(df["Value"]).fit()

df["Forecast"] = model.fittedvalues

df.plot(figsize=(10, 5))

plt.show()
```

6.4 ARIMA (AutoRegressive Integrated Moving Average)

A powerful forecasting model.

```
from statsmodels.tsa.arima.model import ARIMA


model = ARIMA(df["Value"], order=(2,1,2))  # ARIMA(p,d,q)

model_fit = model.fit()
```

```
df["Forecast"] = model_fit.fittedvalues

df.plot(figsize=(10,5))

plt.show()
```

◆ CHAPTER 7: PERFORMANCE EVALUATION METRICS

7.1 Mean Absolute Error (MAE)

Measures the average absolute difference between actual and predicted values.

$$MAE = \frac{1}{n}\sum |y_i - \hat{y}_i|$$

```
from sklearn.metrics import mean_absolute_error

mae = mean_absolute_error(y_actual, y_predicted)

print(f"MAE: {mae}")
```

7.2 Mean Squared Error (MSE)

Squares the errors before averaging.

$$MSE = \frac{1}{n}\sum (y_i - \hat{y}_i)^2$$

```
from sklearn.metrics import mean_squared_error

mse = mean_squared_error(y_actual, y_predicted)

print(f"MSE: {mse}")
```

7.3 Root Mean Squared Error (RMSE)

$$RMSE = \sqrt{MSE}$$

```
rmse = np.sqrt(mse)

print(f"RMSE: {rmse}")
```

7.4 Mean Absolute Percentage Error (MAPE)

$$MAPE = \frac{1}{n} \sum \left| \frac{y_i - \hat{y}_i}{y_i} \right| \times 100$$

```
mape = np.mean(np.abs((y_actual - y_predicted) / y_actual)) * 100

print(f"MAPE: {mape}%")
```

◆ CHAPTER 8: REAL-WORLD APPLICATIONS OF TIME SERIES ANALYSIS

8.1 Stock Market Prediction

✓ Predicting future stock prices based on historical trends.

8.2 Sales Forecasting

✓ Predicting product demand for inventory management.

8.3 Weather Forecasting

✓ Predicting temperatures, rainfall, and storms.

8.4 Energy Consumption

✓ Forecasting electricity demand to optimize power grids.

8.5 Healthcare

✓ Monitoring patient data trends for disease prediction.

🔹 Chapter 9: Summary

✅ **Time Series Analysis** is crucial for understanding sequential data.

✅ **Key components** include trend, seasonality, and cyclic patterns.

✅ **Visualization techniques** (line plots, decomposition) help understand data.

✅ **Forecasting methods** range from simple moving averages to advanced ARIMA models.

✅ **Performance metrics** (MAE, MSE, RMSE) evaluate model accuracy.

🔹 Chapter 10: Next Steps

- **Try advanced models like LSTMs** for time series forecasting.
- **Use real-world datasets** like stock prices or weather data.
- **Experiment with hyperparameter tuning** for ARIMA models.

☑ Forecasting Models: ARIMA, SARIMA & Exponential Smoothing

📌 Chapter 1: Introduction to Forecasting Models

1.1 What is Forecasting?

Forecasting is the process of predicting **future values** based on historical data. It is widely used in **finance, supply chain, stock markets, sales, weather prediction, and economic analysis**.

1.2 Why is Forecasting Important?

✅ Helps businesses make **data-driven decisions**.
✅ Reduces **uncertainty** in planning and resource allocation.
✅ Improves **inventory management** and demand prediction.

1.3 Common Types of Forecasting Models

1. **Time Series Models** - ARIMA, SARIMA, Exponential Smoothing
2. **Machine Learning Models** - Random Forest, XGBoost, LSTMs
3. **Deep Learning Models** - RNNs, Transformer-based models

This study material focuses on **three traditional time series models**:

1. **ARIMA (AutoRegressive Integrated Moving Average)**
2. **SARIMA (Seasonal ARIMA)**
3. **Exponential Smoothing (ETS)**

📌 Chapter 2: ARIMA (AutoRegressive Integrated Moving Average)

2.1 What is ARIMA?

ARIMA (AutoRegressive Integrated Moving Average) is a powerful forecasting model used for **univariate time series** data.

2.2 How ARIMA Works?

ARIMA is defined by three components:

- **AutoRegressive (AR)** - Uses past values to predict future values.
- **Integrated (I)** - Differencing is applied to make data stationary.
- **Moving Average (MA)** - Uses past forecast errors for predictions.

ARIMA is denoted as **ARIMA(p, d, q)**:

- **p (AR order)** - Number of past observations used.
- **d (Differencing order)** - Number of times differencing is applied.
- **q (MA order)** - Number of past forecast errors used.

2.3 When to Use ARIMA?

✅ When the time series is **non-seasonal**.
✅ When data follows a **trend but no clear seasonal pattern**.
✅ When forecasting **short-term future values**.

2.4 Advantages of ARIMA

✓ Works well for **short-term forecasting**.

✓ Captures **trends and relationships** effectively.

✓ Can be fine-tuned using **p, d, q parameters**.

2.5 Disadvantages of ARIMA

✗ Does not handle **seasonality** well (use SARIMA instead).

✗ Requires **manual parameter tuning**.

✗ Sensitive to **outliers and missing values**.

2.6 Implementing ARIMA in Python

```
import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from statsmodels.tsa.arima.model import ARIMA


# Load dataset (Example: Monthly sales data)

df = pd.read_csv("sales_data.csv", index_col='Date',
parse_dates=True)


# Fit ARIMA model (p=2, d=1, q=2)

model = ARIMA(df['Sales'], order=(2,1,2))

model_fit = model.fit()


# Forecast next 12 periods

forecast = model_fit.forecast(steps=12)
```

```
# Plot results
plt.figure(figsize=(10,5))
plt.plot(df['Sales'], label="Actual Sales")
plt.plot(forecast, label="Forecast", linestyle="dashed")
plt.legend()
plt.title("ARIMA Forecast")
plt.show()
```

📌 Chapter 3: SARIMA (Seasonal ARIMA)

3.1 What is SARIMA?

SARIMA (Seasonal ARIMA) extends ARIMA to handle **seasonality** in time series data.

3.2 How SARIMA Works?

SARIMA is denoted as **SARIMA(p, d, q) × (P, D, Q, m)**:

- **p, d, q** - Standard ARIMA components.
- **P, D, Q** - Seasonal AR, Differencing, and MA components.
- **m** - Seasonal period (e.g., 12 for monthly data, 7 for weekly data).

3.3 When to Use SARIMA?

✅ When the time series exhibits **seasonal patterns** (e.g., sales, temperature, demand).
✅ When ARIMA fails due to **seasonal fluctuations**.

3.4 Advantages of SARIMA

✓ Handles both **trends and seasonality**.

✓ Works well for **time series forecasting with seasonal fluctuations**.

3.5 Disadvantages of SARIMA

✗ Computationally **expensive** for large datasets.

✗ Parameter tuning requires **trial and error**.

3.6 Implementing SARIMA in Python

```
from statsmodels.tsa.statespace.sarimax import SARIMAX


# Fit SARIMA model (p=2, d=1, q=2) x (P=1, D=1, Q=1, m=12)

sarima_model = SARIMAX(df['Sales'], order=(2,1,2),
seasonal_order=(1,1,1,12))

sarima_fit = sarima_model.fit()


# Forecast next 12 periods

sarima_forecast = sarima_fit.forecast(steps=12)


# Plot results

plt.figure(figsize=(10,5))

plt.plot(df['Sales'], label="Actual Sales")

plt.plot(sarima_forecast, label="SARIMA Forecast",
linestyle="dashed")

plt.legend()

plt.title("SARIMA Forecast")
```

```
plt.show()
```

📌 Chapter 4: Exponential Smoothing

4.1 What is Exponential Smoothing?

Exponential Smoothing (ETS) is a simple but effective method for forecasting by assigning **exponentially decreasing weights** to past observations.

4.2 Types of Exponential Smoothing

1. **Simple Exponential Smoothing (SES)** - For data with **no trend or seasonality**.
2. **Holt's Exponential Smoothing** - Handles **trends** in data.
3. **Holt-Winters (Triple) Exponential Smoothing** - Handles both **trends and seasonality**.

4.3 When to Use Exponential Smoothing?

✅ When the time series **lacks strong patterns** (for SES).
✅ When data exhibits **trend and seasonality** (Holt-Winters).
✅ When a **simple and fast method** is required.

4.4 Advantages of Exponential Smoothing

✓ **Easy to implement and computationally efficient**.
✓ Works well for **short-term forecasting**.
✓ Can handle **trend and seasonal components** (Holt-Winters).

4.5 Disadvantages of Exponential Smoothing

✗ Does not handle **complex relationships** like ARIMA/SARIMA.
✗ Requires **manual tuning of parameters**.

4.6 Implementing Exponential Smoothing in Python

```
from statsmodels.tsa.holtwinters import ExponentialSmoothing


# Fit Holt-Winters Exponential Smoothing

hw_model = ExponentialSmoothing(df['Sales'], trend='add',
seasonal='add', seasonal_periods=12)

hw_fit = hw_model.fit()


# Forecast next 12 periods

hw_forecast = hw_fit.forecast(steps=12)


# Plot results

plt.figure(figsize=(10,5))

plt.plot(df['Sales'], label="Actual Sales")

plt.plot(hw_forecast, label="Holt-Winters Forecast",
linestyle="dashed")

plt.legend()

plt.title("Exponential Smoothing Forecast")

plt.show()
```

📌 Chapter 5: Comparison of Forecasting Models

Feature	ARIMA	SARIMA	Exponential Smoothing
Best For	Trend-based data	Seasonal data	Simple patterns
Handles Seasonality?	❌ No	✅ Yes	✅ Yes (Holt-Winters)
Computational Cost	Medium	High	Low
Short-Term Forecasting	✅ Yes	✅ Yes	✅ Yes
Long-Term Forecasting	❌ No	✅ Yes	❌ No

📌 Chapter 6: Summary & Conclusion

6.1 Key Takeaways

✅ **ARIMA** is best for **trend-based, non-seasonal** data.
✅ **SARIMA** is best for **seasonal time series**.
✅ **Exponential Smoothing** is best for **simple short-term forecasting**.

6.2 Next Steps

- Try **auto-tuning parameters** using **Auto-ARIMA**.
- Apply **deep learning-based forecasting (LSTMs, Transformer models)**.

- Experiment with **real-world datasets** (e.g., stock prices, sales data).

☑ LSTM FOR TIME SERIES PREDICTION

📌 CHAPTER 1: INTRODUCTION TO TIME SERIES FORECASTING

1.1 What is Time Series Prediction?

Time series prediction involves forecasting future values based on past observations. A **time series** is a sequence of data points recorded at regular time intervals (e.g., daily stock prices, temperature readings, sales data).

1.2 Why Use LSTMs for Time Series?

Traditional machine learning models (e.g., ARIMA, Linear Regression) fail to capture **long-term dependencies**. **Long Short-Term Memory (LSTM) networks** are designed to handle **sequential data**, making them **ideal for time series forecasting**.

1.3 Applications of Time Series Forecasting

- **Stock Market Prediction** 📈
- **Weather Forecasting** 🌦️
- **Sales & Demand Forecasting** 🛒
- **Energy Consumption Prediction** ⚡
- **Healthcare Monitoring (ECG, BP Trends)** 🏥

📌 CHAPTER 2: UNDERSTANDING LSTMS

2.1 What is an LSTM?

LSTM (Long Short-Term Memory) is a special type of **Recurrent Neural Network (RNN)** that solves the **vanishing gradient problem,** allowing it to learn long-term dependencies.

2.2 LSTM Cell Structure

An LSTM cell consists of:

1. **Forget Gate**: Decides which past information to forget.
2. **Input Gate**: Updates the cell state with new information.
3. **Cell State**: Stores long-term memory.
4. **Output Gate**: Produces the final output.

2.3 Why LSTMs for Time Series?

✅ Can learn patterns from sequential data.
✅ Handles **long-term dependencies** better than RNNs.
✅ Suitable for **multivariate forecasting** (multiple features).

📌 Chapter 3: Preparing Data for LSTM

3.1 Steps in Time Series Forecasting

1. **Load Dataset** 📥
2. **Preprocess Data** (scaling, handling missing values)
3. **Convert Data into Sequences** (LSTMs need sequential inputs)
4. **Train the LSTM Model** 🤖
5. **Make Predictions & Evaluate Performance** 📊

📌 Chapter 4: Implementing LSTM for Time Series Prediction

We will use **Python, TensorFlow, and Keras** to build an LSTM model for **predicting stock prices**.

4.1 Install Required Libraries

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

4.2 Import Libraries

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
```

4.3 Load & Visualize Dataset

```
# Load dataset (Example: Apple stock prices)
df = pd.read_csv('AAPL.csv', usecols=['Date', 'Close'])
df['Date'] = pd.to_datetime(df['Date'])
df.set_index('Date', inplace=True)

# Plot closing prices
plt.figure(figsize=(10,5))
plt.plot(df['Close'], label="Closing Price")
plt.xlabel("Date")
plt.ylabel("Price")
plt.title("Stock Price Over Time")
```

```
plt.legend()
plt.show()
```

📌 Chapter 5: Preprocessing for LSTM

5.1 Scale the Data

LSTMs perform better with **scaled data**.

```
scaler = MinMaxScaler(feature_range=(0,1))
df_scaled = scaler.fit_transform(df[['Close']])
```

5.2 Convert Data into Sequences

LSTMs require **input sequences** (X) and **target values** (Y).

```
def create_sequences(data, time_steps=50):
    X, Y = [], []
    for i in range(len(data) - time_steps):
        X.append(data[i:i+time_steps])
        Y.append(data[i+time_steps])
    return np.array(X), np.array(Y)

# Define time steps
time_steps = 50
X, Y = create_sequences(df_scaled, time_steps)

# Split into training & testing sets
```

```
split = int(0.8 * len(X))

X_train, Y_train = X[:split], Y[:split]

X_test, Y_test = X[split:], Y[split:]


print("Training Data Shape:", X_train.shape, Y_train.shape)

print("Testing Data Shape:", X_test.shape, Y_test.shape)
```

📌 Chapter 6: Build & Train LSTM Model

6.1 Define the LSTM Model

```
model = Sequential([

  LSTM(50, activation='relu', return_sequences=True, input_shape=(time_steps, 1)),

  LSTM(50, activation='relu', return_sequences=False),

  Dense(25),

  Dense(1)

])


# Compile the model

model.compile(optimizer='adam', loss='mean_squared_error')


# Display model summary

model.summary()
```

6.2 Train the Model

history = model.fit(X_train, Y_train, epochs=20, batch_size=32, validation_data=(X_test, Y_test))

📌 Chapter 7: Evaluate & Make Predictions

7.1 Plot Training Loss

plt.plot(history.history['loss'], label='Training Loss')

plt.plot(history.history['val_loss'], label='Validation Loss')

plt.legend()

plt.title("Model Loss Over Epochs")

plt.show()

7.2 Make Predictions

Y_pred = model.predict(X_test)

Y_pred = scaler.inverse_transform(Y_pred) # Convert back to original scale

Y_test_actual = scaler.inverse_transform(Y_test.reshape(-1,1))

Plot actual vs predicted values

plt.figure(figsize=(10,5))

plt.plot(Y_test_actual, label="Actual Price")

plt.plot(Y_pred, label="Predicted Price")

plt.xlabel("Time")

plt.ylabel("Price")

plt.title("Stock Price Prediction")

```
plt.legend()

plt.show()
```

📌 Chapter 8: Advanced LSTM Techniques

8.1 Improving Performance

- **Use More Data** 📊 – Train on a longer time period.
- **Hyperparameter Tuning** 🎛️ – Adjust LSTM layers, neurons, and learning rate.
- **Bidirectional LSTM** 🔁 – Captures both past and future trends.

8.2 Using LSTM for Multivariate Forecasting

Instead of using **only the closing price**, add **features like volume, moving averages**.

```
df['MA_50'] = df['Close'].rolling(50).mean()  # 50-day Moving Average

df.dropna(inplace=True)  # Drop NaN values
```

Modify create_sequences() to include multiple features.

📌 Chapter 9: Summary & Next Steps

9.1 Key Takeaways

✅ **LSTMs excel in time series prediction** by capturing long-term dependencies.
✅ **Preprocessing is crucial** (scaling, sequence creation).
✅ **Hyperparameter tuning** improves model performance.

✅ **Advanced LSTMs (Bidirectional, Attention-based)** enhance accuracy.

9.2 Next Steps

- Try **Bidirectional LSTM** for better context learning.
- Use **Transformer models (like BERT for time series)**.
- Deploy the trained model as a **Web App (using Flask/Streamlit).**

☑ Stock Price Prediction using Machine Learning

📌 Chapter 1: Introduction to Stock Price Prediction

1.1 What is Stock Price Prediction?

Stock price prediction involves using **machine learning (ML) models** to analyze historical stock data and forecast future stock prices. It combines **data science, financial analysis, and predictive modeling**.

1.2 Why Use Machine Learning for Stock Prediction?

✅ Identifies **patterns and trends** in stock prices.
✅ Reduces **human biases** in decision-making.
✅ Helps in **risk management** and portfolio optimization.
✅ Improves **short-term and long-term trading strategies**.

1.3 Challenges in Stock Price Prediction

- **Stock prices are highly volatile** (affected by news, market trends, etc.).
- **Data quality matters** (historical prices, indicators, news sentiment).
- **Risk of overfitting** if models are too complex.

📌 Chapter 2: Types of Machine Learning Models for Stock Prediction

2.1 Traditional vs. Machine Learning Approaches

Approach	Description	Examples
Traditional	Uses statistical methods	Moving Averages, ARIMA
ML-Based	Uses AI to learn patterns	Linear Regression, LSTMs

2.2 Common ML Models Used

1. **Linear Regression** – Predicts future prices based on historical trends.
2. **Decision Trees & Random Forests** – Handles non-linearity in data.
3. **Support Vector Machines (SVMs)** – Finds optimal trends in stock price data.
4. **Recurrent Neural Networks (RNNs) / LSTMs** – Captures time-series dependencies.
5. **XGBoost** – Powerful for structured financial data.

📌 CHAPTER 3: STOCK PRICE PREDICTION USING PYTHON

3.1 Install Required Libraries

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

3.2 Import Libraries

```
import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import yfinance as yf

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LinearRegression
```

```
from sklearn.metrics import mean_squared_error
```

📌 Chapter 4: Load and Prepare Stock Data

4.1 Download Stock Data using Yahoo Finance

```
# Download Apple (AAPL) stock data from Yahoo Finance
df = yf.download('AAPL', start='2015-01-01', end='2023-01-01')


# Display first few rows
print(df.head())
```

4.2 Select Features

```
# Selecting only 'Close' price for prediction
df = df[['Close']]


# Create a new column 'Target' (Shifted Close Price)
df['Target'] = df['Close'].shift(-1)


# Drop NaN values
df.dropna(inplace=True)


# Display dataset
print(df.head())
```

📌 Chapter 5: Splitting Data and Training a Machine Learning Model

5.1 Splitting the Data

```
# Splitting into features (X) and target (y)

X = df[['Close']]

y = df['Target']


# Splitting into training and test sets

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

5.2 Train a Linear Regression Model

```
# Train the model

model = LinearRegression()

model.fit(X_train, y_train)


# Make predictions

y_pred = model.predict(X_test)
```

5.3 Evaluate the Model

```
# Calculate Mean Squared Error

mse = mean_squared_error(y_test, y_pred)

print(f"Mean Squared Error: {mse:.2f}")
```

📌 Chapter 6: Visualizing the Predictions

```
plt.figure(figsize=(10,5))

plt.plot(y_test.values, label='Actual Prices', color='blue')

plt.plot(y_pred, label='Predicted Prices', color='red',
linestyle='dashed')

plt.xlabel("Time")

plt.ylabel("Stock Price")

plt.title("Stock Price Prediction - Linear Regression")

plt.legend()

plt.show()
```

📌 Chapter 7: Advanced Techniques

7.1 Using Moving Averages for Smoother Predictions

```
df['SMA_50'] = df['Close'].rolling(window=50).mean()

df['SMA_200'] = df['Close'].rolling(window=200).mean()


plt.figure(figsize=(10,5))

plt.plot(df['Close'], label='Stock Price', color='blue')

plt.plot(df['SMA_50'], label='50-Day SMA', color='green')

plt.plot(df['SMA_200'], label='200-Day SMA', color='red')

plt.title("Stock Price with Moving Averages")

plt.legend()

plt.show()
```

7.2 Using LSTM for Deep Learning-Based Prediction

LSTMs (Long Short-Term Memory networks) are used for time-series forecasting.

```
from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import LSTM, Dense


# Reshape data for LSTM

X_train_reshaped = np.reshape(X_train.values, (X_train.shape[0], X_train.shape[1], 1))

X_test_reshaped = np.reshape(X_test.values, (X_test.shape[0], X_test.shape[1], 1))


# Define LSTM model

model = Sequential([

  LSTM(units=50, return_sequences=True, input_shape=(X_train_reshaped.shape[1], 1)),

  LSTM(units=50, return_sequences=False),

  Dense(units=25),

  Dense(units=1)

])


# Compile model

model.compile(optimizer='adam', loss='mean_squared_error')
```

```
# Train the model
model.fit(X_train_reshaped, y_train, epochs=10, batch_size=32)
```

📌 Chapter 8: Summary

Model	Best For	Advantages	Disadvantages
Linear Regression	Basic trend prediction	Simple & interpretable	Can't capture complex patterns
Decision Trees	Handling non-linear data	Works with missing data	Prone to overfitting
Random Forest	Robust stock forecasting	Reduces overfitting	Computationally expensive
LSTMs (Deep Learning)	Time-series forecasting	Captures dependencies in stock data	Needs large datasets

---

📌 Chapter 9: Conclusion & Next Steps

9.1 Key Takeaways

✅ Stock prices can be predicted using **ML techniques like Linear Regression, Decision Trees, and LSTMs**.

✅ **Feature engineering** (like Moving Averages) improves model accuracy.

✅ **LSTMs are powerful** but require **more data and computing power**.

9.2 Next Steps

🚀 **Try different ML models** (SVM, XGBoost, Deep Learning).
🚀 **Use fundamental/technical indicators** (RSI, MACD).
🚀 **Apply NLP to analyze stock-related news sentiment**.

☑ AI FOR FRAUD DETECTION & RISK MANAGEMENT

📌 CHAPTER 1: INTRODUCTION TO AI IN FRAUD DETECTION & RISK MANAGEMENT

1.1 What is Fraud Detection & Risk Management?

Fraud detection and risk management refer to the processes used to **identify, prevent, and mitigate fraudulent activities** and financial risks in various industries. These activities include **credit card fraud, insurance fraud, money laundering, identity theft, and cyber fraud**.

1.2 Why Use AI for Fraud Detection & Risk Management?

Traditional fraud detection methods rely on **rule-based systems**, which are: ❌ **Slow and reactive**
❌ **Prone to human errors**
❌ **Inefficient at detecting new fraud patterns**

✅ **AI-based fraud detection** uses **Machine Learning (ML) and Deep Learning** to: ✓ Detect fraud in **real-time**.
✓ Identify **hidden patterns** in large datasets.
✓ Adapt to **new fraud techniques** without human intervention.
✓ Reduce **false positives** in fraud detection.

1.3 Key Applications of AI in Fraud & Risk Management

Industry	Application
Banking & Finance	Credit card fraud, transaction monitoring
E-commerce	Fake accounts, fraudulent transactions

Healthcare	Insurance fraud, medical claims fraud
Cybersecurity	Identity theft, phishing attacks
Gaming & Gambling	Account takeovers, bonus abuse fraud

📌 Chapter 2: AI Techniques Used in Fraud Detection

2.1 Machine Learning (ML) for Fraud Detection

ML algorithms can analyze **large datasets** and **learn fraud patterns**. They classify transactions as **fraudulent (1) or legitimate (0)** based on historical data.

2.2 Supervised vs. Unsupervised Learning

Approach	Description	Example Algorithm
Supervised Learning	Uses labeled fraud data to train models	Logistic Regression, Decision Trees, Random Forest
Unsupervised Learning	Detects anomalies in **unlabeled** data	K-Means Clustering, Autoencoders, Isolation Forest
Reinforcement Learning	Learns from past fraud cases and improves decisions	Deep Q-Learning

2.3 Deep Learning for Fraud Detection

✅ **Recurrent Neural Networks (RNNs)** – Used for detecting fraud in **sequential transactions** (e.g., credit card fraud).

✅ **Long Short-Term Memory (LSTM)** – Detects **time-series fraud patterns**.

✅ **Autoencoders** – Used for **anomaly detection** in cybersecurity threats.

2.4 Natural Language Processing (NLP)

✅ **Text analysis for fraud detection** in emails, chat messages, and online reviews.
✅ **Detects phishing attempts** by analyzing patterns in scam messages.

📌 Chapter 3: Fraud Detection Workflow Using AI

3.1 Data Collection

- **Transaction logs** (banking, e-commerce, cryptocurrency)
- **Customer behavior data** (login locations, time of transactions)
- **Financial records** (credit history, loan applications)

3.2 Data Preprocessing

- **Handling missing values**
- **Feature engineering** (e.g., calculating transaction frequency)
- **Converting categorical data** (e.g., transaction type) into numerical values

3.3 Training AI Models

- **Choose an algorithm** (e.g., Random Forest, Neural Networks)
- **Split data into training & test sets** (80%-20% rule)
- **Train model to classify transactions as 'fraud' or 'legitimate'**

3.4 Real-Time Fraud Detection

- AI model **monitors transactions in real-time**

- If fraud probability **> threshold,** trigger **alert/block transaction**
- If fraud probability **low,** transaction proceeds normally

---

📌 Chapter 4: Implementing AI for Fraud Detection

4.1 Install Dependencies

```
!pip install numpy pandas scikit-learn matplotlib seaborn tensorflow keras
```

4.2 Load and Explore Fraud Dataset

```
import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns


# Load dataset (example: Credit Card Fraud dataset)

df = pd.read_csv("creditcard.csv")


# Display first few rows

print(df.head())


# Check for missing values

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

4.3 Data Preprocessing

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Features and target variable
X = df.drop(columns=['Class'])  # Features
y = df['Class']  # Target: 0 (legit), 1 (fraud)

# Scale the data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)

print("Training set size:", X_train.shape)
print("Test set size:", X_test.shape)
```

4.4 Train a Machine Learning Model for Fraud Detection

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report
```

```
# Train a Random Forest model
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)

# Evaluate model performance
print("Accuracy:", accuracy_score(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

4.5 Using Deep Learning for Fraud Detection (LSTM)

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout

# Reshape data for LSTM
X_train_lstm = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))
X_test_lstm = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))
```

```
# Build LSTM model
lstm_model = Sequential([
  LSTM(64, return_sequences=True, input_shape=(X_train.shape[1], 1)),
  Dropout(0.2),
  LSTM(32, return_sequences=False),
  Dense(1, activation='sigmoid')
])

# Compile model
lstm_model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train model
lstm_model.fit(X_train_lstm, y_train, epochs=5, batch_size=32, validation_data=(X_test_lstm, y_test))
```

📌 Chapter 5: Real-World AI Fraud Detection Use Cases

5.1 AI in Banking

✅ **Credit card fraud detection** using ML and deep learning models.
✅ **Loan risk assessment** – AI evaluates customers' credit scores for loan approvals.

5.2 AI in Cybersecurity

✅ **AI-powered anomaly detection** to stop hacking attempts.
✅ **AI detects phishing attacks** based on email patterns.

5.3 AI in Insurance

✅ **Automated claims processing** using AI-powered fraud detection.
✅ **AI flagging suspicious claims** based on customer history.

📌 Chapter 6: Challenges & Future of AI in Fraud Detection

6.1 Challenges

❌ **Imbalanced Datasets** – Fraud cases are rare, making training difficult.
❌ **False Positives** – Overly aggressive fraud detection can block real customers.
❌ **Evolving Fraud Techniques** – Fraudsters constantly develop new methods.

6.2 Future Trends

🚀 **AI-Powered Blockchain for Security**
🚀 **Explainable AI (XAI) for better decision-making**
🚀 **Deep Reinforcement Learning for adaptive fraud detection**

📌 Chapter 7: Summary

✅ **AI fraud detection reduces risks in banking, e-commerce, and cybersecurity.**
✅ **ML and DL techniques detect fraud patterns in large datasets.**
✅ **Real-time fraud detection is essential for stopping cyber**

threats.

✅ **LSTM models work well for sequential fraud analysis.**

📌 Chapter 8: Next Steps

🚀 Try implementing **AI-powered fraud detection on real-world datasets**.

🚀 Use **Autoencoders for Anomaly Detection** in cybersecurity.

🚀 Deploy a **Fraud Detection API for banking applications**.

📌 ✨ ASSIGNMENT 1:

🎯 TRAIN AN **ARIMA MODEL** FOR STOCK PRICE PREDICTION.

📌 ✨ Assignment Solution 1: Train an ARIMA Model for Stock Price Prediction

🎯 Objective

The goal of this assignment is to **train an ARIMA model** to predict stock prices based on historical stock market data. ARIMA (AutoRegressive Integrated Moving Average) is a powerful statistical method for **time series forecasting**.

We will:

- ✅ Load historical stock price data
- ✅ Perform **data preprocessing** and visualization
- ✅ **Check stationarity** and apply transformations if needed
- ✅ Train an **ARIMA model** and evaluate its performance
- ✅ Make **future predictions**

🛠 Step 1: Install and Import Required Libraries

We will use **pandas, matplotlib, statsmodels, and yfinance** to train our ARIMA model.

1.1 Install Dependencies (if not installed)

```
!pip install pandas numpy matplotlib statsmodels yfinance
```

1.2 Import Required Libraries

```
import pandas as pd

import numpy as np
```

```
import matplotlib.pyplot as plt

import statsmodels.api as sm

import yfinance as yf

from statsmodels.tsa.stattools import adfuller

from statsmodels.tsa.arima.model import ARIMA

from sklearn.metrics import mean_absolute_error,
mean_squared_error
```

📥 Step 2: Load Stock Price Data

We will use **Yahoo Finance (yfinance)** to fetch historical stock price data.

2.1 Load Stock Data for a Specific Company

```
# Define the stock symbol and time period

stock_symbol = "AAPL"  # Apple Inc.

start_date = "2018-01-01"

end_date = "2023-12-31"


# Fetch stock data

df = yf.download(stock_symbol, start=start_date, end=end_date)


# Display first few rows

print(df.head())
```

2.2 Select the Closing Price

```
df = df[['Close']]  # We only need the closing price

df = df.asfreq('B')  # Convert to business days (removing non-trading days)

df = df.fillna(method='ffill')  # Fill missing values by forward filling
```

📊 Step 3: Visualize Stock Price Data

3.1 Plot the Stock Price Over Time

```
plt.figure(figsize=(12,6))

plt.plot(df['Close'], label=f"{stock_symbol} Stock Price", color='blue')

plt.xlabel("Date")

plt.ylabel("Stock Price (USD)")

plt.title(f"{stock_symbol} Stock Price Over Time")

plt.legend()

plt.show()
```

📌 Step 4: Check for Stationarity

A stationary time series has **constant mean and variance** over time, which is essential for ARIMA modeling.

4.1 Perform the Augmented Dickey-Fuller (ADF) Test

```
def adf_test(series):
    result = adfuller(series)
    print("ADF Statistic:", result[0])
    print("p-value:", result[1])
```

```
print("Critical Values:", result[4])
if result[1] < 0.05:
    print("✅ The data is stationary (p < 0.05).")
else:
    print("❌ The data is NOT stationary (p > 0.05). Differencing is required.")


# Run ADF test
adf_test(df['Close'])
```

🔄 Step 5: Make Data Stationary (If Needed)

If the ADF test shows **non-stationarity**, we apply **differencing**.

5.1 Apply First Order Differencing

```
df['Close_diff'] = df['Close'].diff().dropna()


# Re-run ADF test
adf_test(df['Close_diff'].dropna())


# Plot differenced data
plt.figure(figsize=(12,6))
plt.plot(df['Close_diff'], label="Differenced Stock Price", color='green')
plt.xlabel("Date")
```

```
plt.ylabel("Differenced Closing Price")

plt.title(f"{stock_symbol} Stock Price After Differencing")

plt.legend()

plt.show()
```

📌 Step 6: Determine ARIMA Model Parameters (p, d, q)

ARIMA has three components:

- **p (AutoRegressive Order)**: Number of lag observations
- **d (Differencing Order)**: Number of times differencing is applied
- **q (Moving Average Order)**: Number of lagged forecast errors

6.1 Plot AutoCorrelation Function (ACF) and Partial AutoCorrelation Function (PACF)

```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf


plt.figure(figsize=(12,6))

plt.subplot(121)

plot_acf(df['Close_diff'].dropna(), lags=20, ax=plt.gca())  # ACF plot

plt.subplot(122)

plot_pacf(df['Close_diff'].dropna(), lags=20, ax=plt.gca())  # PACF plot

plt.show()
```

6.2 Choose ARIMA Order Based on Plots

- **p (AR)**: Determined from the PACF plot (significant lags).
- **q (MA)**: Determined from the ACF plot.
- **d**: Already determined (first differencing = 1).

From the plots, let's assume:

p = 2 # Number of significant lags in PACF

d = 1 # Differencing order from ADF test

q = 2 # Number of significant lags in ACF

🤖 Step 7: Train the ARIMA Model

7.1 Fit ARIMA Model

model = ARIMA(df['Close'], order=(p, d, q))

model_fit = model.fit()

Display model summary

print(model_fit.summary())

📈 Step 8: Forecast Future Stock Prices

8.1 Make Predictions on Test Data

df['Forecast'] = model_fit.fittedvalues

plt.figure(figsize=(12,6))

plt.plot(df['Close'], label="Actual Stock Price", color='blue')

```
plt.plot(df['Forecast'], label="ARIMA Forecast", color='red')
plt.xlabel("Date")
plt.ylabel("Stock Price (USD)")
plt.title(f"ARIMA Model - {stock_symbol} Stock Price Prediction")
plt.legend()
plt.show()
```

8.2 Evaluate Model Performance

```
actual = df['Close'].dropna()
predicted = df['Forecast'].dropna()

mae = mean_absolute_error(actual, predicted)
mse = mean_squared_error(actual, predicted)
rmse = np.sqrt(mse)

print(f"Mean Absolute Error (MAE): {mae:.2f}")
print(f"Mean Squared Error (MSE): {mse:.2f}")
print(f"Root Mean Squared Error (RMSE): {rmse:.2f}")
```

📌 Step 9: Make Future Predictions

9.1 Forecast Stock Prices for the Next 30 Days

```
forecast_steps = 30  # Number of days to forecast

future_dates = pd.date_range(df.index[-1],
periods=forecast_steps+1, freq='B')[1:]
```

```
forecast = model_fit.forecast(steps=forecast_steps)

forecast_series = pd.Series(forecast, index=future_dates)


# Plot the forecast

plt.figure(figsize=(12,6))

plt.plot(df['Close'], label="Historical Stock Price", color='blue')

plt.plot(forecast_series, label="Future Prediction", color='orange', linestyle="dashed")

plt.xlabel("Date")

plt.ylabel("Stock Price (USD)")

plt.title(f"Future Stock Price Prediction for {stock_symbol}")

plt.legend()

plt.show()
```

✅ Final Summary

Step	Description
Step 1	Install and import necessary libraries
Step 2	Load stock price data using Yahoo Finance
Step 3	Visualize stock price trends
Step 4	Check stationarity using the **ADF test**
Step 5	Apply differencing if data is non-stationary

Step 6	Determine **ARIMA(p,d,q)** using ACF & PACF plots
Step 7	Train an **ARIMA model**
Step 8	Evaluate model performance & visualize predictions
Step 9	Forecast future stock prices

🌟 Conclusion

- ARIMA is a powerful tool for **stock price forecasting**.
- **Stationarity is crucial** for accurate predictions.
- **Hyperparameter tuning (p,d,q) is needed** to improve accuracy.
- The model can be extended to **real-time stock analysis**.

📌 ✨ ASSIGNMENT 2: 🎯 IMPLEMENT LSTM FOR FORECASTING ELECTRICITY CONSUMPTION TRENDS.

📌 ✨ Assignment Solution 2: Implement LSTM for Forecasting Electricity Consumption Trends

---

🎯 Objective

The goal of this assignment is to **implement a Long Short-Term Memory (LSTM) model** using **TensorFlow/Keras** to **forecast electricity consumption trends** based on historical data. LSTMs are powerful for time series forecasting as they can learn **long-term dependencies** in data.

🛠️ Step 1: Install and Import Necessary Libraries

We will use:

- **Pandas** for data handling.
- **NumPy** for numerical computations.
- **Matplotlib** for visualization.
- **Scikit-learn** for data preprocessing.
- **TensorFlow/Keras** for building the LSTM model.

🔹 Install Dependencies (if not installed)

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

🔹 Import Required Libraries

```
import pandas as pd

import numpy as np
```

```
import matplotlib.pyplot as plt

from sklearn.preprocessing import MinMaxScaler

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import LSTM, Dense, Dropout

from tensorflow.keras.optimizers import Adam
```

📥 Step 2: Load and Explore the Dataset

For this assignment, we assume the dataset contains electricity consumption data in **hourly or daily intervals**.

🔹 Load the Electricity Consumption Data

```
# Load dataset (Assume CSV file with 'Date' and 'Consumption' columns)

df = pd.read_csv("electricity_consumption.csv", parse_dates=['Date'], index_col='Date')


# Display first few rows

print(df.head())


# Check for missing values

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

🔹 Visualize the Data

```
plt.figure(figsize=(12,5))

plt.plot(df, label="Electricity Consumption")
```

```
plt.xlabel("Date")

plt.ylabel("Consumption (kWh)")

plt.title("Electricity Consumption Over Time")

plt.legend()

plt.show()
```

🧹 Step 3: Data Preprocessing

LSTM models require **normalized data** and **input sequences** for training.

🔹 Normalize the Data

```
scaler = MinMaxScaler(feature_range=(0,1))  # Scale between 0 and 1

df_scaled = scaler.fit_transform(df)


# Convert back to DataFrame

df_scaled = pd.DataFrame(df_scaled, index=df.index, columns=['Consumption'])
```

🔹 Create Sequences for LSTM

We need to prepare the data as **input-output sequences** where:

- **X (Input)** contains past n timestamps.
- **y (Output)** is the next consumption value.

```
def create_sequences(data, time_steps=24):

  X, y = [], []
```

```
    for i in range(len(data) - time_steps):
        X.append(data[i:i+time_steps])  # Last 'time_steps' values
        y.append(data[i+time_steps])    # Next time-step value
    return np.array(X), np.array(y)

# Define time steps (e.g., 24 hours for daily forecasting)
time_steps = 24

# Create sequences
X, y = create_sequences(df_scaled.values, time_steps)

# Display shape
print(f"Input Shape: {X.shape}, Output Shape: {y.shape}")
```

📊 Step 4: Train-Test Split

Split data into **training and testing sets** (80% train, 20% test).

```
train_size = int(len(X) * 0.8)

# Train and test sets
X_train, y_train = X[:train_size], y[:train_size]
X_test, y_test = X[train_size:], y[train_size:]
```

```
# Display shapes
print(f"Training Data: {X_train.shape}, Testing Data: {X_test.shape}")
```

📱 Step 5: Build the LSTM Model

Now, let's create a **stacked LSTM model** with:

- **3 LSTM layers** with units=50
- **Dropout layers** to prevent overfitting
- **Dense output layer** for predicting electricity consumption

```
# Define LSTM model
model = Sequential([
    LSTM(units=50, return_sequences=True, input_shape=(X_train.shape[1], 1)),
    Dropout(0.2),

    LSTM(units=50, return_sequences=True),
    Dropout(0.2),

    LSTM(units=50),
    Dropout(0.2),

    Dense(units=1)  # Output layer
])
```

```
# Compile the model
model.compile(optimizer=Adam(learning_rate=0.001), loss='mse')


# Display model summary
model.summary()
```

📌 Step 6: Train the Model

```
# Train the model
history = model.fit(X_train, y_train, epochs=20, batch_size=32, validation_data=(X_test, y_test))
```

📈 Step 7: Evaluate Model Performance

🔹 Plot Training Loss

```
plt.figure(figsize=(10,5))
plt.plot(history.history['loss'], label="Training Loss")
plt.plot(history.history['val_loss'], label="Validation Loss")
plt.title("LSTM Training Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.show()
```

🔹 Make Predictions

```
# Make predictions
y_pred = model.predict(X_test)

# Rescale predictions back to original scale
y_pred_rescaled = scaler.inverse_transform(y_pred)
y_test_rescaled = scaler.inverse_transform(y_test.reshape(-1,1))

# Plot actual vs. predicted
plt.figure(figsize=(12,5))
plt.plot(y_test_rescaled, label="Actual Consumption", linestyle="solid")
plt.plot(y_pred_rescaled, label="Predicted Consumption", linestyle="dashed")
plt.title("Electricity Consumption Forecasting with LSTM")
plt.xlabel("Time")
plt.ylabel("Consumption (kWh)")
plt.legend()
plt.show()
```

📌 Step 8: Forecast Future Electricity Consumption

To forecast **future consumption,** we use the last time_steps values.

```
# Get the last known values from the dataset
last_known_values = df_scaled.values[-time_steps:]
```

```
# Reshape input for LSTM
input_seq = np.array([last_known_values])
print(f"Input Sequence Shape: {input_seq.shape}")

# Predict next consumption value
future_prediction = model.predict(input_seq)

# Convert back to original scale
future_value = scaler.inverse_transform(future_prediction.reshape(-1,1))

print(f"Forecasted Electricity Consumption: {future_value[0][0]:.2f} kWh")
```

✅ Final Summary

Step	Description
Step 1	Install and import necessary libraries
Step 2	Load and explore the electricity consumption dataset
Step 3	Normalize the data and create LSTM sequences
Step 4	Split data into training and testing sets
Step 5	Build the LSTM model using TensorFlow/Keras

Step 6	Train the model on the dataset
Step 7	Evaluate model performance and visualize results
Step 8	Forecast future electricity consumption

🌟 Conclusion

- **LSTMs effectively capture temporal dependencies** in electricity consumption data.
- **The trained model can predict future electricity trends** using historical data.
- **The accuracy can be improved using hyperparameter tuning** and more data.
- **This model can be extended to real-world energy forecasting applications**.

📌 Next Steps

- **Tune LSTM hyperparameters** (e.g., units, epochs, batch_size).
- **Compare LSTM with other models** (ARIMA, SARIMA, XGBoost).
- **Deploy the model using Flask or FastAPI** for real-time predictions.
- **Use multivariate forecasting** (e.g., include temperature, weather conditions, and holidays).