ACC 424 Accounting Information System

Predictive analytics for financial forecasting and business intelligence
Python notes for 7.3-7.5

Section 001 MW 9:30 AM – 10:45 AM at Rm. 257 over Jan 13 – May 07
Section 002 MW 11:00 AM – 12:15 AM at Rm. 127 over Jan 13 – May 07

7.3. Time Series Forecasting with Synthetic Data

Step 1: Simulating Financial Time Series

We generate synthetic stock price data to model real-world financial behavior without relying on external datasets.

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

# Generate synthetic stock prices
np.random.seed(42)
dates = pd.date_range(start="2023-01-01", periods=365)
trend = 0.001 * np.arange(len(dates))
noise = np.random.normal(0, 0.5, len(dates))
prices = 100 + trend + noise.cumsum()

df = pd.DataFrame({"Price": prices}, index=dates)
df.plot(title="Simulated Stock Prices")
plt.show()
```

First, we import numpy, pandas, and matplotlib for numerical operations, data handling, and visualization. By setting np.random.seed(42), we ensure reproducibility, meaning the same **"random"** data is generated every time the code runs. We then create a 365-day timeline starting from January 1, 2023, using pd.date_range.

To simulate realistic **price** movements, we construct three components: a **linear trend** (0.001 * np.arange(len(dates)) for a steady 0.1% daily increase), **random noise** (np.random.normal(0, 0.5) to mimic market volatility), and a **cumulative sum of noise** (noise.cumsum()) to replicate the non-stationary "**random walk**" behavior typical of financial markets.

Combining these with a base price of 100, we store the results in a pandas DataFrame with dates as the index, then plot the synthetic prices to visualize the simulated trend and volatility. This approach provides a controlled, **self-contained** dataset for teaching core concepts like trends, noise, and **non-stationarity**—essential for understanding real-world financial time series.

Step 2: Feature Engineering

```
df["Returns"] = df["Price"].pct_change()
```

We compute the percentage change between each day's price and the previous day. Returns are fundamental in finance for analyzing performance and risk.

Formula: $\mathrm{Return}_t = \frac{\mathrm{Price}_t - \mathrm{Price}_{t-1}}{\mathrm{Price}_{t-1}}$

A new column Returns with values like 0.01 (1% gain) or -0.005 (0.5% loss). By calculating **daily returns**, we convert absolute prices into relative changes, which are more informative for analyzing performance and risk.

```
df["MA_10"] = df["Price"].rolling(10).mean()
```

We calculate the **10-day moving average** (MA) of prices. At each point, it takes the average of the last 10 days' prices. This calculation purpose is to **smooth out** short-term **noise** to reveal underlying trends. A new column MA_10 where each value is the average of the prior 10 days.

Day	Price	MA_10	
1	100	NaN	
...	...	...	
10	105	102.3	*(average of days 1-10)*

Note that the first 9 rows are NaN, i.e., missing (because not enough history to compute).

The **10-day moving average** (MA) smooths out short-term noise, creating a **trend-following** feature that helps distinguish **signal** from market **randomness**.

```
df["Volatility"] = df["Returns"].rolling(20).std() * np.sqrt(252)
```

- **rolling(20).std()**: Computes the **20-day rolling standard deviation** of returns (short-term volatility).
- *** np.sqrt(252)**: **Annalizes** the volatility (252 trading days/year).
- **Purpose:** Measures **risk**—higher volatility = riskier asset.

A new column Volatility showing annualized volatility (e.g., 0.25 = 25% annual volatility). The **20-day rolling volatility** (annualized) quantifies **risk** by measuring price **fluctuation** magnitude, a critical input for risk management and strategy optimization.

Feature engineering transforms raw price data into meaningful, structured inputs (**features**) that enhance a model's ability to detect financial patterns and relationships. The engineered features—returns, moving averages, and volatility—are **not present** in the raw data but are derived from it to expose **hidden** insights and improve **predictive** power. Essentially, feature engineering bridges raw data and actionable intelligence by creating variables that better represent the underlying economic realities models need to capture.

Step 3: ARIMA Forecasting

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

We import the ARIMA class from the statsmodels library, a tool for time series forecasting.

```python
model = ARIMA(df["Price"].dropna(), order=(2,1,1))
```

We initialize the ARIMA Model.

- **Input Data**: df[“Price”].dropna()
  - Use the synthetic price data from 7.3.1.
  - .dropna() ensures **no missing** values (critical for ARIMA).
- **Order Parameters**: order=(p, d, q)
  - **p=2 (Autoregressive term)**:
    - The model uses **2 prior time steps** (**lags**) to predict the next value.
    - Formula: $Price_t=\alpha+\beta_1 Price_{t-1}+\beta_2 Price_{t-2}+\epsilon t$
  - **d=1 (Differencing term)**:
    - Applies **first-order differencing** to make the data **stationary** (removes trends). ARIMA assumes the time series is **stationary**—meaning its statistical properties (mean, variance, autocorrelation) do not change over time.
    - Computes: $\Delta Price_t=Price_t-Price_{t-1}$
  - **q=1 (Moving Average term)**:
    - Accounts for **1 lagged forecast error** (unexpected shocks).
    - Formula: $\epsilon_t=\theta_1\epsilon_{t-1}+\omega_t$

We then fit the model.

```python
results = model.fit()
```

  - The model estimates optimal coefficients (β_1,β_2,θ_1) using **maximum likelihood estimation (MLE)**.

- Internally, it:
    1. Differences the data. The model first transforms your original price series into a stationary version by computing differences between consecutive points. **Differencing** ensures these patterns are measured on stable, **comparable** data.
    2. Fits AR and MA components to the differenced series. The AR terms (β) capture **momentum** effects ("prices keep rising"); the MA term (θ) corrects for sudden **shocks** ("yesterday's unexpected drop").

- **Output**:
    - A **fitted** model object (results) containing coefficients, residuals, and statistical diagnostics.

```
forecast = results.forecast(steps=5)
```

- **steps=5**: Predicts prices for the next **5 days** beyond the **training** data.
- **How It Works**:
    - Uses the estimated AR and MA terms to project future values iteratively.
    - For each step t+1to t+5:
        1. Predicts Price_{t+1} using lags and prior errors.
        2. Updates errors for the next prediction.

7.4. Business Intelligence Application - Customer Segmentation

Customer segmentation demonstrates how banks and financial institutions **segment** customers into groups based on their consumer behavior (e.g., shared behaviors, needs, and financial patterns) using **K-Means Clustering**. By applying **K-Means Clustering**, banks can uncover **hidden** structures in their customer base and tailor services accordingly. Segmentation helps predict **loan defaults** (e.g., high spenders with low income = risky).

```
from sklearn.cluster import KMeans
import numpy as np
import matplotlib.pyplot as plt

# Simulate client data: Account Balance & Monthly Transactions
np.random.seed(42)
data = np.column_stack([
    np.random.lognormal(mean=3, sigma=0.5, size=100),  # Account Balance ($)
    np.random.poisson(lam=10, size=100)                # Transactions/month
])
```

We begin by importing necessary libraries (KMeans for clustering, numpy for numerical operations, and matplotlib for visualization).

The data simulation creates two key customer attributes: **account balances** (simulated using a lognormal distribution with mean=3 and sigma=0.5 to reflect real-world skewness where most customers have modest balances, and a few have large ones) and **monthly transactions** (modeled using a Poisson distribution with lam = 10 to represent discrete, count-based events like transactions).

```
# Apply K-Means Clustering
kmeans = KMeans(n_clusters=3)
kmeans.fit(data)
labels = kmeans.labels_
```

K-Means clustering works by iteratively **grouping** similar data points together based on their **features**. The algorithm starts by randomly placing **three** center points (called centroids) in the data space representing our customer segments. Each customer's account balance and transaction history is then compared to these **centroids** using straight-line distance calculations. Customers get assigned to their nearest centroid, forming initial groups.

After this first grouping, the algorithm recalculates new centroid positions by finding the **average account balance** and **average transaction count** of all customers in each group. These updated centroids now better represent their clusters' true **centers**. The process repeats - customers get reassigned to the nearest updated centroids, then centroids get recalculated again. This cycle continues until the centroids stop moving significantly, meaning we've found stable, well-defined customer segments.

For our banking data, this results in three distinct customer groups where members share similar financial behaviors. The final centroids represent typical profiles for each segment, like a "medium-value" customer with a $5,000 average balance and 12 monthly transactions. The algorithm automatically discovers these **natural groupings** by mathematically minimizing the **variation** within each cluster while maximizing **differences** between clusters. This makes K-Means particularly useful for financial applications where clear customer segmentation drives **targeted services** and **risk assessment**.

7.5. Business Intelligence Application - Fraud Detection with Anomaly Simulation

```
normal_tx = np.random.normal(loc=50, scale=10, size=90)
fraud_tx = np.random.uniform(low=500, high=5000, size=10)
transactions = np.concatenate([normal_tx, fraud_tx]).reshape(-1, 1)
```

We implement anomaly detection using **Isolation Forest**, an algorithm for identifying fraudulent transactions. We begin by simulating transaction data - creating 90 normal transactions averaging 50 with typical variation, plus 10 abnormally high transactions *between* 500-$5000 to represent potential fraud cases. These are combined into a single dataset that the Isolation Forest model will process later.

This creates a realistic transaction dataset where:

- Normal transactions follow a Gaussian distribution centered at 50(±10 variation)
- Fraudulent transactions are uniformly distributed between 500−5000
- The combined dataset contains 100 total transactions (90 normal + 10 fraud)
- The reshape (-1, 1) converts the data into a 2D array format required by scikit-learn

Model Configuration:

```
clf = IsolationForest(contamination=0.1)
```

This initializes the Isolation Forest with:

- contamination=0.1 indicating we expect ~10% of transactions to be **fraudulent**
- **Default** parameters for number of trees (100) and samples per tree (256)
- Automatic anomaly threshold determination based on the **contamination parameter**

Training and Detection:

```
fraud_pred = clf.fit_predict(transactions)
```

The fit_predict() method performs two operations:

1. **Learns** the **normal transaction patterns** by building **isolation trees**
2. Scores each transaction, returning:
   - 1 for normal transactions
   - -1 for detected anomalies

The algorithm works by:

1. Randomly selecting a transaction amount feature
2. Randomly selecting a **split value** between min/max observed values
3. Building isolation trees where anomalies require **fewer** splits to isolate
4. Aggregating results across all trees to compute **anomaly** scores

The algorithm works by isolating observations through **random feature selection** and **splitting**. **Normal** transactions cluster together and require more partitions to isolate, while **anomalous** transactions stand out and can be separated with fewer splits. We configure the model expecting about 10% of transactions to be **fraudulent** (contamination=0.1). When applied to our simulated data, it successfully flags most of the high-value transactions as **suspicious**, demonstrating its effectiveness at spotting unusual patterns that could indicate **fraud**.