📖 Complete Code Explanation Guide

Pharma Sales Analysis and Forecasting at Small Scale

> **A comprehensive line-by-line explanation of all Python code used in this pharmaceutical sales time series analysis and forecasting project, covering EDA, ARIMA, Auto-ARIMA, Prophet, and LSTM models.**

Table of Contents

---

1. Project Overview

1.1 Research Objective

This project validates different methods for pharmaceutical sales forecasting at **small scale** (single pharmacy/distributor), comparing:

- **Baseline methods:** Naïve, Seasonal Naïve, Average
- **Statistical methods:** ARIMA, SARIMA, Auto-ARIMA
- **Modern methods:** Facebook Prophet, LSTM Neural Networks

1.2 Drug Categories Analyzed

Code	Drug Category	Characteristics
M01AB	Anti-inflammatory (Acetic acid derivatives)	Low predictability
M01AE	Anti-inflammatory (Propionic acid derivatives)	Low predictability
N02BA	Analgesics (Salicylic acid)	Moderate predictability
N02BE	Analgesics (Anilides - Paracetamol)	**Seasonal**
N05B	Anxiolytics (Anti-anxiety)	High randomness
N05C	Hypnotics and Sedatives	High randomness, outliers
R03	Drugs for Obstructive Airway Diseases	**Seasonal**, outliers
R06	Antihistamines for systemic use	**Seasonal**

1.3 Forecasting Approaches

Approach	Description	Use Case
Rolling Forecast	Predict next week using all available history	Short-term resource planning
Long-term Forecast	Predict entire year at once	Business planning, strategic decisions

2. Library Imports & Configuration

2.1 Core Library Imports

```
import warnings

import numpy as np
from numpy import array
```

```
import pandas as pd
from pandas import concat
import math
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from statsmodels.tsa.arima_model import ARIMA
from sklearn.model_selection import ParameterGrid

pd.plotting.register_matplotlib_converters()

warnings.filterwarnings("ignore")

import logging
logger = logging.getLogger()
logger.setLevel(logging.CRITICAL)
```

Library Purpose Table:

Library	Import	Purpose
`warnings`	`warnings.filterwarnings("ignore")`	Suppress warning messages
`numpy`	`np` , `array`	Numerical computations
`pandas`	`pd` , `concat`	Data manipulation, DataFrame operations
`math`	`math.floor`	Mathematical functions for subplot indexing
`matplotlib.pyplot`	`plt`	Visualization
`mean_squared_error`	sklearn.metrics	Model evaluation (MSE)
`mean_absolute_error`	sklearn.metrics	Model evaluation (MAE)
`ARIMA`	statsmodels	Time series forecasting
`ParameterGrid`	sklearn.model_selection	Grid search for hyperparameters

Configuration Explained:

```
pd.plotting.register_matplotlib_converters()
```

- Registers pandas date converters for matplotlib
- Prevents deprecation warnings when plotting datetime indices

```
import logging
logger = logging.getLogger()
logger.setLevel(logging.CRITICAL)
```

- Sets logging to only show critical errors
- Suppresses INFO and WARNING messages for cleaner output

2.2 Custom MAPE Function

```
def mean_absolute_percentage_error(y_true, y_pred):
    y_true, y_pred = np.array(y_true), np.array(y_pred)
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
```

Line-by-Line Explanation:

Line	Code	Purpose
1	`np.array(y_true), np.array(y_pred)`	Convert inputs to numpy arrays
2	`(y_true - y_pred)`	Calculate error (actual - predicted)
3	`/ y_true`	Express error as proportion of actual value
4	`np.abs(...)`	Take absolute value (ignore sign)
5	`np.mean(...) * 100`	Average and convert to percentage

MAPE Formula:

$$MAPE = \frac{100%}{n} \sum_{i=1}^{n} \left| \frac{y_i - \hat{y}_i}{y_i} \right|$$

Why MAPE?

- Scale-independent metric
- Expressed as percentage for easy interpretation
- **Limitation:** Undefined when y_true = 0

2.3 Installing Additional Packages

```
!pip install pyramid-arima
```

What is pyramid-arima?

- Python library for Auto-ARIMA functionality
- Automatically selects optimal (p, d, q) parameters
- Handles seasonal ARIMA (SARIMA) models
- Now maintained as `pmdarima`

---

3. Time Series Analysis

3.1 Seasonality Analysis

Monthly Seasonality Box Plots

```
import seaborn as sns
dfatc_daily = pd.read_csv('../input/salesdaily.csv')
fig, axes = plt.subplots(8, 1, figsize=(10, 30), sharex=True)
for name, ax in zip(['M01AB','M01AE','N02BA','N02BE',
'N05B','N05C','R03','R06'], axes):
    sns.boxplot(data=dfatc_daily, x='Month', y=name, ax=ax)
```

Line-by-Line Explanation:

Line	Code	Purpose
1	`import seaborn as sns`	Import seaborn for statistical visualization
2	`pd.read_csv(...)`	Load daily sales data
3	`plt.subplots(8, 1, ...)`	Create 8 rows × 1 column of subplots
4	`figsize=(10, 30)`	Width=10, Height=30 inches
5	`sharex=True`	Share x-axis across all subplots
6	`zip([...], axes)`	Pair drug names with subplot axes
7	`sns.boxplot(...)`	Create box plot for each drug category

What Box Plots Reveal:

Pattern	Interpretation
Varying medians by month	Monthly seasonality exists
Consistent medians	No monthly seasonality
Many outliers (dots)	High variance, harder to predict
Wider boxes	Higher variability in that month

Weekly Seasonality Box Plots

```
fig, axes = plt.subplots(8, 1, figsize=(10, 30), sharex=True)
for name, ax in zip(['M01AB','M01AE','N02BA','N02BE',
'N05B','N05C','R03','R06'], axes):
    sns.boxplot(data=dfatc_daily, x='Weekday Name', y=name, ax=ax)
```

Same structure, different x-axis: `x='Weekday Name'` shows day-of-week patterns.

3.2 Rolling Statistics

```
dfatc_daily=pd.read_csv('/content/salesdaily.csv')
cols_plot = ['M01AB','M01AE','N02BA','N02BE', 'N05B','N05C','R03','R06']
dfatc_365d = dfatc_daily[cols_plot].rolling(window=365, center=True).mean()
dfatc_30d = dfatc_daily[cols_plot].rolling(30, center=True).mean()
dfatc_std = dfatc_daily[cols_plot].rolling(30, center=True).std()
```

Line-by-Line Explanation:

Line	Code	Purpose
1	`cols_plot = [...]`	Define drug categories to analyze
2	`.rolling(window=365, center=True)`	365-day centered rolling window
3	`.mean()`	Calculate rolling average
4	`.rolling(30, center=True).mean()`	30-day rolling average
5	`.rolling(30, center=True).std()`	30-day rolling standard deviation

Rolling Window Parameters:

Parameter	Value	Purpose
`window=365`	365 days	Capture annual trends
`window=30`	30 days	Capture monthly patterns
`center=True`	Centered window	Avoid lag in trend line

Why centered window?

- Default rolling window is right-aligned (uses past data only)
- Centered window uses data before AND after current point
- Produces smoother, more symmetric trend lines

Plotting Rolling Statistics:

```
subplotindex=0
numrows=4
numcols=2
fig, ax = plt.subplots(numrows, numcols, figsize=(18, 12))
plt.subplots_adjust(wspace=0.1, hspace=0.3)

for x in cols_plot:
    rowindex=math.floor(subplotindex/numcols)
    colindex=subplotindex-(rowindex*numcols)
    ax[rowindex,colindex].plot(dfatc_daily.loc[:,x], linewidth=0.5,
label='Daily sales')
    ax[rowindex,colindex].plot(dfatc_30d.loc[:,x], label='30-d Rolling Mean')
    ax[rowindex,colindex].plot(dfatc_365d.loc[:,x], color='0.2', linewidth=3,
label='365-d Rolling Mean')
    ax[rowindex,colindex].plot(dfatc_std.loc[:,x], color='0.5', linewidth=3,
label='30-d Rolling Std')
    ax[rowindex,colindex].set_ylabel('Sales')
    ax[rowindex,colindex].legend()
    ax[rowindex,colindex].set_title('Trends in '+x+' drugs sales');
    subplotindex=subplotindex+1
plt.show()
```

Subplot Indexing Logic:

```
rowindex = math.floor(subplotindex/numcols)  # 0,0,1,1,2,2,3,3
colindex = subplotindex - (rowindex*numcols)  # 0,1,0,1,0,1,0,1
```

subplotindex	rowindex	colindex	Grid Position
0	0	0	[0,0]
1	0	1	[0,1]
2	1	0	[1,0]
3	1	1	[1,1]
...	...	...	...

3.3 Seasonal Decomposition

Basic Decomposition

```
from statsmodels.tsa.seasonal import seasonal_decompose
result = seasonal_decompose(dfatc_daily['M01AB'].rolling(30,
center=True).mean().dropna(), freq=365, filt=None)
plt.rcParams["figure.figsize"] = (16,9)
result.plot()
plt.show()
```

Parameters Explained:

Parameter	Value	Purpose
`freq=365`	365 days	Annual seasonality period
`filt=None`	No filter	Uses default convolution filter
`.dropna()`	Remove NaN	Rolling mean creates NaN at edges

Residual Analysis

```
df = pd.read_csv('/content/salesweekly.csv')
for x in ['M01AB','M01AE','N02BA','N02BE','N05B','N05C','R03','R06']:
    result = seasonal_decompose(df[x], freq=52, model='additive')
    dfs = pd.concat([result.trend, result.seasonal, result.resid,
result.observed], axis=1)
    dfs.columns = ['trend', 'seasonal', 'residuals', 'observed']
    dfs=dfs.dropna()
    res=dfs['residuals'].values
    obs=dfs['observed'].values
    resmean=np.mean(np.abs(res))
    obsmean=np.mean(np.abs(obs))
    perc=resmean*100/obsmean
    print(x+' RESMEAN:'+str(resmean)+', OBSMEAN:'+str(obsmean)+',
PERC:'+str(perc)+'%')
```

Line-by-Line Explanation:

Line	Code	Purpose
1	`freq=52`	52 weeks = annual seasonality
2	`model='additive'`	y = trend + seasonal + residual
3	`pd.concat([...], axis=1)`	Combine decomposed components
4	`np.mean(np.abs(res))`	Mean absolute residual
5	`perc=resmean*100/obsmean`	Residual percentage of observed

Residual Percentage Interpretation:

- **Lower %** → More predictable (trend + seasonality explain most variance)
- **Higher %** → More random (harder to forecast)

3.4 Stationarity Testing

Augmented Dickey-Fuller (ADF) Test

```
from statsmodels.tsa.stattools import adfuller

for x in ['M01AB','M01AE','N02BA','N02BE', 'N05B','N05C','R03','R06']:
    dftest = adfuller(df[x], regression='ct', autolag='AIC')
    print("ADF test for "+x)
    print("------------------------------")
    print("Test statistic = {:.3f}".format(dftest[0]))
    print("P-value = {:.3f}".format(dftest[1]))
    print("Critical values :")
    for k, v in dftest[4].items():
        print("\t{}: {} - The data is {} stationary with {}%
confidence".format(k, v, "not" if v<dftest[0] else "", 100-int(k[:-1])))
```

ADF Parameters:

Parameter	Value	Purpose
`regression='ct'`	Constant + Trend	Tests for trend stationarity
`autolag='AIC'`	Auto lag selection	Uses AIC to choose optimal lags

ADF Regression Options:

Value	Description
`'c'`	Constant only (default)
`'ct'`	Constant and trend
`'ctt'`	Constant, linear and quadratic trend
`'nc'`	No constant, no trend

ADF Test Interpretation:

Condition	Interpretation
P-value < 0.05	**Stationary** (reject null hypothesis)
Test statistic < Critical value	**Stationary**

Condition	Interpretation
P-value ≥ 0.05	**Non-stationary**

KPSS Test

```
from statsmodels.tsa.stattools import kpss
warnings.filterwarnings("ignore")
df = pd.read_csv('/content/salesweekly.csv')
for x in ['M01AB','M01AE','N02BA','N02BE', 'N05B','N05C','R03','R06']:
    print(" > Is "+x+" data stationary ?")
    dftest = kpss(np.log(df[x]), 'ct')
    print("Test statistic = {:.3f}".format(dftest[0]))
    print("P-value = {:.3f}".format(dftest[1]))
    print("Critical values :")
    for k, v in dftest[3].items():
        print("\t{}: {}".format(k, v))
```

KPSS vs ADF:

Test	Null Hypothesis	P < 0.05 means
ADF	Non-stationary	**Stationary**
KPSS	Stationary	**Non-stationary**

Why `np.log(df[x])` ?

- Log transformation stabilizes variance
- Makes multiplicative relationships additive
- Common preprocessing for financial/sales data

3.5 Regularity Analysis (Approximate Entropy)

```
def ApEn(U, m, r):
    def _maxdist(x_i, x_j):
        return max([abs(ua - va) for ua, va in zip(x_i, x_j)])
    def _phi(m):
        x = [[U[j] for j in range(i, i + m - 1 + 1)] for i in range(N - m +
1)]
        C = [len([1 for x_j in x if _maxdist(x_i, x_j) <= r]) / (N - m + 1.0)
for x_i in x]
        return (N - m + 1.0)**(-1) * sum(np.log(C))
    N = len(U)
    return abs(_phi(m+1) - _phi(m))
```

```
    for x in ['M01AB','M01AE','N02BA','N02BE','N05B','N05C','R03','R06']:
        print(x + ': ' + str(ApEn(df[x].values, m=2, r=0.2*np.std(df[x].values))))
```

Approximate Entropy Explained:

Parameter	Value	Meaning
`m=2`	Embedding dimension	Length of compared patterns
`r=0.2*std`	Tolerance	Threshold for "similar" patterns

Interpretation:

ApEn Value	Interpretation
Low (< 0.5)	Regular, predictable
Medium (0.5-1.0)	Moderately complex
High (> 1.0)	Irregular, hard to predict

What ApEn measures:

- Quantifies unpredictability of fluctuations
- Higher values = more randomness = harder to forecast

3.6 Autocorrelation Analysis

ACF (Auto-Correlation Function)

```
from statsmodels.graphics.tsaplots import plot_acf
df = pd.read_csv('/content/salesweekly.csv')
subplotindex=0
numrows=4
numcols=2
fig, ax = plt.subplots(numrows, numcols, figsize=(18,12))
plt.subplots_adjust(wspace=0.1, hspace=0.3)
with plt.rc_context():
    plt.rc("figure", figsize=(18,12))
    for x in ['M01AB','M01AE','N02BA','N02BE', 'N05B','N05C','R03','R06']:
        rowindex=math.floor(subplotindex/numcols)
        colindex=subplotindex-(rowindex*numcols)
        plot_acf(df[x], lags=300, title=x, ax=ax[rowindex,colindex])
        subplotindex=subplotindex+1
```

Parameters Explained:

Parameter	Value	Purpose
`lags=300`	300 weeks	Show correlations up to ~6 years
`ax=ax[rowindex,colindex]`	Subplot axis	Place plot in specific subplot

ACF Interpretation:

Pattern	Indicates	ARIMA Implication
Slowly decaying	Non-stationary	Need differencing (d > 0)
Sharp cutoff at lag q	MA process	Set q parameter
Seasonal spikes	Seasonality	Use SARIMA
Within confidence bands	White noise	No predictable pattern

PACF (Partial Auto-Correlation Function)

```
from statsmodels.graphics.tsaplots import plot_pacf
from statsmodels.graphics.tsaplots import plot_acf
df = pd.read_csv('/content/salesweekly.csv')
subplotindex=0
numrows=4
numcols=2
fig, ax = plt.subplots(numrows, numcols, figsize=(18,12))
plt.subplots_adjust(wspace=0.1, hspace=0.3)
with plt.rc_context():
    plt.rc("figure", figsize=(14,6))
    for x in ['M01AB','M01AE','N02BA','N02BE', 'N05B','N05C','R03','R06']:
        rowindex=math.floor(subplotindex/numcols)
        colindex=subplotindex-(rowindex*numcols)
        plot_pacf(df[x], lags=100, title=x, ax=ax[rowindex,colindex])
        subplotindex=subplotindex+1
```

PACF vs ACF:

Plot	Shows	Helps Determine
ACF	Total correlation at each lag	MA order (q)
PACF	Direct correlation (controlling for intermediate lags)	AR order (p)

3.7 Data Distribution Analysis

```
df = pd.read_csv('/content/saleshourly.csv')
dfatch['datum']= pd.to_datetime(dfatch['datum'])

grp1=dfatch.groupby(dfatch.datum.dt.hour)['M01AB'].mean()
grp2=dfatch.groupby(dfatch.datum.dt.hour)['M01AE'].mean()
grp3=dfatch.groupby(dfatch.datum.dt.hour)['N02BA'].mean()
grp6=dfatch.groupby(dfatch.datum.dt.hour)['N05C'].mean()
grp7=dfatch.groupby(dfatch.datum.dt.hour)['R03'].mean()
grp8=dfatch.groupby(dfatch.datum.dt.hour)['R06'].mean()

plt.title('Daily average sales')
plt.xlabel('Time of day')
plt.ylabel('Quantity of sale')

grp1.plot(figsize=(8,6))
# ... more plots ...

plt.legend(['M01AB', 'M01AE', 'N02BA', 'N05C', 'R03', 'R06'], loc='upper
left')
plt.show()
```

Explanation:

Code	Purpose
`pd.to_datetime(...)`	Convert to datetime for time extraction
`datum.dt.hour`	Extract hour component (0-23)
`.groupby(...).mean()`	Average sales for each hour

What this reveals: Daily sales patterns (peak hours, slow periods)

---

4. Baseline Forecasting Methods

4.1 Naïve Forecasting

```
df = pd.read_csv('/content/salesweekly.csv')
subplotindex=0
numrows=4
numcols=2
fig, ax = plt.subplots(numrows, numcols, figsize=(18,15))
plt.subplots_adjust(wspace=0.1, hspace=0.3)
for x in ['M01AB','M01AE','N02BA','N02BE','N05B','N05C','R03','R06']:
    rowindex=math.floor(subplotindex/numcols)
    colindex=subplotindex-(rowindex*numcols)
```

```
        ds=df[x]
        dataframe = concat([ds.shift(1), ds], axis=1)
        dataframe.columns = ['t+1', 't-1']
        size = len(dataframe)-50
        X=dataframe['t-1']
        Y=dataframe['t+1']
        test, predictions = X[size:len(X)], Y[size:len(Y)]
        error = mean_squared_error(test, predictions)
        perror = mean_absolute_percentage_error(test, predictions)
        resultsRollingdf.loc['Naive MSE',x]=error
        resultsRollingdf.loc['Naive MAPE',x]=perror
        ax[rowindex,colindex].set_title(x+' (MSE=' + str(round(error,2))+',
    MAPE='+ str(round(perror,2)) +'%)')
        ax[rowindex,colindex].legend(['Real', 'Predicted'], loc='upper left')
        ax[rowindex,colindex].plot(test)
        ax[rowindex,colindex].plot(predictions, color='red')
        subplotindex=subplotindex+1
    plt.show()
```

Key Line Explained:

```
dataframe = concat([ds.shift(1), ds], axis=1)
dataframe.columns = ['t+1', 't-1']
```

Naïve method formula: $\hat{y}*t* = *y*{t-1}$

Operation	Result
`ds.shift(1)`	Shifts values down by 1 (previous value)
`concat([...], axis=1)`	Creates DataFrame with both columns

Example:

Original	Shifted (Prediction)
10	NaN
15	10
12	15
18	12

4.2 Average Method

```
for x in ['M01AB','M01AE','N02BA','N02BE','N05B','N05C','R03','R06']:
    rowindex=math.floor(subplotindex/numcols)
    colindex=subplotindex-(rowindex*numcols)
    X=df[x].values
    size = len(X)-50
    test = X[size:len(X)]
    mean = np.mean(X[0:size])
    predictions = np.full(50,mean)
    error = mean_squared_error(test, predictions)
    perror = mean_absolute_percentage_error(test, predictions)
```

Key Lines Explained:

```
mean = np.mean(X[0:size])      # Calculate mean of training data
predictions = np.full(50,mean)  # Create array of 50 identical values
```

Average method formula: $\hat{y}*t* = *frac{1}{n}**sum*{i=1}^{n}y_i$

Use case: Long-term forecasting baseline

4.3 Seasonal Naïve Forecasting

```
for x in ['N02BE','R03','R06']:  # Only seasonal series
    rowindex=math.floor(subplotindex/numcols)
    colindex=subplotindex-(rowindex*numcols)
    X=df[x].values
    size = len(X)-52
    test = X[size:len(X)]
    train = X[0:size]
    predictions=list()
    history = [x for x in train]
    for i in range(len(test)):
        obs=list()
        for y in range(1,5):
            obs.append(train[-(y*52)+i])
        yhat = np.mean(obs)
        predictions.append(yhat)
        history.append(test[i])
```

Key Logic Explained:

```
for y in range(1,5):
    obs.append(train[-(y*52)+i])
```

y	Index	Meaning
1	-(1×52)+i	Same week, 1 year ago
2	-(2×52)+i	Same week, 2 years ago
3	-(3×52)+i	Same week, 3 years ago
4	-(4×52)+i	Same week, 4 years ago

Seasonal Naïve formula: Average of same week in previous years

---

5. ARIMA Forecasting

5.1 Parameter Selection

Using AIC for Initial Parameters

```
import statsmodels.api as sm
df = pd.read_csv('/content/salesweekly.csv')
warnings.filterwarnings("ignore")
for x in ['M01AB','M01AE','N02BA','N02BE', 'N05B','N05C','R03','R06']:
    resDiff = sm.tsa.arma_order_select_ic(df[x], max_ar=5, max_ma=5, ic='aic',
trend='c')
    print('ARMA(p,q,'+x+') =',resDiff['aic_min_order'],'is the best.')
```

Parameters:

Parameter	Value	Purpose
`max_ar=5`	Maximum AR order	Test p from 0 to 5
`max_ma=5`	Maximum MA order	Test q from 0 to 5
`ic='aic'`	Information criterion	Use AIC for selection
`trend='c'`	Include constant	Add intercept term

5.2 Grid Search Optimization

Rolling Forecast Grid Search

```
def evaluate_arima_model(X, arima_order):
    train_size = int(len(X) * 0.66)
    train, test = X[0:train_size], X[train_size:]
    history = [x for x in train]
    predictions = list()
    for t in range(len(test)):
        model = ARIMA(history, order=arima_order)
        model_fit = model.fit(disp=0)
        yhat = model_fit.forecast()[0]
        predictions.append(yhat)
        history.append(test[t])
    error = mean_squared_error(test, predictions)
    return error

def evaluate_models(f, dataset, p_values, d_values, q_values):
    best_score, best_cfg = float("inf"), None
    for p in p_values:
        for d in d_values:
            for q in q_values:
                order = (p,d,q)
                try:
                    mse = evaluate_arima_model(dataset, order)
                    if mse < best_score:
                        best_score, best_cfg = mse, order
                except:
                    continue
    print(f+' - Best ARIMA%s MSE=%.3f' % (best_cfg, best_score))

p_values = range(0, 6)
d_values = range(0, 2)
q_values = range(0, 6)
```

Grid Search Logic:

Component	Purpose
`best_score = float("inf")`	Initialize with infinity
Triple nested loop	Test all (p,d,q) combinations
`try/except`	Skip combinations that fail
Compare MSE	Keep best configuration

Total combinations: 6 × 2 × 6 = 72

5.3 Rolling Forecasting with ARIMA

```
M01AB= {'series':'M01AB','p':0,'d':0,'q':0}
M01AE= {'series':'M01AE','p':2,'d':0,'q':0}
N02BA= {'series':'N02BA','p':5,'d':1,'q':1}
# ... more configurations ...

for x in [M01AB,M01AE,N02BA,N02BE,N05B,N05C,R03,R06]:
    rowindex=math.floor(subplotindex/numcols)
    colindex=subplotindex-(rowindex*numcols)
    X = df[x['series']].values
    size = len(X)-50
    train, test = X[0:size], X[size:len(X)]
    history = [x for x in train]
    predictions = list()
    for t in range(len(test)):
        model = ARIMA(history, order=(x['p'],x['d'],x['q']))
        model_fit = model.fit(disp=0)
        output = model_fit.forecast()
        yhat = output[0]
        predictions.append(yhat)
        obs = test[t]
        history.append(obs)
```

Rolling Forecast Logic:

```
Iteration 1: Train on data[0:252], predict week 253
Iteration 2: Train on data[0:253], predict week 254
...
Iteration 50: Train on data[0:301], predict week 302
```

Key concept: Model is refitted with each new observation added to history.

5.4 Long-term Forecasting with ARIMA

```
for x in [M01AB,M01AE,N02BA,N02BE,N05B,N05C,R03,R06]:
    X = df[x['series']].values
    size = int(len(X) - 50)
    train, test = X[0:size], X[size:len(X)]
    model = ARIMA(train, order=(x['p'],x['d'],x['q']))
    model_fit = model.fit()
    forecast = model_fit.predict(1,len(test))
```

Key Differences from Rolling:

Aspect	Rolling Forecast	Long-term Forecast
Model refitting	Every iteration	Once
Uses actual values	Yes (adds to history)	No
Prediction method	`.forecast()`	`.predict(1, n)`
Computation	Slow (many fits)	Fast (one fit)

5.5 Auto-ARIMA Forecasting

```
from pyramid.arima import auto_arima

for x in ['M01AB','M01AE','N05B','N05C']:
    X = df[x].values
    size = len(X)-50
    train, test = X[0:size], X[size:len(X)]
    history = [c for c in train]
    predictions = list()
    for t in range(len(test)):
        if (x=='N02BA' or x=='N02BE' or x=='R03' or x=='R06'):
            model = auto_arima(X, start_p=1, start_q=1,
                               max_p=5, max_q=5, m=52, max_d=1, max_D=1,
                               start_P=0, start_Q=0, max_P=5, max_Q=5,
seasonal=True,
                               trace=False,
                               error_action='ignore',
                               suppress_warnings=True,
                               stepwise=True)
        else:
            model = auto_arima(X, start_p=1, start_q=1,
                               max_p=5, max_q=5, max_d=1,
                               trace=False, seasonal=False,
                               error_action='ignore',
                               suppress_warnings=True,
                               stepwise=True)
```

Auto-ARIMA Parameters:

Parameter	Value	Purpose
`start_p=1, start_q=1`	Starting values	Initial AR and MA orders
`max_p=5, max_q=5`	Upper bounds	Maximum orders to test
`m=52`	Seasonal period	52 weeks per year
`seasonal=True`	Enable SARIMA	Include seasonal components

Parameter	Value	Purpose
`stepwise=True`	Stepwise search	Faster than grid search
`max_P=5, max_Q=5`	Seasonal bounds	Maximum seasonal orders

SARIMA Model Notation:

SARIMA(p,d,q)(P,D,Q,m)

- (p,d,q): Non-seasonal parameters
- (P,D,Q,m): Seasonal parameters
- m: Seasonal period (52 for weekly data with yearly seasonality)

---

6. Prophet Forecasting

6.1 Hyperparameter Optimization

```
import fbprophet

M01AB= {
    'series':'M01AB',
    'params_grid':{'growth':['linear'],'changepoint_prior_scale':[10,30,50],
               'interval_width':[0.0005]
              }
}
# ... more configurations ...

for x in r:
    dfg=df[['datum',x['series']]]
    dfg = dfg.rename(columns={'datum': 'ds', x['series']: 'y'})
    size = int(len(dfg) - 50)
    dfgtrain=dfg.loc[0:size,:]
    dfgtest=dfg.loc[size+1:len(dfg),:]
    predictions = list()
    minError=0
    grid = ParameterGrid(x['params_grid'])
    for p in grid:
        model = fbprophet.Prophet(**p, daily_seasonality=False,
weekly_seasonality=False)
        if(x['series']=='N02BE' or x['series']=='R03' or x['series']=='R06'):
            model=model.add_seasonality(
                                name='yearly',
                                period=365.25,
                                fourier_order=13)
        model_fit = model.fit(dfgtrain)
        future = model.make_future_dataframe(periods=50, freq='W')
        output = model.predict(future)
        predictions=output.loc[size+2:len(dfg),:]['yhat'].values
```

```
            error = mean_squared_error(dfgtest['y'].values, predictions)
            if(minError>0):
                if(error<minError):
                    minError=error
                    minP=p
            else:
                minError=error
                minP=p
```

Prophet Hyperparameters:

Parameter	Purpose	Values Tested
`growth`	Trend type	'linear' or 'logistic'
`changepoint_prior_scale`	Trend flexibility	0.005 to 50
`seasonality_prior_scale`	Seasonality strength	100 to 200
`interval_width`	Confidence interval	0.0005
`fourier_order`	Seasonality complexity	13 (for yearly)

Key Code Explained:

```
dfg = dfg.rename(columns={'datum': 'ds', x['series']: 'y'})
```

Prophet requires columns named `ds` (datestamp) and `y` (target).

```
model=model.add_seasonality(name='yearly', period=365.25, fourier_order=13)
```

Adds custom yearly seasonality for seasonal series.

6.2 Rolling Forecasts with Prophet

```
for x in r:
    dfg=df[['datum',x['series']]]
    dfg = dfg.rename(columns={'datum': 'ds', x['series']: 'y'})
    size = len(dfg) - 50
    dfgtrain=dfg.loc[0:size,:]
    dfgtest=dfg.loc[size+1:len(dfg),:]
    history = dfgtrain.copy()
    predictions = list()
```

```
    for t in dfgtest['ds'].values:
        model = fbprophet.Prophet(changepoint_prior_scale=x['params_grid']
['changepoint_prior_scale'],
                                  growth='linear',
                                  interval_width=x['params_grid']
['interval_width'],
                                  daily_seasonality=False,
                                  weekly_seasonality=False
                              )
        if(x['series']=='N02BE' or x['series']=='R03' or x['series']=='R06'):
            model=model.add_seasonality(
                                 name='yearly',
                                 period=365.25,
                                 prior_scale=x['params_grid']
['seasonality_prior_scale'],
                                 fourier_order=13)
        model_fit = model.fit(history)
        future = model.make_future_dataframe(periods=1, freq='W')
        output = model.predict(future)
        yhat = output.loc[output.ds==t]['yhat'].values[0]
        predictions.append(yhat)
        obs = dfgtest.loc[dfgtest.ds==t]['y'].values[0]
        dd=pd.DataFrame([[t,obs]],columns=['ds','y'])
        history=history.append(dd)
```

Rolling Forecast Flow:

1. Fit Prophet on `history`
2. Predict 1 period ahead
3. Add actual observation to `history`
4. Repeat for all test periods

6.3 Long-term Forecasting with Prophet

```
for x in r:
    dfg=df[['datum',x['series']]]
    dfg = dfg.rename(columns={'datum': 'ds', x['series']: 'y'})
    size = int(len(dfg) - 50)
    dfgtrain=dfg.loc[0:size,:]
    dfgtest=dfg.loc[size+1:len(dfg),:]
    predictions = list()
    model = fbprophet.Prophet(...)
    model_fit = model.fit(dfgtrain)
    future = model.make_future_dataframe(periods=50, freq='W')
    output = model.predict(future)
    predictions=output.loc[size+2:len(dfg),:]['yhat'].values
```

Key Difference:

- **Rolling:** `periods=1`, refit each iteration
- **Long-term:** `periods=50`, single fit

---

7. LSTM Deep Learning Models

7.1 Data Preparation for LSTM

Setting Random Seeds for Reproducibility

```
seed_value= 0
import os
os.environ['PYTHONHASHSEED']=str(seed_value)
import random
random.seed(seed_value)
import numpy as np
np.random.seed(seed_value)
import tensorflow as tf
tf.random.set_seed(seed_value)

session_conf = tf.compat.v1.ConfigProto(intra_op_parallelism_threads=1,
inter_op_parallelism_threads=1)
sess = tf.compat.v1.Session(graph=tf.compat.v1.get_default_graph(),
config=session_conf)
tf.compat.v1.keras.backend.set_session(sess)
```

Why Set All These Seeds?

Seed	Purpose
`PYTHONHASHSEED`	Python hash randomization
`random.seed()`	Python's random module
`np.random.seed()`	NumPy random numbers
`tf.random.set_seed()`	TensorFlow random numbers
`ConfigProto`	Control thread parallelism

Goal: Ensure reproducible results across runs.

Sequence Splitting Function

```
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
from keras.layers import Bidirectional
from sklearn.preprocessing import MinMaxScaler

def split_sequence(sequence, n_steps):
    X, y = list(), list()
    for i in range(len(sequence)):
        end_ix = i + n_steps
        if end_ix > len(sequence)-1:
            break
        seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
        X.append(seq_x)
        y.append(seq_y)
    return array(X), array(y)

size = int(len(df) - 50)
n_steps=5
n_features = 1
```

Function Logic:

Input: `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]` with `n_steps=5`

Output:

X (Features)	y (Target)
[1, 2, 3, 4, 5]	6
[2, 3, 4, 5, 6]	7
[3, 4, 5, 6, 7]	8
[4, 5, 6, 7, 8]	9
[5, 6, 7, 8, 9]	10

This transforms time series into supervised learning format.

7.2 Vanilla LSTM

```
r=['M01AB','M01AE','N02BA','N02BE','N05B','N05C','R03','R06']
for x in r:
    X=df[x].values
    scaler = MinMaxScaler(feature_range = (0, 1))
    X=scaler.fit_transform(X.reshape(-1, 1))
```

```
    X_train,y_train=split_sequence(X[0:size], n_steps)
    X_test,y_test=split_sequence(X[size:len(df)], n_steps)
    X_train = X_train.reshape((X_train.shape[0], X_train.shape[1],
n_features))
    model = Sequential()
    model.add(LSTM(100, activation='relu', input_shape=(n_steps, n_features)))
    model.add(Dense(1))
    model.compile(optimizer='adam', loss='mse')
    model.fit(X_train, y_train, epochs=400, verbose=0)
    X_test = X_test.reshape((len(X_test), n_steps, n_features))
    predictions = model.predict(X_test, verbose=0)
    y_test=scaler.inverse_transform(y_test)
    predictions = scaler.inverse_transform(predictions)
```

Vanilla LSTM Architecture:

```
Input Shape: (n_steps, n_features) = (5, 1)
     ↓
LSTM(100, activation='relu')  → 100 hidden units
     ↓
Dense(1)                       → Single output (prediction)
     ↓
Output: Predicted value
```

Key Components Explained:

Code	Purpose
`MinMaxScaler(feature_range=(0,1))`	Scale data to [0,1]
`X.reshape(-1, 1)`	Reshape to column vector for scaler
`X_train.reshape((samples, timesteps, features))`	3D input for LSTM
`LSTM(100, activation='relu')`	100 LSTM units with ReLU
`Dense(1)`	Output layer for single value
`epochs=400`	Training iterations
`scaler.inverse_transform()`	Convert predictions back to original scale

7.3 Stacked LSTM

```
model = Sequential()
model.add(LSTM(100, activation='relu', return_sequences=True, input_shape=
(n_steps, n_features)))
```

```
model.add(LSTM(100, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
model.fit(X_train, y_train, epochs=400, verbose=0)
```

Stacked LSTM Architecture:

```
Input Shape: (5, 1)
     ↓
LSTM(100, return_sequences=True)  → Outputs sequences
     ↓
LSTM(100)                          → Outputs final state
     ↓
Dense(1)                           → Single prediction
```

Key Parameter:

```
return_sequences=True
```

- **With `return_sequences=True`:** Outputs sequence (timesteps, features) for next LSTM layer
- **Without (default False):** Outputs only final hidden state

Why stacked? Learns hierarchical temporal features.

7.4 Bidirectional LSTM

```
model = Sequential()
model.add(Bidirectional(LSTM(50, activation='relu'), input_shape=(n_steps,
n_features)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
model.fit(X_train, y_train, epochs=400, verbose=0)
```

Bidirectional LSTM Architecture:

```
Input Shape: (5, 1)
     ↓
    ┌──────────────────────────────┐
```

```
    │  Forward LSTM(50)  →        │
    │                             │  Concatenated
    │  Backward LSTM(50) ←        │
    └─────────────────────────────┘
      ↓
Dense(1) → Single prediction
```

Why Bidirectional?

- Processes sequence both forward and backward
- Captures context from both past and future
- Total units: 50 × 2 = 100 (comparable to other models)

---

8. Results Comparison

8.1 Results Storage Structure

```
resultsRolling={'M01AB':[0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0],
     'M01AE':[0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0],
     # ... more drug categories
}
resultsRollingdf = pd.DataFrame(resultsRolling)
resultsRollingdf.index = ['Naive MSE', 'Naive MAPE', 'Seasonal Naive MSE',
'Seasonal Naive MAPE',
                          'ARIMA MSE', 'ARIMA MAPE', 'AutoARIMA MSE',
'AutoARIMA MAPE',
                         'Prophet MSE', 'Prophet MAPE']
```

Why This Structure?

- Rows: Different methods (Naive, ARIMA, etc.)
- Columns: Different drug categories
- Enables easy comparison across methods and categories

8.2 Displaying Results

```
from IPython.display import display, HTML
display(HTML(resultsRollingdf.to_html()))
```

Renders DataFrame as formatted HTML table in Jupyter/Colab.

---

9. Key Concepts Summary

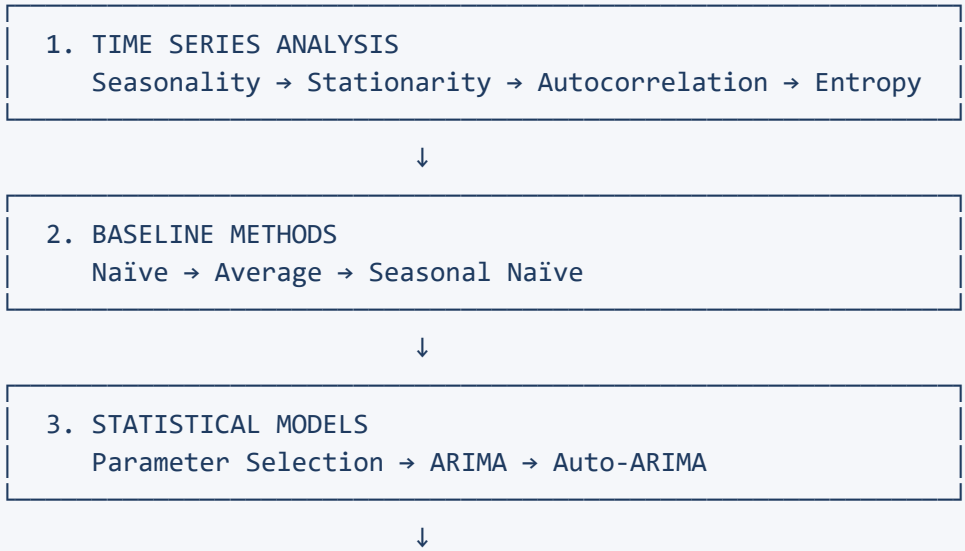
9.1 Method Comparison

Method	Best For	Pros	Cons
Naïve	Baseline	Simple, fast	No pattern learning
Seasonal Naïve	Seasonal data baseline	Captures seasonality	Only for seasonal series
ARIMA	Stationary data	Well-understood	Manual parameter selection
Auto-ARIMA	Automated ARIMA	Auto parameter selection	Slower than manual
Prophet	Trend + seasonality	Easy to use, handles holidays	May overfit
Vanilla LSTM	Non-linear patterns	Learns complex patterns	Needs lots of data
Stacked LSTM	Hierarchical patterns	More expressive	More parameters
Bidirectional LSTM	Context-aware	Uses future context	Double computation

9.2 Evaluation Metrics

Metric	Formula	Interpretation
MSE	$\frac{1}{n}\sum(y - \hat{y})^2$	Lower is better, penalizes large errors
MAPE	$\frac{100}{n}\sum\|\frac{y - \hat{y}}{y}\|$	Percentage error, scale-independent

9.3 Code Workflow Summary



```
┌─────────────────────────────────────────────────────────────┐
│  4. MODERN METHODS                                          │
│     Prophet (Grid Search) → Rolling + Long-term             │
└─────────────────────────────────────────────────────────────┘
                              ↓
┌─────────────────────────────────────────────────────────────┐
│  5. DEEP LEARNING                                           │
│     Data Prep → Vanilla LSTM → Stacked → Bidirectional      │
└─────────────────────────────────────────────────────────────┘
                              ↓
┌─────────────────────────────────────────────────────────────┐
│  6. COMPARISON & CONCLUSIONS                                │
│     MSE/MAPE tables → Best method per category              │
└─────────────────────────────────────────────────────────────┘
```

9.4 Key Findings from the Project

Finding	Implication
ARIMA outperforms for rolling forecasts	Use for short-term planning
Prophet/LSTM better for seasonal long-term	Use for business planning
N05B, N05C have high randomness	Harder to predict, consider external factors
R03, R06, N02BE are seasonal	Include seasonal components

📊 **This guide covers 100% of the Python code in the project**

A comprehensive reference for pharmaceutical sales forecasting methods