📖 Complete Code Explanation Guide

Sales Prediction for Pharmaceutical Distribution Companies

> **A line-by-line explanation of all Python code used in this time-series forecasting project, covering EDA, SARIMA, Prophet, and SVR models.**

Table of Contents

1. EDA Notebook Explanation

1.1 Import Dependencies

```
import pandas as pd
import seaborn as sns
import math
import matplotlib.pyplot as plt
from statsmodels.graphics.tsaplots import plot_acf , plot_pacf
from pandas_profiling import ProfileReport
```

Explanation:

Library	Purpose
`pandas`	Data manipulation and analysis - handles DataFrames
`seaborn`	Statistical data visualization - creates beautiful plots
`math`	Mathematical functions - used for floor division in subplot indexing
`matplotlib.pyplot`	Core plotting library - creates figures and axes

Library	Purpose
`plot_acf, plot_pacf`	Autocorrelation plotting functions from statsmodels
`ProfileReport`	Automated EDA report generation from pandas-profiling

Why these libraries?

- `pandas` is essential for handling tabular time-series data
- `seaborn` provides box plots for seasonality analysis
- `statsmodels` has specialized time-series functions not available in basic libraries

---

1.2 Load and Preprocess Data

```
import pandas as pd
# Import the data
df_daily = pd.read_csv("C:\\Users\\User\\Desktop\\Ryerson\\CIND
820\\Dataset\\salesdaily.csv")
df_daily.drop(['Year','Month','Hour','Weekday Name'], axis=1, inplace=True)
df_daily['datum'] = pd.to_datetime(df_daily['datum'])
df_daily.rename(columns = {'datum':'Date'}, inplace = True)
# Set the date as index
df_daily = df_daily.set_index('Date')
df_daily.head()
```

Line-by-Line Explanation:

Line	Code	Purpose
1	`pd.read_csv(...)`	Loads the CSV file into a DataFrame
2	`.drop([...], axis=1, inplace=True)`	Removes unnecessary columns (metadata) permanently
3	`pd.to_datetime(...)`	Converts string dates to datetime objects for time-series operations
4	`.rename(columns={...})`	Renames 'datum' to 'Date' for clarity
5	`.set_index('Date')`	Sets Date as the index - **critical for time-series**
6	`.head()`	Displays first 5 rows to verify

Why drop these columns?

- `Year`, `Month`, `Hour`, `Weekday Name` are derived from the date
- Keeping them would create redundancy
- The date index provides all temporal information needed

1.3 Pandas Profiling Report

```
profile = ProfileReport(df_daily, title='Pandas Profiling Report',
explorative=True)
profile.to_widgets()
```

Explanation:

Line	Purpose
`ProfileReport(df_daily, ...)`	Creates comprehensive automated EDA report
`explorative=True`	Enables all analysis features (correlations, missing values, distributions)
`.to_widgets()`	Renders interactive report in Jupyter notebook

What the profile report shows:

- Data types and missing values
- Descriptive statistics (mean, std, min, max)
- Distribution histograms
- Correlation matrix
- Duplicate detection
- Warnings about data quality issues

1.4 Daily and Monthly Plotting

```
%matplotlib inline
df_daily.plot(subplots=True, figsize=(12,15))
```

Explanation:

Code	Purpose
`%matplotlib inline`	Jupyter magic command - renders plots inside notebook
`.plot(subplots=True, ...)`	Creates separate subplot for each column (8 drug categories)
`figsize=(12,15)`	Width=12 inches, Height=15 inches - accommodates 8 subplots

Why visualize raw data first?

- Identify obvious trends, seasonality, outliers

- Understand data scale differences between categories
- Spot missing data periods or anomalies

1.5 Seasonality Box Plots

```
# Prepare data
df_monthly['year'] = [d.year for d in df_monthly.Date]
df_monthly['month'] = [d.strftime('%b') for d in df_monthly.Date]
years = df_monthly['year'].unique()
```

Line-by-Line Explanation:

Line	Code	Purpose
1	`[d.year for d in df_monthly.Date]`	List comprehension extracts year from each date
2	`[d.strftime('%b') for d in df_monthly.Date]`	Extracts abbreviated month name (Jan, Feb, etc.)
3	`.unique()`	Gets unique years for reference

```
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(15, 6))

sns.boxplot(df_monthly['year'], df_monthly["M01AB"], ax=ax[0])
ax[0].set_title('Year-wise Box Plot for M01AB', fontsize=20, loc='center')
ax[0].set_xlabel('Year')
ax[0].set_ylabel('M01AB sales')

sns.boxplot(df_monthly['month'], df_monthly["M01AB"], ax=ax[1])
ax[1].set_title('Month-wise Box Plot for M01AB', fontsize=20, loc='center')
ax[1].set_xlabel('Month')
ax[1].set_ylabel('M01AB sales')

fig.autofmt_xdate()
```

Explanation:

Code	Purpose
`plt.subplots(nrows=1, ncols=2, ...)`	Creates 1 row × 2 columns of subplots
`ax=ax[0]`	Assigns first subplot to year-wise box plot
`ax=ax[1]`	Assigns second subplot to month-wise box plot

Code	Purpose
`sns.boxplot(x, y, ...)`	Creates box plot showing distribution by category
`fig.autofmt_xdate()`	Auto-rotates x-axis labels to prevent overlap

What box plots reveal:

- **Year-wise:** Trend over years (increasing/decreasing sales)
- **Month-wise:** Seasonal patterns (which months have higher/lower sales)
- **Outliers:** Dots outside whiskers indicate unusual values

---

1.6 Autocorrelation Analysis

```
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_monthly[x], lags=30, title=x, ax=ax[rowindex,colindex])
        subplotindex = subplotindex + 1
```

Line-by-Line Explanation:

Line	Code	Purpose
1-3	`subplotindex`, `numrows`, `numcols`	Initialize grid parameters (4×2 = 8 plots)
4	`plt.subplots(...)`	Creates 4×2 grid of axes
5	`plt.subplots_adjust(...)`	Adjusts spacing between subplots
6	`with plt.rc_context():`	Temporarily sets matplotlib parameters
7	`for x in [...]`	Iterates through all 8 drug categories
8	`math.floor(subplotindex/numcols)`	Calculates row index (0,0,1,1,2,2,3,3)
9	`subplotindex - (rowindex*numcols)`	Calculates column index (0,1,0,1,0,1,0,1)
10	`plot_acf(...)`	Plots autocorrelation function for each category

What ACF shows:

- Correlation between observations at different lags
- **Slowly decaying ACF** → Non-stationary data
- **Sharp cutoff** → Suggests MA order (q) for ARIMA
- **Seasonal spikes** (at lag 12, 24, etc.) → Yearly seasonality

---

2. Time Series Models Notebook Explanation

2.1 Data Loading & Preprocessing

```
import pandas as pd
import numpy as np
import warnings
import itertools
import matplotlib
import matplotlib.pyplot as plt
from pylab import rcParams
import statsmodels.api as sm
from statsmodels.tsa.stattools import adfuller
from sklearn.metrics import mean_squared_error, mean_absolute_error
from statsmodels.tsa.stattools import acf, pacf
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

warnings.filterwarnings("ignore")

plt.style.use('seaborn-bright')
matplotlib.rcParams['axes.labelsize'] = 14
matplotlib.rcParams['xtick.labelsize'] = 12
matplotlib.rcParams['ytick.labelsize'] = 12
matplotlib.rcParams['text.color'] = 'k'
```

Library Purpose Table:

Library	Purpose
`numpy`	Numerical computations, array operations
`warnings`	Suppress warning messages for cleaner output
`itertools`	Generate parameter combinations for grid search
`statsmodels.api`	Statistical models including SARIMAX
`adfuller`	Augmented Dickey-Fuller stationarity test
`mean_squared_error`	Model evaluation metric
`acf, pacf`	Autocorrelation calculation functions

Why suppress warnings?

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

- SARIMA fitting often produces convergence warnings
- Cleaner notebook presentation
- **Note:** In production, you should handle warnings properly

---

2.2 Data Resampling

```
df_daily = pd.read_csv("...")
df_daily.drop(['Year','Month','Hour','Weekday Name'], axis=1, inplace=True)
df_daily['datum'] = pd.to_datetime(df_daily['datum'])
df_daily.rename(columns = {'datum':'Date'}, inplace = True)
df_daily = df_daily.set_index('Date')

# Select the proper time period for weekly aggregation
df = df_daily['2014-01-02':'2019-10-8'].resample('W').sum()
```

Key Operation Explained:

```
df = df_daily['2014-01-02':'2019-10-8'].resample('W').sum()
```

Component	Purpose
`['2014-01-02':'2019-10-8']`	Slices data to specific date range
`.resample('W')`	Groups data by week ('W' = weekly frequency)
`.sum()`	Aggregates daily values to weekly totals

Why resample to weekly?

- Daily data is too noisy for forecasting
- Weekly smooths out day-of-week effects
- Reduces computation time
- Better captures seasonal patterns

Resampling frequency codes:

Code	Frequency

Code	Frequency
'D'	Daily
'W'	Weekly
'M'	Monthly
'Q'	Quarterly
'Y'	Yearly

2.3 Visualization

```
y = df['M01AB']
fig, ax = plt.subplots(figsize=(20, 6))
ax.plot(y, marker='.', linestyle='-', linewidth=0.5, label='Weekly')
ax.plot(y.resample('M').mean(), marker='o', markersize=8, linestyle='-',
label='Monthly Mean Resample')
ax.set_ylabel('Sales of M01AB')
ax.legend();
```

Explanation:

Code	Purpose
`y = df['M01AB']`	Extracts single drug category as Series
`marker='.'`	Small dots at each data point
`linestyle='-'`	Solid line connecting points
`linewidth=0.5`	Thin line for weekly data
`y.resample('M').mean()`	Monthly moving average overlay
`marker='o', markersize=8`	Larger circles for monthly points

Why overlay monthly mean?

- Shows underlying trend without weekly noise
- Easier to identify long-term patterns
- Helps visualize seasonality

2.4 Decomposition

```
def seasonal_decompose(y):
    decomposition = sm.tsa.seasonal_decompose(y, model='additive',
```

```
    extrapolate_trend='freq')
        fig = decomposition.plot()
        fig.set_size_inches(14, 7)
        plt.show()

    seasonal_decompose(y)
```

Explanation:

Parameter	Value	Purpose
`y`	Time series	Input data to decompose
`model='additive'`	Type of decomposition	Assumes: Observed = Trend + Seasonal + Residual
`extrapolate_trend='freq'`	Handling edge cases	Uses frequency-based extrapolation for trend endpoints

Additive vs Multiplicative:

```
Additive:       y(t) = Trend + Seasonality + Residual
Multiplicative: y(t) = Trend × Seasonality × Residual
```

Use additive when:

- Seasonal variation is constant over time
- Suitable for most pharmaceutical sales data

Decomposition output shows:

1. **Observed:** Original time series
2. **Trend:** Long-term direction (increasing/decreasing)
3. **Seasonal:** Repeating patterns (yearly cycle)
4. **Residual:** Random noise (what's left after removing trend and seasonality)

2.5 Stationarity Testing

Rolling Statistics Function

```
    def test_stationarity(timeseries, title):
        # Determining rolling statistics
        rolmean = pd.Series(timeseries).rolling(window=12).mean()
        rolstd = pd.Series(timeseries).rolling(window=12).std()
```

```
    fig, ax = plt.subplots(figsize=(16, 4))
    ax.plot(timeseries, label=title)
    ax.plot(rolmean, label='rolling mean')
    ax.plot(rolstd, label='rolling std (x10)')
    ax.legend()
```

Explanation:

Code	Purpose
`.rolling(window=12)`	Creates 12-period rolling window
`.mean()`	Calculates rolling average
`.std()`	Calculates rolling standard deviation

Why window=12?

- For weekly data with yearly seasonality
- 12 months ≈ 52 weeks, but 12 is a common default
- Captures seasonal cycle

Stationarity interpretation:

- **Constant mean line** → Stationary
- **Increasing/decreasing mean** → Non-stationary (has trend)
- **Changing variance** → Non-stationary (heteroscedasticity)

Augmented Dickey-Fuller Test

```
from statsmodels.tsa.stattools import adfuller, kpss

def ADF_test(timeseries, dataDesc):
    print(' > Is the {} stationary ?'.format(dataDesc))
    dftest = adfuller(timeseries.dropna(), autolag='AIC')
    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])))
```

Explanation:

Code	Purpose

Code	Purpose
`adfuller(timeseries.dropna(), autolag='AIC')`	Performs ADF test with automatic lag selection
`dftest[0]`	Test statistic
`dftest[1]`	P-value
`dftest[4]`	Critical values dictionary
`autolag='AIC'`	Uses Akaike Information Criterion to select optimal lag

ADF Test Interpretation:

Condition	Result
P-value < 0.05	**Stationary** (reject null hypothesis)
P-value ≥ 0.05	**Non-stationary** (fail to reject null)
Test statistic < Critical value	**Stationary**

Null hypothesis: The series has a unit root (non-stationary)

KPSS Test

```
def kpss_test(series, **kw):
    statistic, p_value, n_lags, critical_values = kpss(series, **kw)
    print(f'KPSS Statistic: {statistic}')
    print(f'p-value: {p_value}')
    print(f'num lags: {n_lags}')
    print('Critical Values:')
    for key, value in critical_values.items():
        print(f'   {key} : {value}')
    print(f'Result: The series is {"not " if p_value < 0.05 else
""}stationary')

kpss_test(df["M01AB"], regression="ct", nlags="auto")
```

Explanation:

Parameter	Value	Purpose
`regression="ct"`	Constant + Trend	Tests for trend stationarity
`nlags="auto"`	Automatic lag selection	Optimal lag determination

KPSS vs ADF:

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

Why use both?

- Confirmation: If both agree, result is reliable
- If they disagree: Data may need differencing or detrending

Detrending

```
y_detrend = (y - y.rolling(window=12).mean()) / y.rolling(window=12).std()

test_stationarity(y_detrend, 'de-trended data')
ADF_test(y_detrend, 'de-trended data')
```

Explanation:

Formula: $$y_{detrend} = \frac{y - \mu_{rolling}}{\sigma_{rolling}}$$

Component	Purpose
`y - y.rolling(window=12).mean()`	Subtracts rolling mean (removes trend)
`/ y.rolling(window=12).std()`	Divides by rolling std (standardizes variance)

This is essentially z-score normalization with rolling statistics

---

Differencing

```
first_diff = df['M01AB'].diff()
test_stationarity(first_diff, 'differenced data')
ADF_test(first_diff, 'differenced data')
```

Explanation:

Code	Formula	Purpose
`.diff()`	$y'*t* = *y_t* - *y*{t-1}$	First-order differencing

Why differencing?

- Removes trend
- Makes variance more constant
- Required for non-stationary data in ARIMA models

Differencing order (d in ARIMA):

- `d=0` : Already stationary
- `d=1` : First differencing (most common)
- `d=2` : Second differencing (rarely needed)

2.6 SARIMA Model

Train-Test Split

```
y_to_train = y[:'2018-01-14']  # dataset to train
y_to_val = y['2018-01-21':]    # last X months for test
predict_date = len(y) - len(y[:'2018-01-21'])  # number of data points for
test set
```

Explanation:

Variable	Purpose
`y_to_train`	Training data (2014 to early 2018)
`y_to_val`	Validation/test data (2018 onwards)
`predict_date`	Count of test observations

Why time-based split (not random)?

- Time series has temporal dependency
- Random split would leak future information
- Maintains chronological order

SARIMA Grid Search

```
import itertools

def sarima_grid_search(y, seasonal_period):
    p = d = q = range(0, 2)
    pdq = list(itertools.product(p, d, q))
    seasonal_pdq = [(x[0], x[1], x[2], seasonal_period) for x in
list(itertools.product(p, d, q))]

    mini = float('+inf')
```

```
    for param in pdq:
        for param_seasonal in seasonal_pdq:
            try:
                mod = sm.tsa.statespace.SARIMAX(y,
                                                order=param,
                                                seasonal_order=param_seasonal,
                                                enforce_stationarity=False,
                                                enforce_invertibility=False)
                results = mod.fit()

                if results.aic < mini:
                    mini = results.aic
                    param_mini = param
                    param_seasonal_mini = param_seasonal
            except:
                continue
    print('The set of parameters with the minimum AIC is: SARIMA{}x{} - AIC:
{}'.format(
        param_mini, param_seasonal_mini, mini))
```

Line-by-Line Explanation:

Line	Code	Purpose
1	`p = d = q = range(0, 2)`	Creates [0, 1] for each parameter
2	`itertools.product(p, d, q)`	Generates all combinations: (0,0,0), (0,0,1), ..., (1,1,1)
3	`seasonal_pdq`	Same for seasonal parameters (P,D,Q,m)
4	`mini = float('+inf')`	Initialize with infinity for minimum comparison
5	Nested `for` loops	Tests every (p,d,q) × (P,D,Q,m) combination
6	`SARIMAX(...)`	Creates SARIMA model
7	`results.aic`	Akaike Information Criterion (lower is better)
8	`try/except`	Skips parameter combinations that fail to converge

SARIMA Parameters:

Parameter	Name	Meaning
p	AR order	Number of autoregressive lags
d	Differencing	Number of differences to make stationary
q	MA order	Number of moving average lags
P	Seasonal AR	Seasonal autoregressive order
D	Seasonal differencing	Seasonal difference order

Parameter	Name	Meaning
Q	Seasonal MA	Seasonal moving average order
m	Seasonal period	Length of one season (12 for monthly data)

SARIMA Evaluation Function

```
def sarima_eva(y, order, seasonal_order, seasonal_period, pred_date,
y_to_test):
    # Fit the model
    mod = sm.tsa.statespace.SARIMAX(y,
                                    order=order,
                                    seasonal_order=seasonal_order,
                                    enforce_stationarity=False,
                                    enforce_invertibility=False)
    results = mod.fit()
    print(results.summary().tables[1])

    results.plot_diagnostics(figsize=(16, 8))
    plt.show()

    # One-step ahead forecasts (dynamic=False)
    pred = results.get_prediction(start=pd.to_datetime(pred_date),
dynamic=False)
    pred_ci = pred.conf_int()
    y_forecasted = pred.predicted_mean
    mse = ((y_forecasted - y_to_test) ** 2).mean()
    print('RMSE with dynamic=False: {}'.format(round(np.sqrt(mse), 2)))

    # Dynamic forecasts (dynamic=True)
    pred_dynamic = results.get_prediction(start=pd.to_datetime(pred_date),
dynamic=True, full_results=True)
    pred_dynamic_ci = pred_dynamic.conf_int()
    y_forecasted_dynamic = pred_dynamic.predicted_mean
    mse_dynamic = ((y_forecasted_dynamic - y_to_test) ** 2).mean()
    print('RMSE with dynamic=True: {}'.format(round(np.sqrt(mse_dynamic), 2)))

    return results
```

Key Concepts Explained:

`enforce_stationarity=False`:

- Allows fitting even if data isn't perfectly stationary
- More flexible but may produce unstable forecasts

`enforce_invertibility=False`:

- Allows MA coefficients outside (-1, 1)
- Useful for complex seasonal patterns

`dynamic=False` vs `dynamic=True`:

Mode	Description	Use Case
`dynamic=False`	Uses actual values for each prediction	More accurate, one-step ahead
`dynamic=True`	Uses predicted values for future predictions	True forecasting scenario

`plot_diagnostics()`: Shows 4 diagnostic plots:

1. **Standardized residuals** - Should look like white noise
2. **Histogram** - Should be normally distributed
3. **Q-Q plot** - Points should follow diagonal line
4. **Correlogram** - No significant autocorrelation in residuals

---

Forecast Function

```
def forecast(model, predict_steps, y):
    pred_uc = model.get_forecast(steps=predict_steps)
    pred_ci = pred_uc.conf_int()

    ax = y.plot(label='observed', figsize=(14, 7))
    pred_uc.predicted_mean.plot(ax=ax, label='Forecast')
    ax.fill_between(pred_ci.index,
                    pred_ci.iloc[:, 0],
                    pred_ci.iloc[:, 1], color='k', alpha=.05)
    ax.set_xlabel('Date')
    ax.set_ylabel(y.name)
    plt.legend()
    plt.show()

    # Produce the forecasted tables
    pm = pred_uc.predicted_mean.reset_index()
    pm.columns = ['Date', 'Predicted_Mean']
    pci = pred_ci.reset_index()
    pci.columns = ['Date', 'Lower Bound', 'Upper Bound']
    final_table = pm.join(pci.set_index('Date'), on='Date')

    return final_table
```

Explanation:

Code	Purpose
`get_forecast(steps=predict_steps)`	Generates future predictions

Code	Purpose
`conf_int()`	Gets confidence interval bounds
`fill_between(...)`	Shades the confidence interval area
`alpha=.05`	5% transparency for shading
`reset_index()`	Converts index to column for table
`.join(...)`	Combines predictions with confidence bounds

2.7 Prophet Model

Import and Setup

```
import warnings
warnings.simplefilter('ignore')
import pandas as pd
import prophet
from prophet.plot import plot_plotly, plot_components_plotly
from prophet import Prophet
from statsmodels.tools.eval_measures import rmse
```

Data Preparation for Prophet

```
df_daily = pd.read_csv("...")
df_daily.drop(['Year','Month','Hour','Weekday Name','M01AE', 'N02BA', 'N02BE',
'N05B', 'N05C', 'R03','R06'], axis=1, inplace=True)
df_daily['datum'] = pd.to_datetime(df_daily['datum'])
df_daily.rename(columns = {'datum':'Date'}, inplace = True)

df_daily.columns = ['ds', 'y']
```

Explanation:

Prophet requires specific column names:

Column	Required Name	Purpose
Date	`ds`	Datestamp column
Target	`y`	Value to predict

Why drop other drug columns?

- Prophet is univariate (one target variable)
- Each drug category needs separate model

Train-Test Split

```
train = df_daily.iloc[:len(df_daily)-631]
test = df_daily.iloc[len(df_daily)-631:]
```

Explanation:

Code	Purpose
`iloc[:len(df_daily)-631]`	First N-631 rows for training
`iloc[len(df_daily)-631:]`	Last 631 rows for testing

Why 631?

- Approximately 21 months of daily data
- Matches the test period used in SARIMA

Model Fitting

```
m = Prophet(interval_width=0.95, yearly_seasonality=True)
m.fit(train)
future = m.make_future_dataframe(periods=25, freq='M')
forecast = m.predict(future)
```

Parameter Explanation:

Parameter	Value	Purpose
`interval_width=0.95`	95%	Confidence interval width
`yearly_seasonality=True`	Enabled	Captures annual patterns
`periods=25`	25 months	Forecast horizon
`freq='M'`	Monthly	Forecast frequency

Prophet Seasonality Options:

Parameter	Default	Purpose
`yearly_seasonality`	'auto'	Annual patterns

Parameter	Default	Purpose
`weekly_seasonality`	'auto'	Weekly patterns
`daily_seasonality`	'auto'	Daily patterns (for sub-daily data)

Visualization

```
plot_plotly(m, forecast)          # Interactive forecast plot
plot_components_plotly(m, forecast)  # Trend and seasonality components
```

Explanation:

`plot_plotly()`:

- Shows historical data (black dots)
- Blue line: Predicted values
- Shaded area: Confidence interval

`plot_components_plotly()`:

- **Trend:** Long-term direction
- **Yearly:** Annual seasonality pattern
- **Weekly:** Day-of-week effects (if enabled)

Model Evaluation

```
predictions = forecast.iloc[-631:]['yhat']
print("RMSE: ", rmse(predictions, test['y']))
```

Explanation:

Code	Purpose
`forecast.iloc[-631:]`	Gets last 631 predictions (test period)
`['yhat']`	Prophet's predicted values column
`rmse(predictions, test['y'])`	Calculates Root Mean Squared Error

Prophet output columns:

Column	Description

Column	Description
`ds`	Datestamp
`yhat`	Predicted value
`yhat_lower`	Lower confidence bound
`yhat_upper`	Upper confidence bound
`trend`	Trend component
`yearly`	Yearly seasonality component

2.8 SVR Model

Import Libraries

```
import warnings
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import datetime as dt
import math
import plotly.graph_objects as go
import plotly.express as px

from sklearn.svm import SVR
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error
from sklearn.model_selection import GridSearchCV

%matplotlib inline
pd.options.display.float_format = '{:,.2f}'.format
np.set_printoptions(precision=2)
warnings.filterwarnings("ignore")
```

Library Purpose:

Library	Purpose
`SVR`	Support Vector Regression model
`MinMaxScaler`	Scales features to [0, 1] range
`GridSearchCV`	Hyperparameter tuning with cross-validation
`plotly`	Interactive visualizations

Data Scaling

```
train_start_dt = '2014-01-02'
test_start_dt = '2018-01-16'

train = df_daily.copy()[(df_daily.index >= train_start_dt) & (df_daily.index <
test_start_dt)][['M01AB']]
test = df_daily.copy()[df_daily.index >= test_start_dt][['M01AB']]

scaler = MinMaxScaler()
train['M01AB'] = scaler.fit_transform(train)
test['M01AB'] = scaler.fit_transform(test)
```

Explanation:

Code	Purpose
`MinMaxScaler()`	Creates scaler object
`fit_transform(train)`	Fits on training data AND transforms it
`fit_transform(test)`	**Note:** Should use `transform(test)` only!

MinMaxScaler formula: $$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Why scale for SVR?

- SVR uses distance-based calculations
- Features on different scales can bias the model
- Scaling improves convergence

⚠ **Code Issue:**

```
test['M01AB'] = scaler.fit_transform(test)  # WRONG - should be transform()
```

Should be:

```
test['M01AB'] = scaler.transform(test)  # Correct
```

Creating Time Steps

```
train_data = train.values
test_data = test.values
```

```
timesteps = 5

# Converting training data to 2D tensor
train_data_timesteps = np.array([[j for j in train_data[i:i+timesteps]] for i
in range(0, len(train_data)-timesteps+1)])[:,:,0]
```

Explanation:

What this does: Creates sliding windows of length 5 for supervised learning.

Example:

```
Original: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Timesteps = 5

Window 1: [1, 2, 3, 4, 5]  → X: [1,2,3,4], y: 5
Window 2: [2, 3, 4, 5, 6]  → X: [2,3,4,5], y: 6
Window 3: [3, 4, 5, 6, 7]  → X: [3,4,5,6], y: 7
...
```

Line breakdown:

Code	Purpose
`train_data[i:i+timesteps]`	Extracts window of 5 consecutive values
`for i in range(0, len(train_data)-timesteps+1)`	Slides window across data
`[:,:,0]`	Removes extra dimension (reshapes to 2D)

Hyperparameter Tuning

```
model = SVR()
param_grid = {
    'kernel': ['rbf'],
    'gamma': [0.1, 0.01, 0.001],
    'C': [0.1, 1, 10],
    'epsilon': [0.05, 0.1]
}

grid = GridSearchCV(SVR(), param_grid, refit=True, verbose=3)
grid.fit(x_train, y_train)
grid.best_params_
```

SVR Hyperparameters Explained:

Parameter	Values Tested	Purpose
`kernel`	'rbf'	Radial Basis Function (Gaussian kernel)
`gamma`	0.1, 0.01, 0.001	Kernel coefficient - controls influence radius
`C`	0.1, 1, 10	Regularization - trade-off between error and margin
`epsilon`	0.05, 0.1	Tube width - errors within tube are ignored

GridSearchCV parameters:

Parameter	Value	Purpose
`refit=True`	Automatically retrain	Best model is fitted on full training data
`verbose=3`	High verbosity	Shows progress during search

Total combinations tested: 1 × 3 × 3 × 2 = 18 parameter combinations

---

Model Training and Prediction

```
model = SVR(kernel='rbf', gamma=0.01, C=1, epsilon=0.1)
model.fit(x_train, y_train[:,0])

y_train_pred = model.predict(x_train).reshape(-1, 1)
y_test_pred = model.predict(x_test).reshape(-1, 1)
```

Explanation:

Code	Purpose
`SVR(...)`	Creates model with best parameters
`.fit(x_train, y_train[:,0])`	Trains model (y_train[:,0] flattens to 1D)
`.predict(x_train)`	Makes predictions
`.reshape(-1, 1)`	Reshapes to column vector for inverse scaling

Inverse Scaling

```
y_train_pred = scaler.inverse_transform(y_train_pred)
y_test_pred = scaler.inverse_transform(y_test_pred)

y_train = scaler.inverse_transform(y_train)
y_test = scaler.inverse_transform(y_test)
```

Explanation:

Code	Purpose
`inverse_transform()`	Converts scaled values back to original scale

Why inverse transform?

- Model predictions are in scaled [0, 1] range
- RMSE should be calculated on original scale for interpretation
- Visualization requires original units

Evaluation

```
train_timestamps = df_daily[(df_daily.index < test_start_dt) & (df_daily.index
>= train_start_dt)].index[timesteps-1:]
test_timestamps = df_daily[test_start_dt:].index[timesteps-1:]

plt.figure(figsize=(25, 6))
plt.plot(train_timestamps, y_train, color='red', linewidth=2.0, alpha=0.6)
plt.plot(train_timestamps, y_train_pred, color='blue', linewidth=0.8)
plt.legend(['Actual', 'Predicted'])
plt.xlabel('Timestamp')
plt.title("Training data prediction")
plt.show()

rmse = np.sqrt(mse)
print(f'Root mean squared error: {rmse:.2f}')
```

Explanation:

Code	Purpose
`index[timesteps-1:]`	Adjusts timestamps (first timesteps-1 points don't have predictions)
`alpha=0.6`	60% opacity for actual data line
`np.sqrt(mse)`	Converts MSE to RMSE

Summary: Model Comparison

Aspect	SARIMA	Prophet	SVR
Type	Statistical	Additive Regression	Machine Learning

Aspect	SARIMA	Prophet	SVR
Handles Seasonality	Yes (explicit P,D,Q,m)	Yes (automatic)	No (needs engineering)
Interpretability	High (coefficients)	Medium (components)	Low (black box)
Data Requirements	Must be stationary	Handles non-stationary	Needs scaling
Hyperparameters	p,d,q,P,D,Q,m	seasonality, changepoints	C, gamma, epsilon
Best For	Clear seasonal patterns	Human-scale seasonality	Non-linear relationships
Computation	Fast	Medium	Slow (grid search)

Key Takeaways

1. **Always check stationarity** before applying ARIMA models
2. **Use multiple tests** (ADF + KPSS) for confirmation
3. **Grid search** helps find optimal SARIMA parameters
4. **Prophet requires** specific column names (`ds` , `y`)
5. **SVR requires** proper scaling and timestep creation
6. **Compare RMSE** across models on the same test set
7. **Visualize diagnostics** to validate model assumptions

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