Sales Forecasting and Optimizng Warehouse Operations for Walmart Stores

Mamatha Singh

9 min read · Just now

Introduction:

Retailers like Walmart handle vast inventories and face challenges in predicting sales and managing warehouses efficiently. Poor forecasting can result in overstocking or stockouts, directly affecting profits and customer satisfaction. This project analyzes three years of Walmart sales data to uncover patterns, forecast trends, and optimize warehouse operations using advanced predictive analytics and interactive dashboards.

Key questions addressed:

1.How do holidays affect sales?

2.How do economic factors influence weekly sales?

3.Can sales velocity improve warehouse stock planning?

Data Summary:

The dataset includes sales and economic indicators for 45 Walmart stores. It integrates three files:

1.Stores: IDs, types, and sizes.

2.Features: Weekly data on fuel prices, CPI, unemployment rates, and holidays.

3.Sales: Weekly department sales with holiday flags.

Methodology:

Data Preparation:

Cleaned datasets and combined them into a central table of 421,570 rows.

Removed irrelevant columns like promotional markdowns.

Exploratory Analysis:

Sales spiked around holidays, emphasizing their importance in forecasting.

Visualized correlations between economic indicators and sales.

Feature Engineering:

Mapped holidays using a Nixtala library.

Introduced Sales Velocity, calculated as sales per store size.

Dashboard Development:

Created a multi-layer dashboard in Streamlit, hosted on HuggingFace Spaces.

Features include holiday trends, top stores, and sales velocity visualizations.

```
def calculate_sales_velocity(df, store_number=None, dept_number=None):
    """
    Calculates and normalizes sales velocity for overall data, store, or departm

    Args:
    df (pd.DataFrame): The input DataFrame.
    store_number (int, optional): The store number to filter by. Defaults to Non
    dept_number (int, optional): The department number to filter by. Defaults to

    Returns:
    float: The normalized sales velocity.
    """
    velocity_df = df.copy()

    if store_number and dept_number:
        velocity_df = velocity_df.query('Store == @store_number and Dept == @dep
    elif store_number:
        velocity_df = velocity_df.query('Store == @store_number')

    velocity_df = velocity_df.groupby(['Date'])['Weekly_Sales'].mean().reset_ind

    n_days = velocity_df["Date"].nunique()
    Sales_n = velocity_df["Weekly_Sales"].sum()
    win_rate = 1  # Assuming since every week has sales

    sales_velocity = (Sales_n * win_rate) / n_days

    norm = (sales_velocity - velocity_df["Weekly_Sales"].min()) / (velocity_df["
     # Format norm to two decimal places
    formatted_norm = f"{norm:.2f}"

    return formatted_norm

calculate_sales_velocity(combined_df, store_number=33, dept_number= 33)
```

This shows how sales velocity changes over time, aiding warehouse stock decisions.

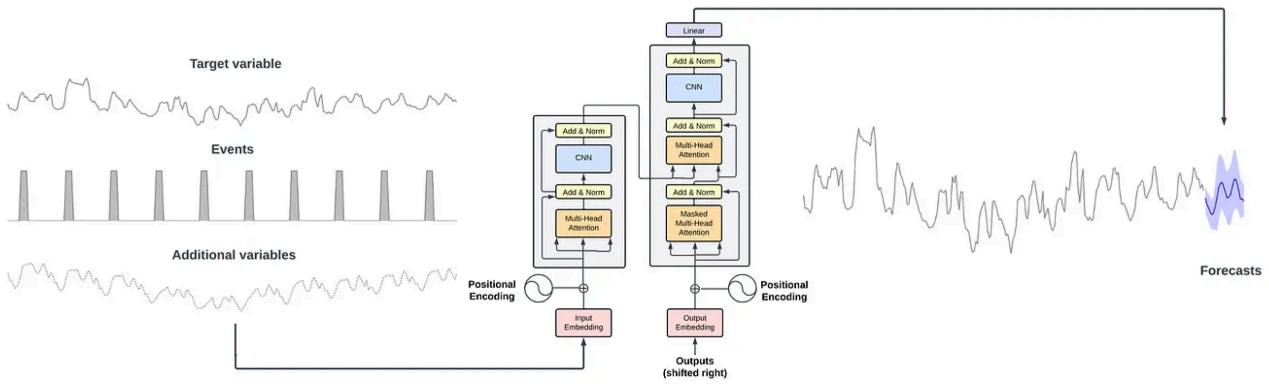
- High Sales Velocity: Requires frequent restocking to avoid stockouts, with adjustments for delivery and sorting processes in multi-warehouse setups.
- Low Sales Velocity: Calls for minimal restocking to prevent overstock. Instead, focus on advertising, customer retention, and improving customer experience to boost sales.

Predictive Models :

We have built predictive models, including ARIMA, Random Forest, XGBoost, and TimeGPT.

TimeGPT

- TimeGPT is a production-ready generative pretrained transformer for time series. It's capable of accurately predicting various domains such as retail, electricity, finance, and IoT with just a few lines of code.
- TimeGPT model is not based on any existing large language model(LLM). Instead, it is independently trained on a vast amount of time series data, and the large transformer model is designed to minimize the forecasting error.



- The architecture uses an encoder-decoder setup with several layers, where each layer includes shortcuts (residual connections) and normalization to improve performance. At the end, a linear layer converts the decoder's output into the forecasted values. The idea is that attention-based methods can effectively learn patterns from past data and predict possible future outcomes.

Below is the code for TimeGPT.

```
from nixtla import NixtlaClient

nixtla_client = NixtlaClient(
    # defaults to os.environ.get("NIXTLA_API_KEY")
    api_key = 'nixak-odp8YhH8CEjt2wGiZ5zP5XCHoeMAFCj0hGGzpBDIA2QIcEZz4RqI98kxn7d
)

# Overall forcast
def get_forecasts(df):

      #df= combined_df.copy()

      #input_df = df.groupby(['Date'])['Weekly_Sales'].mean().reset_index()

      fcst_store = nixtla_client.forecast(
          df= df,
          h = 24,                                #  Forecast horizon of 24 tim
```

```
            freq='W-FRI',                           # Frequency of date is Weekly
            time_col='Date',
            model='timegpt-1-long-horizon',         # Long horizon model
            target_col='Weekly_Sales',
            finetune_steps=15,                       # Model will go through 5 ite
            X_df= df.drop(columns=['Weekly_Sales'])[:24], #Exogenous Features
            level = [50,90,80],
            add_history = True
        )

        fig = nixtla_client.plot(
            df = df,
            forecasts_df = fcst_store,
            time_col = 'Date',
            target_col = 'Weekly_Sales',
            engine = 'plotly',
            #max_insample_length= 24,
            level = [50,90,80]
          )

        fig.update_layout(
              title_text="Sales forcast for next 6 months",  # Your existing title
              height=250,
              width=1000
          )
        return fig, fcst_store

    fig, fcst_df = get_forecasts(df)
    fig
```

Random Forest

```
    aggregated_df['Date'] = pd.to_datetime(aggregated_df['Date'], format='%d/%m/%Y')

    # Add lag features to incorporate the effect of previous weeks' sales:

    aggregated_df['Lag_1_Week'] = aggregated_df['Weekly_Sales'].shift(1)
    aggregated_df['Lag_2_Weeks'] = aggregated_df['Weekly_Sales'].shift(2)

    # Add rolling averages
    from sklearn.impute import SimpleImputer

    aggregated_df['Rolling_4_Week'] = aggregated_df['Weekly_Sales'].rolling(window=4
    # aggregated_df['Rolling_5_Week'] = aggregated_df['Weekly_Sales'].rolling(window

    # Impute NaNs (using mean strategy)
    imputer = SimpleImputer(strategy='mean')  # You can also use 'median' if preferr
    aggregated_df[['Lag_1_Week', 'Lag_2_Weeks', 'Rolling_4_Week']] = imputer.fit_tra
        aggregated_df[['Lag_1_Week', 'Lag_2_Weeks', 'Rolling_4_Week']]
    )

    # Check if NaNs are handled
    print(aggregated_df.isnull().sum())

    # Extract date-related features:
    aggregated_df['Year'] = aggregated_df['Date'].dt.year

    aggregated_df['Month'] = aggregated_df['Date'].dt.month
    aggregated_df['Day_of_Week'] = aggregated_df['Date'].dt.dayofweek  # 0 = Monday,

    from sklearn.model_selection import train_test_split

    # Filter training data (2010 and 2011) and testing data (2012)
    train_df = aggregated_df[aggregated_df['Year'].isin([2010, 2011])]
    test_df = aggregated_df[aggregated_df['Year'] == 2012]

    # Separate features (X) and target (y) for training and testing sets
    X_train = train_df.drop(columns=['Weekly_Sales', 'Date','Year'])
    y_train = train_df[['Weekly_Sales', 'Date']]
    X_test = test_df.drop(columns=['Weekly_Sales', 'Date', 'Year'])
    y_test = test_df[['Weekly_Sales', 'Date']]

    from sklearn.ensemble import RandomForestRegressor
    import xgboost as xgb
    from sklearn.metrics import mean_absolute_error, mean_squared_error, explained_v
    import numpy as np
    import plotly.graph_objects as go

    # Define SMAPE function to calculate metrics
    def smape(y_true, y_pred):
        # Ensure no division by zero and calculate SMAPE
        denominator = (np.abs(y_true) + np.abs(y_pred))
        diff = np.abs(y_true - y_pred) / denominator
        diff[denominator == 0] = 0  # Handle cases where both y_true and y_pred are
        return 2 * np.mean(diff)

    # Initialize and fit the Random Forest model
    rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
    rf_model.fit(X_train, y_train['Weekly_Sales'])

    # Predict on the test data
    y_pred = rf_model.predict(X_test)
```

```python
# Evaluate the model performance
mae = mean_absolute_error(y_test['Weekly_Sales'], y_pred)
rmse = mean_squared_error(y_test['Weekly_Sales'], y_pred, squared=False)
smape_score = smape(y_test['Weekly_Sales'], y_pred)

# Print the evaluation metrics
print(f"SMAPE: {smape_score}")
print("Mean Absolute Error (MAE):", mae)
print("Root Mean Squared Error (RMSE):", rmse)

# Sort the data by date to avoid zigzag patterns
y_test_sorted = y_test.sort_values(by='Date')
y_pred_sorted = [y for _, y in sorted(zip(y_test['Date'], y_pred))]

# Create a Plotly figure
fig = go.Figure()

# Add actual weekly sales
fig.add_trace(go.Scatter(
    x=y_test_sorted['Date'],
    y=y_test_sorted['Weekly_Sales'],
    mode='lines',
    name='Actual',
    line=dict(color='blue')
))

# Add predicted weekly sales
fig.add_trace(go.Scatter(
    x=y_test_sorted['Date'],
    y=y_pred_sorted,
    mode='lines',
    name='Predicted',
    line=dict(color='red')
))

# Update layout
fig.update_layout(
    title='Predicted vs Actual Weekly Sales (Random Forest)',
    xaxis_title='Week',
    yaxis_title='Weekly Sales',
    legend_title='Legend',
    template='plotly_white',
    width=1000,
    height=600
)

# Show plot
fig.show()
```

Without Hyperparameter tuning the performance of RandomForest is like this MAE — 2103.04 and SMAPE — 0.12.

```
# Hyperparameter tuning for RandomForest
from sklearn.model_selection import GridSearchCV

param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [10, 20, 30, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['log2', 'sqrt'],
    'bootstrap': [True, False]
}

rf = RandomForestRegressor(random_state=42)

grid_search_rf = GridSearchCV(estimator=rf, param_grid=param_grid, cv=3, n_jobs=
grid_search_rf.fit(X_train, y_train['Weekly_Sales'])

best_params_rf = grid_search_rf.best_estimator_
# best_params_
print(f"Best parameters: {best_params_rf}")
y_pred= best_params_rf.predict(X_test)
# y_pred= grid_search.predict(X_test)

y_pred= best_params_rf.predict(X_test)

# Evaluate the model performance
mae = mean_absolute_error(y_test['Weekly_Sales'], y_pred)
rmse = mean_squared_error(y_test['Weekly_Sales'], y_pred, squared=False)
smape_score = smape(y_test['Weekly_Sales'], y_pred)

# Print the evaluation metrics
print(f"SMAPE: {smape_score}")
print("Mean Absolute Error (MAE):", mae)
print("Root Mean Squared Error (RMSE):", rmse)

# Sort the data by date to avoid zigzag patterns
y_test_sorted = y_test.sort_values(by='Date')
y_pred_sorted = [y for _, y in sorted(zip(y_test['Date'], y_pred))]

# Create a Plotly figure
fig = go.Figure()
```

```
# Add actual weekly sales
fig.add_trace(go.Scatter(
    x=y_test_sorted['Date'],
    y=y_test_sorted['Weekly_Sales'],
    mode='lines',
    name='Actual',
    line=dict(color='blue')
))

# Add predicted weekly sales
fig.add_trace(go.Scatter(
    x=y_test_sorted['Date'],
    y=y_pred_sorted,
    mode='lines',
    name='Predicted',
    line=dict(color='red')
))

# Update layout
fig.update_layout(
    title='Predicted vs Actual Weekly Sales (Random Forest)',
    xaxis_title='Week',
    yaxis_title='Weekly Sales',
    legend_title='Legend',
    template='plotly_white',
    width=1000,
    height=600
)

# Show plot
fig.show()
```

After Hyperparameter tuning The performance of RandomForest is like this MAE- 1909.10 and SMAPE — 0.11

XG Boost

```
# Initialize and fit the XGBoost model
xg_model = xgb.XGBRegressor(n_estimators=100, random_state=42)
xg_model.fit(X_train, y_train['Weekly_Sales'])
```

```python
# Predict on the test data
y_pred_xg = xg_model.predict(X_test)

# Evaluate the model
xg_rmse = mean_squared_error(y_test['Weekly_Sales'], y_pred_xg, squared=False)
xg_mae = mean_absolute_error(y_test['Weekly_Sales'], y_pred_xg)  # Mean Absolute
smape_score = smape(y_test['Weekly_Sales'], y_pred_xg)

# Print the evaluation metrics
print(f"SMAPE: {smape_score}")
print(f"XGBoost RMSE: {xg_rmse}")
print(f"XGBoost MAE: {xg_mae}")

# Sort the data by date to ensure the plot is smooth
y_test_sorted = y_test.sort_values(by='Date')
y_pred_xg_sorted = [y for _, y in sorted(zip(y_test['Date'], y_pred_xg))]

# Create a Plotly figure
fig = go.Figure()

# Add actual weekly sales
fig.add_trace(go.Scatter(
    x=y_test_sorted['Date'],
    y=y_test_sorted['Weekly_Sales'],
    mode='lines',
    name='Actual',
    line=dict(color='blue')
))

# Add predicted weekly sales
fig.add_trace(go.Scatter(
    x=y_test_sorted['Date'],
    y=y_pred_xg_sorted,
    mode='lines',
    name='Predicted',
    line=dict(color='red')
))

# Update layout
fig.update_layout(
    title='XGBoost Predicted vs Actual Weekly Sales',
    xaxis_title='Week',
    yaxis_title='Weekly Sales',
    legend_title='Legend',
    template='plotly_white',
    width=1000,
    height=600
)
```

```
# Show plot
fig.show()
```

Without Hyperparameter tunning The performance of XGboost is like this MAE- 1957.20 and SMAPE — 0.11

```
from sklearn.model_selection import RandomizedSearchCV
from sklearn.metrics import make_scorer, mean_squared_error
import numpy as np

# Define the XGBRegressor
xgb_model = XGBRegressor(objective='reg:squarederror', random_state=42)

# Parameter grid for RandomizedSearchCV
param_grid = {
    'n_estimators': [100, 200, 300],
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 5, 7],
    'min_child_weight': [1, 3, 5],
    'subsample': [0.8, 1.0],
    'colsample_bytree': [0.8, 1.0]
}
# Define RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=xgb_model,
    param_distributions=param_grid,
    n_iter=50,  # Number of parameter settings to sample
    scoring=make_scorer(mean_squared_error, greater_is_better=False),  # Using M
    cv=5,  # 5-fold cross-validation
    verbose=2,
    random_state=42,
    n_jobs=-1  # Use all available processors
)
# Fit RandomizedSearchCV
random_search.fit(X_train, y_train['Weekly_Sales'])

# Output best parameters and score
print("Best Parameters:", random_search.best_params_)
print("Best Score (Negative MSE):", random_search.best_score_)

# Predict using the best model
best_model_xgb = random_search.best_estimator_
y_pred_xg = best_model_xgb.predict(X_test)
```

```python
# Evaluate the model
xg_rmse = mean_squared_error(y_test['Weekly_Sales'], y_pred_xg, squared=False)
xg_mae = mean_absolute_error(y_test['Weekly_Sales'], y_pred_xg)  # Mean Absolute
smape_score = smape(y_test['Weekly_Sales'], y_pred_xg)

# Print the evaluation metrics
print(f"SMAPE: {smape_score}")
print(f"XGBoost RMSE: {xg_rmse}")
print(f"XGBoost MAE: {xg_mae}")

# Sort the data by date to ensure the plot is smooth
y_test_sorted = y_test.sort_values(by='Date')
y_pred_xg_sorted = [y for _, y in sorted(zip(y_test['Date'], y_pred_xg))]

# Create a Plotly figure
fig = go.Figure()

# Add actual weekly sales
fig.add_trace(go.Scatter(
    x=y_test_sorted['Date'],
    y=y_test_sorted['Weekly_Sales'],
    mode='lines',
    name='Actual',
    line=dict(color='blue')
))

# Add predicted weekly sales
fig.add_trace(go.Scatter(
    x=y_test_sorted['Date'],
    y=y_pred_xg_sorted,
    mode='lines',
    name='Predicted',
    line=dict(color='red')
))

# Update layout
fig.update_layout(
    title='XGBoost Predicted vs Actual Weekly Sales',
    xaxis_title='Week',
    yaxis_title='Weekly Sales',
    legend_title='Legend',
    template='plotly_white',
    width=1000,
    height=600
)

# Show plot
fig.show()
```

After Hyperparameter tunning using GridSearchCV. The performance of XGboost is like this MAE- 662.72 and SMAPE — 0.03

Ensemble of Random Forest and XGBoost

```
from sklearn.ensemble import VotingRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
import matplotlib.pyplot as plt

# Create the ensemble model (Voting Regressor)
ensemble_model = VotingRegressor(estimators=[('rf', best_params_rf), ('xg', best

# Fit the ensemble model
ensemble_model.fit(X_train, y_train['Weekly_Sales'])

# Predict on the test data
y_pred_ensemble = ensemble_model.predict(X_test)

# Evaluate the ensemble model
ensemble_rmse = mean_squared_error(y_test['Weekly_Sales'], y_pred_ensemble, squa
ensemble_mae = mean_absolute_error(y_test['Weekly_Sales'], y_pred_ensemble)  # M
smape_score = smape(y_test['Weekly_Sales'], y_pred_ensemble)

# Print the evaluation metrics
print(f"SMAPE: {smape_score}")
print(f"Ensemble Model RMSE: {ensemble_rmse}")
print(f"Ensemble Model MAE: {ensemble_mae}")

# # Plot Predicted vs Actual values
# plt.figure(figsize=(10, 6))
# plt.plot(y_test['Date'], y_test['Weekly_Sales'], color='blue', label='Actual')
# plt.plot(y_test['Date'], y_pred_xg, color='red', label='Predicted')
# # plt.scatter(y_test['Weekly_Sales'], y_pred_ensemble, color='blue', alpha=0.6
# # plt.plot([min(y_test['Weekly_Sales']), max(y_test['Weekly_Sales'])], [min(y_
# plt.title('Predicted vs Actual Values (Ensemble Model)')
# plt.xlabel('Actual Values')
# plt.ylabel('Predicted Values')

# # Add a legend to the plot
# plt.legend()
```

```python
# # Show the plot
# plt.show()

import plotly.graph_objects as go

# Sort the data by date to ensure smooth plotting
y_test_sorted = y_test.sort_values(by='Date')
y_pred_ensemble_sorted = [y for _, y in sorted(zip(y_test['Date'], y_pred_ensemb

# Create a Plotly figure
fig = go.Figure()

# Add actual weekly sales
fig.add_trace(go.Scatter(
    x=y_test_sorted['Date'],
    y=y_test_sorted['Weekly_Sales'],
    mode='lines',
    name='Actual',
    line=dict(color='blue')
))

# Add ensemble model predicted weekly sales
fig.add_trace(go.Scatter(
    x=y_test_sorted['Date'],
    y=y_pred_ensemble_sorted,
    mode='lines',
    name='Predicted (Ensemble)',
    line=dict(color='red')
))

# Update layout
fig.update_layout(
    title='Ensemble Model Predicted vs Actual Weekly Sales',
    xaxis_title='Week',
    yaxis_title='Weekly Sales',
    legend_title='Legend',
    template='plotly_white',
    width=1000,
    height=600
)

# Show plot
fig.show()
```

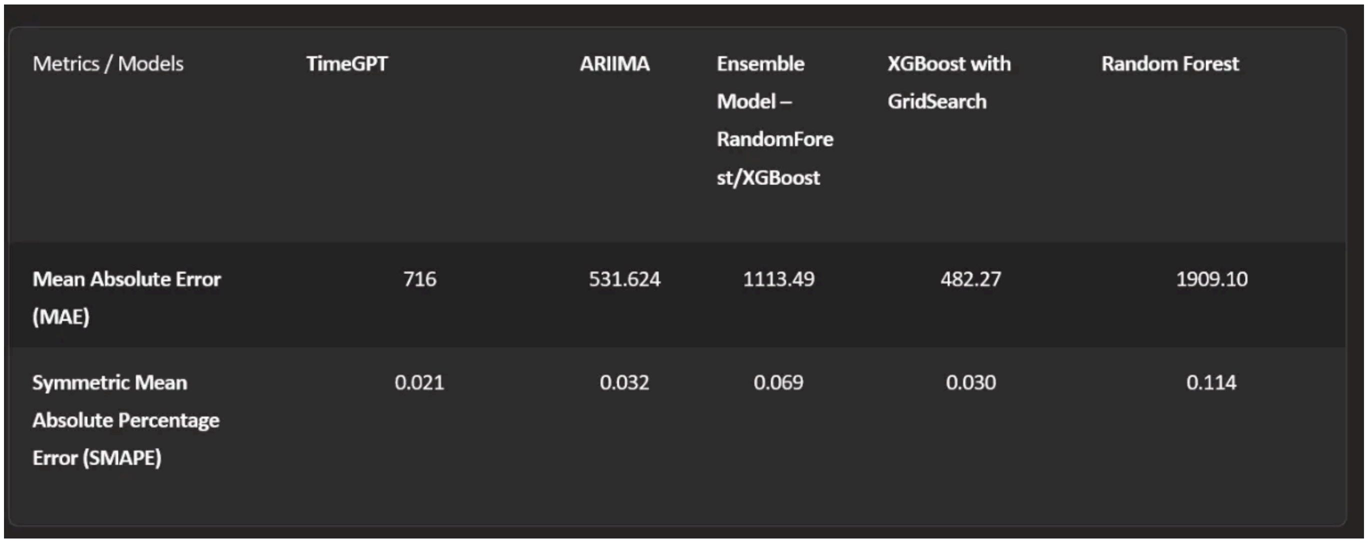
Performance of ensemble model on Random Forest and XGBoost is MAE — 1113.49 and SMAPE — 0.069

ARIMA

```
df_ariima['Date'] = pd.to_datetime(df_ariima['Date'], format='%y/%m/%d')
# sales data
df_ariima = df_ariima.groupby(['Date'])['Weekly_Sales'].mean().reset_index()

import warnings
warnings.filterwarnings('ignore')
import pandas as pd
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_absolute_error
from sklearn.model_selection import GridSearchCV
import numpy as np

# Select the time series for Weekly_Sales
weekly_sales = df_ariima

# Split data into train and test sets (80% train, 20% test)
train_size = int(len(weekly_sales) * 0.8)
train, test = weekly_sales.iloc[:train_size], weekly_sales.iloc[train_size:]



p = list(range(1,24))
d = list(range(0,3))
q = list(range(1,3))

# hyperparameters
def evaluate_models(dataset, p_values, d_values, q_values):
    dataset = dataset.astype('float32')
    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:
            model = ARIMA(train, order=(p,d,q))
            model_fit = model.fit()
            forecast = model_fit.forecast(steps=len(test))
            mae = mean_absolute_error(test, forecast)
            if mae < best_score:
                best_score, best_cfg = mae, order
                print('ARIMA%s MAE=%.3f' % (order,mae))
            except:
              continue
      print('Best ARIMA%s MAE=%.3f' % (best_cfg, best_score))
      return best_cfg


best_param = evaluate_models(train, p, d, q)
print(best_param)
# Fit ARIMA model - adjust (p, d, q) as needed
model = ARIMA(train['Weekly_Sales'], order=best_param)
fitted_model = model.fit()

# Forecast for the test period
forecast = fitted_model.forecast(steps=len(test['Weekly_Sales']))

# Calculate MAE
mae = mean_absolute_error(test['Weekly_Sales'], forecast)

# Calculate SMAPE
def smape(y_true, y_pred):
    return  np.mean(2 * np.abs(y_true - y_pred) / (np.abs(y_true) + np.abs(y_pre

smape_value = smape(test['Weekly_Sales'], forecast)

# Output results
print(f"MAE: {mae}")
print(f"SMAPE: {smape_value}")
```

Performance of ARIMA model, MAE — 531.624 and SMAPE — 0.032

Contingency Table



Metrics / Models	TimeGPT	ARIIMA	Ensemble Model – RandomForest/XGBoost	XGBoost with GridSearch	Random Forest
Mean Absolute Error (MAE)	716	531.624	1113.49	482.27	1909.10
Symmetric Mean Absolute Percentage Error (SMAPE)	0.021	0.032	0.069	0.030	0.114

From the above table, We can see TimeGPT is performing well among over models, with best SMAPE scores and giving promising results by graphs.

Conclusion:

We can conclude with specific takeaways:

- Sales forecasting can prevent overstocking or stockouts.
- Economic indicators like unemployment and CPI have tangible impacts.
- Metrics like Sales Velocity are actionable tools for warehouse management.

Timeseries Timegpt Random Forest Sales Forecasting Arima

M

Written by Mamatha Singh

0 Followers · 1 Following

Edit profile