Inventory Management Dataset

According to the UN Environment Programme (UNEP), we waste one one billion tons of food annually. Food waste from restaurants accounts for about 15% of all the food that ends up in landfills. This is an issue that has significant economic, human, and planetary consequences. One of the main reasons why restaurants produce so much food waste is because of improper inventory management, which leads to excess inventory that gets thrown away.

Using an ML model to analyze the history of past sales to predict the number of sales of each menu item can help restaurants optimize their inventory. With a better approximation of expected sales, restaurants can avoid excess inventory and food waste.

Dataset

Link - https://data.mendeley.com/datasets/7s3ys9ntnz/1

The dataset that we are analyzing contains over two years of daily sales from a restaurant in Stuttgart, Germany. Number of servings sold for 7 products (CALAMARI, FISCH, GARNELEN, HAEHNCHEN, KOEFTE, LAMM, STEAK). Features include calendar informations such as weekday or special events, information derived from historical sales as well as weather from the local weather station.

ML Algorithms

1. Poisson Regression
    - Our ML output will be a count (number of sales for each menu item), so Poisson Regression will be a good fit in this case.
2. Tree Based Model
    - We want to capture complex non linear relationships between historical sales, weather, and menu sales. Our dataset contains many dimensions, so a tree based model would be a good fit to capture these complex interactions.
3. Time series
    - The data is contained as a sequence that was collected within the span of 2 years. This would be an ideal model for our dataset because we are predicting future sales based on past observations of sales, weather patterns, and seasonal patterns

Data Preparation

In [495…
```
import pandas as pd
import seaborn as sns
import numpy as np
import warnings
import matplotlib.pyplot as plt
from sklearn.feature_selection import VarianceThreshold
from sklearn.linear_model import PoissonRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, mean_absolute_error
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint
from sklearn.preprocessing import StandardScaler
from xgboost import XGBRegressor
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.statespace.sarimax import SARIMAX
from sklearn.feature_selection import RFE
from sklearn.model_selection import TimeSeriesSplit
from statsmodels.tsa.statespace.varmax import VARMAX
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
warnings.filterwarnings('ignore')
```

In [496…
```
pd.set_option('display.max_columns', None)

df = pd.read_csv('data_preprocessed.csv', delimiter=';', encoding='utf-8')
df.head()
```

Out[496]:

	Unnamed: 0	DEMAND_DATE	Montag	Dienstag	Mittwoch	Donnerstag	Freitag	Samstag	Sonntag	MONTH_JAN	MONTH_FEB	M
0	1	10.4.2013	0	0	0	0	1	0	0	0	0	
1	2	10.5.2013	0	0	0	0	0	1	0	0	0	
2	3	10.6.2013	0	0	0	0	0	0	1	0	0	
3	4	10.7.2013	1	0	0	0	0	0	0	0	0	
4	5	10.8.2013	0	1	0	0	0	0	0	0	0	

In [497…
```
df.shape
```

Out[497]:
```
(760, 293)
```

In [498…
```
df.describe()
```

Out[498]:

	Unnamed: 0	Montag	Dienstag	Mittwoch	Donnerstag	Freitag	Samstag	Sonntag	MONTH_JAN	MONTH_
count	760.000000	760.000000	760.000000	760.000000	760.000000	760.000000	760.000000	760.000000	760.000000	760.000
mean	380.500000	0.143421	0.143421	0.139474	0.142105	0.143421	0.144737	0.143421	0.081579	0.073
std	219.537392	0.350733	0.350733	0.346668	0.349388	0.350733	0.352067	0.350733	0.273902	0.261
min	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000
25%	190.750000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000
50%	380.500000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000
75%	570.250000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000
max	760.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000

In [499…
```
missing_values = df.isnull().sum()
print(missing_values[missing_values > 0])
```

```
Series([], dtype: int64)
```

In [500…
```
duplicate_rows = df[df.duplicated()]
print(f"Number of duplicate rows: {duplicate_rows.shape[0]}")
```

```
Number of duplicate rows: 0
```

In [501…
```
df = df.drop(columns=['Unnamed: 0', 'TOTAL_FISCHPROD', 'TOTAL_FLEISCH'])

df_time_series = df[['DEMAND_DATE', 'TOTAL']]
df = df.drop(columns=['DEMAND_DATE', 'TOTAL'])

df.head()
```

Out[501]:

	Montag	Dienstag	Mittwoch	Donnerstag	Freitag	Samstag	Sonntag	MONTH_JAN	MONTH_FEB	MONTH_MAR	MONTH_APR	M
0	0	0	0	0	1	0	0	0	0	0	0	
1	0	0	0	0	0	1	0	0	0	0	0	
2	0	0	0	0	0	0	1	0	0	0	0	
3	1	0	0	0	0	0	0	0	0	0	0	
4	0	1	0	0	0	0	0	0	0	0	0	

In [489…

```
#Dropping columns with low variance

selector = VarianceThreshold(threshold=0.01)
selector.fit(df)

low_variance_columns = df.columns[~selector.get_support()]
print(f"Low variance columns: {low_variance_columns}")

df = df.drop(columns=low_variance_columns)
```

```
Low variance columns: Index(['ISOUTLIER_HIGH', 'ISOUTLIER_LOW'], dtype='object')
```

In [490…

```
# Drop highly correlated columns
correlation_matrix = df.corr().abs()
upper_tri = correlation_matrix.where(
    np.triu(np.ones(correlation_matrix.shape), k=1).astype(np.bool)
)

to_drop = [column for column in upper_tri.columns if any(upper_tri[column] > 0.8)]
df = df.drop(columns=to_drop)
print(f"Dropped columns due to high correlation: {to_drop}")
```

```
Dropped columns due to high correlation: ['TOTAL_FISCHPROD_DEMAND_T1', 'TOTAL_FISCHPROD_DEMAND_T2', 'TOTAL_FI
SCHPROD_DEMAND_T3', 'TOTAL_FISCHPROD_DEMAND_T4', 'TOTAL_FISCHPROD_DEMAND_T5', 'TOTAL_FISCHPROD_DEMAND_T6', 'T
OTAL_FISCHPROD_DEMAND_T7', 'TOTAL_FLEISCH_DEMAND_T1', 'TOTAL_FLEISCH_DEMAND_T2', 'TOTAL_FLEISCH_DEMAND_T3',
'TOTAL_FLEISCH_DEMAND_T4', 'TOTAL_FLEISCH_DEMAND_T5', 'TOTAL_FLEISCH_DEMAND_T6', 'TOTAL_FLEISCH_DEMAND_T7',
'TOTAL_DEMAND_T1', 'TOTAL_DEMAND_T2', 'TOTAL_DEMAND_T3', 'TOTAL_DEMAND_T4', 'TOTAL_DEMAND_T5', 'TOTAL_DEMAND_
T6', 'TOTAL_DEMAND_T7', 'CALAMARI_CUM_DEMAND_T3', 'CALAMARI_CUM_DEMAND_T4', 'CALAMARI_CUM_DEMAND_T5', 'CALAMA
RI_CUM_DEMAND_T6', 'CALAMARI_CUM_DEMAND_T7', 'FISCH_CUM_DEMAND_T3', 'FISCH_CUM_DEMAND_T4', 'FISCH_CUM_DEMAND_
T5', 'FISCH_CUM_DEMAND_T6', 'FISCH_CUM_DEMAND_T7', 'GARNELEN_CUM_DEMAND_T3', 'GARNELEN_CUM_DEMAND_T4', 'GARNE
LEN_CUM_DEMAND_T5', 'GARNELEN_CUM_DEMAND_T6', 'GARNELEN_CUM_DEMAND_T7', 'HAEHNCHEN_CUM_DEMAND_T3', 'HAEHNCHEN
_CUM_DEMAND_T4', 'HAEHNCHEN_CUM_DEMAND_T5', 'HAEHNCHEN_CUM_DEMAND_T6', 'HAEHNCHEN_CUM_DEMAND_T7', 'KOEFTE_CUM
_DEMAND_T3', 'KOEFTE_CUM_DEMAND_T4', 'KOEFTE_CUM_DEMAND_T5', 'KOEFTE_CUM_DEMAND_T6', 'KOEFTE_CUM_DEMAND_T7',
'LAMM_CUM_DEMAND_T3', 'LAMM_CUM_DEMAND_T4', 'LAMM_CUM_DEMAND_T5', 'LAMM_CUM_DEMAND_T6', 'LAMM_CUM_DEMAND_T7',
'STEAK_CUM_DEMAND_T3', 'STEAK_CUM_DEMAND_T4', 'STEAK_CUM_DEMAND_T5', 'STEAK_CUM_DEMAND_T6', 'STEAK_CUM_DEMAND
_T7', 'TOTAL_FISCHPROD_CUM_DEMAND_T2', 'TOTAL_FISCHPROD_CUM_DEMAND_T3', 'TOTAL_FISCHPROD_CUM_DEMAND_T4', 'TOT
AL_FISCHPROD_CUM_DEMAND_T5', 'TOTAL_FISCHPROD_CUM_DEMAND_T6', 'TOTAL_FISCHPROD_CUM_DEMAND_T7', 'TOTAL_FLEISCH
_CUM_DEMAND_T2', 'TOTAL_FLEISCH_CUM_DEMAND_T3', 'TOTAL_FLEISCH_CUM_DEMAND_T4', 'TOTAL_FLEISCH_CUM_DEMAND_T5',
'TOTAL_FLEISCH_CUM_DEMAND_T6', 'TOTAL_FLEISCH_CUM_DEMAND_T7', 'TOTAL_CUM_DEMAND_T2', 'TOTAL_CUM_DEMAND_T3',
'TOTAL_CUM_DEMAND_T4', 'TOTAL_CUM_DEMAND_T5', 'TOTAL_CUM_DEMAND_T6', 'TOTAL_CUM_DEMAND_T7', 'TOTAL_HML_DEMAND
_T7', 'TOTAL_NO_DAYS_ABOVE_7D_MEAN', 'GARNELEN_NO_DAYS_BELOW_7D_MEAN', 'HAEHNCHEN_NO_DAYS_BELOW_7D_MEAN', 'KO
EFTE_NO_DAYS_BELOW_7D_MEAN', 'LAMM_NO_DAYS_BELOW_7D_MEAN', 'STEAK_NO_DAYS_BELOW_7D_MEAN', 'TOTAL_FISCHPROD_NO
_DAYS_BELOW_7D_MEAN', 'TOTAL_FLEISCH_NO_DAYS_BELOW_7D_MEAN', 'TOTAL_NO_DAYS_BELOW_7D_MEAN', 'CALAMARI_MEAN_SA
ME_WDAY_DEMANDS_W3', 'CALAMARI_MEAN_SAME_WDAY_DEMANDS_W4', 'FISCH_MEAN_SAME_WDAY_DEMANDS_W3', 'FISCH_MEAN_SAM
E_WDAY_DEMANDS_W4', 'GARNELEN_MEAN_SAME_WDAY_DEMANDS_W2', 'GARNELEN_MEAN_SAME_WDAY_DEMANDS_W3', 'GARNELEN_MEA
N_SAME_WDAY_DEMANDS_W4', 'HAEHNCHEN_MEAN_SAME_WDAY_DEMANDS_W2', 'HAEHNCHEN_MEAN_SAME_WDAY_DEMANDS_W3', 'HAEHN
CHEN_MEAN_SAME_WDAY_DEMANDS_W4', 'KOEFTE_MEAN_SAME_WDAY_DEMANDS_W2', 'KOEFTE_MEAN_SAME_WDAY_DEMANDS_W3', 'KOE
FTE_MEAN_SAME_WDAY_DEMANDS_W4', 'LAMM_MEAN_SAME_WDAY_DEMANDS_W2', 'LAMM_MEAN_SAME_WDAY_DEMANDS_W3', 'LAMM_MEA
N_SAME_WDAY_DEMANDS_W4', 'STEAK_MEAN_SAME_WDAY_DEMANDS_W2', 'STEAK_MEAN_SAME_WDAY_DEMANDS_W3', 'STEAK_MEAN_SA
ME_WDAY_DEMANDS_W4', 'TOTAL_FISCHPROD_MEAN_SAME_WDAY_DEMANDS_W2', 'TOTAL_FISCHPROD_MEAN_SAME_WDAY_DEMANDS_W
3', 'TOTAL_FISCHPROD_MEAN_SAME_WDAY_DEMANDS_W4', 'TOTAL_FLEISCH_MEAN_SAME_WDAY_DEMANDS_W2', 'TOTAL_FLEISCH_ME
AN_SAME_WDAY_DEMANDS_W3', 'TOTAL_FLEISCH_MEAN_SAME_WDAY_DEMANDS_W4', 'TOTAL_MEAN_SAME_WDAY_DEMANDS_W2', 'TOTA
L_MEAN_SAME_WDAY_DEMANDS_W3', 'TOTAL_MEAN_SAME_WDAY_DEMANDS_W4', 'SONNE', 'SONNE_T1', 'SONNE_T2', 'SONNE_T3',
'LUFTTEMPERATUR_T1', 'LUFTTEMPERATUR_T2', 'LUFTTEMPERATUR_T3', 'WIND_MEAN_OVER_LAST_T2', 'WIND_MEAN_OVER_LAST
_T3', 'WIND_MEAN_OVER_LAST_T4', 'WIND_MEAN_OVER_LAST_T5', 'WIND_MEAN_OVER_LAST_T6', 'WIND_MEAN_OVER_LAST_T7',
'BEWOELKUNG_MEAN_OVER_LAST_T2', 'BEWOELKUNG_MEAN_OVER_LAST_T3', 'BEWOELKUNG_MEAN_OVER_LAST_T4', 'BEWOELKUNG_M
EAN_OVER_LAST_T5', 'BEWOELKUNG_MEAN_OVER_LAST_T6', 'BEWOELKUNG_MEAN_OVER_LAST_T7', 'NIEDERSCHLAG_MEAN_OVER_LA
ST_T3', 'NIEDERSCHLAG_MEAN_OVER_LAST_T4', 'NIEDERSCHLAG_MEAN_OVER_LAST_T5', 'NIEDERSCHLAG_MEAN_OVER_LAST_T6',
'NIEDERSCHLAG_MEAN_OVER_LAST_T7', 'SONNE_MEAN_OVER_LAST_T2', 'SONNE_MEAN_OVER_LAST_T3', 'SONNE_MEAN_OVER_LAST
_T4', 'SONNE_MEAN_OVER_LAST_T5', 'SONNE_MEAN_OVER_LAST_T6', 'SONNE_MEAN_OVER_LAST_T7', 'LUFTTEMPERATUR_MEAN_O
VER_LAST_T2', 'LUFTTEMPERATUR_MEAN_OVER_LAST_T3', 'LUFTTEMPERATUR_MEAN_OVER_LAST_T4', 'LUFTTEMPERATUR_MEAN_OV
ER_LAST_T5', 'LUFTTEMPERATUR_MEAN_OVER_LAST_T6', 'LUFTTEMPERATUR_MEAN_OVER_LAST_T7', 'NIEDERSCHLAG_HML_OVER_T
7', 'WIND_NO_DAYS_BELOW_7D_MEAN', 'BEWOELKUNG_NO_DAYS_BELOW_7D_MEAN', 'NIEDERSCHLAG_NO_DAYS_BELOW_7D_MEAN',
'SONNE_NO_DAYS_BELOW_7D_MEAN', 'LUFTTEMPERATUR_NO_DAYS_BELOW_7D_MEAN']
```

In [491…

```
df.head()
```

Out[491]:

	Montag	Dienstag	Mittwoch	Donnerstag	Freitag	Samstag	Sonntag	MONTH_JAN	MONTH_FEB	MONTH_MAR	MONTH_APR	M
0	0	0	0	0	1	0	0	0	0	0	0	
1	0	0	0	0	0	1	0	0	0	0	0	
2	0	0	0	0	0	0	1	0	0	0	0	
3	1	0	0	0	0	0	0	0	0	0	0	
4	0	1	0	0	0	0	0	0	0	0	0	

Poisson Regression

For the first iteration, we fit a Poisson Regression model to each menu item to predict sales, using the same features for each model.

In [408…

```
from sklearn.model_selection import TimeSeriesSplit
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
from sklearn.linear_model import PoissonRegressor

menu_items = ['CALAMARI', 'FISCH', 'GARNELEN', 'HAEHNCHEN', 'KOEFTE', 'LAMM', 'STEAK']

X = df.drop(columns=menu_items)
features = X
y = df[menu_items]

# Scale features
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Set up TimeSeriesSplit
tscv = TimeSeriesSplit(n_splits=5)

for item in menu_items:
    train_scores = []
    test_scores = []

    for train_index, test_index in tscv.split(X):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y.iloc[train_index][item], y.iloc[test_index][item]
```

```
        model = PoissonRegressor(max_iter=200)
        model.fit(X_train, y_train)

        y_train_pred = model.predict(X_train)
        y_test_pred = model.predict(X_test)

        train_mse = mean_squared_error(y_train, y_train_pred)
        test_mse = mean_squared_error(y_test, y_test_pred)
        test_scores.append(test_mse)
        train_scores.append(train_mse)

    avg_train_mse = sum(train_scores) / len(train_scores)
    avg_test_mse = sum(test_scores) / len(test_scores)


    train_mse = mean_squared_error(y_train, y_train_pred)
    test_mse = mean_squared_error(y_test, y_test_pred)

    print(f'{item} Avg Train MSE: {avg_train_mse:.4f}, Avg Test MSE: {avg_test_mse:.4f}\n')
```

```
CALAMARI Avg Train MSE: 4.3733, Avg Test MSE: 11.8854

FISCH Avg Train MSE: 4.3431, Avg Test MSE: 8.2688

GARNELEN Avg Train MSE: 9.3740, Avg Test MSE: 29.7110

HAEHNCHEN Avg Train MSE: 38.6646, Avg Test MSE: 287.8877

KOEFTE Avg Train MSE: 26.5360, Avg Test MSE: 140.8644

LAMM Avg Train MSE: 45.5634, Avg Test MSE: 217.1821

STEAK Avg Train MSE: 32.8859, Avg Test MSE: 357.1320
```

Results analysis: MSE: We get high values of mean squared error for haehnchen (chicken), lamm (lamb), steak. We also see a large overfitting as the test MSE for all categories are significantly higher than the train MSE. I would like to further look into why these values are high. This could possibly be caused by these items having more variabilities in sales or the features not capturing the nuances of these items as well

The model performs relatively well in predicting the sales of calamari, fisch (fish), and garnelen (shrimp), outputting a lower MSE

Train vs test scores: We can see for some of the menu items, like fisch (fish), garnelen (shrimp), haehnchen(chicken) that they perform better for the training set compared to the test set.

Hyperparameter Tuning

To help the model with overfitting, we can add a regularization method. The PoissonRegressor class allows for L2 regularization. We can also adjust the max_iter value to tune convergence.

GridSearchCV is a cross-validation technique that we will use to find the optimal parameter values in a grid. We will use this library to tune our hyperparameters, finding the optimal values.

In [409…

```
param_grid = {
    'alpha': [0.01, 0.1, 1.0, 10.0, 100.0],
    'max_iter': [1000, 1500, 2000, 2500, 3000],
}

for item in menu_items:
    y_item = y[item]

    model = PoissonRegressor()
    grid_search = GridSearchCV(
        estimator=model,
        param_grid=param_grid,
        cv=tscv,
        scoring='neg_mean_squared_error',
        n_jobs=-1,
    )

    grid_search.fit(X, y_item)

    # Get best parameters and model
    best_params = grid_search.best_params_
    print(f"Best parameters for {item} :{best_params}")
    best_model = grid_search.best_estimator_

    split_index = int(len(df) * 0.8)
    X_train = X[:split_index]
    X_test = X[split_index:]
    y_train = y_item[:split_index]
    y_test = y_item[split_index:]
```

```
    y_train_pred = best_model.predict(X_train)
    y_pred = best_model.predict(X_test)

    train_mse = mean_squared_error(y_train, y_train_pred)
    test_mse = mean_squared_error(y_test, y_pred)

    print(f"Train MSE: {train_mse:.4f}")
    print(f"Test MSE: {test_mse:.4f}\n")
```

```
Best parameters for CALAMARI :{'alpha': 100.0, 'max_iter': 1000}
Train MSE: 8.1955
Test MSE: 4.5404

Best parameters for FISCH :{'alpha': 10.0, 'max_iter': 1000}
Train MSE: 6.3633
Test MSE: 4.2153

Best parameters for GARNELEN :{'alpha': 10.0, 'max_iter': 1000}
Train MSE: 14.8100
Test MSE: 14.3984

Best parameters for HAEHNCHEN :{'alpha': 100.0, 'max_iter': 1000}
Train MSE: 86.5244
Test MSE: 109.7315

Best parameters for KOEFTE :{'alpha': 100.0, 'max_iter': 1000}
Train MSE: 57.1914
Test MSE: 68.5894

Best parameters for LAMM :{'alpha': 100.0, 'max_iter': 1000}
Train MSE: 102.5174
Test MSE: 91.9662

Best parameters for STEAK :{'alpha': 100.0, 'max_iter': 1000}
Train MSE: 67.4207
Test MSE: 56.7634
```

The MSE lowered significantly for all the categories. However, the model still seems to be overfitting in some categories.

Feature Evaluation

We will perform feature selection to try to mitigate the overfitting. We can use the coefficients to analyze which features have the strongest importance in the model's performance, and select the top 10% of them. This will reduce noise and potentially help

with overfitting.

In [52]:
```
coefficients = best_model.coef_
feature_importance = abs(coefficients)

importance_df = pd.DataFrame({
    'Feature': features.columns,
    'Coefficient': coefficients,
    'Importance': feature_importance
}).sort_values(by='Importance', ascending=False)

top_features = importance_df.head(int(273* 0.10))['Feature'].tolist()
print(top_features)
```

```
['Samstag', 'STEAK_DEMAND_T7', 'HAEHNCHEN_DEMAND_T7', 'Sonntag', 'CALAMARI_MEAN_SAME_WDAY_DEMANDS_W2', 'KOEFTE_DEMAND_T7', 'LAMM_DEMAND_T7', 'YEAR_2013', 'CALAMARI_DEMAND_T7', 'WEEKEND', 'FISCH_MEAN_SAME_WDAY_DEMANDS_W2', 'GARNELEN_DEMAND_T7', 'FISCH_DEMAND_T7', 'Montag', 'LUFTTEMPERATUR', 'MONTH_SEP', 'LAMM_DEMAND_T5', 'MONTH_NOV', 'Dienstag', 'ISHOLIDAY', 'YEAR_2015', 'LAMM_DEMAND_T2', 'STEAK_DEMAND_T1', 'SONNE_HML_OVER_T7', 'LAMM_DEMAND_T3', 'STEAK_DEMAND_T6', 'CALAMARI_DEMAND_T1']
```

In [53]:
```
top_feature_indices = [features.columns.get_loc(f) for f in top_features]
X_top_features = X[:, top_feature_indices]

for item in menu_items:
    y_item = y[item]

    model = PoissonRegressor()
    grid_search = GridSearchCV(
        estimator=model,
        param_grid=param_grid,
        cv=tscv,  # Use TimeSeriesSplit for cross-validation
        scoring='neg_mean_squared_error',  # Metric to evaluate the model
        n_jobs=-1  # Use all available CPUs
    )

    grid_search.fit(X_top_features, y_item)

    # Get best parameters and model
    best_params = grid_search.best_params_
    print(f"Best parameters for {item} with top features: {best_params}")
    best_model = grid_search.best_estimator_

    # Evaluate the best model
    split_index = int(len(df) * 0.8)
    X_train = X_top_features[:split_index]
```

```
    X_test = X_top_features[split_index:]
    y_train = y_item[:split_index]
    y_test = y_item[split_index:]

    y_train_pred = best_model.predict(X_train)
    y_pred = best_model.predict(X_test)

    train_mse = mean_squared_error(y_train, y_train_pred)
    test_mse = mean_squared_error(y_test, y_pred)

    print(f"Train MSE with top features: {train_mse:.4f}")
    print(f"Test MSE with top features: {test_mse:.4f}\n")
```

```
Best parameters for CALAMARI with top features: {'alpha': 10.0, 'max_iter': 1000}
Train MSE with top features: 7.2896
Test MSE with top features: 4.1257

Best parameters for FISCH with top features: {'alpha': 1.0, 'max_iter': 1000}
Train MSE with top features: 6.3992
Test MSE with top features: 4.3659

Best parameters for GARNELEN with top features: {'alpha': 10.0, 'max_iter': 1000}
Train MSE with top features: 16.6125
Test MSE with top features: 15.9592

Best parameters for HAEHNCHEN with top features: {'alpha': 10.0, 'max_iter': 1000}
Train MSE with top features: 70.6049
Test MSE with top features: 105.0817

Best parameters for KOEFTE with top features: {'alpha': 10.0, 'max_iter': 1000}
Train MSE with top features: 49.2375
Test MSE with top features: 68.9013

Best parameters for LAMM with top features: {'alpha': 10.0, 'max_iter': 1000}
Train MSE with top features: 87.6822
Test MSE with top features: 86.8723

Best parameters for STEAK with top features: {'alpha': 10.0, 'max_iter': 1000}
Train MSE with top features: 55.7043
Test MSE with top features: 50.4318
```

Diminishing the number of features to the top 10% most important features seemed to help a lot with overfitting. The MSE also went down a bit from the last model that we predicted from.

Random Forest

In [54]:
```
# Define the parameter grid for hyperparameter tuning
params = {
    'n_estimators': [50, 100, 150],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['auto', 'sqrt', 'log2']
}

tscv = TimeSeriesSplit(n_splits=5)

for item in menu_items:
    y = df[item]

    mse_scores_train = []
    mse_scores_test = []

    all_y_train_pred = []
    all_y_test_pred = []
    all_y_train_true = []
    all_y_test_true = []

    for train_index, test_index in tscv.split(X_top_features):
        X_train, X_test = X_top_features[train_index], X_top_features[test_index]
        y_train, y_test = y.iloc[train_index], y.iloc[test_index]

        model = RandomForestRegressor(random_state=42)
        grid_search = GridSearchCV(
            estimator=model,
            param_grid=params,
            cv=3,
            scoring='neg_mean_squared_error',
            n_jobs=-1
        )
        grid_search.fit(X_train, y_train)

        best_model = grid_search.best_estimator_

        y_train_pred = best_model.predict(X_train)
        y_test_pred = best_model.predict(X_test)
```

```
        mse_train = mean_squared_error(y_train, y_train_pred)
        mse_test = mean_squared_error(y_test, y_test_pred)

        mse_scores_train.append(mse_train)
        mse_scores_test.append(mse_test)

        all_y_train_pred.extend(y_train_pred)
        all_y_test_pred.extend(y_test_pred)
        all_y_train_true.extend(y_train)
        all_y_test_true.extend(y_test)

    avg_mse_train = sum(mse_scores_train) / len(mse_scores_train)
    avg_mse_test = sum(mse_scores_test) / len(mse_scores_test)

    print(f"Best parameters for {item} :{grid_search.best_params_}")
    print(f"Average Train MSE: {avg_mse_train:.4f}")
    print(f"Average Test MSE: {avg_mse_test:.4f}\n")
```

```
Best parameters for CALAMARI :{'max_depth': None, 'max_features': 'log2', 'min_samples_leaf': 4, 'min_samples
_split': 2, 'n_estimators': 150}
Average Train MSE: 4.4584
Average Test MSE: 6.0254

Best parameters for FISCH :{'max_depth': 10, 'max_features': 'log2', 'min_samples_leaf': 4, 'min_samples_spli
t': 2, 'n_estimators': 50}
Average Train MSE: 3.0892
Average Test MSE: 6.3536

Best parameters for GARNELEN :{'max_depth': 10, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_s
plit': 10, 'n_estimators': 150}
Average Train MSE: 7.0995
Average Test MSE: 18.5810

Best parameters for HAEHNCHEN :{'max_depth': 20, 'max_features': 'log2', 'min_samples_leaf': 1, 'min_samples_
split': 5, 'n_estimators': 100}
Average Train MSE: 19.8043
Average Test MSE: 94.8375

Best parameters for KOEFTE :{'max_depth': 10, 'max_features': 'log2', 'min_samples_leaf': 1, 'min_samples_spl
it': 10, 'n_estimators': 150}
Average Train MSE: 20.6918
Average Test MSE: 56.1709

Best parameters for LAMM :{'max_depth': None, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_spl
it': 10, 'n_estimators': 100}
Average Train MSE: 26.8727
Average Test MSE: 104.6050

Best parameters for STEAK :{'max_depth': 10, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_spli
t': 5, 'n_estimators': 150}
Average Train MSE: 21.4422
Average Test MSE: 66.1955
```

Random Forest Regression gives us similar results as Poisson Regression, but we can see here that overfitting is a much larger problem. This is a common issue because of Random Forest's flexibility, which allows the model to capture complex patterns but also makes it more susceptible to capturing noise within the data, causing it to overfit to the training data.

XGBoost

In [55]:
```
from sklearn.model_selection import TimeSeriesSplit, GridSearchCV
from sklearn.metrics import mean_squared_error
from xgboost import XGBRegressor

param_grid = {
    'n_estimators': [100, 200],
    'learning_rate': [0.01, 0.1],
    'max_depth': [3, 5],
    'subsample': [0.7, 0.8],
    'colsample_bytree': [0.7, 0.8],
    'min_child_weight': [1, 3],
    'gamma': [0, 0.1],
    'lambda': [0, 1],
    'alpha': [0, 1]
}

for item in menu_items:
    y = df[item]

    tscv = TimeSeriesSplit(n_splits=5)

    model = XGBRegressor(objective='reg:squarederror')
    grid_search = GridSearchCV(
        estimator=model,
        param_grid=param_grid,
        cv=tscv,
        scoring='neg_mean_squared_error',
        n_jobs=-1,
    )

    grid_search.fit(X_top_features, y)

    best_params = grid_search.best_params_
    print(f"Best parameters for {item}: {best_params}")
    best_model = grid_search.best_estimator_

    y_pred = best_model.predict(X_top_features[tscv.get_n_splits():])

    y_train = y[:-len(y_pred)]
    y_test = y[-len(y_pred):]

    train_mse = mean_squared_error(y_train, best_model.predict(X_top_features[:-len(y_pred)]))
    test_mse = mean_squared_error(y_test, y_pred)
```

```
    print(f"Train MSE: {train_mse}")
    print(f"Test MSE: {test_mse}\n")
```

```
Best parameters for CALAMARI: {'alpha': 0, 'colsample_bytree': 0.7, 'gamma': 0, 'lambda': 1, 'learning_rate':
0.01, 'max_depth': 3, 'min_child_weight': 1, 'n_estimators': 200, 'subsample': 0.7}
Train MSE: 2.9590394983714416
Test MSE: 5.164862514571149

Best parameters for FISCH: {'alpha': 1, 'colsample_bytree': 0.8, 'gamma': 0.1, 'lambda': 1, 'learning_rate':
0.01, 'max_depth': 3, 'min_child_weight': 3, 'n_estimators': 200, 'subsample': 0.7}
Train MSE: 0.8395606756522739
Test MSE: 5.288939551645853

Best parameters for GARNELEN: {'alpha': 0, 'colsample_bytree': 0.8, 'gamma': 0, 'lambda': 0, 'learning_rate':
0.01, 'max_depth': 3, 'min_child_weight': 3, 'n_estimators': 100, 'subsample': 0.7}
Train MSE: 24.848164786230154
Test MSE: 16.154454542670873

Best parameters for HAEHNCHEN: {'alpha': 1, 'colsample_bytree': 0.7, 'gamma': 0, 'lambda': 1, 'learning_rat
e': 0.01, 'max_depth': 3, 'min_child_weight': 1, 'n_estimators': 200, 'subsample': 0.8}
Train MSE: 31.219736649170226
Test MSE: 67.02939114746582

Best parameters for KOEFTE: {'alpha': 0, 'colsample_bytree': 0.7, 'gamma': 0.1, 'lambda': 1, 'learning_rate':
0.01, 'max_depth': 5, 'min_child_weight': 1, 'n_estimators': 200, 'subsample': 0.7}
Train MSE: 15.787731704912584
Test MSE: 32.66704601223124

Best parameters for LAMM: {'alpha': 1, 'colsample_bytree': 0.8, 'gamma': 0.1, 'lambda': 0, 'learning_rate':
0.01, 'max_depth': 3, 'min_child_weight': 3, 'n_estimators': 200, 'subsample': 0.8}
Train MSE: 67.75551329617738
Test MSE: 76.06063484390435

Best parameters for STEAK: {'alpha': 1, 'colsample_bytree': 0.8, 'gamma': 0.1, 'lambda': 1, 'learning_rate':
0.01, 'max_depth': 5, 'min_child_weight': 1, 'n_estimators': 200, 'subsample': 0.8}
Train MSE: 12.026724873802596
Test MSE: 30.488669640088954
```

So far, XGBoost has given the best results (lowest Test MSE), though it does overfit on the data.

Time Series Forecasting

We will use the SARIMA (Seasonal autoregressive integrated moving average) model to predict future sales. This model is designed to capture trends and seasonality. Naturally, restaurant sales follow a period of seasonality trends with slow and busy seasons, correlating with the time of year (holidays), and weekdays vs. weekends.

SARIMA(p,d,q)×(P,D,Q)m

Non-Seasonal Component (p, d, q):

p: Autoregressive (AR) part p is the number of lag observations in the model (number of autoregressive terms).

d: Differencing (I) part d is the number of times the raw observations are differenced to make the time series stationary.

q Moving Average (MA) part: The parameter q is the size of the moving average window (number of lagged forecast errors).

Seasonal Component (P, D, Q):

P: Seasonal Autoregressive (SAR) part: The parameter P is the number of seasonal autoregressive terms.

D: Seasonal Differencing (SI) part: The parameter D is the number of seasonal differences.

Q: Seasonal Moving Average (SMA) part: The parameterQ is the number of seasonal moving average terms.

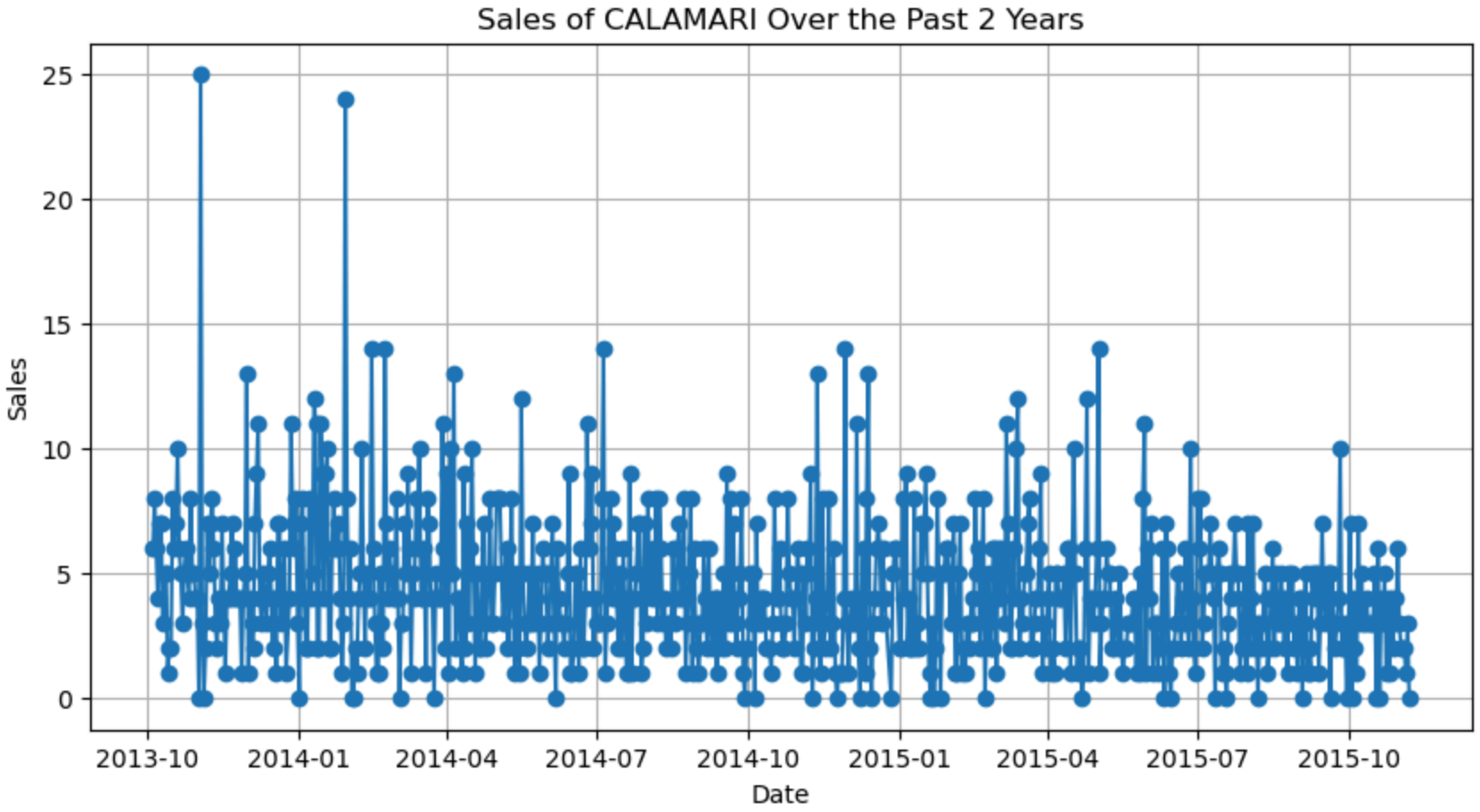
m: Seasonal Period (m): The number of periods per season (e.g., 7 for weekly trends).

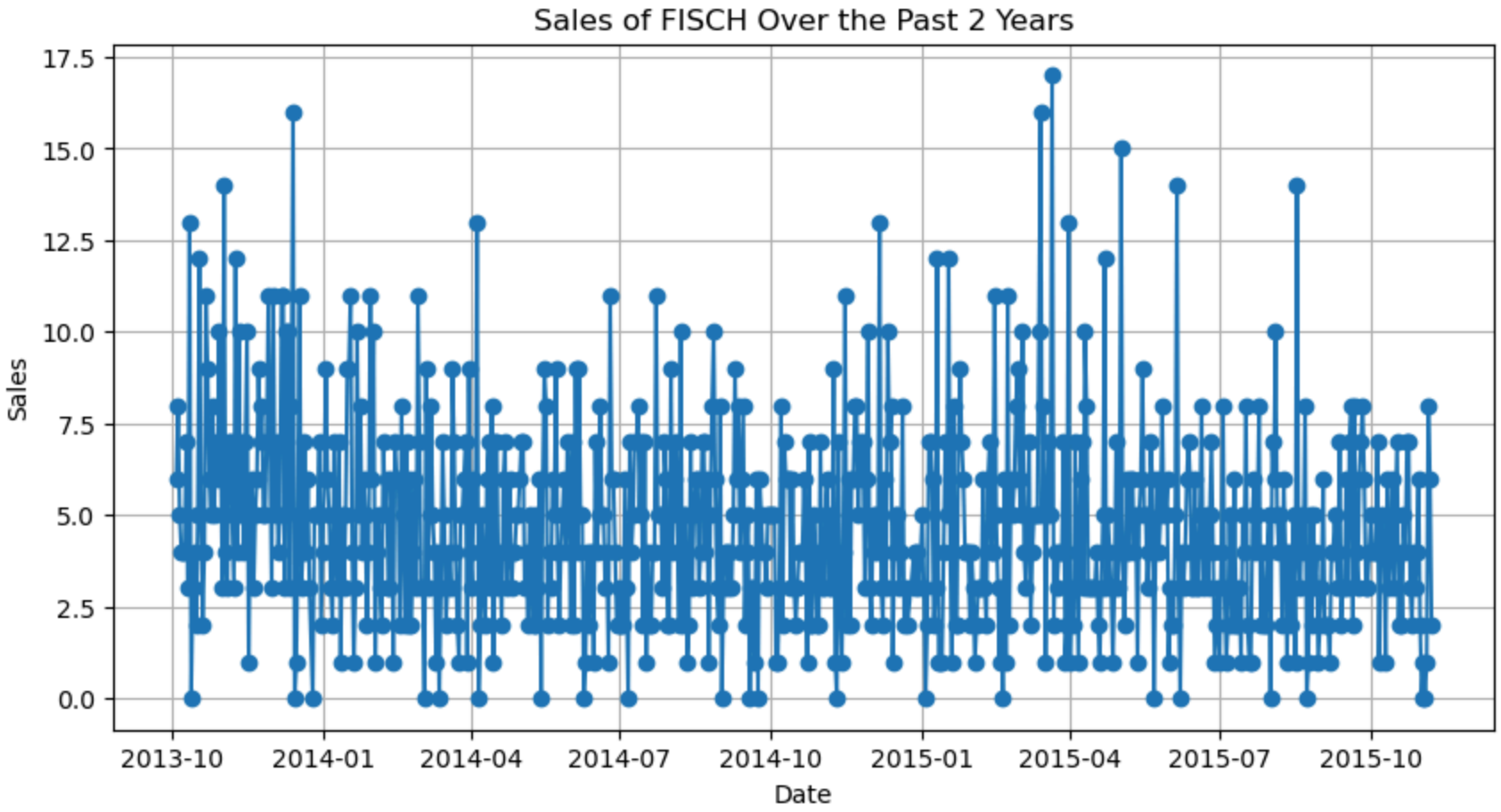
In [504…
```
# Convert DEMAND_DATE into datetime
df_time_series = pd.concat([df_time_series, df], axis=1)
df_time_series['DEMAND_DATE'] = pd.to_datetime(df_time_series['DEMAND_DATE'], format='%m.%d.%Y')
df_time_series = df_time_series.set_index('DEMAND_DATE')
df_time_series.head()

df_exog = df_time_series[['LUFTTEMPERATUR', 'ISHOLIDAY', 'WEEKEND']]
df_time_series = df_time_series[menu_items]
```

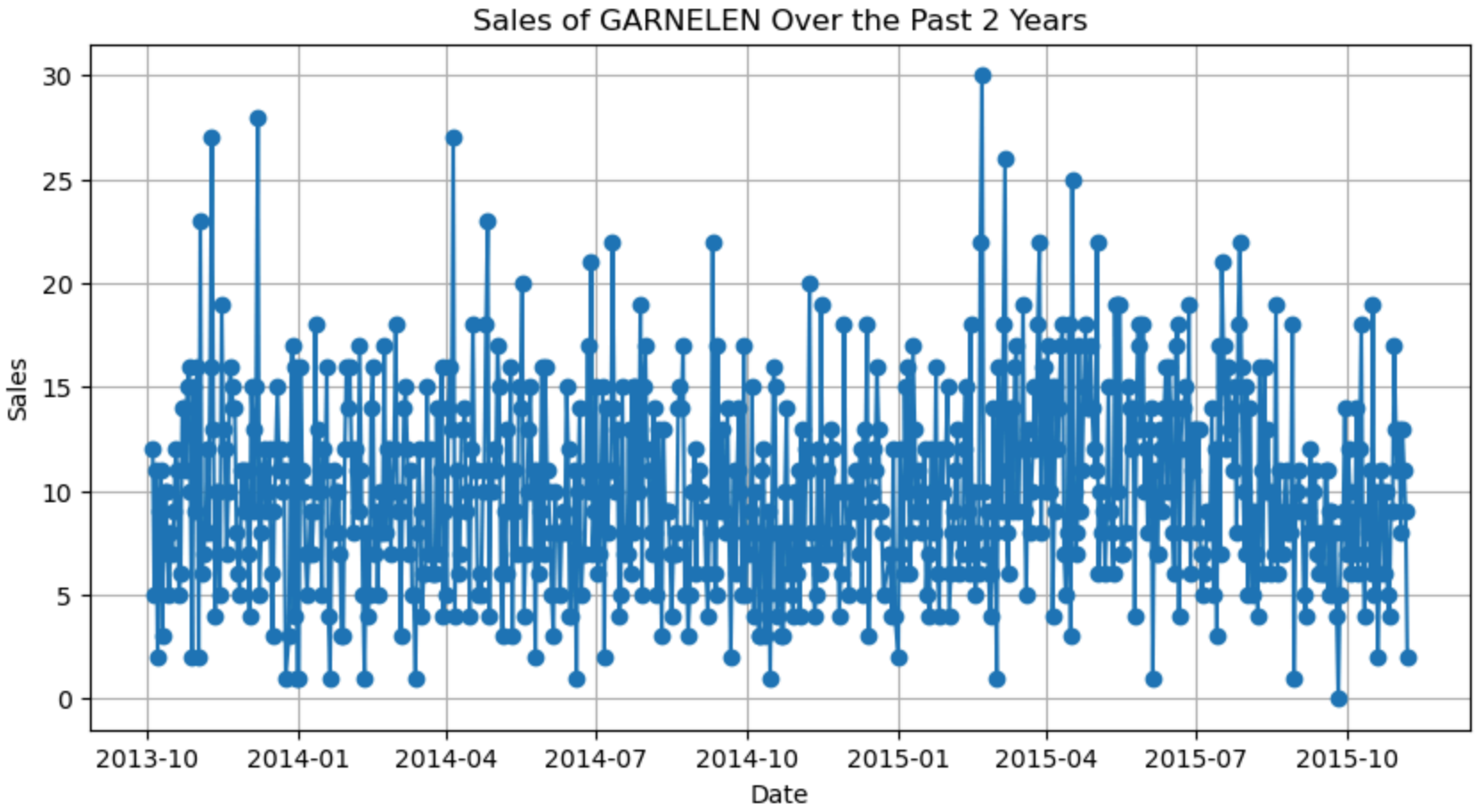
In [505…
```
plt.figure(figsize=(14,5))

for item in menu_items:
    plt.figure(figsize=(10, 5))
    plt.plot(df_time_series.index, df_time_series[item], marker='o')
```

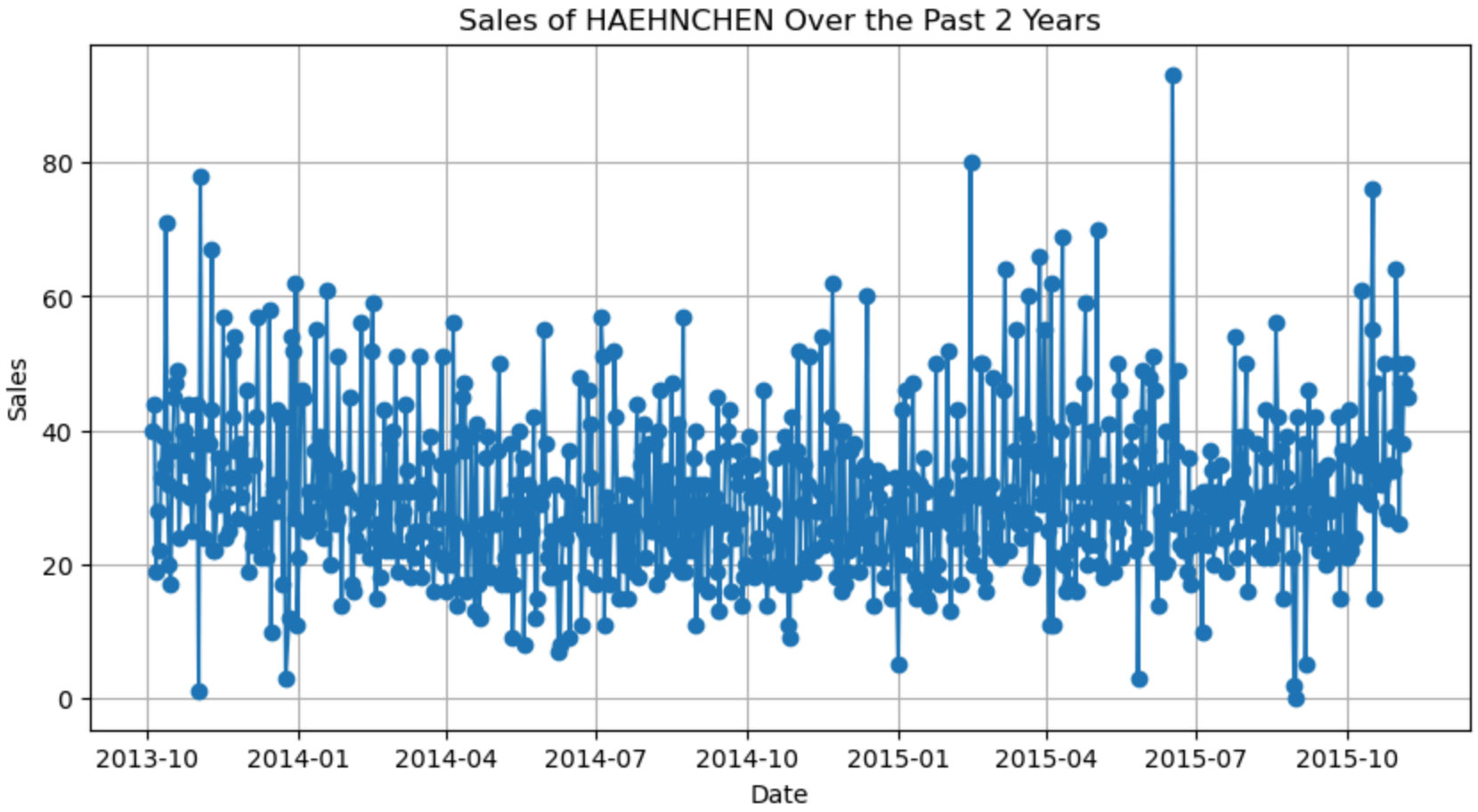
```
    plt.title(f'Sales of {item} Over the Past 2 Years')
    plt.xlabel('Date')
    plt.ylabel('Sales')
    plt.grid(True)
    plt.show()
```

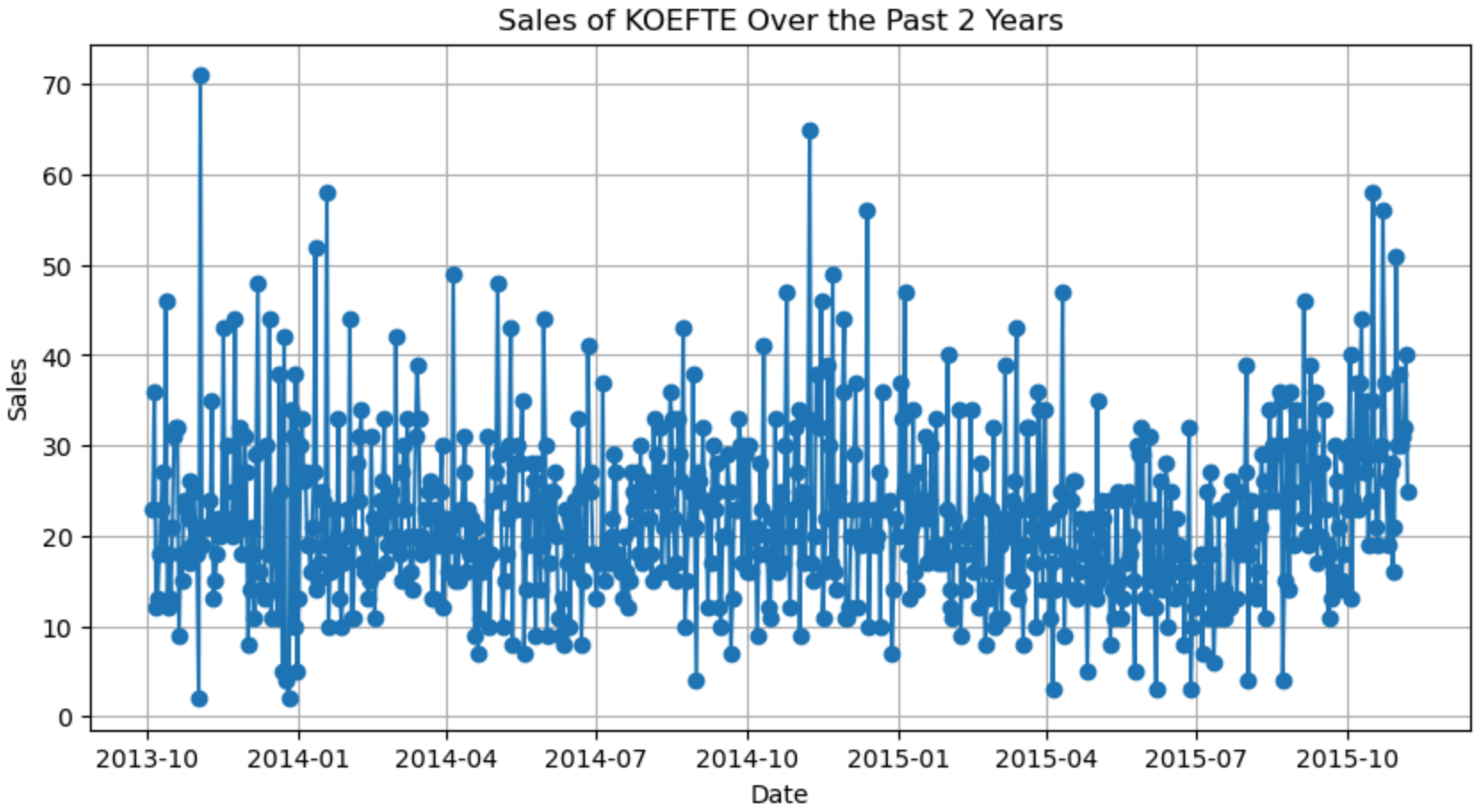
```
<Figure size 1400x500 with 0 Axes>
```

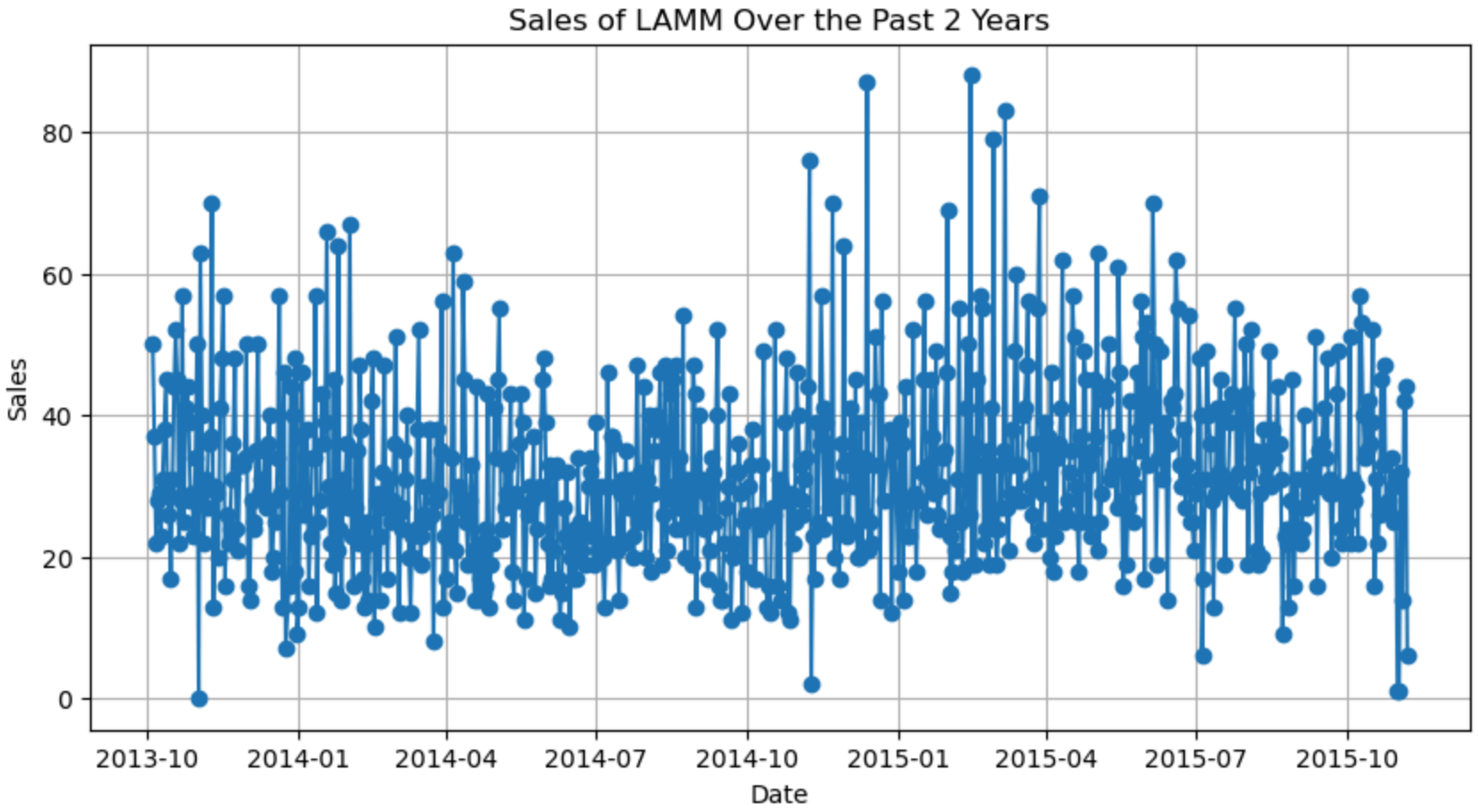


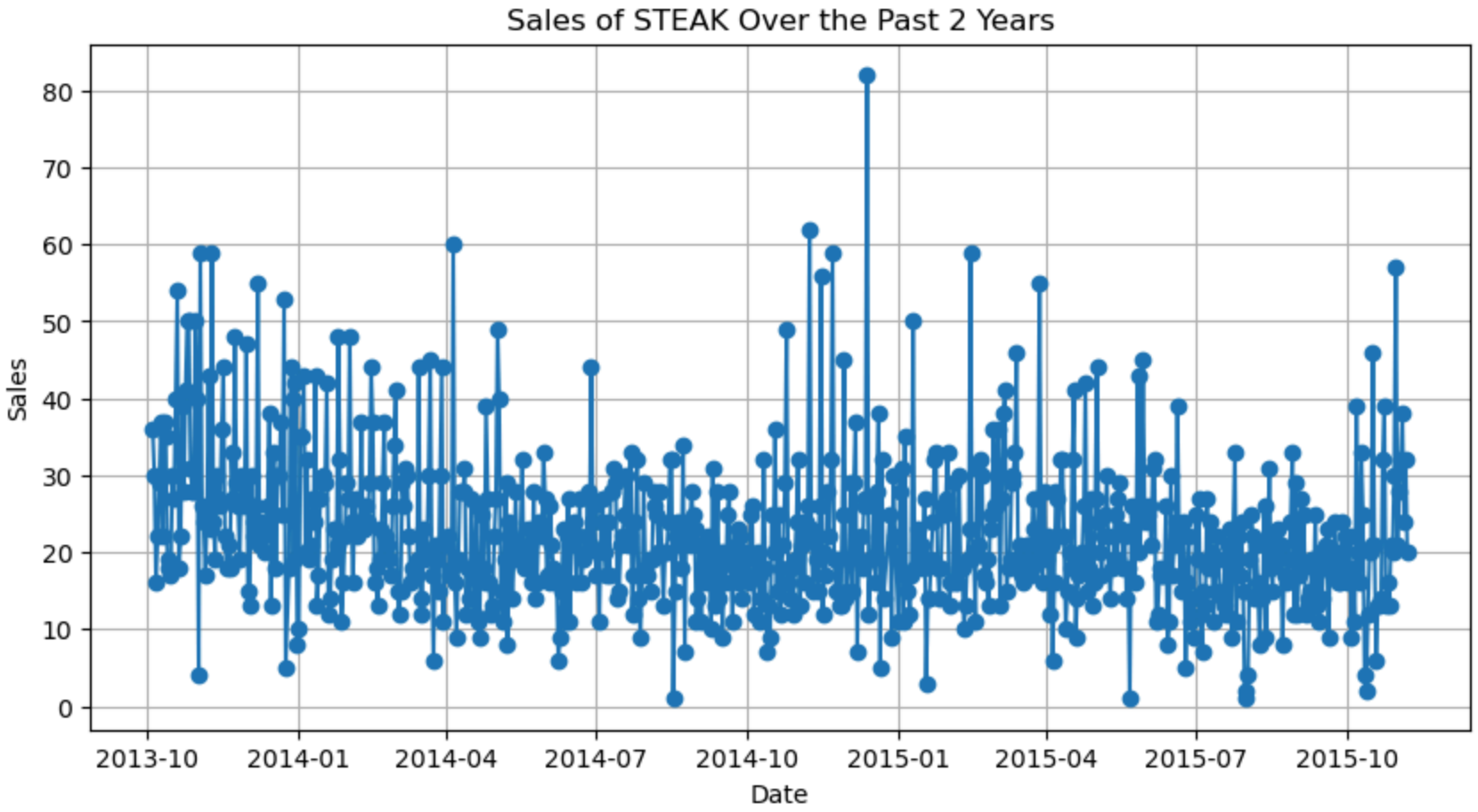

Sales of FISCH Over the Past 2 Years
Sales
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
2013-10
2014-01
2014-04
2014-07
2014-10
2015-01
2015-04
2015-07
2015-10
Date


Sales of GARNELEN Over the Past 2 Years
Sales
Date
0
5
10
15
20
25
30
2013-10
2014-01
2014-04
2014-07
2014-10
2015-01
2015-04
2015-07
2015-10


Sales of HAEHNCHEN Over the Past 2 Years
Sales
0
20
40
60
80
2013-10
2014-01
2014-04
2014-07
2014-10
2015-01
2015-04
2015-07
2015-10
Date


Sales of KOEFTE Over the Past 2 Years
Sales
Date
0
10
20
30
40
50
60
70
2013-10
2014-01
2014-04
2014-07
2014-10
2015-01
2015-04
2015-07
2015-10


Sales of LAMM Over the Past 2 Years
Sales
0
20
40
60
80
2013-10
2014-01
2014-04
2014-07
2014-10
2015-01
2015-04
2015-07
2015-10
Date



SARIMA model requires data to be stationary. We can ues the Augmented Dickey-Fuller (ADF) Test to check if the data is stationary. (stationary if p-value < 0.05). A stationary time series means that the data will have a steady statistical trend across time.

Check Stationarity

In [417…

```
def adf_test(series, signif=0.05):
    result = adfuller(series)
    return result[1] < signif

for item in menu_items:
    print(f"{item} stationary:", adf_test(df_time_series[item]))
```

```
CALAMARI stationary: True
FISCH stationary: True
GARNELEN stationary: True
HAEHNCHEN stationary: False
KOEFTE stationary: False
LAMM stationary: True
STEAK stationary: True
```

Haehnchen and Koefte are not stationary, so we need to difference the data.

In [420…

```
not_stationary = ['HAEHNCHEN', 'KOEFTE']
for item in not_stationary:
    df_time_series[item] = df_time_series[item].diff().dropna()
    print(f"{item} stationary:", adf_test(df_diff[item]))
```

```
HAEHNCHEN stationary: True
KOEFTE stationary: True
```

Now all the menu items are stationary, so we can continue on to fit the model

Finding best paramaters (p, d, q) and (P, D, Q)

To find the optimal values for p and q, we iteratively find the AIC values of a SARIMA model with p and q values within the range of 0 and 2, and pick the values that return the lowest model AIC value. The AIC value is a statistical relative measure of the quality of a model. A lower value indicates a better model fit.

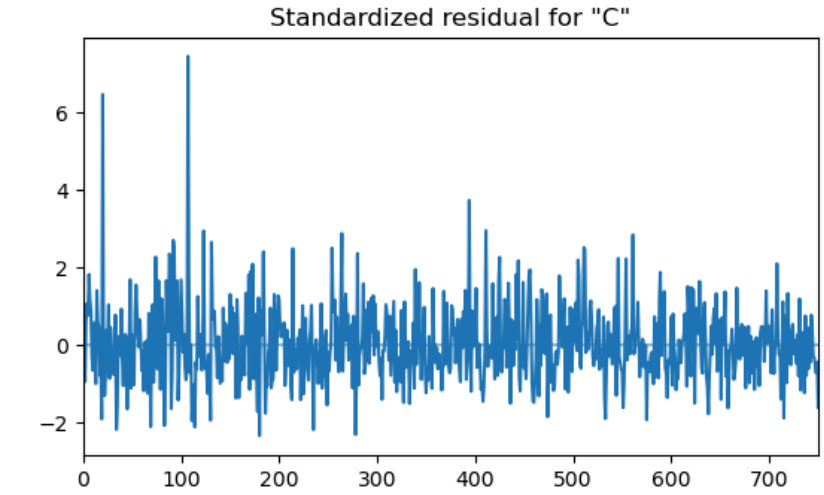
In [423…

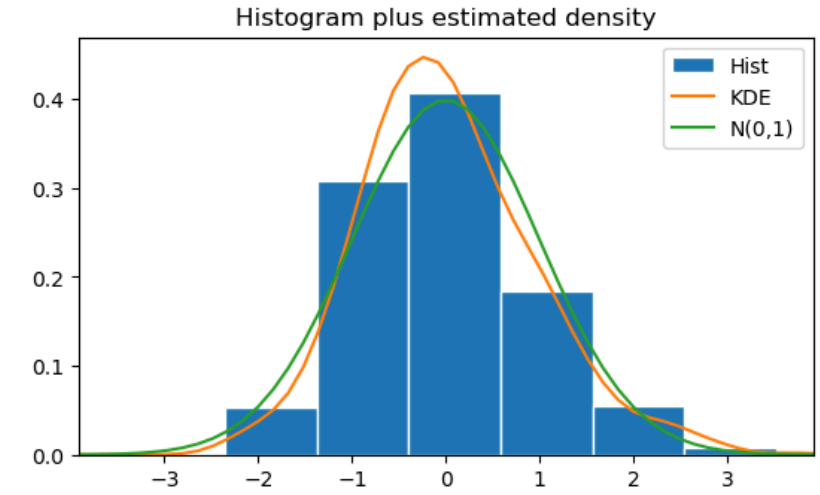
```
import itertools
from statsmodels.tsa.statespace.sarimax import SARIMAX
import matplotlib.pyplot as plt

p_values = range(0, 2)
q_values = range(0, 2)
P_values = range(0, 2)
Q_values = range(0, 2)
seasonal_period = 7  # Weekly seasonality

def find_best_params(plot_diagnostics=True):
    best_params = {}

    for item in menu_items:
        print(f"Evaluating {item}")
        best_aic = float('inf')
```

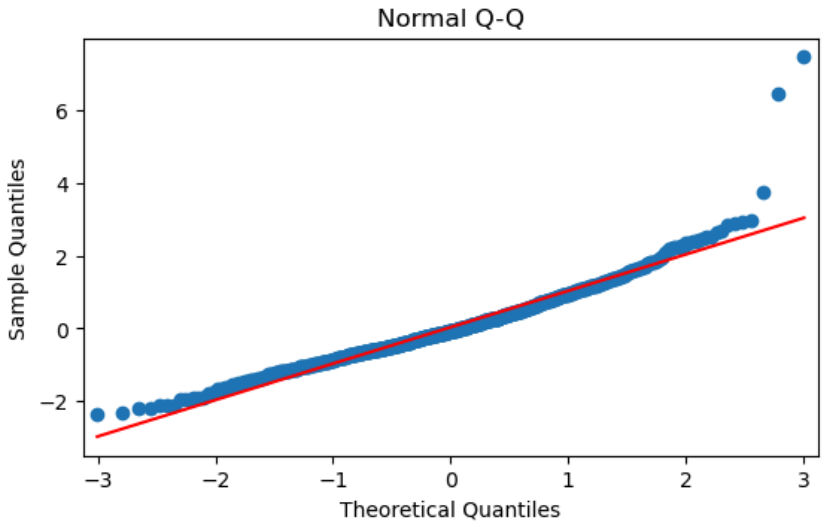
```
        best_p = best_q = best_P = best_Q = None

        d = 1 if item in not_stationary else 0
        D = 1 if item in not_stationary else 0

        for p, q, P, Q in itertools.product(p_values, q_values, P_values, Q_values):
            try:
                model = SARIMAX(
                    df[item],
                    order=(p, d, q),
                    seasonal_order=(P, D, Q, seasonal_period),
                    enforce_stationarity=False,
                    enforce_invertibility=False
                )
                sarima_model = model.fit(disp=False)

                if sarima_model.aic < best_aic:
                    best_aic = sarima_model.aic
                    best_p, best_q, best_P, best_Q = p, q, P, Q

            except Exception as e:
                print(f"Error with p={p}, d={d}, q={q}, P={P}, D={D}, Q={Q}: {e}")
                continue

        best_params[item] = {
            'p': best_p,
            'd': d,
            'q': best_q,
            'P': best_P,
            'D': D,
            'Q': best_Q
        }

        best_model = SARIMAX(
            df[item],
            order=(best_p, d, best_q),
            seasonal_order=(best_P, D, best_Q, seasonal_period),
            enforce_stationarity=False,
            enforce_invertibility=False
        )
        best_sarima_model = best_model.fit(disp=False)
        print(f"Best model for {item}: p={best_p}, d={d}, q={best_q}, P={best_P}, D={D}, Q={best_Q}")

        if plot_diagnostics:
            best_sarima_model.plot_diagnostics(figsize=(14, 8))
```

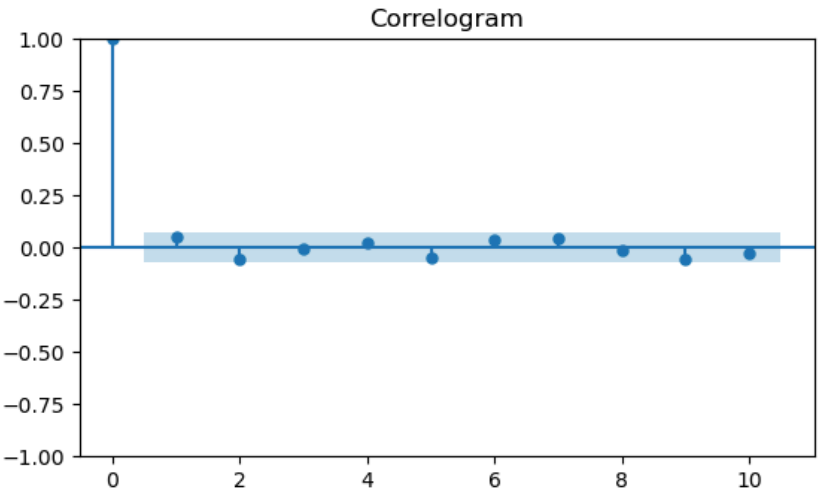
```
            plt.show()

    return best_params

best_params = find_best_params()
print(best_params)
```

```
Evaluating CALAMARI
Best model for CALAMARI: p=1, d=0, q=1, P=1, D=0, Q=1
```

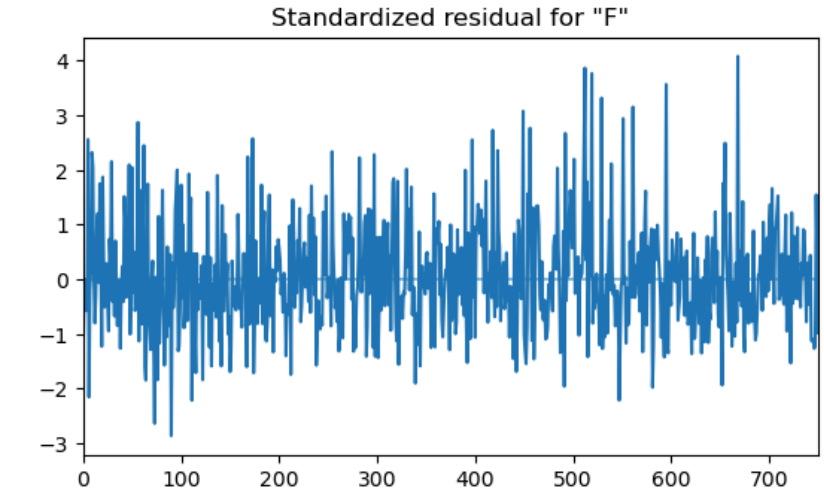


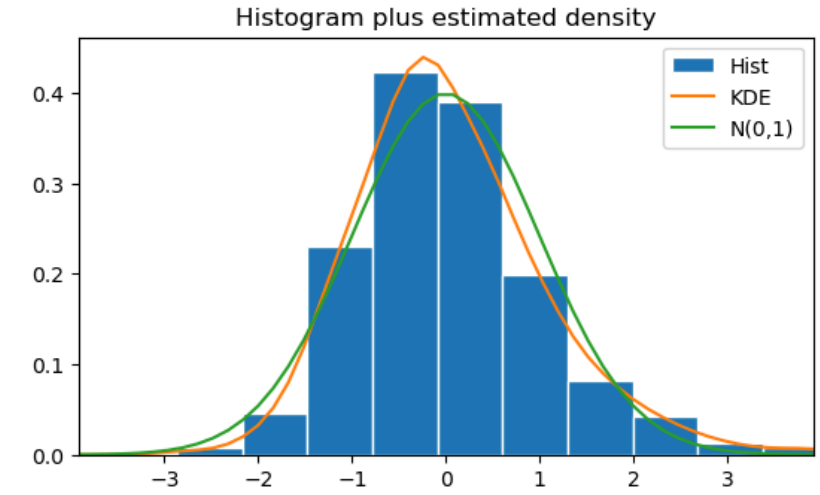


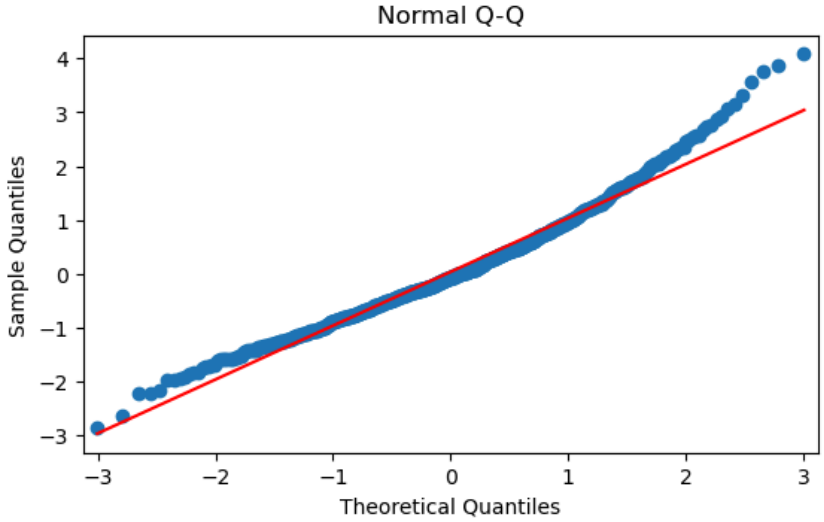


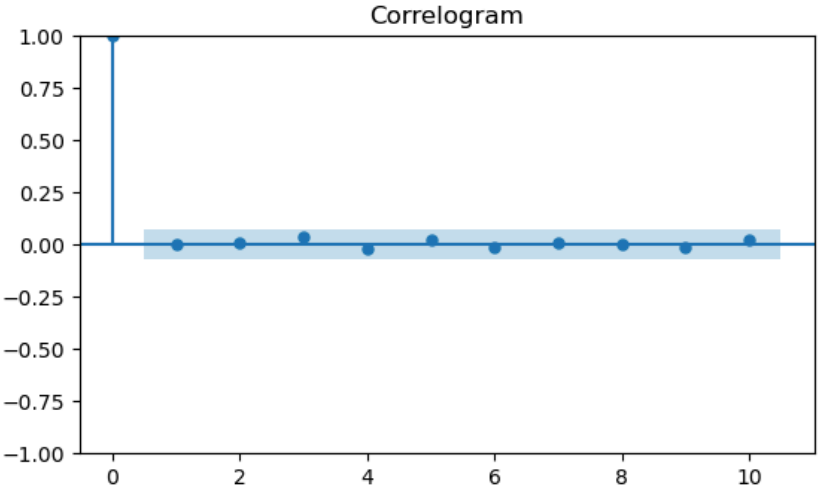



```
Evaluating FISCH
Best model for FISCH: p=0, d=0, q=1, P=1, D=0, Q=1
```

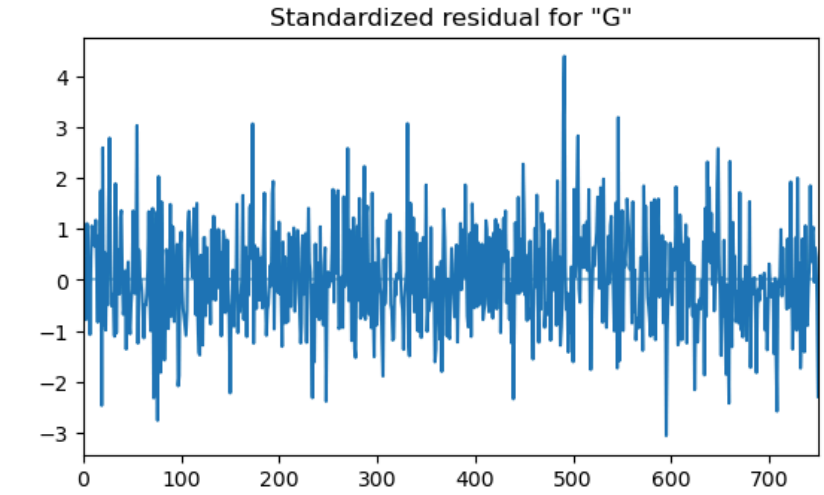


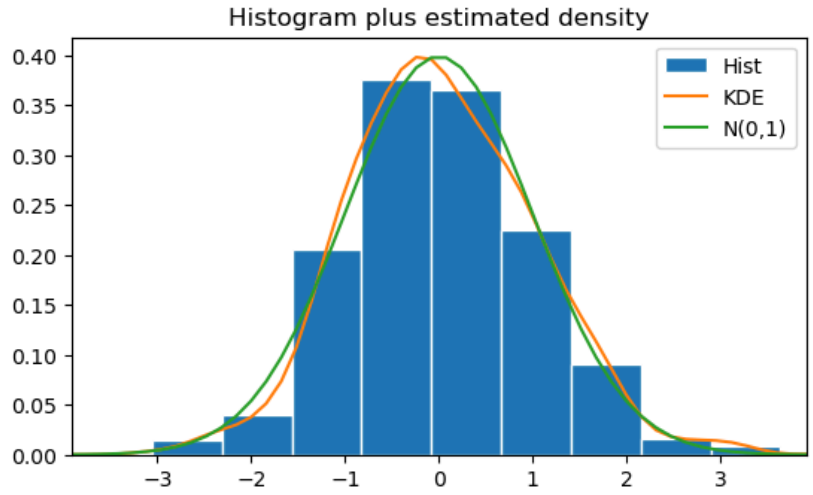


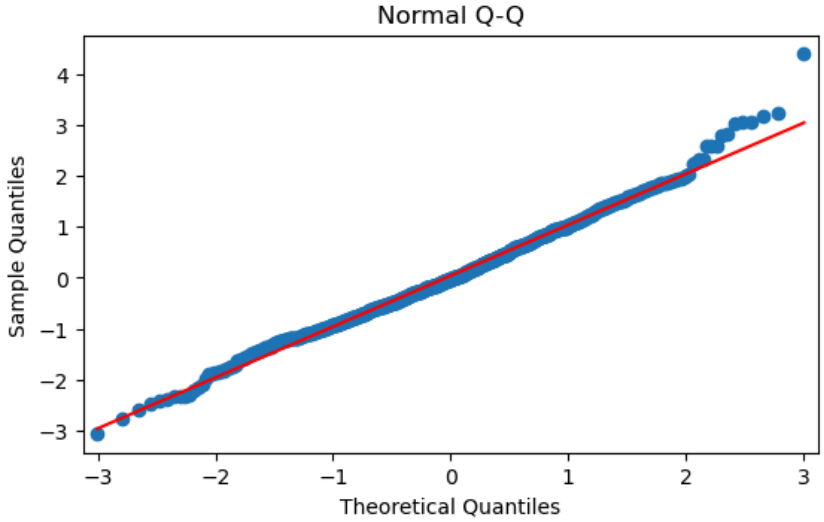


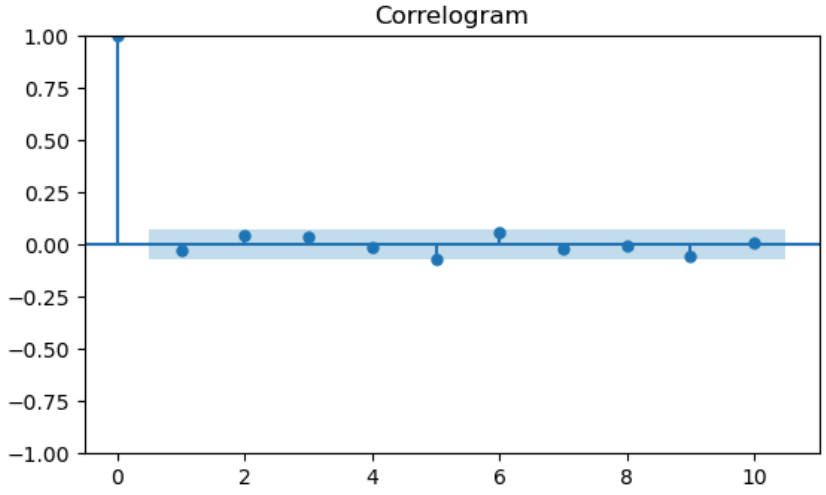



```
Evaluating GARNELEN
Best model for GARNELEN: p=1, d=0, q=1, P=1, D=0, Q=1
```

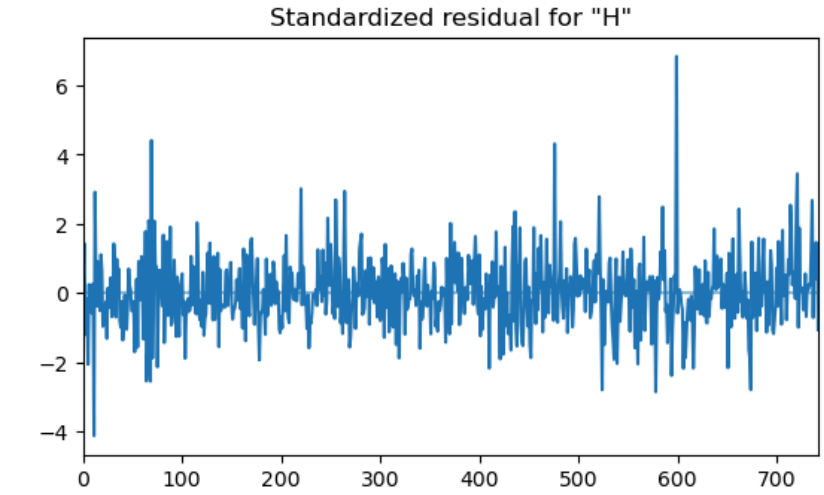


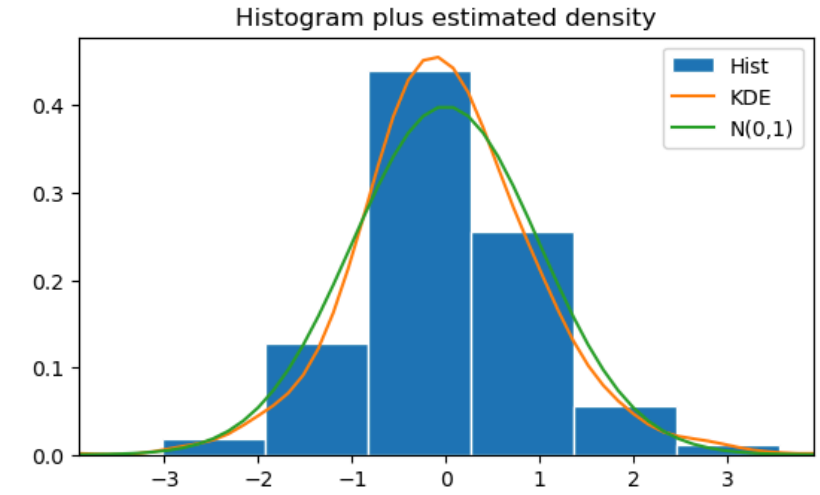


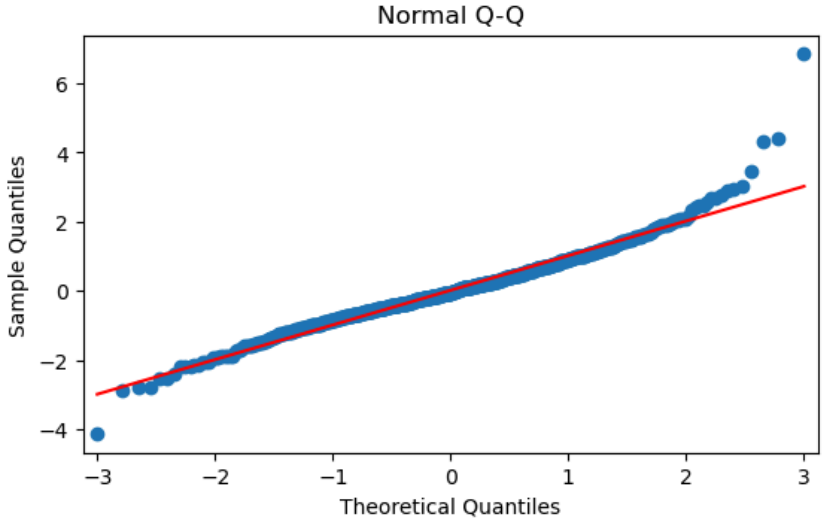


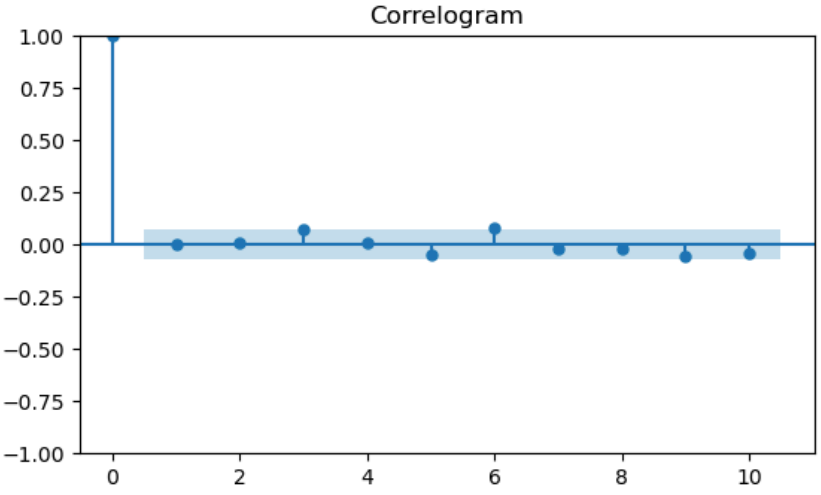



```
Evaluating HAEHNCHEN
Best model for HAEHNCHEN: p=0, d=1, q=1, P=0, D=1, Q=1
```

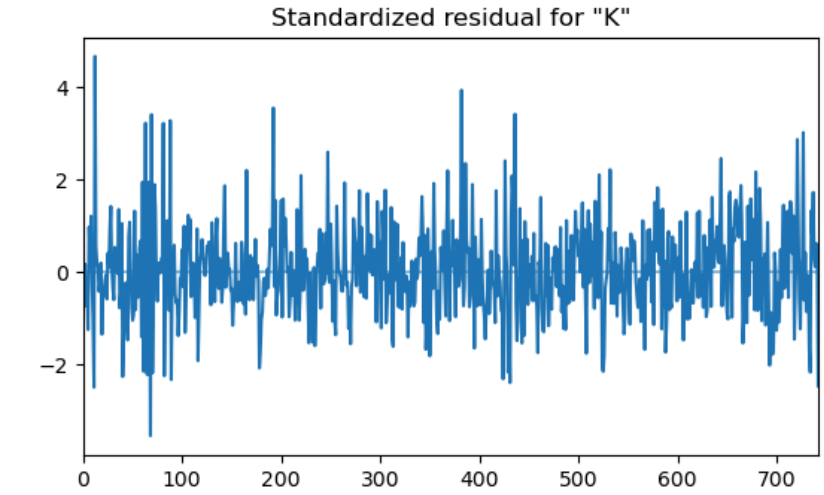


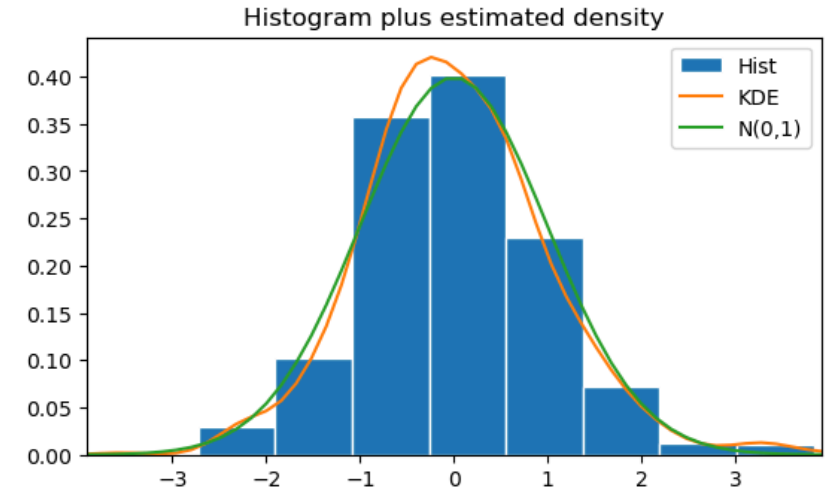


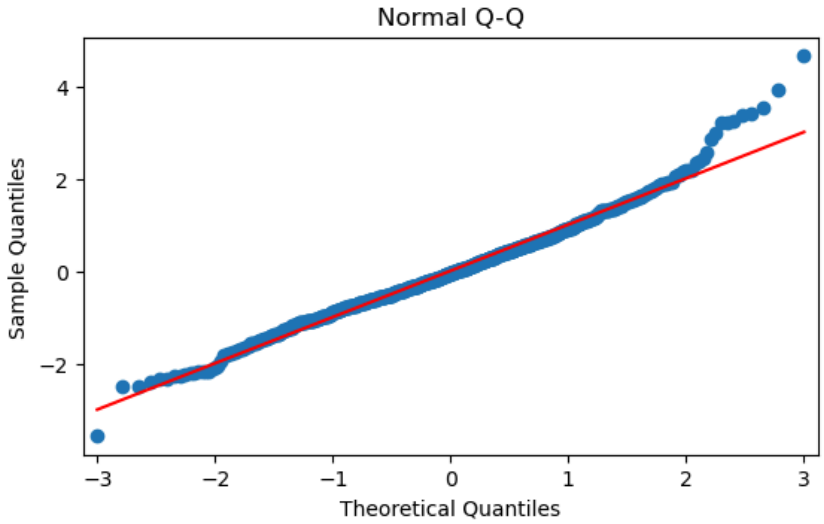


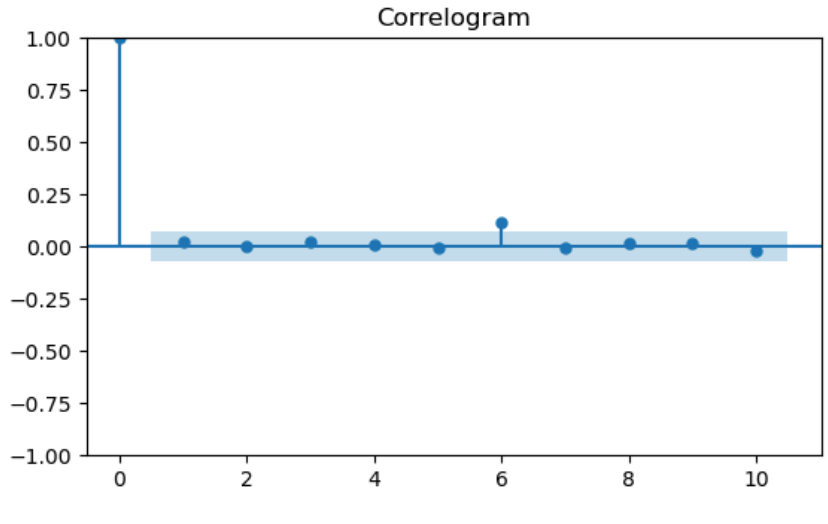



```
Evaluating KOEFTE
Best model for KOEFTE: p=0, d=1, q=1, P=0, D=1, Q=1
```

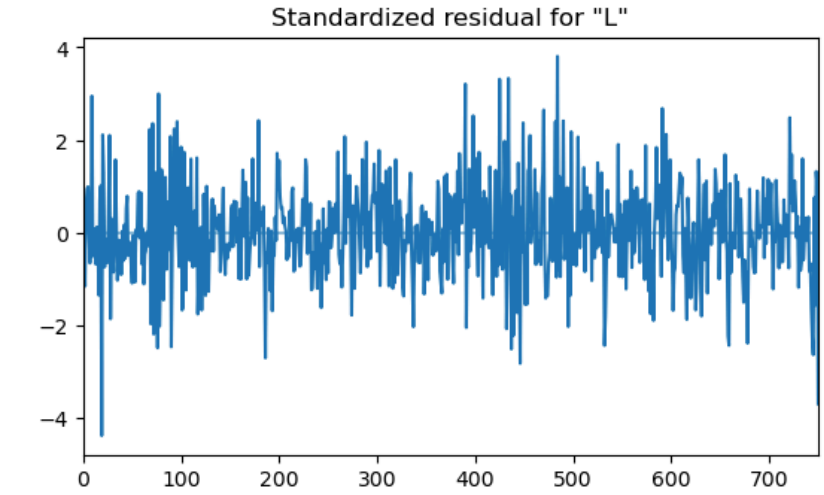


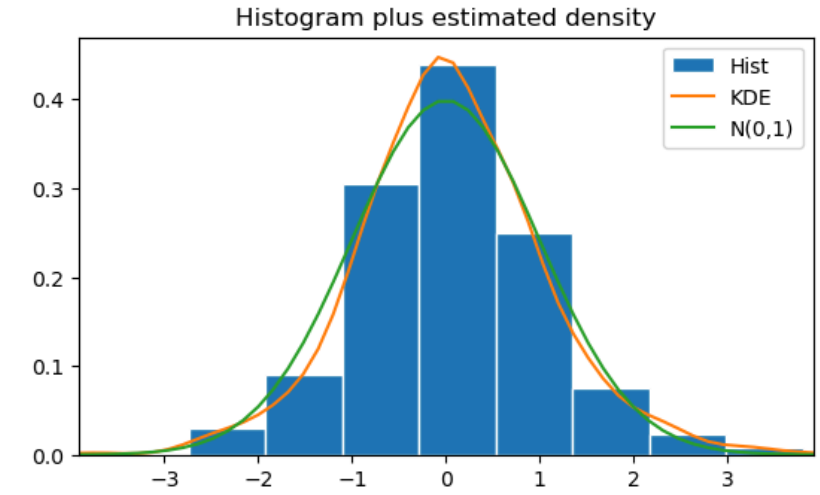


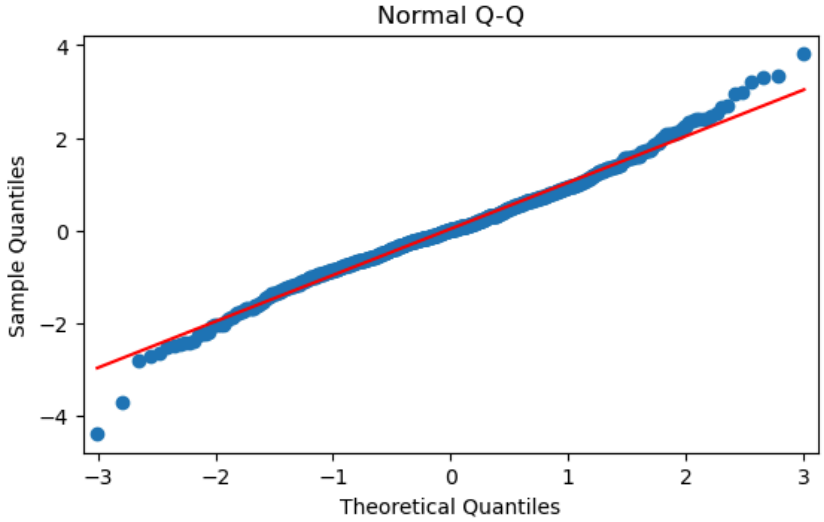


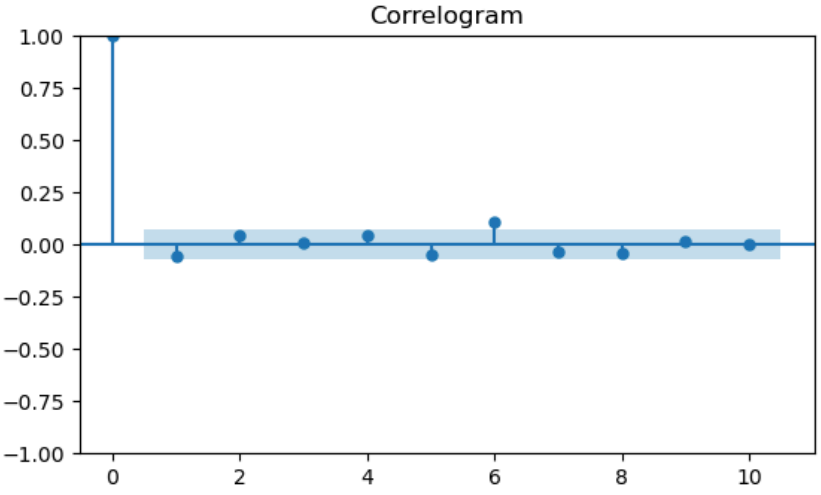



```
Evaluating LAMM
Best model for LAMM: p=1, d=0, q=1, P=1, D=0, Q=1
```

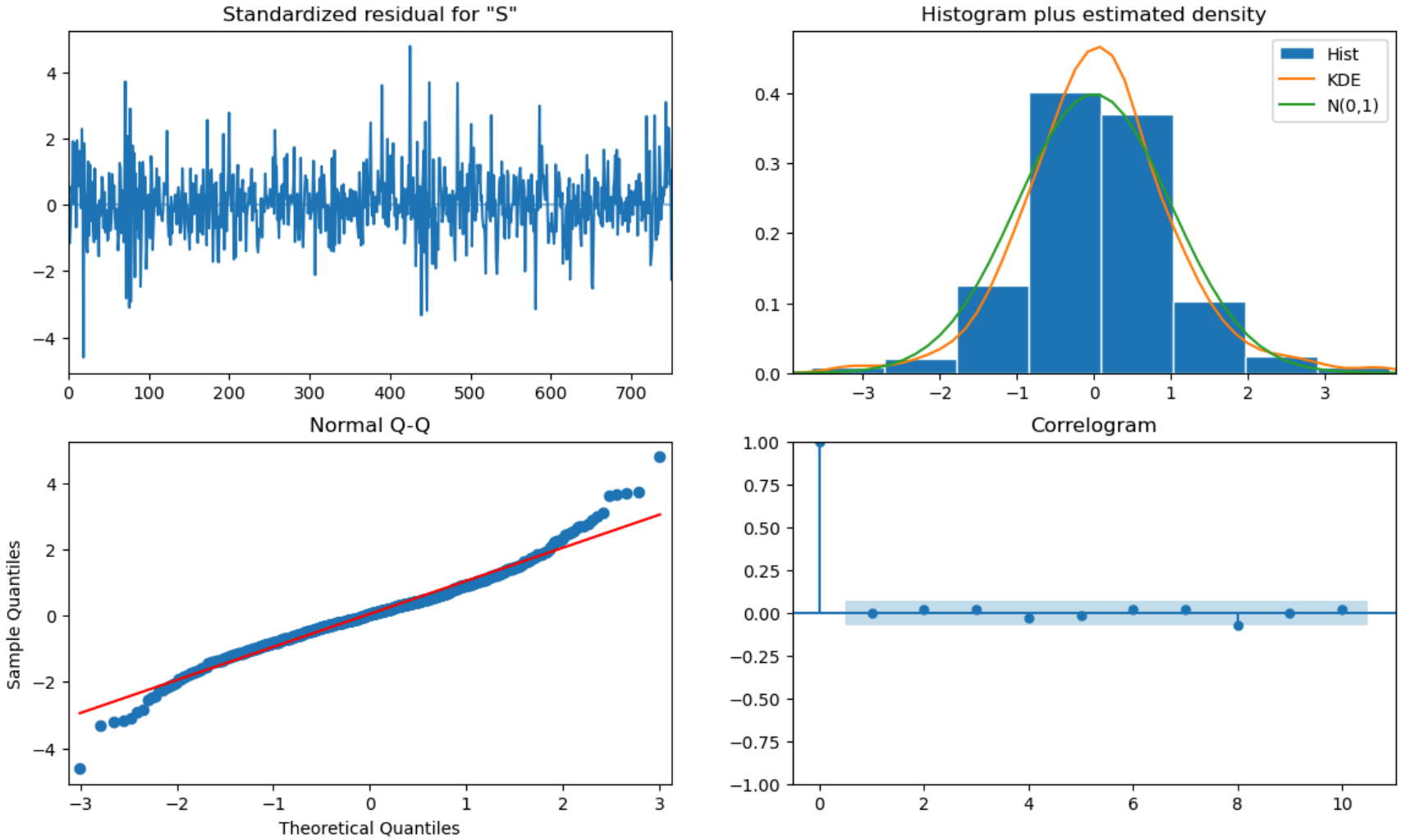









```
Evaluating STEAK
Best model for STEAK: p=0, d=0, q=1, P=1, D=0, Q=1
```



{'CALAMARI': {'p': 1, 'd': 0, 'q': 1, 'P': 1, 'D': 0, 'Q': 1}, 'FISCH': {'p': 0, 'd': 0, 'q': 1, 'P': 1, 'D': 0, 'Q': 1}, 'GARNELEN': {'p': 1, 'd': 0, 'q': 1, 'P': 1, 'D': 0, 'Q': 1}, 'HAEHNCHEN': {'p': 0, 'd': 1, 'q': 1, 'P': 0, 'D': 1, 'Q': 1}, 'KOEFTE': {'p': 0, 'd': 1, 'q': 1, 'P': 0, 'D': 1, 'Q': 1}, 'LAMM': {'p': 1, 'd': 0, 'q': 1, 'P': 1, 'D': 0, 'Q': 1}, 'STEAK': {'p': 0, 'd': 0, 'q': 1, 'P': 1, 'D': 0, 'Q': 1}}

To analyze the diagnostic plots that were outputted, we check for:

Standardized Residuals: Residuals should behave like white noise (i.e., no obvious patterns, constant variance, and mean around zero). Any patterns or trends may suggest the model is missing some information.

Histogram plus KDE (Kernel Density Estimate): This plot should follow a normal bell curve.

Normal Q-Q Plot: Points should fall approximately along the 45-degree line. Significant deviations indicate that the residuals are not normally distributed, which may affect the validity of confidence intervals and forecasts.

Correlogram of the Residuals (ACF Plot):This plot shows the autocorrelation of the residuals at various lags. There should not be any significant spikes outside the shaded confidence interval area.

The plots that were outputted for each menu item after finding the best parameters fulfill all the requirements that we are looking for in the plot diagnostics to make sure that the model has a quality fit.

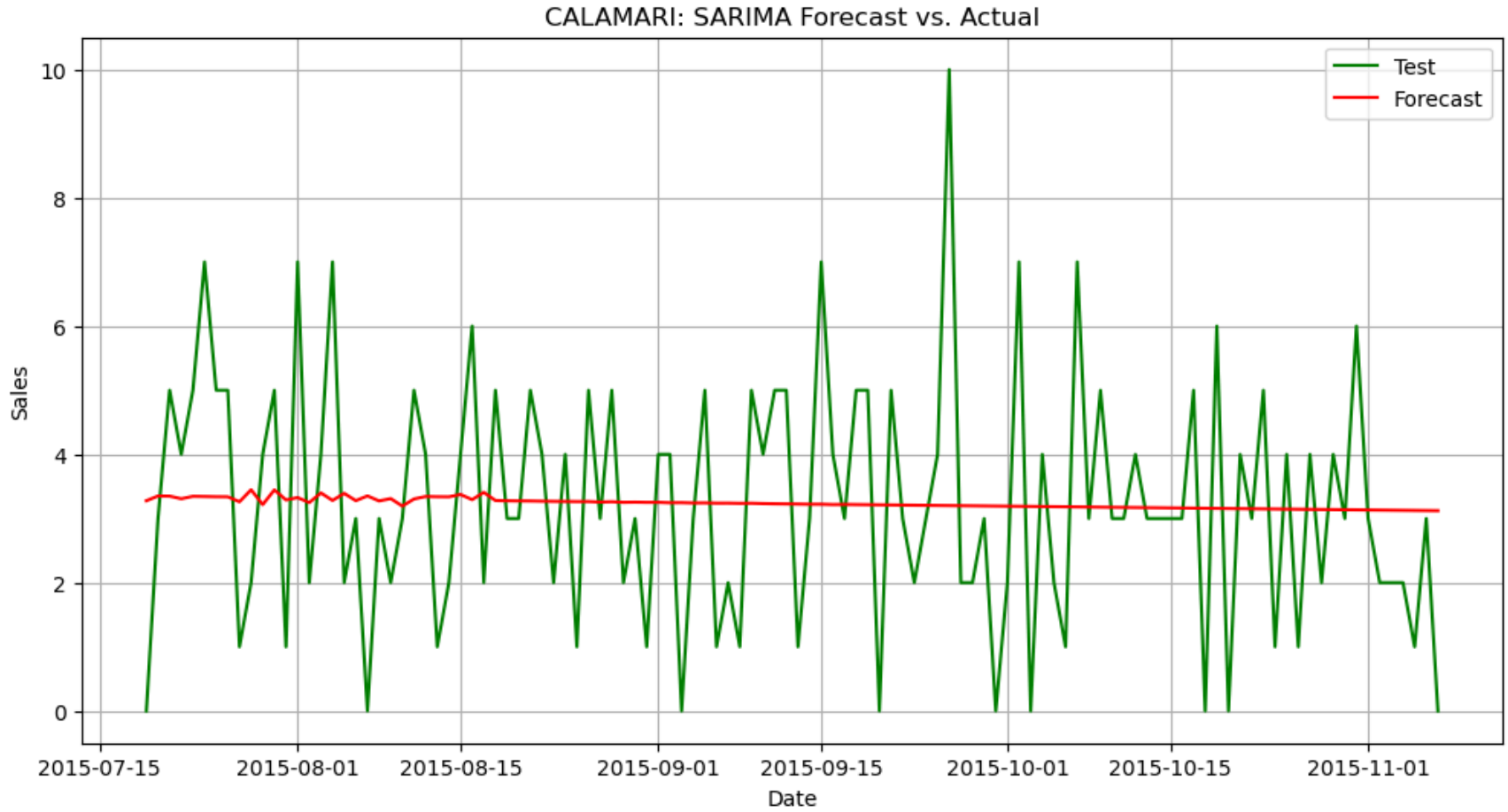
Model Evaluation

In [ ]:
```
def evaluate_sarima(item, train, test, test_size, params, m):

    model = SARIMAX(
        df_time_series[item],
        order=(params[item]['p'], params[item]['d'], params[item]['q']),
        seasonal_order=(params[item]['P'], params[item]['D'], params[item]['Q'], m),
        enforce_stationarity=False,
        enforce_invertibility=False
    )

    sarima_model = model.fit(disp=False)

    forecast = sarima_model.get_forecast(steps=test_size)
    forecast_index = test.index
    forecast_values = forecast.predicted_mean

    plt.figure(figsize=(12, 6))
    plt.plot(test.index, test[item], label='Test', color='green')
    plt.plot(forecast_index, forecast_values, label='Forecast', color='red')
    plt.title(f'{item}: SARIMA Forecast vs. Actual')
    plt.xlabel('Date')
    plt.ylabel('Sales')
    plt.legend()
    plt.grid(True)
    plt.show()

    mse = round(mean_squared_error(test[item], forecast_values), 2)
    print(f'{item} Mean Squared Error: {mse}')

    return mse
```

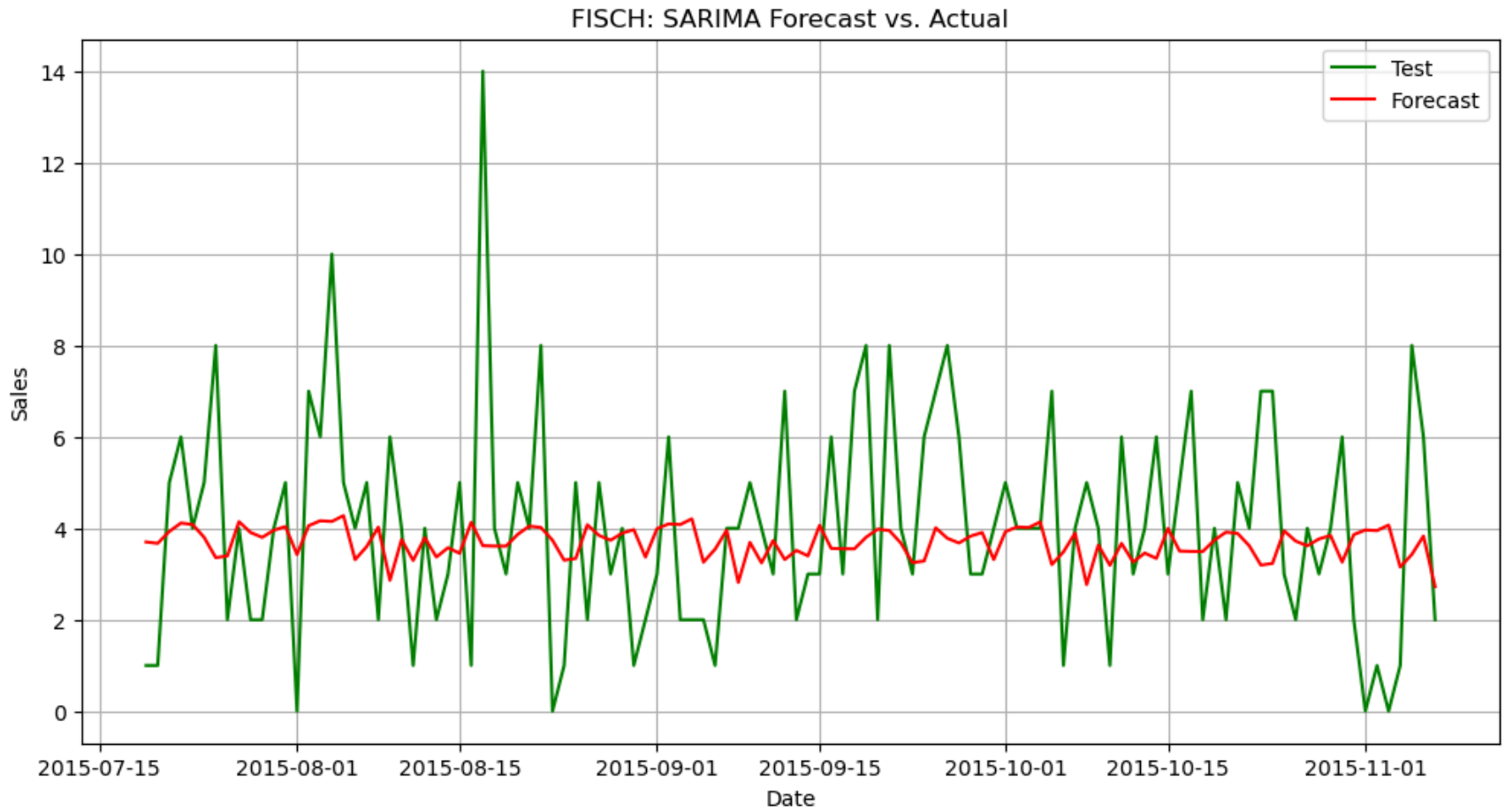
In [327…
```
def run(m):
    mse_all_items = {}

    for item in menu_items:
        test_size = 7 * 16 # Last 16 weeks
        train_end = len(df_time_series) - test_size
        train = df_time_series.iloc[:train_end]
        test = df_time_series.iloc[train_end:]

        mse = evaluate_sarima(item, train, test, test_size, best_params, m)
        mse_all_items[item] = mse

    return mse_all_items
```

Comparing Monthly Trends (m = 30) vs. Weekly Trends (m = 7)

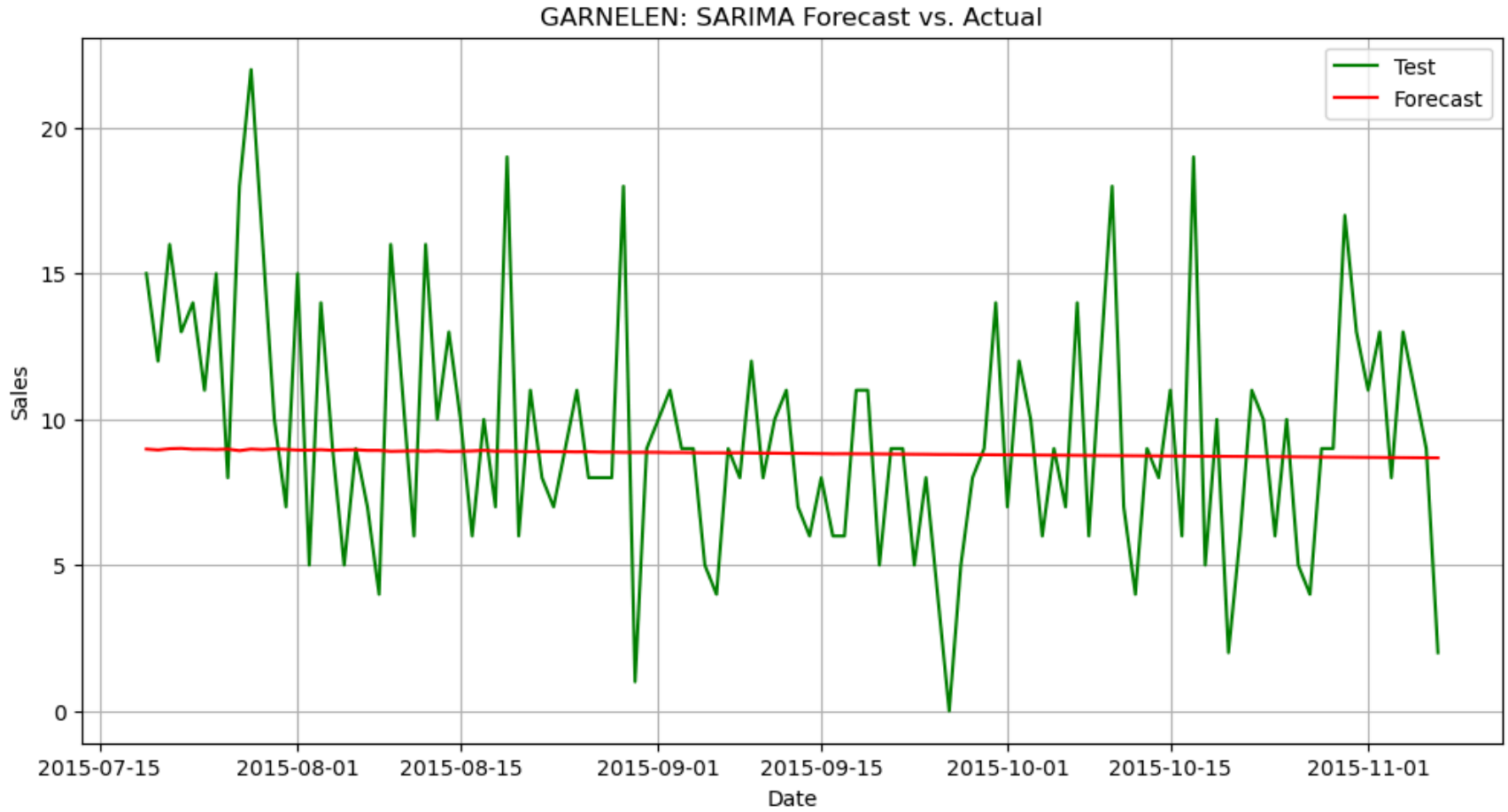
In [328…
```
run(m=30)
```



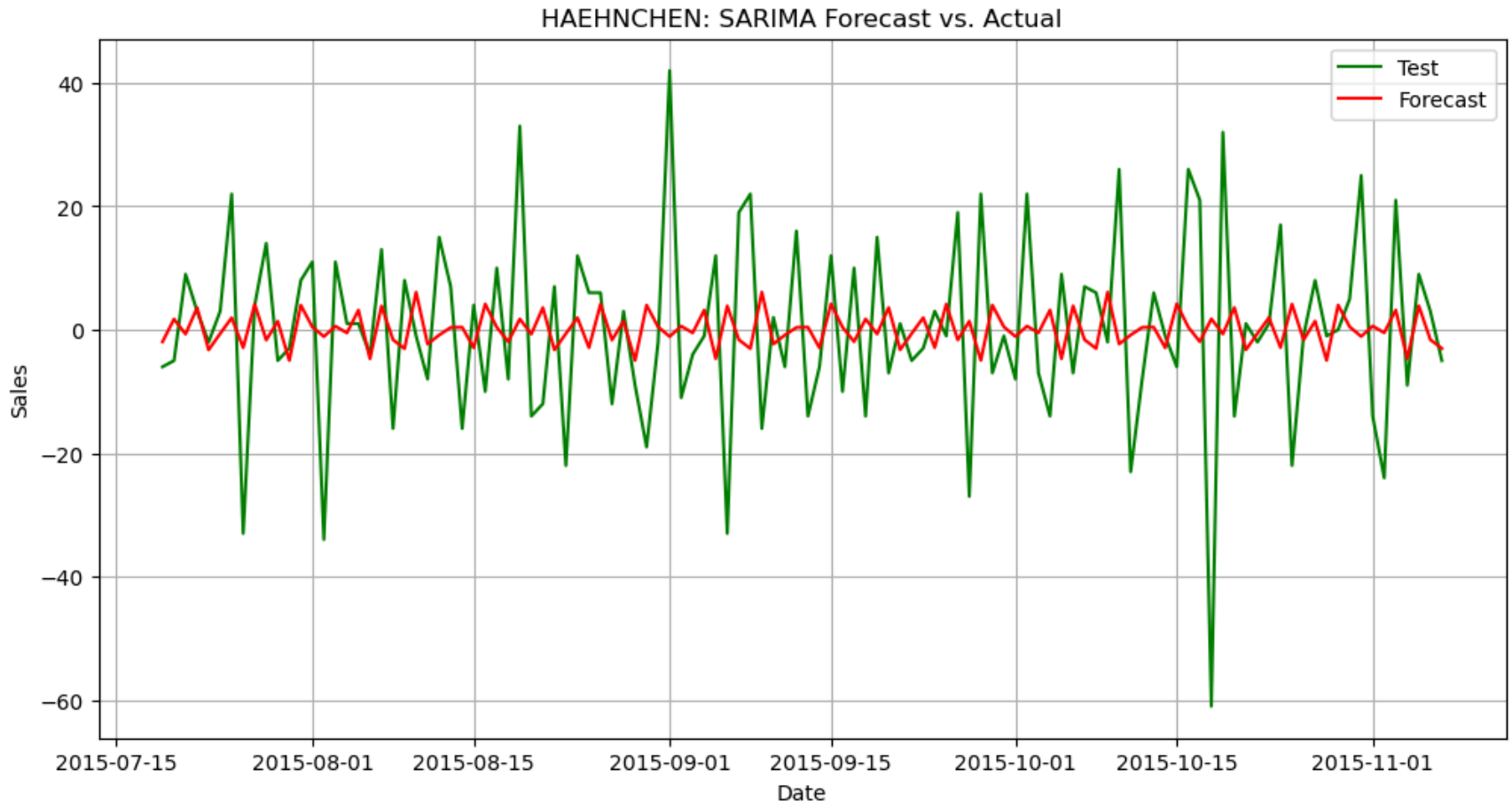

```
CALAMARI Mean Squared Error: 3.63
```



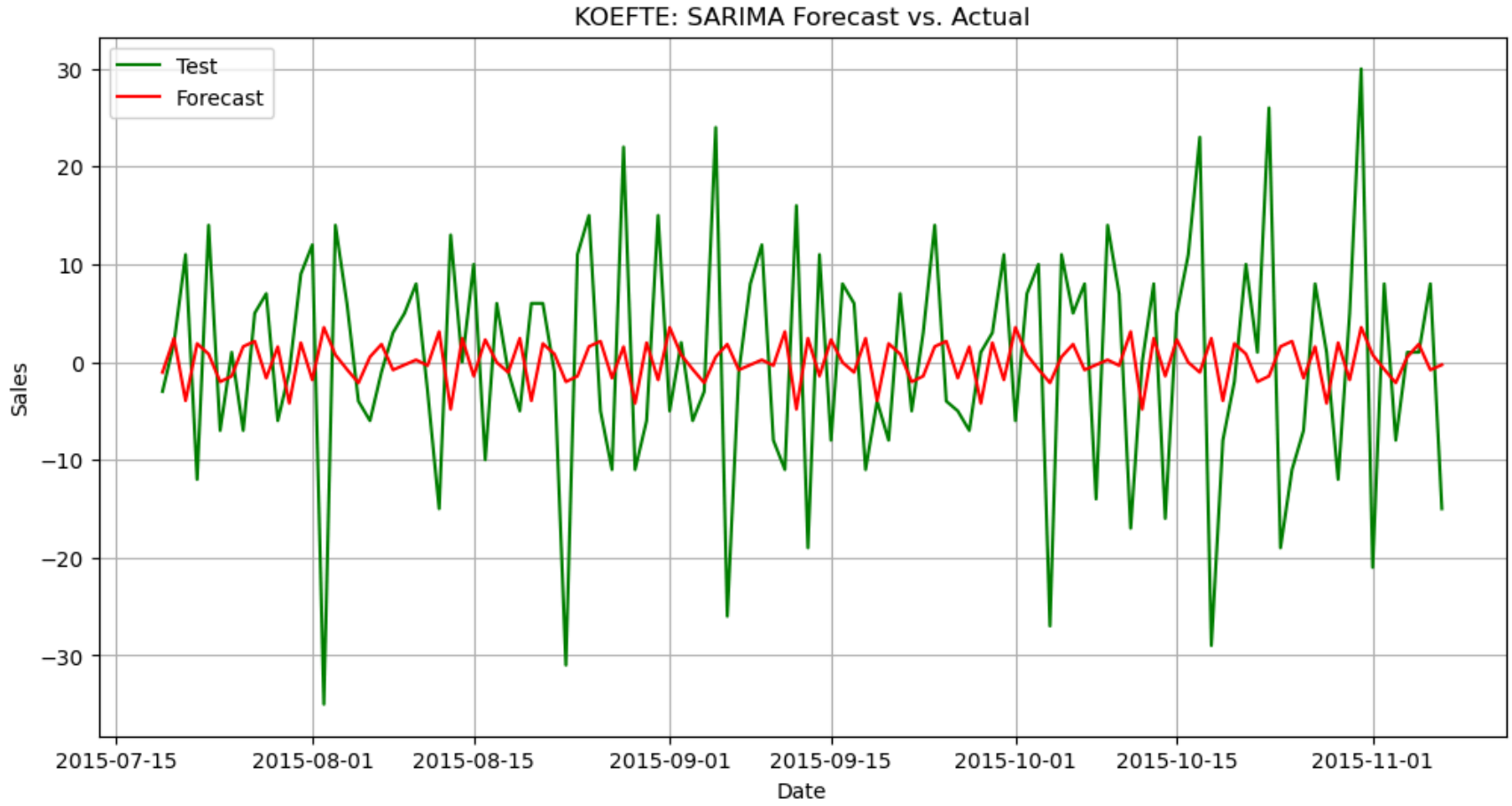

```
FISCH Mean Squared Error: 5.67
```



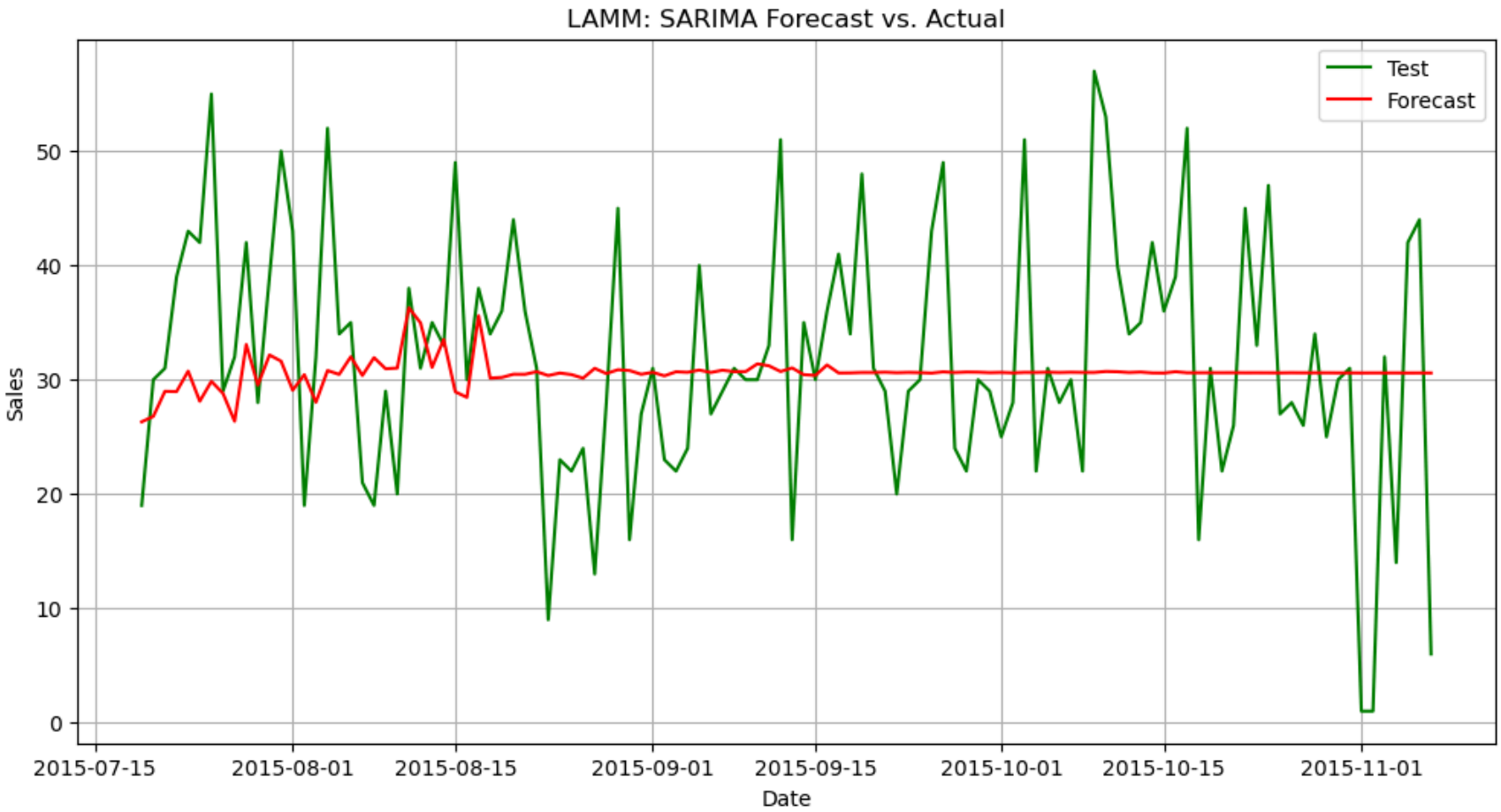

```
GARNELEN Mean Squared Error: 16.92
```



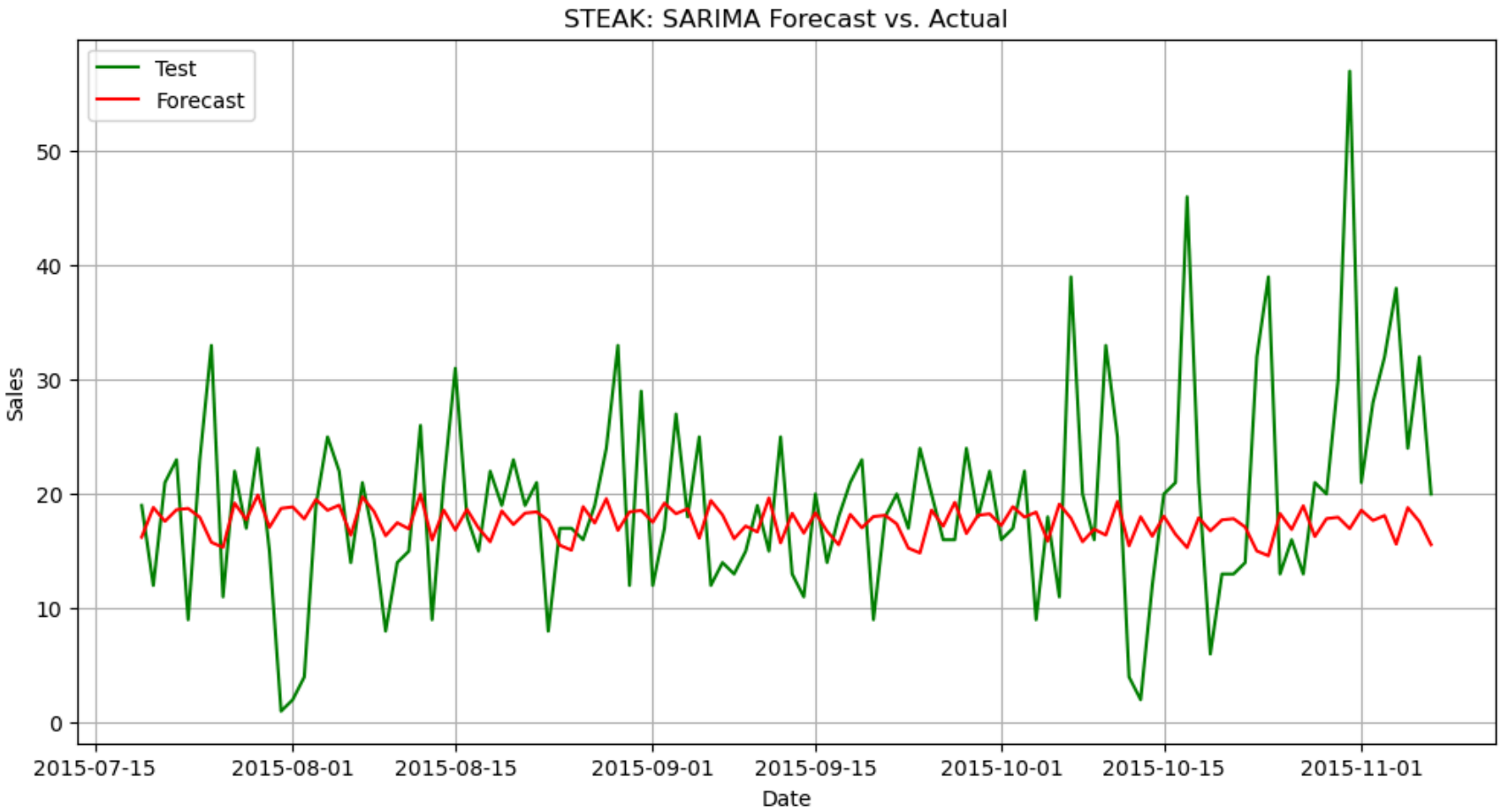

```
HAEHNCHEN Mean Squared Error: 255.04
```




```
KOEFTE Mean Squared Error: 155.85
```




```
LAMM Mean Squared Error: 120.16
```

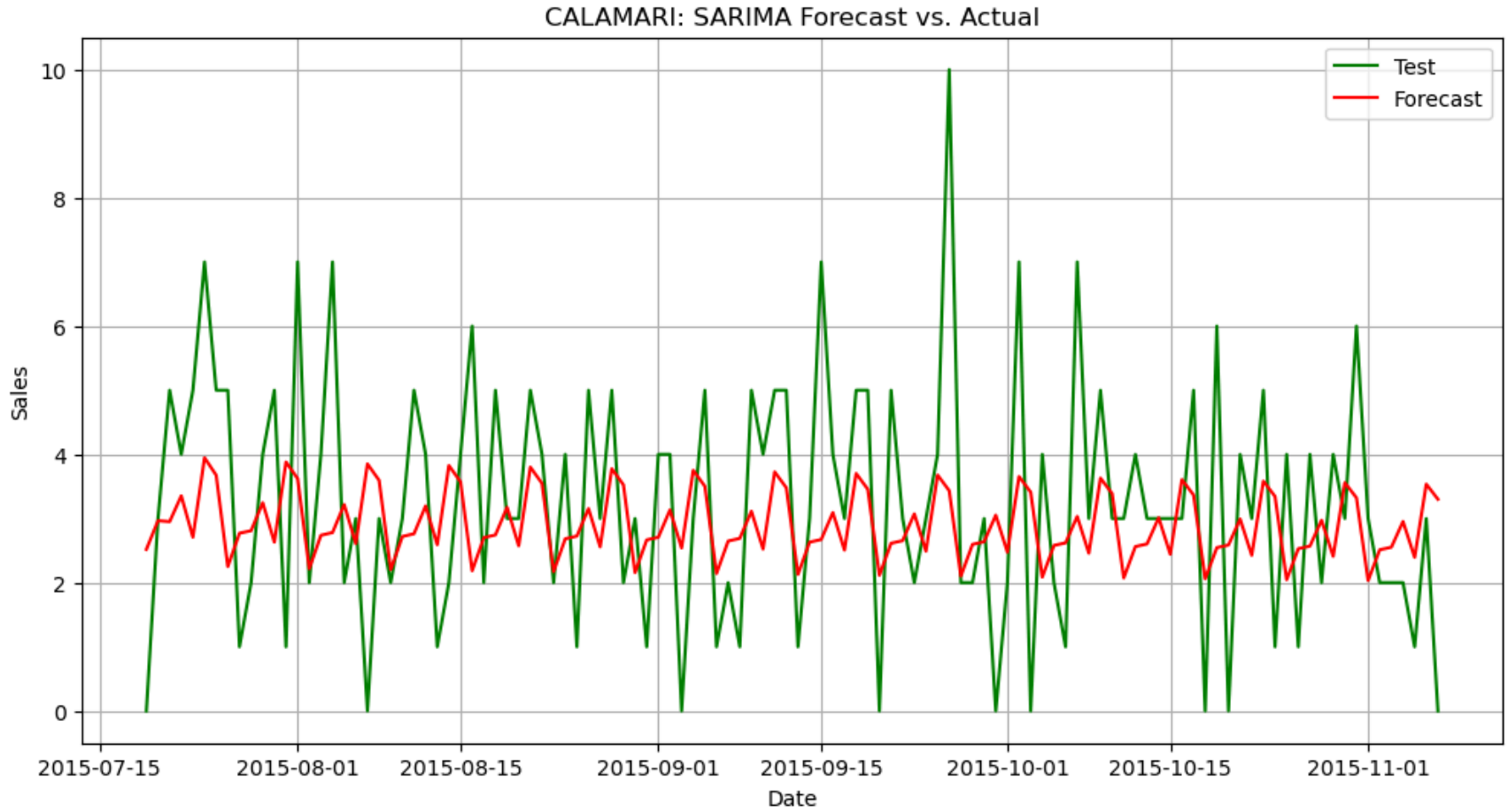



```
STEAK Mean Squared Error: 84.47
```

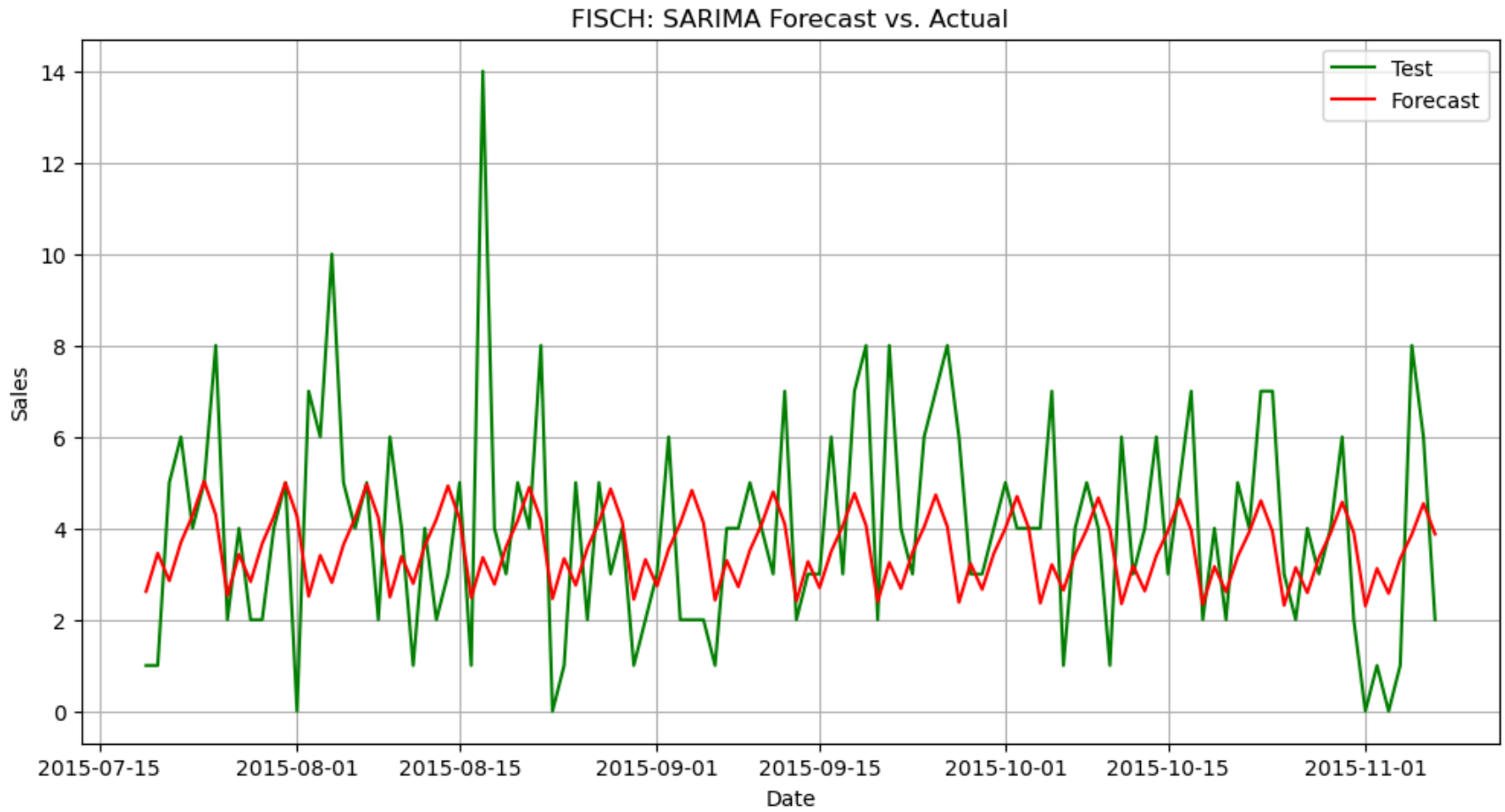
Out[328]:
```
{'CALAMARI': 3.63,
 'FISCH': 5.67,
 'GARNELEN': 16.92,
 'HAEHNCHEN': 255.04,
 'KOEFTE': 155.85,
 'LAMM': 120.16,
 'STEAK': 84.47}
```

The monthly seasonal period does not seem to perform very well, as it does not correctly capture the trends that the data exhibits. We can see this in LAMM for example, as the model becomes stagnant at the end of the month, ignoring the dips and peaks in the data.

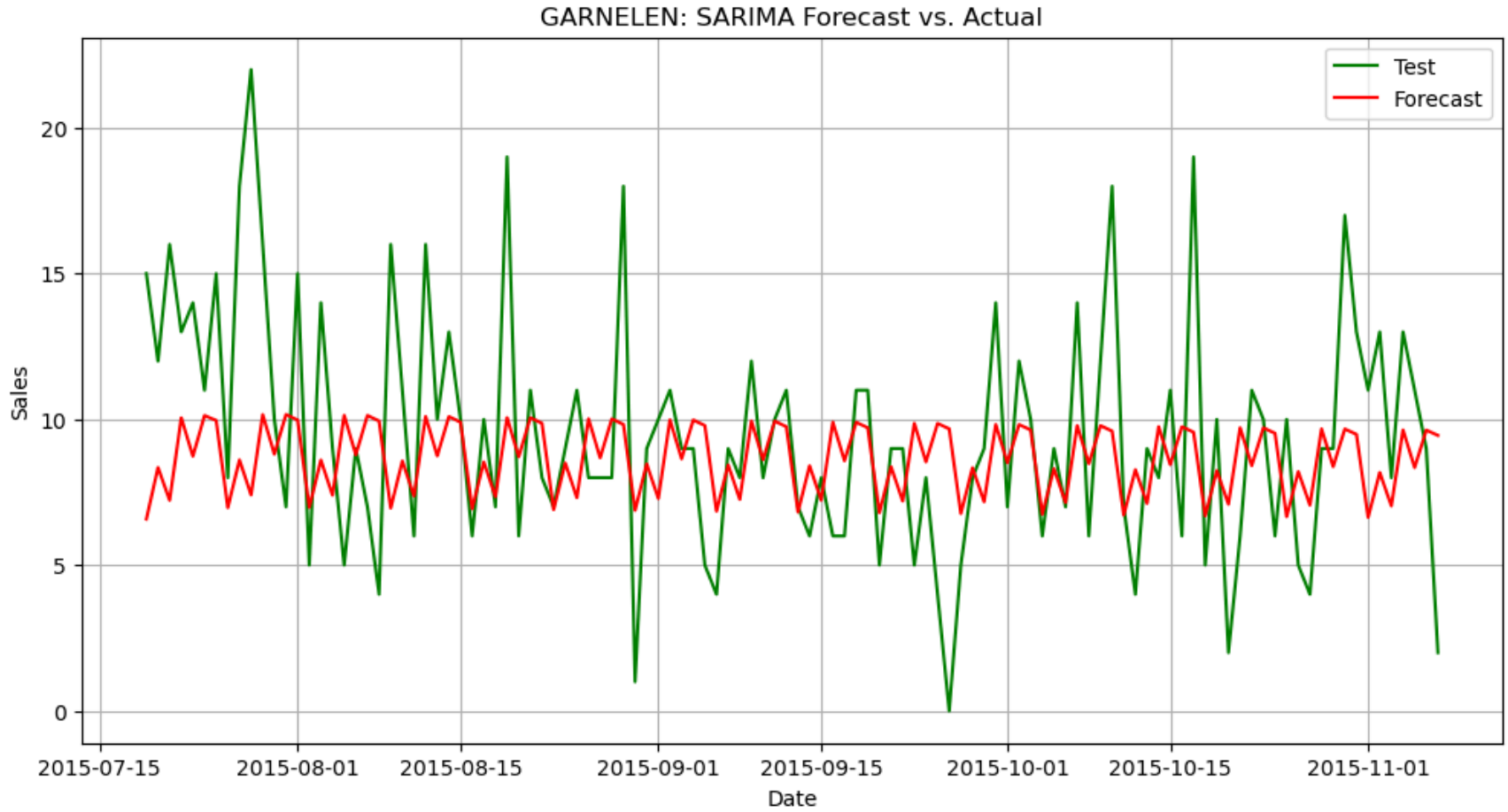
In [329…
```
run(m=7)
```



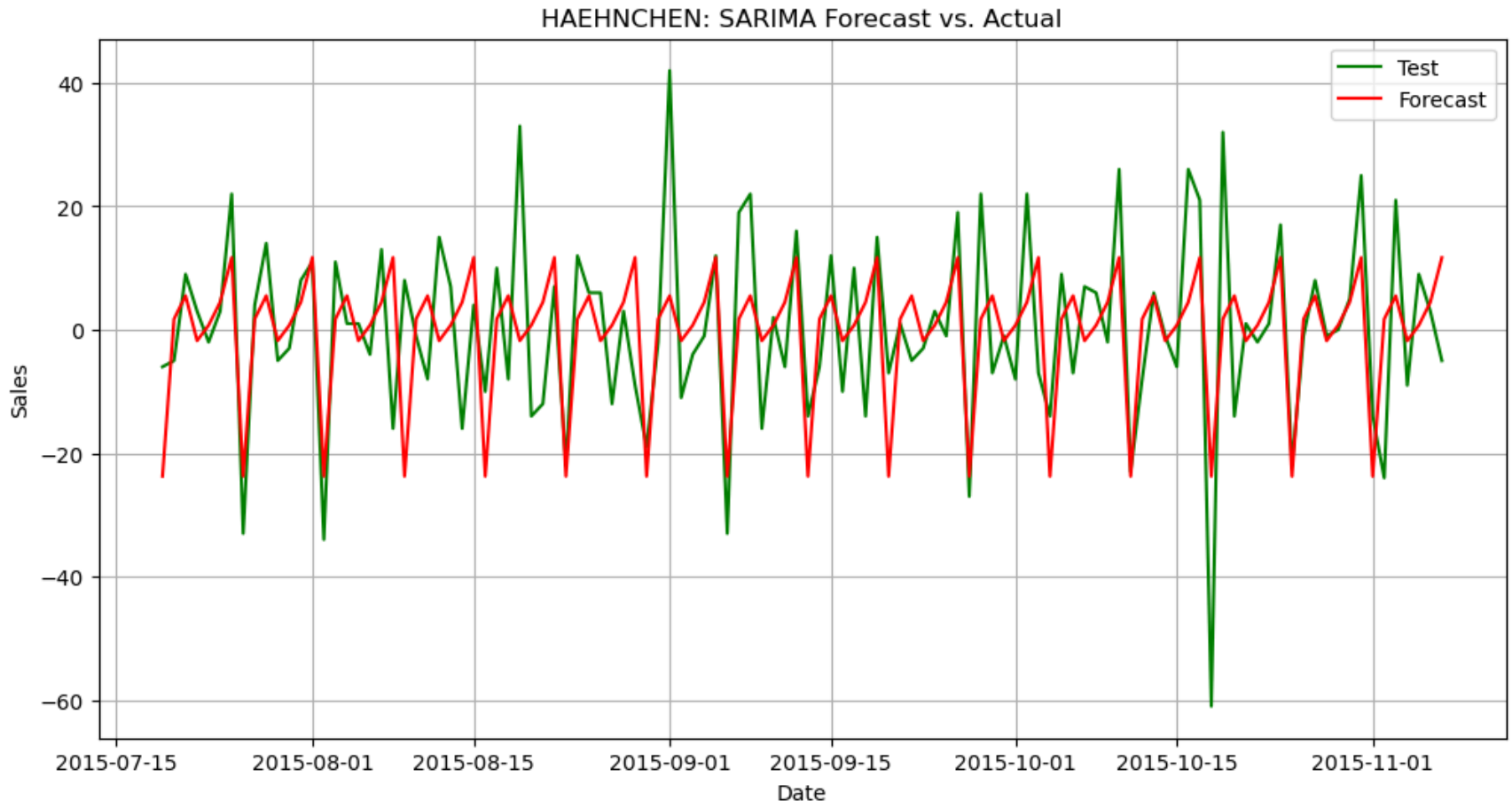

```
CALAMARI Mean Squared Error: 3.45
```



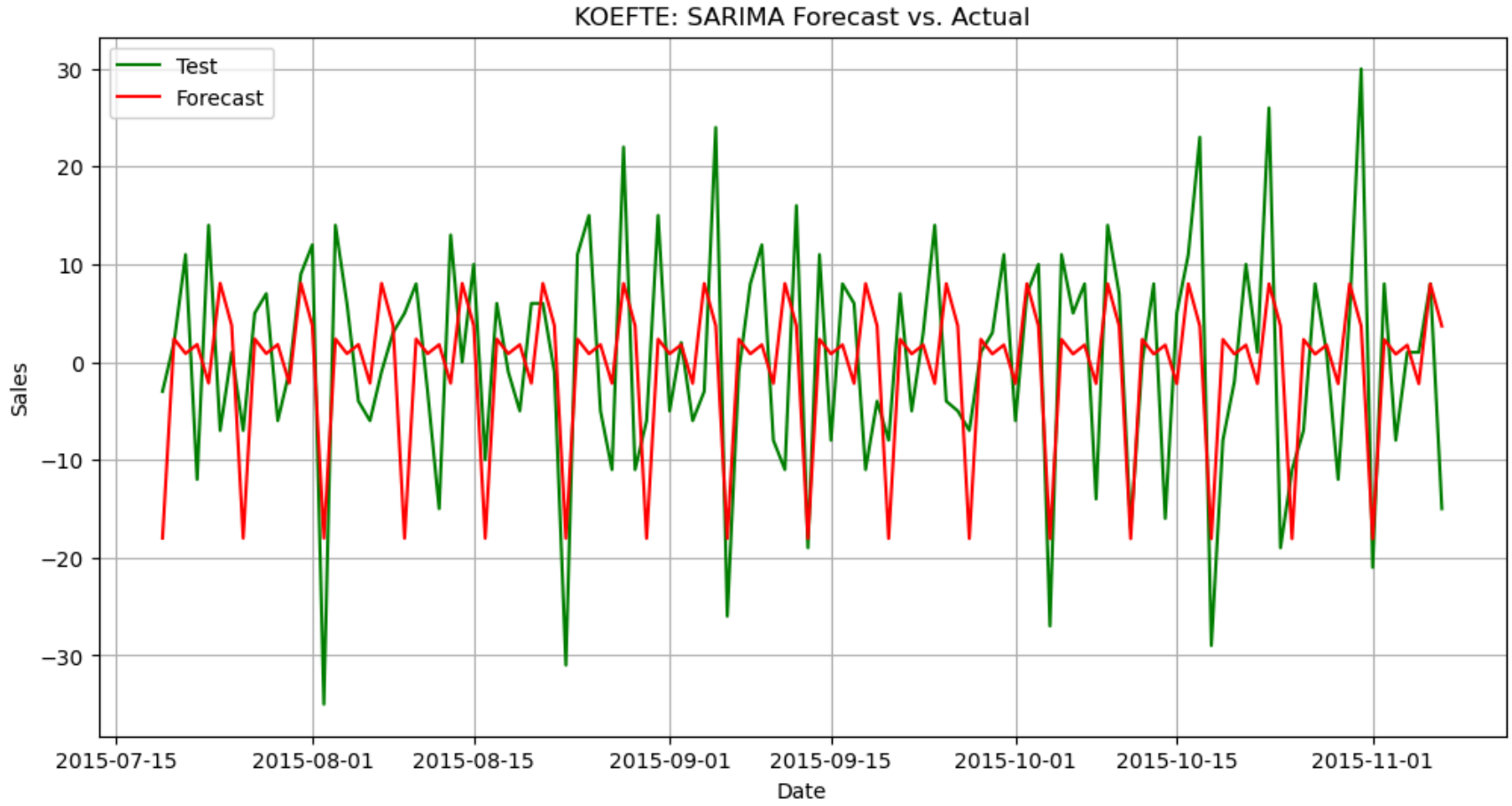

```
FISCH Mean Squared Error: 5.32
```



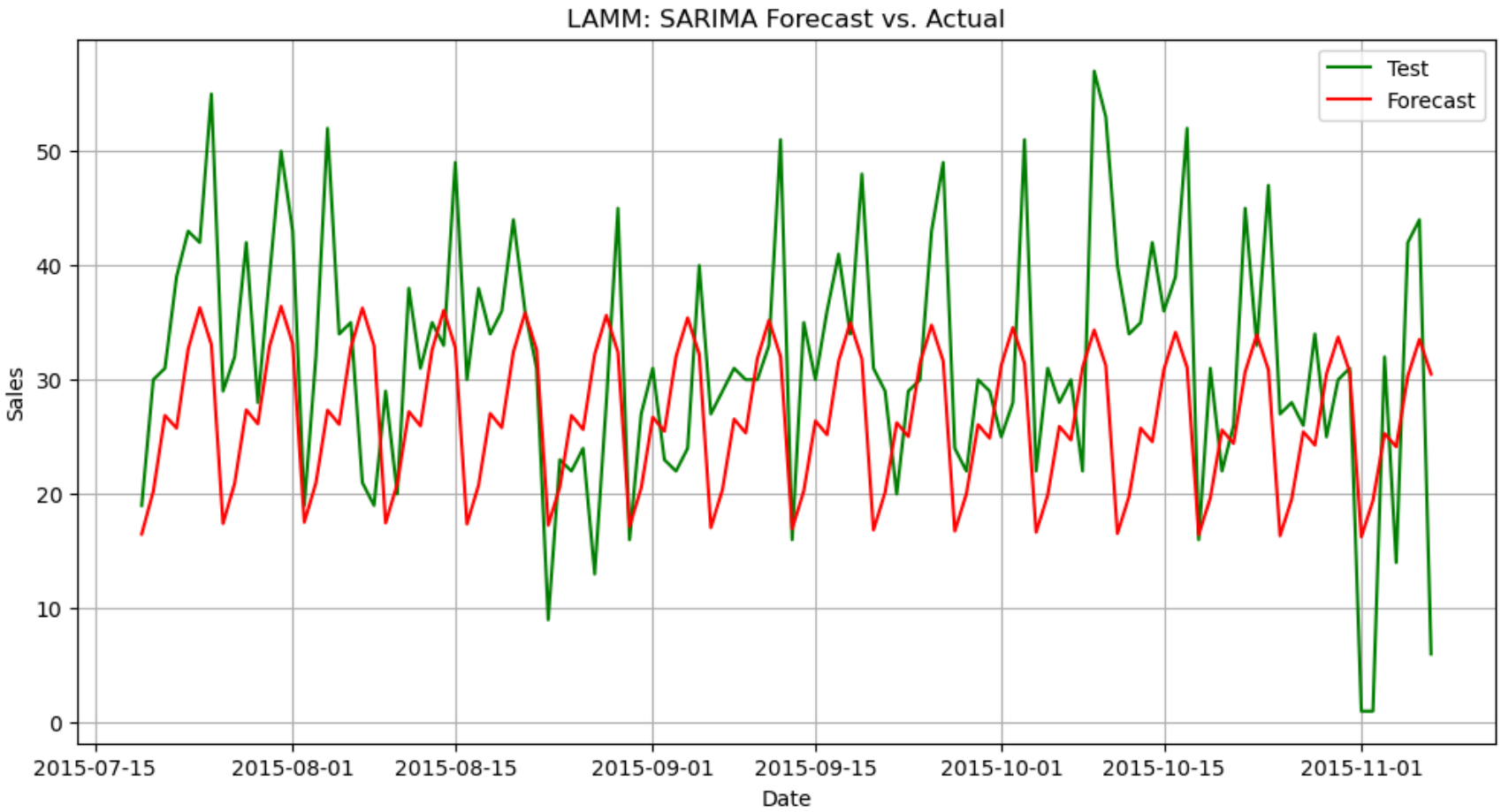

```
GARNELEN Mean Squared Error: 16.44
```



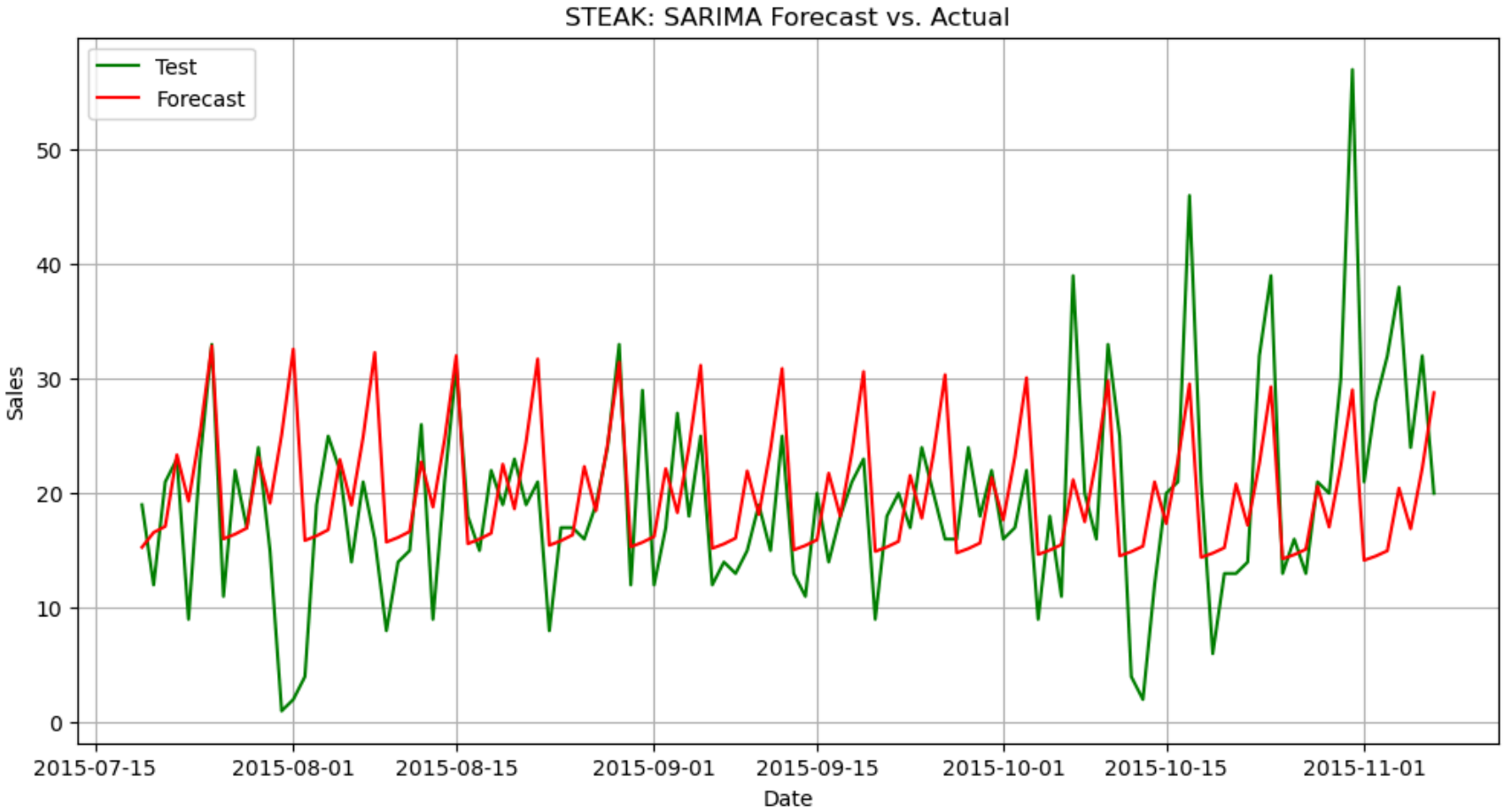

```
HAEHNCHEN Mean Squared Error: 155.18
```




```
KOEFTE Mean Squared Error: 101.42
```




```
LAMM Mean Squared Error: 117.89
```




```
STEAK Mean Squared Error: 67.3
```

Out[329]:
```
{'CALAMARI': 3.45,
 'FISCH': 5.32,
 'GARNELEN': 16.44,
 'HAEHNCHEN': 155.18,
 'KOEFTE': 101.42,
 'LAMM': 117.89,
 'STEAK': 67.3}
```

The visualization for a weekly seasonal period fits the trends in the data much better than a monthly seasonal period. We can see that generally the spikes follow the natural dips and peaks of weekly sales (higher sales on weekends, followed by a decrease in sales on the weekdays).

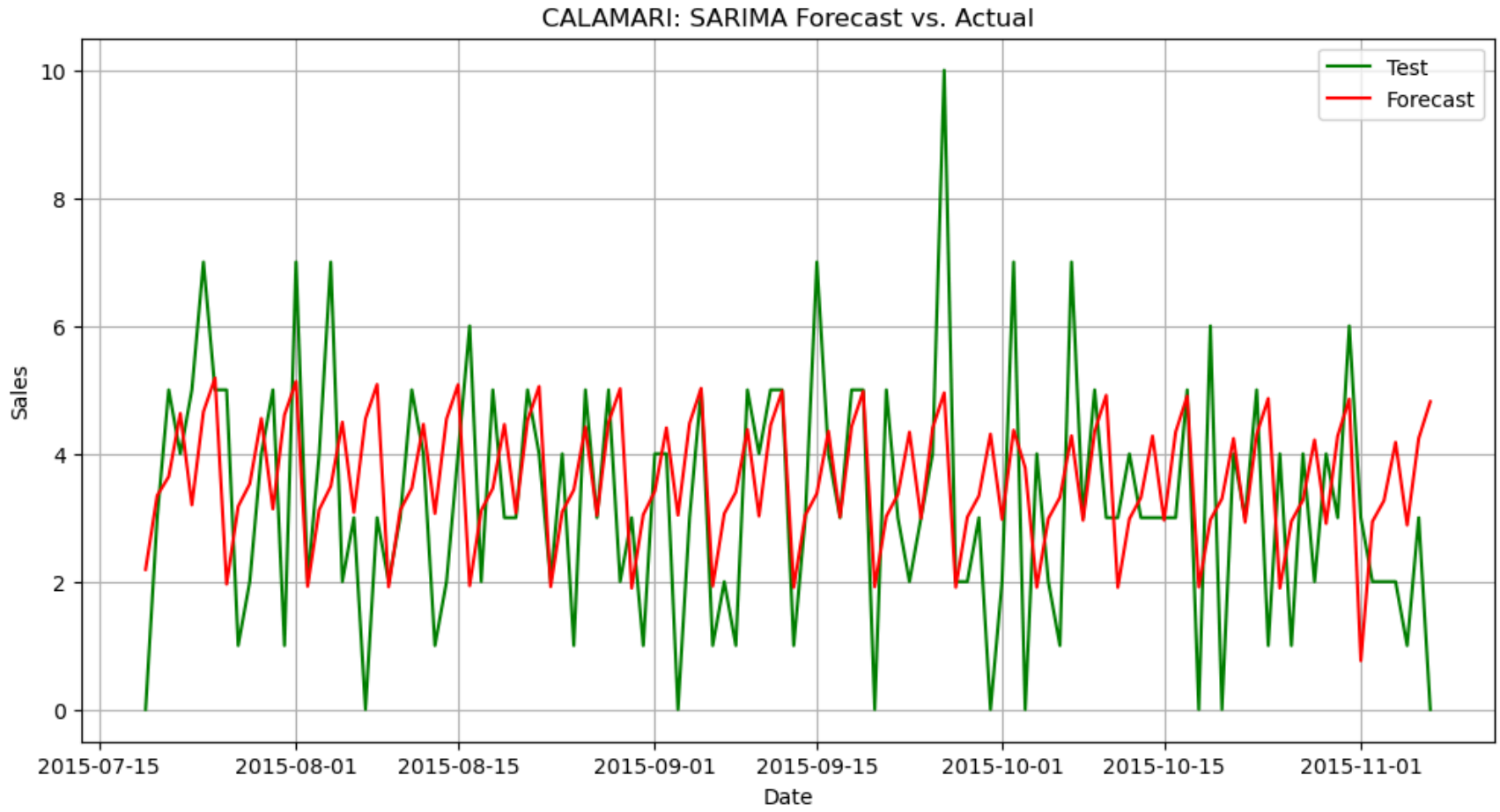
In [506…

```
def evaluate_sarima_exog(item, train, test, test_size, params, m, train_exog, test_exog):

    model = SARIMAX(
        train[item],
        order=(params[item]['p'], params[item]['d'], params[item]['q']),
        seasonal_order=(params[item]['P'], params[item]['D'], params[item]['Q'], m),
        exog=train_exog,
        enforce_stationarity=False,
        enforce_invertibility=False
    )

    sarima_model = model.fit(disp=False)

    forecast = sarima_model.get_forecast(steps=test_size, exog=test_exog)
    forecast_index = test.index
    forecast_values = forecast.predicted_mean

    plt.figure(figsize=(12, 6))
    plt.plot(test.index, test[item], label='Test', color='green')
    plt.plot(forecast_index, forecast_values, label='Forecast', color='red')
    plt.title(f'{item}: SARIMA Forecast vs. Actual')
    plt.xlabel('Date')
    plt.ylabel('Sales')
    plt.legend()
    plt.grid(True)
    plt.show()

    mse = round(mean_squared_error(test[item], forecast_values), 2)
    print(f'{item} Mean Squared Error: {mse}')

    return mse

def run_exog(m):
    mse_all_items = {}

    for item in menu_items:
        test_size = 7 * 16  # Last 16 weeks
        train_end = len(df_time_series) - test_size
        train = df_time_series.iloc[:train_end]
        test = df_time_series.iloc[train_end:]

        train_exog = df_exog.iloc[:train_end]
        test_exog = df_exog.iloc[train_end:]
```

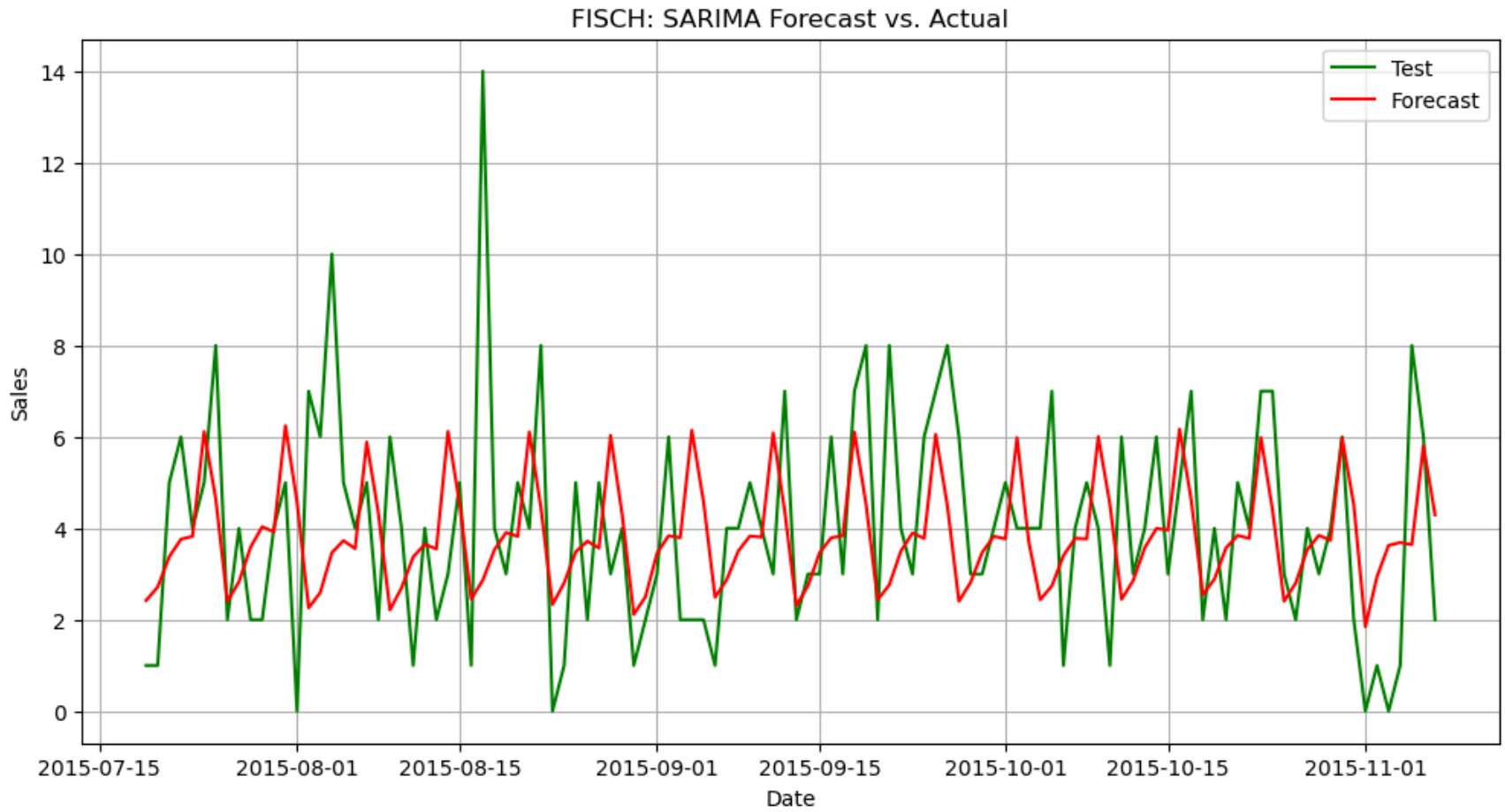
```
        mse = evaluate_sarima_exog(item, train, test, test_size, best_params, m, train_exog, test_exog)
        mse_all_items[item] = mse

    return mse_all_items
```

In [507…

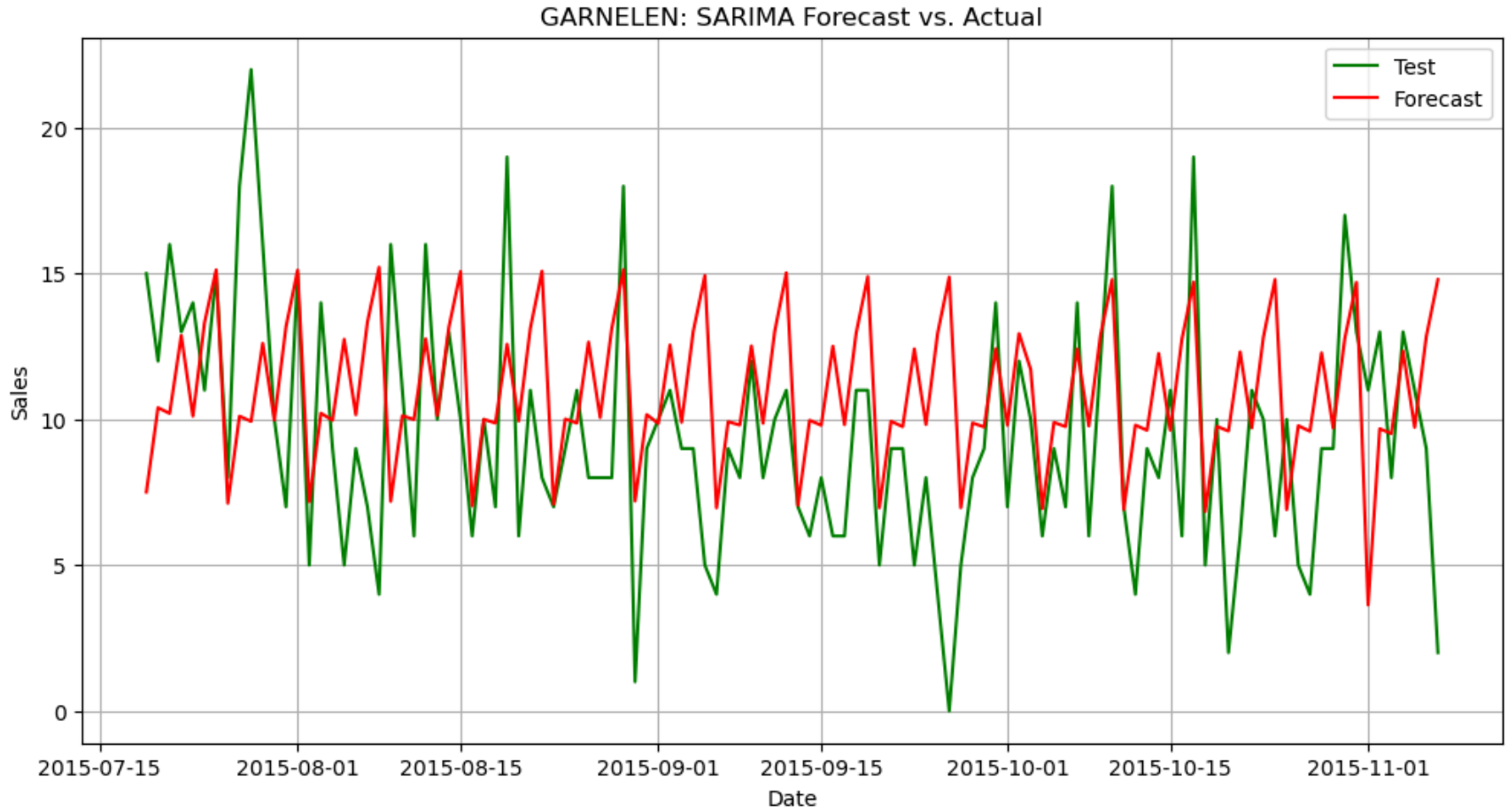
```
run_exog(7)
```



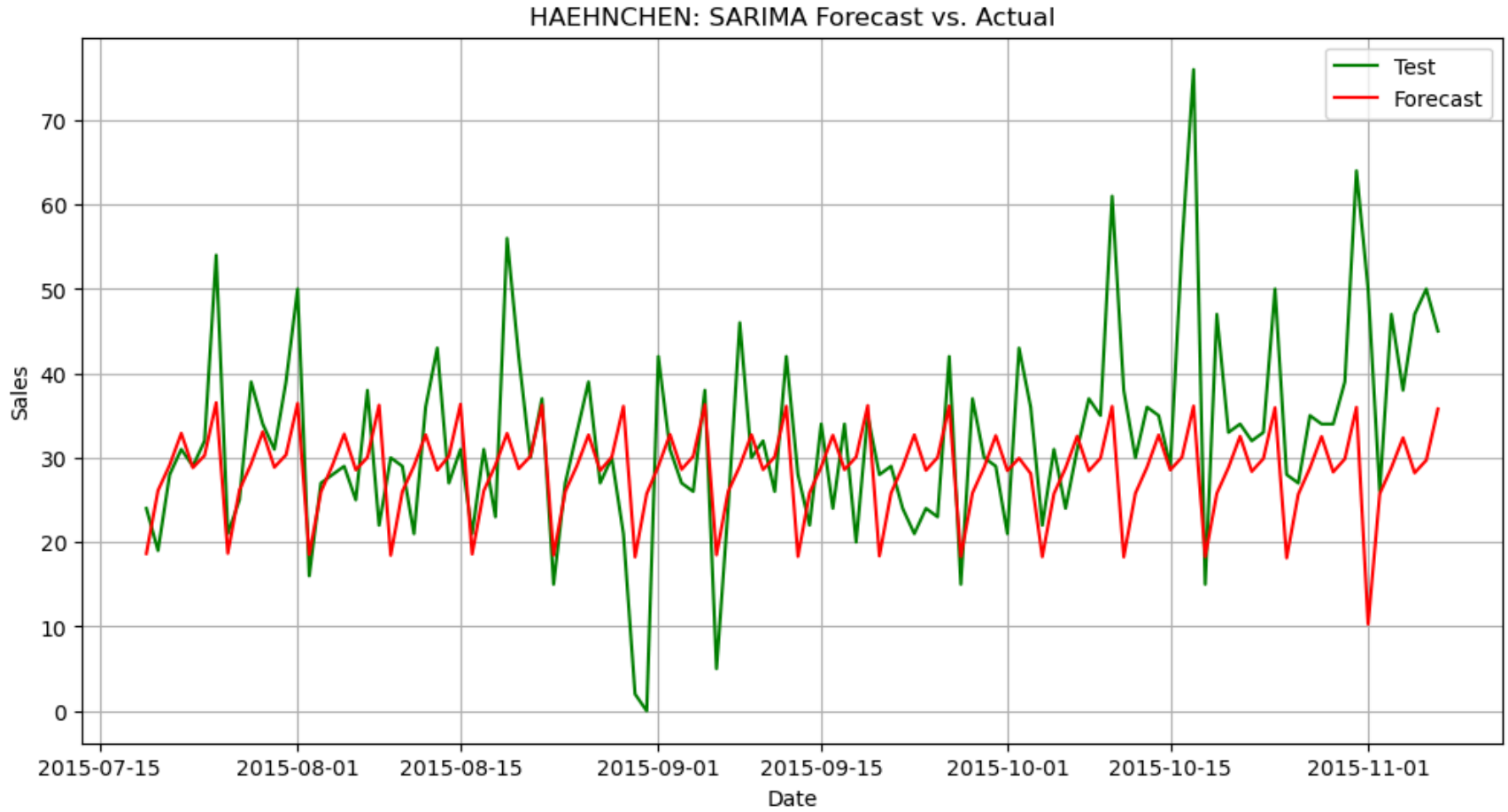

```
CALAMARI Mean Squared Error: 3.56
```



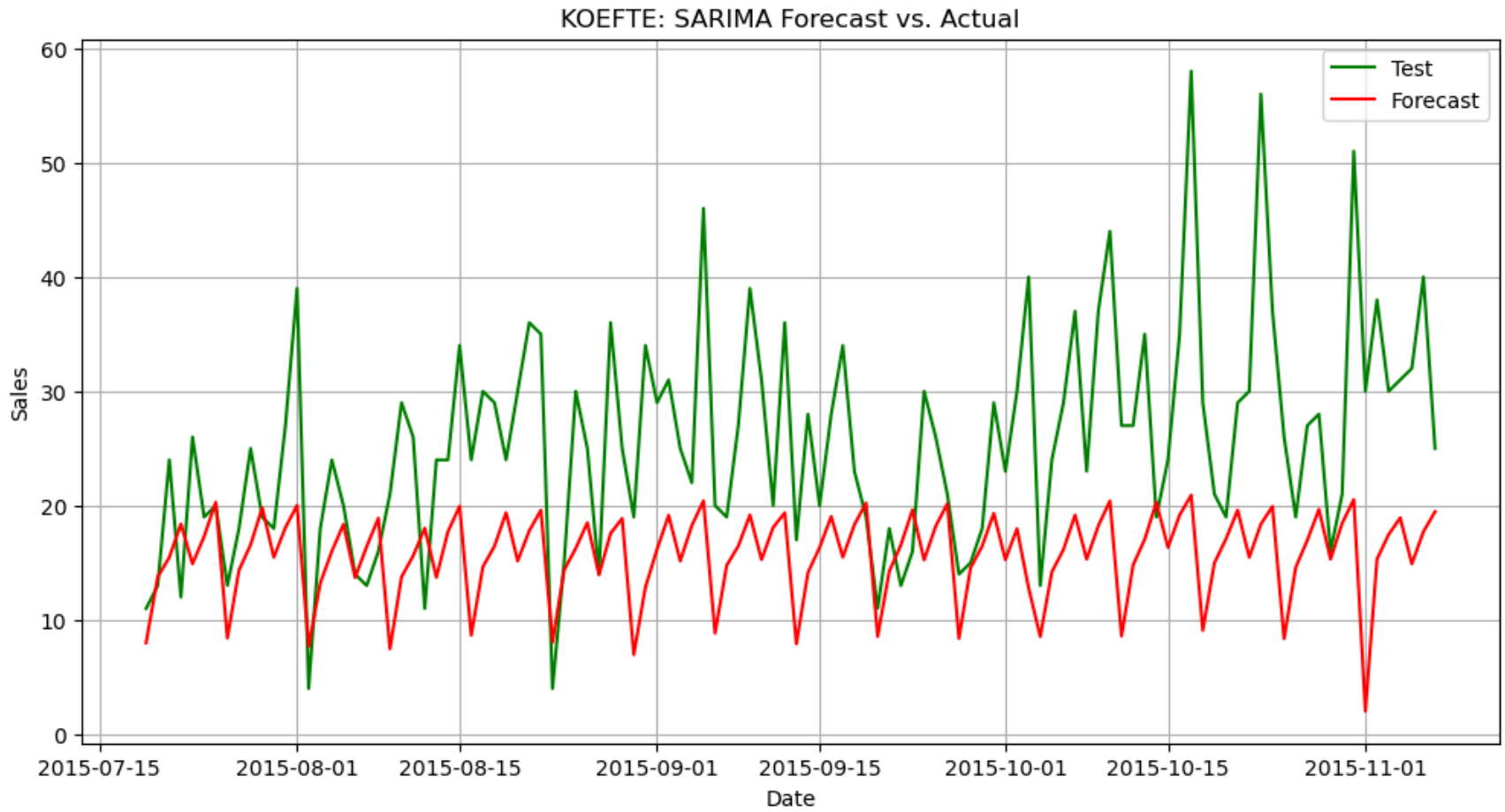

```
FISCH Mean Squared Error: 5.6
```



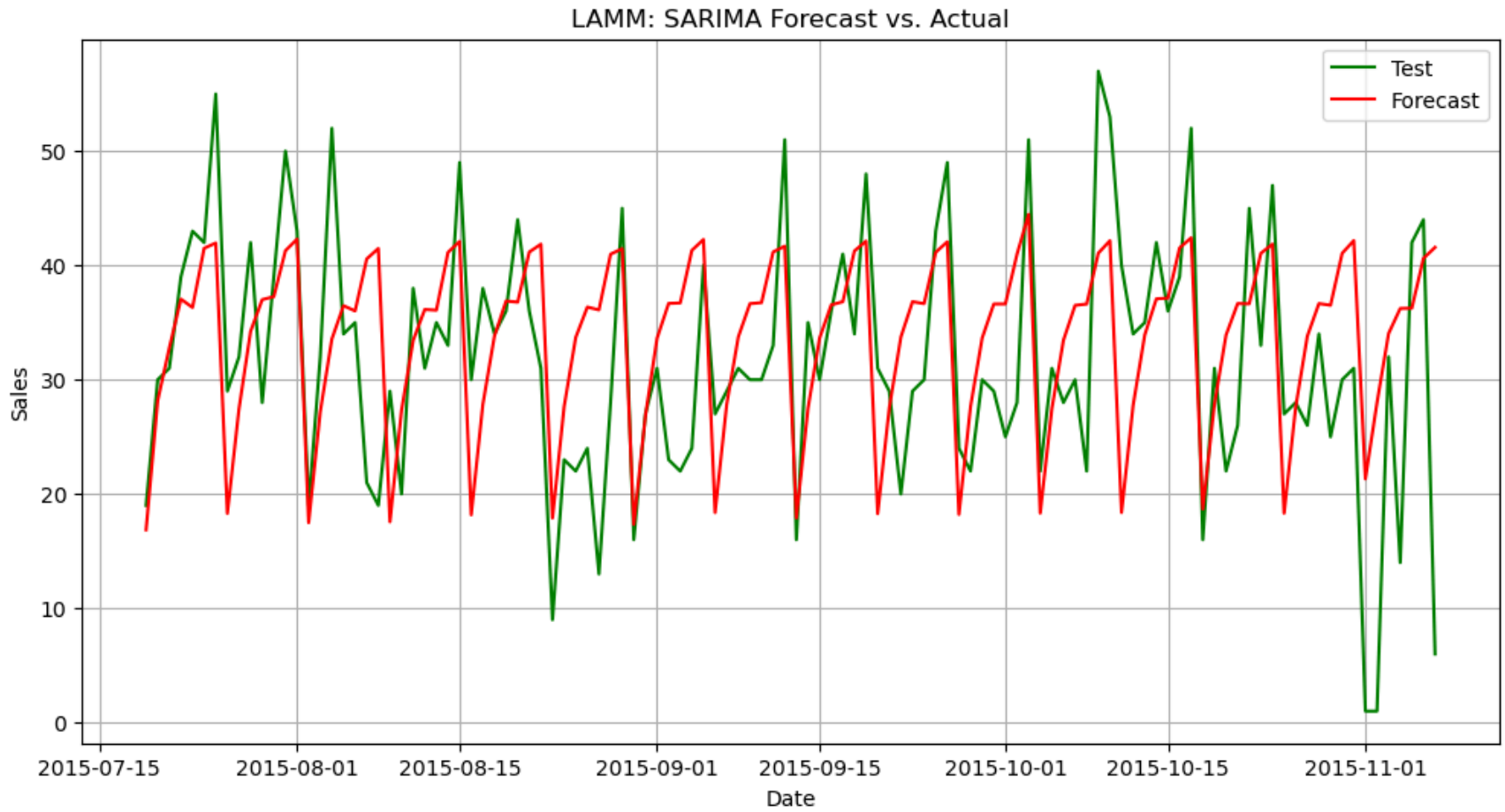

```
GARNELEN Mean Squared Error: 20.55
```



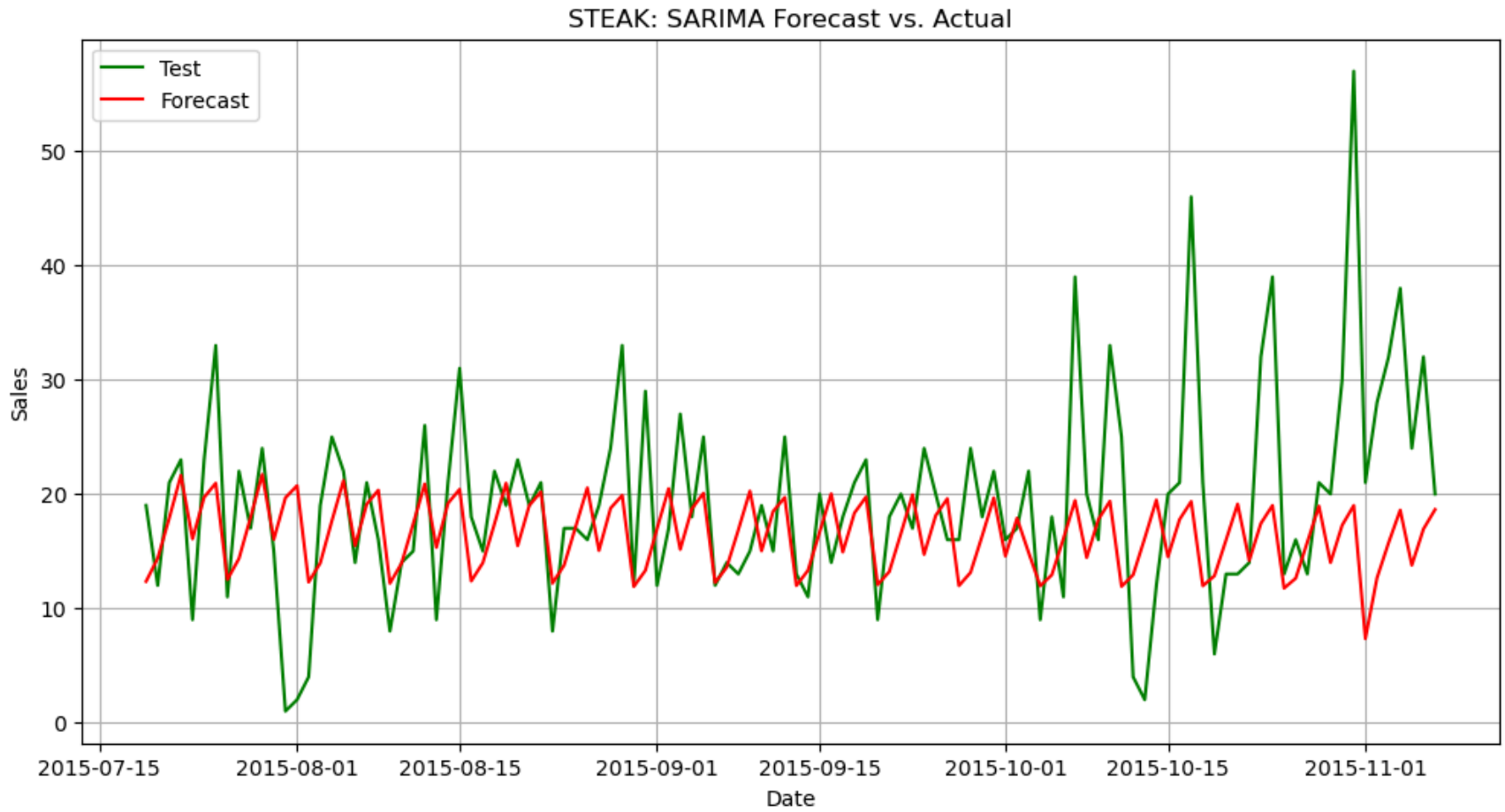

```
HAEHNCHEN Mean Squared Error: 119.78
```




```
KOEFTE Mean Squared Error: 171.68
```




```
LAMM Mean Squared Error: 99.79
```




```
STEAK Mean Squared Error: 76.28
```

Out[507]:
```
{'CALAMARI': 3.56,
 'FISCH': 5.6,
 'GARNELEN': 20.55,
 'HAEHNCHEN': 119.78,
 'KOEFTE': 171.68,
 'LAMM': 99.79,
 'STEAK': 76.28}
```

In [ ]: