Weekly Report 19 (09/02/2024 - 09/06/2024)

Project: Time Series Store Sales

Introduction

This project focuses on predicting daily sales for multiple retail stores using historical time series data. The data incorporates a variety of factors, including store transactions, promotions, holidays, and external economic indicators like oil prices. The project applies advanced time series forecasting methods to capture trends, seasonality, and external influences on store sales, providing valuable insights for more accurate sales forecasting in a dynamic retail environment.

Objective

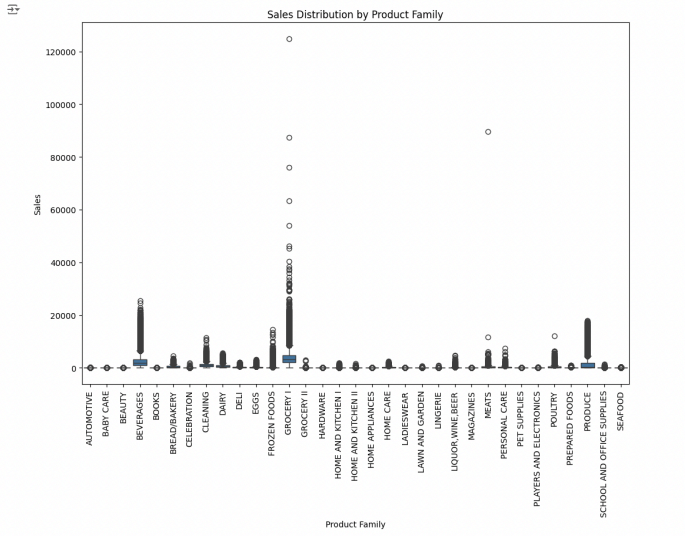
The objective of this project is to develop a robust forecasting model capable of accurately predicting daily sales across numerous retail locations. By leveraging techniques such as statistical methods and machine learning, the goal is to minimize forecasting errors and provide actionable insights into the effects of promotions, holidays, and other variables. The project aims to create a scalable solution that supports strategic decision-making in the retail sector.

Exploratory Data Analysis

<u>Sales Distribution by Product Family</u>

This boxplot presents the distribution of sales across different product families. Categories such as FROZEN FOODS, GROCERY I, and GROCERY II show wider ranges of sales, with outliers indicating exceptional sales events (e.g., peak holiday seasons). This visualization is useful for understanding both the central tendency of sales in different families and the variability, allowing for targeted marketing or inventory decisions based on product family performance.

```
plt.figure(figsize=(12, 8))
sns.boxplot(x='family', y='sales', data=final_merged_data)
plt.title('Sales Distribution by Product Family')
plt.xlabel('Product Family')
plt.ylabel('Sales')
plt.xticks(rotation=90)
plt.show()
```

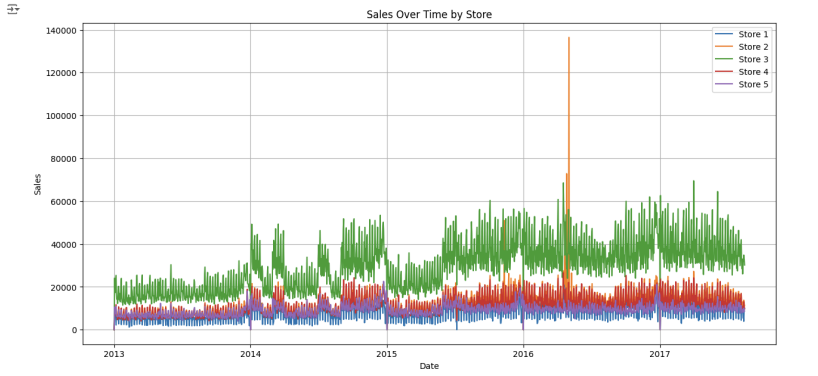


Weekly Report 19 (09/02/2024 - 09/06/2024)

Sales Over Time by Select Stores

This graph highlights sales trends for five specific stores. Store 3 shows a noticeable spike around the year 2016, indicating a potential one-time event or a significant change in store management or promotion. By comparing the sales growth patterns of the selected stores, insights can be drawn on the store-specific strategies that are working, which can be replicated across other locations.

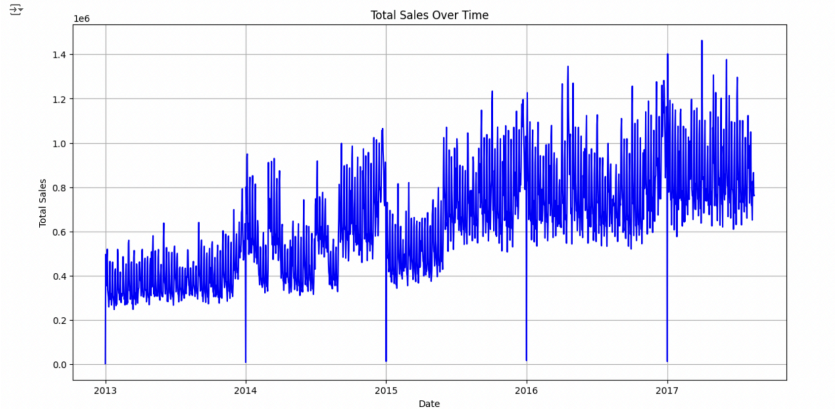
```
# Group by 'date' and 'store_nbr' and sum the sales
sales_by_store_time = final_merged_data.groupby(['date', 'store_nbr'])['sales'].sum().reset_index()

# Plot sales for a few stores
plt.figure(figsize=(15, 7))
for store in sales_by_store_time['store_nbr'].unique()[:5]:  # Plot for the first 5 stores
    store_data = sales_by_store_time[sales_by_store_time['store_nbr'] == store]
    plt.plot(store_data['date'], store_data['sales'], label=f'Store {store}')

plt.title('Sales Over Time by Store')
plt.xlabel('Date')
plt.ylabel('Sales')
plt.legend()
plt.grid(True)
plt.show()
```



Total Sales Over Time

This graph displays the overall sales trends across all stores over the years. The rising sales trend indicates that business is growing steadily. The plot also shows some noticeable dips, likely caused by external factors such as holidays or seasonal shifts. This chart is critical for business forecasting, as it demonstrates the general direction of sales and highlights periods that may require further investigation.

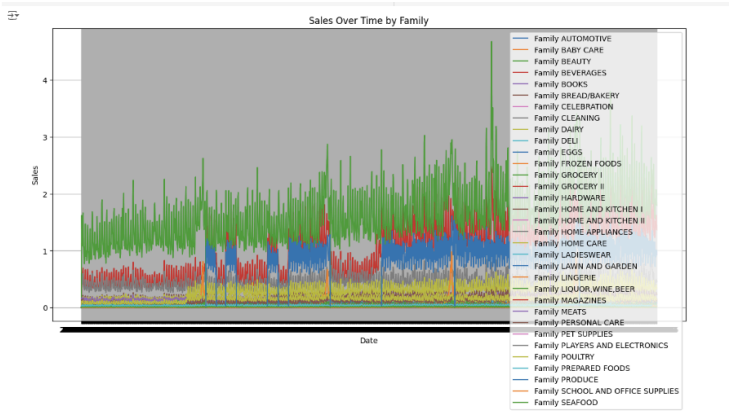
```
# Group the data by 'date' and calculate the total sales for each date
total_sales_by_date = final_merged_data.groupby('date')['sales'].sum()

# Plot the total sales over time
plt.figure(figsize=(14, 7))
plt.plot(total_sales_by_date.index, total_sales_by_date.values, color='blue')
plt.title('Total Sales Over Time')
plt.xlabel('Date')
plt.ylabel('Total Sales')
plt.grid(True)
plt.show()
```



Weekly Report 19 (09/02/2024 - 09/06/2024)

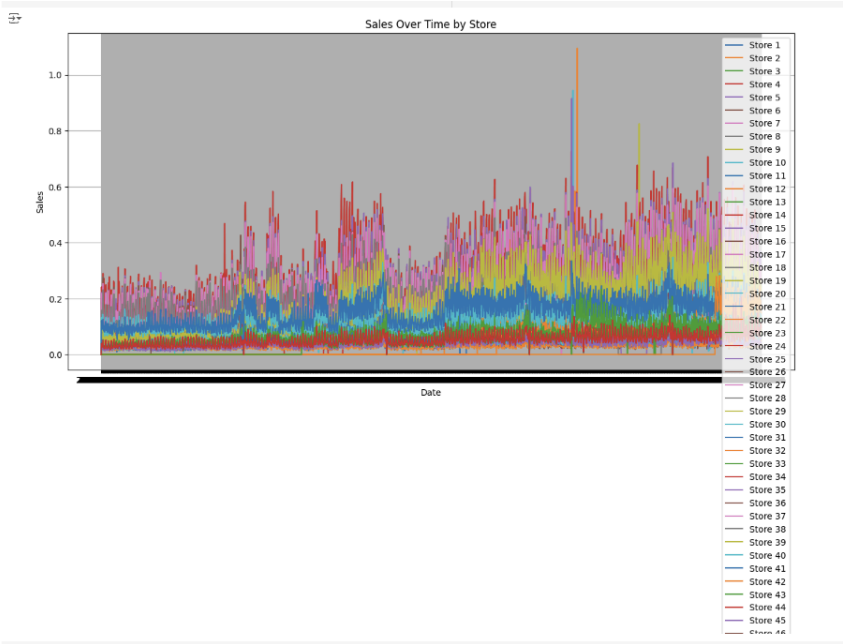
Sales Over Time by Product Family

This visualization demonstrates the sales trends across different product families over time. The x-axis represents the date, and the y-axis shows the normalized sales figures. Categories like AUTOMOTIVE and LIQUOR,WINE,BEER show distinct spikes, indicating seasonal or promotional sales increases. The significant variance across categories suggests that certain product families consistently perform better, likely due to their consumer demand or market positioning.



Sales Over Time by Store

This chart shows sales performance across different stores. Similar to the product family analysis, the x-axis tracks the date, and the y-axis shows normalized sales data. The gradual increase in sales over time is visible across all stores, suggesting overall business growth. However, spikes in specific stores may suggest local promotions, holidays, or events leading to increased sales in those periods. Store 1 and Store 2 appear to dominate in sales compared to others.



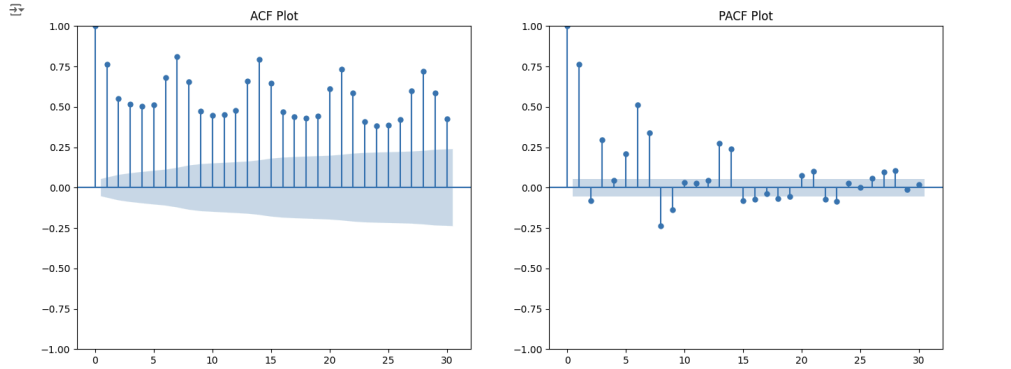
Weekly Report 19 (09/02/2024 - 09/06/2024)

ACF and PACF Plots

These plots provide insights into the time series' underlying patterns. The ACF (Autocorrelation Function) plot shows correlations between the time series values at different lags, with significant spikes suggesting the presence of patterns or dependencies in the data over time. The gradual decline in the ACF indicates non-stationarity, suggesting that differencing may be required to stabilize the series.

The PACF (Partial Autocorrelation Function) plot highlights the direct correlation between a value and its lagged values, without the influence of intermediate lags. The significant spikes at lags 1 and 2 indicate the presence of AR(2) terms, meaning that the current sales figures are influenced by the past two periods.

```
[ ] #--- Pre SARIMA model ---

    import matplotlib.pyplot as plt
    from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

    # Plot ACF and PACF for the training data
    fig, axes = plt.subplots(1, 2, figsize=(16, 6))

    # ACF plot
    plot_acf(daily_sales_train['scaled_sales'], lags=30, ax=axes[0])
    axes[0].set_title('ACF Plot')

    # PACF plot
    plot_pacf(daily_sales_train['scaled_sales'], lags=30, ax=axes[1])
    axes[1].set_title('PACF Plot')

    plt.show()
```



Data Preprocessing

Data Cleaning and Feature Engineering

Handling Missing Values

- Missing data can significantly impact model performance, especially when creating lag features or calculating rolling statistics. To address this, any missing values generated during the lagging or rolling operations were filled with zeros. This ensures that the model can process the data without errors or introducing bias from missing values.

Date Features Extraction

- The date column is converted to a datetime format, allowing for the extraction of key temporal features like year, month, day, day_of_week, and week_of_year. These features provide the model with essential time-related information to understand seasonality and temporal trends in sales.
- Benefit: Capturing periodic trends helps models like SARIMA and CatBoost recognize recurring sales patterns (e.g., weekend spikes or seasonal sales fluctuations).

Fourier Terms for Seasonal Patterns

Weekly Report 19 (09/02/2024 - 09/06/2024)

- Seasonal variations are common in sales data. To capture these patterns, Fourier terms (sin and cos transformations) are introduced for various time periods (day, week, month). These transformations help the model detect repeating cycles, allowing it to account for seasonal peaks or dips in sales.
- Purpose: Fourier terms add predictive power for capturing non-linear cyclical trends.

Lag Features Creation

- Lag features provide the model with information about past sales. For example, 1-day, 3-day, 7-day, and 30-day lagged sales were created. These features enable the model to understand how past performance impacts future sales.
- Additionally, rolling statistics such as the rolling mean and rolling standard deviation for these lag intervals are calculated. This provides a smoother version of past sales and gives the model a sense of sales variability over time.
- Benefit: Lag features and rolling statistics help capture short-term memory, trends, and volatility in sales data.

Sales Change and Trend Features

- To capture the changes in sales from one period to another, percentage change features are calculated (e.g., change from the last day, week, month). These features help the model identify if sales are trending upward or downward over specific time frames.
- Purpose: Tracking percentage changes over different time periods helps the model anticipate sharp shifts in sales based on recent performance.

Store and Family Interaction Terms

- To better understand how different stores perform across different product families, interaction terms between store_nbr and family are created. These terms help capture the relationship between stores and the specific product families they sell.
- Benefit: Interaction terms allow the model to differentiate performance patterns among stores for various product categories.

One-Hot Encoding

- Categorical variables like family are one-hot encoded, converting each product family into a separate binary feature. This ensures that the model can process categorical data without assuming any ordinality.
- Purpose: One-hot encoding allows the model to consider each product family independently, enhancing its ability to differentiate between product categories.

```python
# Convert 'date' to datetime
train_df['date'] = pd.to_datetime(train_df['date'])
test_df['date'] = pd.to_datetime(test_df['date'])

# Extract year, month, day, and other date features
for df in [train_df, test_df]:
    df['year'] = df['date'].dt.year
    df['month'] = df['date'].dt.month
    df['day'] = df['date'].dt.day
    df['day_of_week'] = df['date'].dt.dayofweek
    df['day_of_year'] = df['date'].dt.dayofyear
    df['week_of_year'] = df['date'].dt.isocalendar().week

# Fourier terms for capturing seasonality
def create_fourier_terms(df, period, order):
    t = np.arange(len(df))
    for i in range(1, order + 1):
        df[f'sin_{i}'] = np.sin(2 * np.pi * i * t / period)
        df[f'cos_{i}'] = np.cos(2 * np.pi * i * t / period)
    return df

# Apply Fourier terms to train and test data
train_df = create_fourier_terms(train_df, period=365.25, order=3)
test_df = create_fourier_terms(test_df, period=365.25, order=3)
```

```python
from sklearn.preprocessing import OneHotEncoder

# OneHotEncoding for 'family' column
encoder = OneHotEncoder(sparse=False, handle_unknown='ignore')

# Fit and transform 'family' column for both train and test
train_encoded = encoder.fit_transform(train_df[['family']])
test_encoded = encoder.transform(test_df[['family']])

# Add the encoded columns to train and test data
family_encoded_cols = encoder.get_feature_names_out(['family'])
train_df[family_encoded_cols] = train_encoded
test_df[family_encoded_cols] = test_encoded
```

Weekly Report 19 (09/02/2024 - 09/06/2024)

```
# Create lag features grouped by both family and store_nbr
def create_lag_features(df, lag_days):
    for lag in lag_days:
        df[f'sales_lag_{lag}'] = df.groupby(['store_nbr', 'family'])['scaled_sales'].shift(lag)
    return df

# Apply lag features (1-day, 3-day, and 7-day lags)
train_df = create_lag_features(train_df, lag_days=[1, 3, 7, 14, 30])

# Fill NA values that come from lagging
train_df.fillna(0, inplace=True)

# Add rolling mean and rolling standard deviation
train_df['rolling_mean_3'] = train_df.groupby(['store_nbr', 'family'])['scaled_sales'].shift(1).rolling(window=3).mean()
train_df['rolling_std_3'] = train_df.groupby(['store_nbr', 'family'])['scaled_sales'].shift(1).rolling(window=3).std()
train_df['rolling_mean_7'] = train_df.groupby(['store_nbr', 'family'])['scaled_sales'].shift(1).rolling(window=7).mean()
train_df['rolling_std_7'] = train_df.groupby(['store_nbr', 'family'])['scaled_sales'].shift(1).rolling(window=7).std()
train_df['rolling_mean_14'] = train_df.groupby(['store_nbr', 'family'])['scaled_sales'].shift(1).rolling(window=14).mean()
train_df['rolling_std_14'] = train_df.groupby(['store_nbr', 'family'])['scaled_sales'].shift(1).rolling(window=14).std()
train_df['rolling_mean_30'] = train_df.groupby(['store_nbr', 'family'])['scaled_sales'].shift(1).rolling(window=30).mean()
train_df['rolling_std_30'] = train_df.groupby(['store_nbr', 'family'])['scaled_sales'].shift(1).rolling(window=30).std()

#Trends create
train_df['pct_change_1d'] = train_df.groupby('store_nbr')['sales'].pct_change(1)
train_df['pct_change_3d'] = train_df.groupby('store_nbr')['sales'].pct_change(3)
train_df['pct_change_7d'] = train_df.groupby('store_nbr')['sales'].pct_change(7)
train_df['pct_change_14d'] = train_df.groupby('store_nbr')['sales'].pct_change(14)
train_df['pct_change_30d'] = train_df.groupby('store_nbr')['sales'].pct_change(30)

# Fill NA values in pct_change
train_df.fillna(0, inplace=True)

# Fill NA values in rolling calculations
train_df.fillna(0, inplace=True)

# Drop the original 'family' column from train and test datasets
train_df = train_df.drop('family', axis=1)
test_df = test_df.drop('family', axis=1)
```

```
#Extra Feature Engineering

#seasonal encoding
for df in [train_df]:
    df['sin_year'] = np.sin(2 * np.pi * df['year'] / 12)
    df['cos_year'] = np.cos(2 * np.pi * df['year'] / 12)
    df['sin_month'] = np.sin(2 * np.pi * df['month'] / 12)
    df['cos_month'] = np.cos(2 * np.pi * df['month'] / 12)
    df['sin_day'] = np.sin(2 * np.pi * df['day'] / 31)
    df['cos_day'] = np.cos(2 * np.pi * df['day'] / 31)
    df['sin_day_of_week'] = np.sin(2 * np.pi * df['day_of_week'] / 7)
    df['cos_day_of_week'] = np.cos(2 * np.pi * df['day_of_week'] / 7)
    df['day_of_year_sin'] = np.sin(2 * np.pi * df['day_of_year'] / 365)
    df['day_of_year_cos'] = np.cos(2 * np.pi * df['day_of_year'] / 365)
    df['week_of_year_sin'] = np.sin(2 * np.pi * df['week_of_year'] / 52)
    df['week_of_year_cos'] = np.cos(2 * np.pi * df['week_of_year'] / 52)

#store * family interaction
for col in family_encoded_cols:
    train_df[f'store_{col}_interaction'] = train_df['store_nbr'] * train_df[col]

# Create lag features of sales averaged over stores
train_df['avg_store_sales_lag_3'] = train_df.groupby('store_nbr')['scaled_sales'].transform(lambda x: x.rolling(window=3).mean().shift(1))
train_df['avg_store_sales_lag_7'] = train_df.groupby('store_nbr')['scaled_sales'].transform(lambda x: x.rolling(window=7).mean().shift(1))
train_df['avg_store_sales_lag_30'] = train_df.groupby('store_nbr')['scaled_sales'].transform(lambda x: x.rolling(window=30).mean().shift(1))

#changes seasonality
from statsmodels.tsa.seasonal import seasonal_decompose

decomposition = seasonal_decompose(train_df['scaled_sales'], period=365, model='additive')
train_df['trend'] = decomposition.trend
train_df['seasonal'] = decomposition.seasonal
train_df['residual'] = decomposition.resid

#changes through time
train_df['change_vs_last_day'] = train_df['sales'] - train_df.groupby('store_nbr')['sales'].shift(1)
train_df['change_vs_last_week'] = train_df['sales'] - train_df.groupby('store_nbr')['sales'].shift(7)
train_df['change_vs_last_month'] = train_df['sales'] - train_df.groupby('store_nbr')['sales'].shift(30)
```

Data Splitting

The dataset was divided into two sets: 80% for training and 20% for validation. The training set allows the model to learn patterns, while the validation set assesses its performance on unseen data. This method ensures the model generalizes well to new information and helps prevent overfitting. Also the data is splitted by first 80% of the time and predict the last 20% of the data, this helps the model to understand any underlying trends and pattern to make accurate prediction

```
[ ] from sklearn.model_selection import train_test_split
    from sklearn.metrics import mean_squared_log_error
    from statsmodels.tsa.statespace.sarimax import SARIMAX
    from statsmodels.tsa.arima.model import ARIMA
    import xgboost as xgb
    from catboost import CatBoostRegressor
    import lightgbm as lgb
    import re
    from sklearn.model_selection import GridSearchCV
    from lightgbm import LGBMRegressor

    # Split the data into training and validation sets (80% train, 20% validation)
    split_point = int(len(train_df) * 0.8)

    # Drop 'scaled_sales' as it's the target variable
    X_train = train_df.iloc[:split_point].drop(['sales'], axis=1)
    y_train = train_df.iloc[:split_point]['scaled_sales']
    X_val = train_df.iloc[split_point:].drop(['sales'], axis=1)
    y_val = train_df.iloc[split_point:]['scaled_sales']
```

Weekly Report 19 (09/02/2024 - 09/06/2024)

Model Development and Training

Model 1: ARIMA

The ARIMA model was used for capturing sales patterns by utilizing autoregressive terms, moving average terms, and integrated terms for differencing. The model was trained on aggregated daily sales data, and RMSE was calculated to evaluate performance. However, due to its simplicity, it struggled with capturing complex seasonality in the data.

```
[ ] # ---- Model 1: ARIMA ----

    train_combined = X_train.copy()
    train_combined['scaled_sales'] = y_train

    # Aggregate sales at daily level
    daily_sales_train = train_combined.groupby('date')['scaled_sales'].sum().reset_index()

    # Fit ARIMA model
    arima_model = ARIMA(daily_sales_train['scaled_sales'], order=(1, 1, 2))
    arima_result = arima_model.fit()

    # Prepare validation set similarly (combine X_val and y_val for ARIMA validation)
    val_combined = X_val.copy()
    val_combined['scaled_sales'] = y_val

    # Aggregate sales for validation
    daily_sales_val = val_combined.groupby('date')['scaled_sales'].sum().reset_index()

    # Predict on validation set using ARIMA
    arima_forecast = arima_result.forecast(steps=len(daily_sales_val))

    # Handle potential negative values in predictions
    arima_forecast = np.maximum(arima_forecast, 0)  # Set negative predictions to 0

    # Compute RMSE for ARIMA
    rmsle_arima = np.sqrt(mean_squared_log_error(daily_sales_val['scaled_sales'], arima_forecast))
    print(f"RMSLE for ARIMA: {rmsle_arima}")

⇒ RMSLE for ARIMA: 0.25330137780700684
```

Model 2: SARIMA

To address seasonality, SARIMA was introduced, incorporating seasonal differencing and seasonal autoregressive terms. A grid search was performed to optimize parameters, including seasonal and non-seasonal components. SARIMA showed improved results but still lacked precision for handling multiple interacting factors such as promotions and holidays.

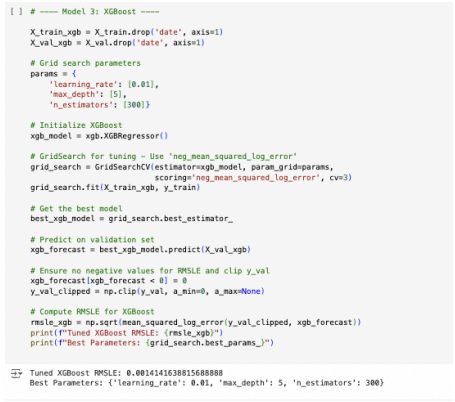
```
[ ] from statsmodels.tsa.stattools import adfuller

    # Perform seasonal differencing with lag 12 for monthly data
    diff_sales = daily_sales_train['scaled_sales'].diff(12).dropna()

    # Perform the ADF test
    adf_test = adfuller(diff_sales)
    print(f'ADF Statistic: {adf_test[0]}')
    print(f'p-value: {adf_test[1]}')

⇒ ADF Statistic: -11.309225625545826
  p-value: 1.2509938031703447e-20
```

```
    # ---- Model 2: SARIMA ----

    import itertools

    # Define ranges for p, d, q parameters
    p = q = range(0, 2)
    d = range(0, 2)

    # Define ranges for seasonal P, D, Q, s parameters (seasonality: 12 months)
    seasonal_pdq = [(x[0], x[1], x[2], 12) for x in list(itertools.product(p, d, q))]

    # Assuming train_combined is available in this scope (if not, you need to define or import it)
    train_combined = X_train.copy()
    train_combined['scaled_sales'] = y_train

    # Aggregate sales at daily level
    daily_sales_train = train_combined.groupby('date')['scaled_sales'].sum().reset_index()

    best_aic = float("inf")
    best_order = None
    best_seasonal_order = None

    # Perform Grid Search over a reduced set of parameters
    for param in itertools.product(p, d, q):
        for param_seasonal in seasonal_pdq:
            try:
                model = SARIMAX(daily_sales_train['scaled_sales'],
                                order=param,
                                seasonal_order=param_seasonal,
                                enforce_stationarity=False,
                                enforce_invertibility=False)
                results = model.fit(maxiter=50, disp=False)

                # Check if the current model is better than the best one
                if results.aic < best_aic:
                    best_aic = results.aic
                    best_order = param
                    best_seasonal_order = param_seasonal
            except Exception as e:
                print(f"Error with parameters {param}, {param_seasonal}: {e}")
                continue

    print(f'Best SARIMA Model: ARIMA{best_order}x{best_seasonal_order}12 - AIC:{best_aic}')

    # Refit SARIMA with the best parameters
    sarima_model = SARIMAX(daily_sales_train['scaled_sales'],
                           order=best_order,
                           seasonal_order=best_seasonal_order,
                           enforce_stationarity=False,
                           enforce_invertibility=False)
    sarima_result = sarima_model.fit(maxiter=100, disp=False)

    # Predict on the validation set
    sarima_forecast = sarima_result.forecast(steps=len(X_val))

    # Ensure no negative values for RMSLE and clip y_val
    sarima_forecast[sarima_forecast < 0] = 0
    y_val_clipped = np.clip(y_val, a_min=0, a_max=None)

    # Compute RMSLE for SARIMA
    rmsle_sarima = np.sqrt(mean_squared_log_error(y_val_clipped, sarima_forecast))
    print(f"RMSLE for SARIMA: {rmsle_sarima}")

⇒ Best SARIMA Model: ARIMA(1, 1, 1)x(0, 0, 1, 12)12 - AIC:3707.33695724917
  RMSLE for SARIMA: 1.8835898494080305
```

Weekly Report 19 (09/02/2024 - 09/06/2024)

Model 3: XGBoost

XGBoost, a tree-based ensemble method, was applied for its ability to capture non-linear relationships and interactions between variables. Hyperparameters such as learning rate, max depth, and number of estimators were tuned using GridSearchCV. The model provided better results than SARIMA, but further improvements were sought.



```
# ---- Model 3: XGBoost ----

X_train_xgb = X_train.drop('date', axis=1)
X_val_xgb = X_val.drop('date', axis=1)

# Grid search parameters
params = {
    'learning_rate': [0.01],
    'max_depth': [5],
    'n_estimators': [300]}

# Initialize XGBoost
xgb_model = xgb.XGBRegressor()

# GridSearch for tuning - Use 'neg_mean_squared_log_error'
grid_search = GridSearchCV(estimator=xgb_model, param_grid=params,
                           scoring='neg_mean_squared_log_error', cv=3)
grid_search.fit(X_train_xgb, y_train)

# Get the best model
best_xgb_model = grid_search.best_estimator_

# Predict on validation set
xgb_forecast = best_xgb_model.predict(X_val_xgb)

# Ensure no negative values for RMSLE and clip y_val
xgb_forecast[xgb_forecast < 0] = 0
y_val_clipped = np.clip(y_val, a_min=0, a_max=None)

# Compute RMSLE for XGBoost
rmsle_xgb = np.sqrt(mean_squared_log_error(y_val_clipped, xgb_forecast))
print(f"Tuned XGBoost RMSLE: {rmsle_xgb}")
print(f"Best Parameters: {grid_search.best_params_}")
```

```
Tuned XGBoost RMSLE: 0.0014141638815688888
Best Parameters: {'learning_rate': 0.01, 'max_depth': 5, 'n_estimators': 300}
```

Model 4: LightGBM

LightGBM, known for its efficiency in handling large datasets and lower memory usage, was selected and tuned using GridSearchCV. The model outperformed previous models in terms of speed and accuracy, with optimized parameters delivering the best RMSE. LightGBM was chosen as the final model due to its scalability and performance.

```
#---- Model 4: LightGBM ----

# Replace special characters in column names (if not already done)
X_train_xgb = X_train_xgb.rename(columns=lambda x: re.sub('[^A-Za-z0-9_]+', '', x))
X_val_xgb = X_val_xgb.rename(columns=lambda x: re.sub('[^A-Za-z0-9_]+', '', x))

# LightGBM Grid search parameters (adjusted)
lgb_params = {
    'learning_rate': [0.01],
    'max_depth': [7],
    'n_estimators': [300],
    'min_data_in_leaf': [70] }

# Initialize LightGBM model
lgb_model = LGBMRegressor()

# GridSearchCV for tuning
grid_search_lgb = GridSearchCV(estimator=lgb_model, param_grid=lgb_params,
                           scoring='neg_mean_squared_log_error', cv=3)
grid_search_lgb.fit(X_train_xgb, y_train)

# Get the best LightGBM model
best_lgb_model = grid_search_lgb.best_estimator_

# Predict on validation set
lgb_forecast = best_lgb_model.predict(X_val_xgb)

# Ensure no negative values and clip y_val
lgb_forecast[lgb_forecast < 0] = 0
y_val_clipped = np.clip(y_val, a_min=0, a_max=None)

# Compute RMSLE for LightGBM
rmsle_lgb = np.sqrt(mean_squared_log_error(y_val_clipped, lgb_forecast))
print(f"Tuned LightGBM RMSLE: {rmsle_lgb}")
print(f"Best Parameters: {grid_search_lgb.best_params_}")
```

```
[LightGBM] [Warning] min_data_in_leaf is set=70, min_child_samples=20 will be ignored. Current value: min_data_in_leaf=70
[LightGBM] [Warning] min_data_in_leaf is set=70, min_child_samples=20 will be ignored. Current value: min_data_in_leaf=70
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.054735 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 12095
[LightGBM] [Info] Number of data points in the train set: 1600473, number of used features: 121
[LightGBM] [Info] Start training from score 0.003004
[LightGBM] [Warning] min_data_in_leaf is set=70, min_child_samples=20 will be ignored. Current value: min_data_in_leaf=70
[LightGBM] [Warning] min_data_in_leaf is set=70, min_child_samples=20 will be ignored. Current value: min_data_in_leaf=70
[LightGBM] [Warning] min_data_in_leaf is set=70, min_child_samples=20 will be ignored. Current value: min_data_in_leaf=70
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.055953 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 12076
[LightGBM] [Info] Number of data points in the train set: 1600473, number of used features: 121
[LightGBM] [Info] Start training from score 0.002685
[LightGBM] [Warning] min_data_in_leaf is set=70, min_child_samples=20 will be ignored. Current value: min_data_in_leaf=70
[LightGBM] [Warning] min_data_in_leaf is set=70, min_child_samples=20 will be ignored. Current value: min_data_in_leaf=70
[LightGBM] [Warning] min_data_in_leaf is set=70, min_child_samples=20 will be ignored. Current value: min_data_in_leaf=70
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.056544 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 11984
[LightGBM] [Info] Number of data points in the train set: 1600474, number of used features: 121
[LightGBM] [Info] Start training from score 0.002207
[LightGBM] [Warning] min_data_in_leaf is set=70, min_child_samples=20 will be ignored. Current value: min_data_in_leaf=70
[LightGBM] [Warning] min_data_in_leaf is set=70, min_child_samples=20 will be ignored. Current value: min_data_in_leaf=70
[LightGBM] [Warning] min_data_in_leaf is set=70, min_child_samples=20 will be ignored. Current value: min_data_in_leaf=70
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.084132 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 12084
[LightGBM] [Info] Number of data points in the train set: 2400710, number of used features: 121
[LightGBM] [Info] Start training from score 0.002632
[LightGBM] [Warning] min_data_in_leaf is set=70, min_child_samples=20 will be ignored. Current value: min_data_in_leaf=70
Tuned LightGBM RMSLE: 0.0011189090611385972
Best Parameters: {'learning_rate': 0.01, 'max_depth': 7, 'min_data_in_leaf': 70, 'n_estimators': 300}
```

Weekly Report 19 (09/02/2024 - 09/06/2024)

Model 5: CatBoost

CatBoost, which handles categorical variables naturally, was also tested. It was tuned similarly using GridSearchCV, and though it performed well, LightGBM was ultimately preferred due to its slightly better performance and faster computation time.

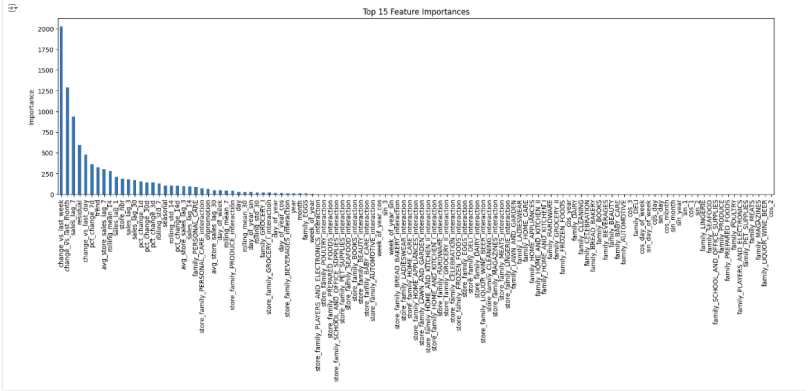
```
[ ] #=== Model 5: CatBoost ===

    # CatBoost Grid search parameters
    cat_params = {
        'learning_rate': [0.01],
        'depth': [9],
        'iterations': [300]}

    # Initialize CatBoost model
    cat_model = CatBoostRegressor(verbose=0)

    # GridSearchCV for tuning
    grid_search_cat = GridSearchCV(estimator=cat_model, param_grid=cat_params,
                                   scoring='neg_mean_squared_log_error', cv=3)
    grid_search_cat.fit(X_train_xgb, y_train)

    # Get the best CatBoost model
    best_cat_model = grid_search_cat.best_estimator_

    # Predict on validation set
    cat_forecast = best_cat_model.predict(X_val_xgb)

    # Ensure no negative values and clip y_val
    cat_forecast[cat_forecast < 0] = 0
    y_val_clipped = np.clip(y_val, a_min=0, a_max=None)

    # Compute RMSLE for CatBoost
    rmsle_cat = np.sqrt(mean_squared_log_error(y_val_clipped, cat_forecast))
    print(f"Tuned CatBoost RMSLE: {rmsle_cat}")
    print(f"Best Parameters: {grid_search_cat.best_params_}")

Tuned CatBoost RMSLE: 0.0013311741582245 91
Best Parameters: {'depth': 9, 'iterations': 300, 'learning_rate': 0.01}
```

Test Data Model Prediction

After evaluating all models, LightGBM was chosen as the final model for sales prediction due to its superior accuracy and ability to handle large datasets efficiently. The model was trained on the full dataset, and feature importance analysis revealed key drivers of sales such as lag features, trends, and store-product family interactions. The model provided actionable insights for predicting sales across retail stores.

submission6.csv — Complete · 2d ago — **2.44706**

```
importances = lgb_model.feature_importances_

# Create a Series for easier visualization, considering all features
feature_importances = pd.Series(importances, index=X_train.columns)

# Sort by importance and select the top 15
top_15_features = feature_importances.sort_values(ascending=False)

# Plot the top 15 features
plt.figure(figsize=(20, 5))
top_15_features.plot(kind='bar')
plt.title('Top 15 Feature Importances')
plt.ylabel('Importance')
plt.show()
```




```
[ ] # Get the names of features with importance greater than 1
    important_features = top_15_features[top_15_features > 1].index.tolist()

    important_features

['change_vs_last_week',
 'change_vs_last_month',
 'sales_lag_7',
 'residual',
 'change_vs_last_day',
 'pct_change_7d',
 'trend',
 'avg_store_sales_lag_7',
 'rolling_mean_14',
 'sales_lag_1',
 'store_nbr',
 'sales_lag_3',
 'sales_lag_30',
 'pct_change_1d',
 'pct_change_30d',
 'pct_change_3d',
 'rolling_std_7',
 'seasonal',
 'rolling_std_14',
 'pct_change_14d',
 'avg_store_sales_lag_3',
 'sales_lag_14',
 'family_PERSONAL_CARE',
 'store_family_PERSONAL_CARE_interaction',
 'onpromotion',
 'avg_store_sales_lag_30',
 'day_of_week',
 'rolling_mean_7',
 'store_family_PRODUCE_interaction',
 'day',
 'rolling_mean_30',
 'day_of_year_sin',
 'rolling_std_30',
 'family_GROCERY_I',
 'store_family_GROCERY_I_interaction',
 'day_of_year',
 'day_of_year_cos',
 'store_family_BEVERAGES_interaction',
 'year',
 'month',
 'family_EGGS',
 'week_of_year']

[ ] #importance feature pct
    (len(important_features)/len(importances))*100

35.8974358974359
```

Weekly Report 19 (09/02/2024 - 09/06/2024)

Feature Reduced for Modeling

Through an iterative process, certain engineered features were identified as not contributing significantly to the model's performance. By removing these unnecessary features, the accuracy of the model improved from an RMSLE score of 2.44706 to 1.87918. This highlights the importance of eliminating redundant or less relevant features to enhance the model's predictive capability and overall efficiency.

```
#Extra Feature Engineering

#seasonal encoding
for df in [train_df]:
    df['sin_year'] = np.sin(2 * np.pi * df['year'] / 12)
    df['cos_year'] = np.cos(2 * np.pi * df['year'] / 12)
    df['sin_month'] = np.sin(2 * np.pi * df['month'] / 12)
    df['cos_month'] = np.cos(2 * np.pi * df['month'] / 12)
    df['sin_day'] = np.sin(2 * np.pi * df['day'] / 31)
    df['cos_day'] = np.cos(2 * np.pi * df['day'] / 31)
    df['sin_day_of_week'] = np.sin(2 * np.pi * df['day_of_week'] / 7)
    df['cos_day_of_week'] = np.cos(2 * np.pi * df['day_of_week'] / 7)
    df['day_of_year_sin'] = np.sin(2 * np.pi * df['day_of_year'] / 365)
    df['day_of_year_cos'] = np.cos(2 * np.pi * df['day_of_year'] / 365)
    df['week_of_year_sin'] = np.sin(2 * np.pi * df['week_of_year'] / 52)
    df['week_of_year_cos'] = np.cos(2 * np.pi * df['week_of_year'] / 52)

#store * family interaction
for col in family_encoded_cols:
    train_df[f'store_{col}_interaction'] = train_df['store_nbr'] * train_df[col]

# Create lag features of sales averaged over stores
train_df['avg_store_sales_lag_3'] = train_df.groupby('store_nbr')['scaled_sales'].transform(lambda x: x.rolling(window=3).mean().shift(1))
train_df['avg_store_sales_lag_7'] = train_df.groupby('store_nbr')['scaled_sales'].transform(lambda x: x.rolling(window=7).mean().shift(1))
train_df['avg_store_sales_lag_30'] = train_df.groupby('store_nbr')['scaled_sales'].transform(lambda x: x.rolling(window=30).mean().shift(1))

#changes seasonality
from statsmodels.tsa.seasonal import seasonal_decompose

decomposition = seasonal_decompose(train_df['scaled_sales'], period=365, model='additive')
train_df['trend'] = decomposition.trend
train_df['seasonal'] = decomposition.seasonal
train_df['residual'] = decomposition.resid

#changes through time
train_df['change_vs_last_day'] = train_df['sales'] - train_df.groupby('store_nbr')['sales'].shift(1)
train_df['change_vs_last_week'] = train_df['sales'] - train_df.groupby('store_nbr')['sales'].shift(7)
train_df['change_vs_last_month'] = train_df['sales'] - train_df.groupby('store_nbr')['sales'].shift(30)
```

Summary of Week 19

What I have learned

For this week, I have learnt much more about the time series in different settings than financial markets and trading. In analyzing sales, there are many factors that need to take into consideration not only pure time series analysis but also many factors like store and different product sales and also trends and seasonality can affect sales. Also not only technical but also this requires some quantitative skills into modeling that I have to smoothed the sales to normalize, how does one feature can affect the whole performance of model, etc.

What is the biggest challenge

The biggest challenge is to find the right method and approach in which in my first attempt, the error rate is significantly high and requires different methods and further analysis. Also by visualizing the time series, not every store or product has the same performance, there will be anomalies and needs to be smoothed for the most objective analysis without taking high value outliers to account.