2024 09 20 - Data Preperation Pipeline

September 20, 2024

1 Corporacion Favorita - New Superb Forecasting Model -

1.1 Data Preperation Pipeline

Made by 4B Consultancy (Janne Heuvelmans, Georgi Duev, Alexander Engelage, Sebastiaan de Bruin) - 2024

In this data pipeline, the data used for forecasting item unit_sales will be processed and finalized before being imported in the machine learning model.
The following steps are made within this notebook:

-0. Import Packages

-1. Load and optimize raw data
-1.1. Functions - Creation of downcast and normalize functions for initial data load
-1.2. Functions - Import raw data from local path
-1.3. Importing raw data

-2. Cleaning data (functions)
-2.1. Return list containing stores with less then 1670 operational days with sales
-2.2. Return list containing stores with cluster=10 in stores df
-2.3. Function to exclude stores with less then 1670 sales days and related to cluster 10

-3. Excluding data based on exploratory data analyses (functions)
-3.1. Function (partly optional) - Excluding stores based on sales units and on cluster type 10
-3.2. Function - Exclude holiday event related to the "Terromoto" volcano event

-4. Enriching datasets for further analysis (functions)
-4.1. Function - Determining holidays per store
-4.2. Function - Determining a count per type of holiday per store
-4.3. Function - Constructing a cartesian sales dataset for each store based on the maximum sales daterange

-5. Constructing final dataset

The structure of this notebook was inspired by: https://hamilton.dagworks.io/en/latest/how-tos/use-in-jupyter-notebook/

1.2 0. Import packages

[1]:
```python
# Importing the libraries
import pandas as pd
import numpy as np
import polars as pl
import os
import sys
import altair as alt
import vegafusion as vf
import sklearn
import time
from datetime import date, datetime, timedelta
from sklearn.pipeline import Pipeline, make_pipeline
```

1.3 1. Load and optimize raw data

1.3.1 1.1. Functions - Creation of downcast and normalize functions for initial data load

Update formatting of features to optimize memory and standardize column names.
Furthermore, get basic information on loaded data and print back to user.

1.1.1. Optimize memory by:
- a) Remove spaces from column names.
- b) Downcasting objects, integers and floats.
- c) Standardize date columns to datetime format.

[2]:
```python
# Data memory optimization function 1 - Removing spaces from the column names
def standardize_column_names(s):
    """Removes spaces from the column names."""

    return s.replace(" ", "")


# Data memory optimization function 2 - Changing datatypes to smaller ones␣
↪(downcasting)
def optimize_memory(df):
    """Optimize memory usage of a DataFrame by converting object columns to␣
↪categorical
    and downcasting numeric columns to smaller types."""

    # Change: Objects to Categorical.
    object_cols = df.select_dtypes(include="object").columns
    if not object_cols.empty:
        print("Change: Objects to Categorical")
        df[object_cols] = df[object_cols].astype("category")
```

```
    # Change: Convert integers to smallest signed or unsigned integer and␣
 ↪floats to smallest.
    for col in df.select_dtypes(include=["int"]).columns:
        if (df[col] >= 0).all():  # Check if all values are non-negative
            df[col] = pd.to_numeric(
                df[col], downcast="unsigned"
            )  # Downcast to unsigned
        else:
            df[col] = pd.to_numeric(df[col], downcast="integer")  # Downcast to␣
 ↪signed

    # Downcast float columns
    for col in df.select_dtypes(include=["float"]).columns:
        df[col] = pd.to_numeric(df[col], downcast="float")

    return df


# Data memory optimization function 4 - Transform date-related columns to␣
 ↪datetime format.
def transform_date_to_datetime(df, i):
    """Transform date-related columns to datetime format."""
    if i != 0:
        if "date" in df.columns:
            print("Change: Transformed 'date' column to Datetime Dtype")
            df["date"] = pd.to_datetime(df["date"]).dt.tz_localize(None).dt.
 ↪floor("D")

    return df
```

1.1.2. Return basic information on each dataframe:
- a) Information on the number of observation and features.
- b) Information on the optimized size of the dataframe.

[3]:
```
# Getting the basic information of the dataframe (number of observations and␣
 ↪features, optimized size)
def df_basic_info(df, dataframe_name):
    print(
        f"The '{dataframe_name}' dataframe contains: {df.shape[0]:,}".
 ↪replace(",", ".")
        + f" observations and {df.shape[1]} features."
    )
    print(
        f"After optimizing by downcasting and normalizing it has optimized size␣
 ↪of    {round(sys.getsizeof(df)/1024/1024/1024, 2)} GB."
    )
```

1.3.2 1.2. Functions - Import raw data from local PATH

Create import data function and apply downcast, normalize functions and give basic information function within the importing function.

[4]:
```
def f_get_data(i=0):

    # Define path.
    c_path = "C:/Users/sebas/OneDrive/Documenten/GitHub/
 ↪Supermarketcasegroupproject/Group4B/data/raw/"

    # c_path = "C:/Users/alexander/Documents/0. Data Science and AI for Experts/
 ↪EAISI_4B_Supermarket/data/raw/"

    # c_path = 'https://www.dropbox.com/scl/fo/4f5xcrzfqlyv3qjzm0kgc/
 ↪AAJkdVC_Wa8NjoTBMwG4gx4?rlkey=gyi9pc4rcmghkzk2wgqyb7y4o&dl=0' Checking if␣
 ↪possible to use c_path of dropbox

    # Identify file.
    v_file = (
        "history-per-year",  # 0
        "holidays_events",  # 1
        "items",  # 2
        "stores",  # 3
    )

    print(f"\nReading file {i}\n")

    # Load data.
    df = (
        pd.read_parquet(c_path + v_file[i] + ".parquet")
        .rename(columns=standardize_column_names)
        .pipe(optimize_memory)
        .pipe(transform_date_to_datetime, i)
    )

    # Return data.
    return df
```

1.3.3 1.3. Importing raw data

Importing parquet files with importing function (downcasting, normalizing and giving basic information)

[5]:
```
# Sales History per year
df_sales = f_get_data(0)

# Holidays
```

```
df_holidays = f_get_data(1)

# Items
df_items = f_get_data(2)

# Stores
df_stores = f_get_data(3)
```

```
Reading file 0


Reading file 1

Change: Objects to Categorical
Change: Transformed 'date' column to Datetime Dtype

Reading file 2

Change: Objects to Categorical

Reading file 3

Change: Objects to Categorical
```

1.4 2. Cleaning data (functions)

1.4.1 2.1. Prepare and clean df_sales

Drop of columns “id”, “year”, “month”, “day” and create a date column based on the columns “year” , “month” and “day”.

[6]:
```
# Prepare df_sales by cleaning up df for merging with holidays by dropping␣
↪unneeded columns
def sales_cleaned(df_sales):
    df_sales["date"] = pd.to_datetime(df_sales[["year", "month", "day"]])
    df_sales = df_sales.drop(columns=["id", "year", "month", "day"])

    return df_sales
```

1.4.2 2.2. Prepare, clean and rename df_items

Renaming columns: “family” to “item_family” and “class” to “item_class”

[7]:
```
# Prepare df_items by cleaning up df by renaming columns for clearity in final␣
↪df
def items_cleaned_renamed(df_items):
```

```
    df_items = df_items.rename(columns={"family": "item_family", "class":␣
↪"item_class"})

    return df_items
```

1.4.3 2.3. Prepare, clean and rename df_stores

Drop of columns "state"
Rename of columns "city" to "store_city", "cluster" to "store_cluster" and "type" to "store_type"

[8]:
```
# Prepare df_stores by cleaning up df by dropping unneeded columns and rename␣
↪columns for clearity in final df
def stores_cleaned_renamed(df_stores):

    df_stores = df_stores.drop(columns=["state", "city"])

    df_stores = df_stores.rename(
        columns={
            "cluster": "store_cluster",
            "type": "store_type",
        }
    )

    return df_stores
```

1.5 3. Excluding data based on exploratory data analyses (functions)

Excluding sales data based on store sales availability
Excluding holiday events related to the "Terromoto" volcano event

1.5.1 3.1. Function (partly optional) - Excluding stores based on sales units and on cluster type 10

3.1.1. Function (optional) - Return list containing stores with less then 1670 operational days with sales
default parameter: store_exclusion_cutoff_number = 1670 days. Based on Exploratory data analysis, 17 stores do not have 1670 days of date present in the sales dataset and either are new stores are were closed for a significant number of days during the timeframe within the sales dataset. It might be functional to make the model only for stores that had sales for all dates (and not new) as that might influence model behavior. This function gives the flexibility as so the user can choose him/herself the cutoff point.

[9]:
```
def stores_exclude_sales_days(df_sales, store_exclusion_cutoff_number=1670):

    # Group the sales data by store and date
    df_sales_grouped = (
        df_sales.groupby(["store_nbr", "date"]).agg({"unit_sales": "sum"}).
↪reset_index()
```

```
    )

    # Count the number of daily sale records per store
    store_count = df_sales_grouped["store_nbr"].value_counts()

    # Get stores with counts less than the exclusion cutoff
    store_count_exclusion = store_count[store_count <␣
↪store_exclusion_cutoff_number]

    # Get the list of store numbers to be excluded
    list_excluded_stores_sales_days = store_count_exclusion.index.tolist()

    return list_excluded_stores_sales_days
```

3.1.2. Function - Return list containing stores with cluster=10 in stores df
From our exploratory data analysis we found that cluster 10 had data issues as it was the only cluster that could was assigned to multiple storetypes. Therefore and because these stores are not part of the top 10 in terms of unit sales, we excluded all stores assigned to cluster 10.

[10]:
```
def stores_exclude_cluster(df_stores, cluster_number=10):

    # Get the list of store numbers that belong to cluster 10

    list_stores_cluster_10 = df_stores[df_stores["cluster"] == cluster_number][
        "store_nbr"
    ].tolist()

    return list_stores_cluster_10
```

3.1.3. Function - Exclude stores with less then X sales days and stores related to cluster 10

[11]:
```
def df_sales_cleaned_stores(df_sales, df_stores,␣
↪store_exclusion_cutoff_number=500):

    # Excluded less then 1670 salesdays
    list_excluded_stores_sales_days = stores_exclude_sales_days(
        df_sales, store_exclusion_cutoff_number
    )

    df_sales = df_sales.drop(
        df_sales[df_sales["store_nbr"].isin(list_excluded_stores_sales_days)].
↪index
    )

    # Cluster 10
    list_stores_cluster_10 = stores_exclude_cluster(df_stores,␣
↪cluster_number=10)
```

```python
    df_sales = df_sales.drop(
        df_sales[df_sales["store_nbr"].isin(list_stores_cluster_10)].index
    )

    return df_sales
```

1.5.2 3.2. Function - Exclude holiday event related to the "Terromoto" volcano event

3.2.1. Function - Create dataframe based on df_holidays with only events containing "Terremoto Manabi"

[12]:
```python
def holiday_filter_vulcano_event(df_holidays, event_substring="Terremoto␣
↪Manabi"):

    # Filter the DataFrame where 'description' contains the event_substring
    df_vulcano_event_filtered = df_holidays[
        df_holidays["description"].str.contains(event_substring)
    ]

    return df_vulcano_event_filtered
```

3.2.2. Function - Exclude the "Terremoto Manabi" from the df_holidays dataframe

[13]:
```python
def df_holidays_cleaned(df_holidays):

    # Exclude holiday_filter_vulcano_event function to return filtered df
    df_vulcano_event_filtered = holiday_filter_vulcano_event(df_holidays)

    # Filter the specific holiday events from the holiday DataFrame
    df_holidays = df_holidays.loc[
        ~df_holidays.index.isin(df_vulcano_event_filtered.index)
    ]

    return df_holidays
```

1.6 4. Enriching datasets for further analysis (functions)

1.6.1 4.1. Function - Determining holidays per store

The holidays dataset contains information on local, regional and national holidays. For each of these types, there is a different key/identifier that corresponds with the stores data found in df_stores (the raw data). To overcome this issue, three separate dataframes are made for each type of holiday where the data is merged (joined) with the stores dataframe. Thereafter, these dataframes are combined as to construct one big dataframe containing all the holidays per store.

4.1.1. Function - Make cleaned versions of the holidays and stores dataframe

[14]:
```python
# Prepare df_holiday and df_stores by cleaning up df for merging with holidays␣
↪by dropping unneeded columns
```

```python
def clean_holidays_stores_prep(df_holidays, df_stores):

    df_holidays_cleaned = df_holidays.drop(
        columns=[
            "description",
            "transferred",
        ]
    )

    df_stores_cleaned = df_stores.drop(columns=["cluster", "type"])

    return df_holidays_cleaned, df_stores_cleaned
```

4.1.2. Function - Create a dataframe with all the local holidays per store

[15]:
```python
def holidays_prep_local(df_holidays, df_stores):

    df_holidays_cleaned, df_stores_cleaned = clean_holidays_stores_prep(
        df_holidays, df_stores
    )

    # select locale 'Local' from holiday df and merge with city stores df
    df_holidays_local = df_holidays_cleaned[df_holidays_cleaned["locale"] ==␣
 ↪"Local"]

    df_holidays_prep_local = df_holidays_local.merge(
        df_stores_cleaned, left_on="locale_name", right_on="city", how="left"
    )

    return df_holidays_prep_local
```

4.1.3. Function - Create a dataframe with all the regional holidays per store

[16]:
```python
def holidays_prep_regional(df_holidays, df_stores):

    df_holidays_cleaned, df_stores_cleaned = clean_holidays_stores_prep(
        df_holidays, df_stores
    )

    # select locale 'Regional' from holiday df and merge with state stores df
    df_holidays_regional = df_holidays_cleaned[
        df_holidays_cleaned["locale"] == "Regional"
    ]

    df_holidays_prep_regional = df_holidays_regional.merge(
        df_stores_cleaned, left_on="locale_name", right_on="state", how="left"
    )
```

```
    return df_holidays_prep_regional
```

4.1.4. Function - Create a dataframe with all the national holidays per store

[17]:
```
def holidays_prep_national(df_holidays, df_stores):

    df_holidays_cleaned, df_stores_cleaned = clean_holidays_stores_prep(
        df_holidays, df_stores
    )

    # Select locale 'Regional' from holiday df and merge with national stores df
    df_holidays_national = df_holidays_cleaned[
        df_holidays_cleaned["locale"] == "National"
    ]

    # Create extra column for merge on "Ecuador"
    df_stores_cleaned["national_merge"] = "Ecuador"

    df_holidays_prep_national = df_holidays_national.merge(
        df_stores_cleaned, left_on="locale_name", right_on="national_merge",␣
 ↪how="left"
    )

    # Drop newly created column national_merge, not needed further
    df_holidays_prep_national = df_holidays_prep_national.drop(
        columns=["national_merge"]
    )

    return df_holidays_prep_national
```

4.1.5. Function - Create a dataframe that merges all the separate dataframe for each type of holiday and store combination

[18]:
```
def holidays_prep_merged(df_holidays, df_stores):

    # Load prep functions from local, Regional and National df's
    df_holidays_prep_local = holidays_prep_local(df_holidays, df_stores)

    df_holidays_prep_regional = holidays_prep_regional(df_holidays, df_stores)

    df_holidays_prep_national = holidays_prep_national(df_holidays, df_stores)

    # Combine local, regional and national dataframes into 1 merged dataframe
    df_holidays_merged = pd.concat(
        [df_holidays_prep_local, df_holidays_prep_regional,␣
 ↪df_holidays_prep_national]
    )
```

```python
    # Clean df_holidays_merged by dropping "locale_name", "city", "state"
    df_holidays_merged = df_holidays_merged.drop(
        columns=["locale_name", "city", "state"]
    )

    # Rename 'type' of holiday to 'holiday_type'
    df_holidays_merged = df_holidays_merged.rename(
        columns={"type": "holiday_type", "locale": "holiday_locale"}
    )

    return df_holidays_merged
```

1.6.2 4.2. Function - Determining a count per type of holiday per store

The dataframe resulting from the function described in 4.1. gives duplicate values because there sometimes are multiple holidays on one date. Duplicate values per date would result in multiple sales rows for each date, making it not workable. Therfore, we transform the holiday and stores combination to contain 3 columns (for each type of holiday, namely, local, regional and national) that count the amount of holidays found for a specific date. Thereby we create a unique list of date and store combinations for all the holidays within the dataset.

4.2.1. Function - Creating unique combination of store and date with three count columns for each type of holiday

[19]:
```python
def holidays_prep_merged_grouped(df_holidays, df_stores):

    # Merge the holiday dataframes and clean the merged dataframe
    df_holidays_merged = holidays_prep_merged(df_holidays, df_stores)

    # Group by date and store_nbr and count the number of holidays per date per␣
 ↪store
    df_holidays_merged_grouped = df_holidays_merged.pivot_table(
        index=["date", "store_nbr"],
        columns="holiday_locale",
        values="holiday_type",
        aggfunc="count",
        observed=True,
    ).reset_index()

    # Remove the name of the columns
    df_holidays_merged_grouped.columns.name = None

    # Fill NaN values with 0
    df_holidays_merged_grouped = df_holidays_merged_grouped.fillna(0)

    # Convert the count columns to Int8-dtype (note the capital 'I'). This␣
 ↪dtype can handle null values, needed to prevent float64 from the merge in␣
 ↪Step 6
```

```python
    # Rename the columns to holiday_local_count,  holiday_regional_count,␣
↪holiday_national_count
    df_holidays_merged_grouped = df_holidays_merged_grouped.astype(
        {"Local": "Int8", "Regional": "Int8", "National": "Int8"}
    ).rename(
        columns={
            "Local": "holiday_local_count",
            "Regional": "holiday_regional_count",
            "National": "holiday_national_count",
        }
    )

    # Let's do an inner join with the original data to get the original date␣
↪and store_nbr combinations back. Therefore we need to make another dataframe.
    df_holidays_merged_grouped_inner = holidays_prep_merged(df_holidays,␣
↪df_stores)
    df_holidays_merged_grouped_inner = (
        df_holidays_merged_grouped_inner.groupby(["date", "store_nbr"])
        .size()
        .reset_index()
        .drop(columns=0)
    )

    df_holidays_merged_grouped = df_holidays_merged_grouped.merge(
        df_holidays_merged_grouped_inner, on=["date", "store_nbr"], how="inner"
    )

    print(
        f"In the orignal unioned holiday dataframe, df_holidays_merged we found␣
↪(including duplicates) {df_holidays_merged.shape[0]} rows"
    )
    print(
        f"In our new adjusted dataframe we have {df_holidays_merged_grouped.
↪shape[0]} rows"
    )
    print(
        f"Thus, we have removed {df_holidays_merged.shape[0] -␣
↪df_holidays_merged_grouped.shape[0]} rows"
    )

    # Might want to filter out the holiday dates that will never be in de␣
↪salesdate range. However, they will be left out anyway when joining with the␣
↪sales data.
    return df_holidays_merged_grouped
```

4.2.2. Function - Filling in NA values for each count column whenever no holiday could be found for a specific holiday date and store combination

```
[20]: #  Fill newly created NaN columns, due to holiday join, with 'no' on thates␣
      ↪where there are now holidays
      def holidays_fill_zero_normal(df):
          """
          Fills the NaN values with 0 for all columns "holiday_local_count",␣
      ↪"holiday_regional_count", "holiday_national_count", in the combined␣
      ↪dataframe.
          It will only fill the columns that are in the original dataframe and not in␣
      ↪the holiday dataframe.
          """

          columns_to_fill = [
              "holiday_local_count",
              "holiday_regional_count",
              "holiday_national_count",
          ]

          df[columns_to_fill] = df[columns_to_fill].fillna(0).astype("int8")

          return df
```

1.6.3 4.3. Function - Constructing a cartesian sales dataset for each store based on the maximum sales daterange

The df_sales dataset contains unit sales data for each store but not all stores have data for each date. To overcome this and make sure each date is present for each store we construct a new dataframe based on the minimum- and maximum date found within the sales dataframe. The result is thus a sales dataframe with each date, store and item combination for the whole timerange.

```
[21]: def filling_dates_cartesian(df):

          # Print first and last date of df
          print(f'First date in df: {df["date"].min()}')
          print(f'Last date in df:  {df["date"].max()}')

          # Calculate memory size and shape size of start df
          df_mem_start = sys.getsizeof(df)
          df_shape_start = df.shape[0] / 1e6
          print(
              f"Start size of df_sales:     {round(df_mem_start/1024/1024/1024, 2)}␣
      ↪GB and start observations:     {round(df_shape_start, 1)} million."
          )

          # Create a complete date range for the entire dataset, it's a datetimeindex␣
      ↪object
          all_dates = pd.date_range(start=df["date"].min(), end=df["date"].max(),␣
      ↪freq="D")
```

```
# Create a multi-index from all possible combinations of 'item_nbr' and␣
↪'date'
all_combinations = pd.MultiIndex.from_product(
    [df["store_nbr"].unique(), df["item_nbr"].unique(), all_dates],
    names=["store_nbr", "item_nbr", "date"],
)

print(
    f"The multi-index (all_combinations of store, date and item) for the␣
↪minimum and maximum dates found result in {round(all_combinations.shape[0]/
↪1e6,1)} million rows, this is the amount of rows we expect in the final␣
↪dataframe."
)

#␣
↪-----------------------------------------------------------------------------------------

# Check for duplicates in the combination of 'store_nbr', 'item_nbr', and␣
↪'date'
# This method is based on boolean indexing, when there's a true value for␣
↪the duplicated method, it will return those rows to the duplicate_rows␣
↪variable
duplicate_rows = df[
    df.duplicated(subset=["store_nbr", "item_nbr", "date"], keep=False)
]
if not duplicate_rows.empty:
    print(
        "Warning: Duplicate entries found in the combination of␣
↪'store_nbr', 'item_nbr', and 'date'."
    )
    print(f"Total dublicate rows {duplicate_rows.shape[0]}")
    print("-" * 71)

#␣
↪-----------------------------------------------------------------------------------------

# Reindex the original DataFrame to include all combinations of␣
↪'store_nbr', 'item_nbr', and 'date'
df_reindexed = df.set_index(["store_nbr", "item_nbr", "date"]).reindex(
    all_combinations
)

# Reset the index to turn the multi-index back into regular columns
df_sales_cartesian = df_reindexed.reset_index()

# Calculate memory size and shape size of final end df
```

```
    df_mem_end = sys.getsizeof(df_sales_cartesian)
    df_mem_change_perc = ((df_mem_end - df_mem_start) / df_mem_start) * 100
    df_mem_change = df_mem_end - df_mem_start

    df_shape_end = df_sales_cartesian.shape[0] / 1e6
    df_shape_change_perc = ((df_shape_end - df_shape_start) / df_shape_start) *␣
↪100
    df_shape_change = df_shape_end - df_shape_start

    print(
        f"Final size of the dataframe is:     {round(df_mem_end/1024/1024/1024,␣
↪2)} GB and end observations:       {round(df_shape_end, 1)} million."
    )
    print(
        f"Change in size of the dataframe is: {round(df_mem_change_perc, 2)} %␣
↪and observations:           {round(df_shape_change_perc, 2)}     %."
    )
    print(
        f"Increased size of the dataframe is: {round(df_mem_change/1024/1024/
↪1024, 2)} GB and increased observations: {round(df_shape_change, 1)} million.
↪"
    )

    return df_sales_cartesian
```

1.7 5. Constructing final dataset

In this step all the datasets will be merged together.

[22]:
```
# Merge datasets
def merge_datasets(df_sales, df_items, df_stores, df_holidays):

    # Basic information of loaded data
    print(
        "Step 1 - Importing, downcasting and normalizing data and optimizing␣
↪memory, the following data has been imported."
    )
    df_basic_info(df_sales, "df_sales")
    print("""""")
    df_basic_info(df_items, "df_items")
    print("""""")
    df_basic_info(df_stores, "df_stores")
    print("""""")
    df_basic_info(df_holidays, "df_holidays")
    print("-" * 100)

    # Sales prep
```

```python
    print(
        "Step 2 - Cleaning sales data and making a cartesian product of the␣
↪sales data and the minimum and maximum dates found in the data."
    )
    df_sales = sales_cleaned(df_sales)
    df_sales = df_sales_cleaned_stores(df_sales, df_stores)
    df_sales_cartesian = filling_dates_cartesian(df_sales)

    print("-" * 100)

    # Holidays prep
    print(
        "Step 3 - Cleaning holiday data and counting the number of holidays per␣
↪date per store for each type of holiday (national, regional, local)."
    )
    df_holidays = df_holidays_cleaned(df_holidays)
    df_holidays_merged_grouped = holidays_prep_merged_grouped(df_holidays,␣
↪df_stores)
    print("-" * 100)

    # Stores prep
    print(
        "Step 4 - Cleaning stores data (read: dropping unnecessary columns and␣
↪renaming columns for clarity)."
    )
    df_stores = stores_cleaned_renamed(df_stores)
    print("-" * 100)

    # Items prep
    print(
        "Step 5 - Cleaning items data  (read: dropping unnecessary columns and␣
↪renaming columns for clarity)."
    )
    df_items = items_cleaned_renamed(df_items)
    print("-" * 100)

    # Holidays merge on sales
    print(
        "Step 6 - Adding holiday data to our cartesian product of sales data␣
↪(with store, item and date combinations) and cleaning up null values for␣
↪count of three holiday columns."
    )

    df_merged = df_sales_cartesian.merge(
        df_holidays_merged_grouped, on=["date", "store_nbr"], how="left"
    )
```

```
    df_merged = holidays_fill_zero_normal(df_merged)
    print("-" * 100)

    # Stores merged with sales+holidays
    print(
        "Step 7 - Adding holiday data to our cartesian product of sales data␣
↪(with store, item and date combinations) and cleaning up null values for␣
↪count of holiday columns."
    )
    df_merged = df_merged.merge(df_stores, on="store_nbr", how="left")

    print("-" * 100)

    # Change the dtype for item_nbr from uint32 to int32, during testing we␣
↪found that the merge was not working properly with uint32
    df_merged["item_nbr"] = df_merged["item_nbr"].astype(int)
    df_items["item_nbr"] = df_items["item_nbr"].astype(int)

    # Items merged with sales+holidays+stores
    print(
        "Step 8 - Adding items data to our cartesian product of sales data␣
↪(with store, item and date combinations) and cleaning up null values for␣
↪count of holiday columns. Remember, in our last step we added a lot of store␣
↪information as well"
    )
    df_final = df_merged.merge(df_items, on="item_nbr", how="left")
    print("-" * 100)

    # Print some referential integrity checks to make sure we have the same␣
↪amount of rows
    print(
        f"The amount of rows in the sales dataframe was {df_sales.shape[0] /
↪1_000_000:.2f} million."
    )
    print(
        f"After making a cartesian product with date, store and item we had a␣
↪total of {df_sales_cartesian.shape[0]/1_000_000:.2f} million rows."
    )
    print(
        f"After mergin with the holidays, stores, and items we have {df_final.
↪shape[0]/1_000_000:.2f} million rows"
    )
    print(
        f"The difference between the incoming and outgoing data from this␣
↪function is {df_sales.shape[0] - df_final.shape[0]} rows"
```

```
    )
    print(
        f'If we compare the outgoing dataframe called "df_final" with the␣
↪cartesian product of sales data and dates we see that the difference is␣
↪{df_sales_cartesian.shape[0] - df_final.shape[0]} rows'
    )
    print(
        f"If the difference is 0, we have a perfect match and we can continue␣
↪with the next steps."
    )

    # f"Final size of the dataframe is:     {round(df_mem_end/1024/1024/1024,␣
↪2)} GB and end observations:       {round(df_shape_end, 1)} million."
    return df_final
```

[23]:
```
# df_sales = df_sales[(df_sales["store_nbr"] == 1)]

# df_sales.info()
```

[24]:
```
df_final = merge_datasets(df_sales, df_items, df_stores, df_holidays)  # --> 2.
↪44 GB
```

```
Step 1 - Importing, downcasting and normalizing data and optimizing memory, the
following data has been imported.
The 'df_sales' dataframe contains: 125.497.040 observations and 8 features.
After optimizing by downcasting and normalizing it has optimized size of    2.1
GB.

The 'df_items' dataframe contains: 4.100 observations and 4 features.
After optimizing by downcasting and normalizing it has optimized size of    0.0
GB.

The 'df_stores' dataframe contains: 54 observations and 5 features.
After optimizing by downcasting and normalizing it has optimized size of    0.0
GB.

The 'df_holidays' dataframe contains: 350 observations and 6 features.
After optimizing by downcasting and normalizing it has optimized size of    0.0
GB.
--------------------------------------------------------------------------------
--------------------
Step 2 - Cleaning sales data and making a cartesian product of the sales data
and the minimum and maximum dates found in the data.
First date in df: 2013-01-01 00:00:00
Last date in df:  2017-08-15 00:00:00
Start size of df_sales:     2.84 GB and start observations:     113.0 million.
The multi-index (all_combinations of store, date and item) for the minimum and
maximum dates found result in 320.2 million rows, this is the amount of rows we
```

```
expect in the final dataframe.
Final size of the dataframe is:     5.67 GB and end observations:       320.2
million.
Change in size of the dataframe is: 99.45 % and observations:           183.43
%.
Increased size of the dataframe is: 2.83 GB and increased observations: 207.2
million.
--------------------------------------------------------------------------------
--------------------
Step 3 - Cleaning holiday data and counting the number of holidays per date per
store for each type of holiday (national, regional, local).
In the orignal unioned holiday dataframe, df_holidays_merged we found (including
duplicates) 8276 rows
In our new adjusted dataframe we have 8091 rows
Thus, we have removed 185 rows
--------------------------------------------------------------------------------
--------------------
Step 4 - Cleaning stores data (read: dropping unnecessary columns and renaming
columns for clarity).
--------------------------------------------------------------------------------
--------------------
Step 5 - Cleaning items data  (read: dropping unnecessary columns and renaming
columns for clarity).
--------------------------------------------------------------------------------
--------------------
Step 6 - Adding holiday data to our cartesian product of sales data (with store,
item and date combinations) and cleaning up null values for count of three
holiday columns.
--------------------------------------------------------------------------------
--------------------
Step 7 - Adding holiday data to our cartesian product of sales data (with store,
item and date combinations) and cleaning up null values for count of holiday
columns.
--------------------------------------------------------------------------------
--------------------
Step 8 - Adding items data to our cartesian product of sales data (with store,
item and date combinations) and cleaning up null values for count of holiday
columns. Remember, in our last step we added a lot of store information as well
--------------------------------------------------------------------------------
--------------------
The amount of rows in the sales dataframe was 112.97 million.
After making a cartesian product with date, store and item we had a total of
320.20 million rows.
After mergin with the holidays, stores, and items we have 320.20 million rows
The difference between the incoming and outgoing data from this function is
-207227074 rows
If we compare the outgoing dataframe called "df_final" with the cartesian
product of sales data and dates we see that the difference is 0 rows
```

If the difference is 0, we have a perfect match and we can continue with the next steps.

[25]:
```
# Sebastiaan code -
# What about the "onpromotion" column, seems that it has a lot of NaN values.␣
↪Are these quality issues or is just that there's no promotion.
# This issue didn't arrive after merging, it was there from the beginning (in␣
↪the df_sales dataframe).
# You would expect that if there's no promotion going on the value to be "False"

# df_sales1 = sales_cleaned(df_sales)

# df_sales1_unique = df_sales1["onpromotion"].unique()
```

2 4. Data Manipulation

X.X. Count nulls per column

[26]:
```
null_counts = df_final.isnull().sum()

type(null_counts)
```

[26]: pandas.core.series.Series

[27]:
```
df_final.info()
# Count nulls per column
null_counts = df_final.isnull().sum()

# Print results
for column, count in null_counts.items():
    print(f"Column '{column}' has {count} null values.")
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 320200096 entries, 0 to 320200095
Data columns (total 13 columns):
 #   Column                  Dtype
---  ------                  -----
 0   store_nbr               uint8
 1   item_nbr                int32
 2   date                    datetime64[ns]
 3   unit_sales              float32
 4   onpromotion             boolean
 5   holiday_local_count     int8
 6   holiday_national_count  int8
 7   holiday_regional_count  int8
 8   store_type              category
 9   store_cluster           uint8
 10  item_family             category
```

```
 11  item_class               uint16
 12  perishable               uint8
dtypes: boolean(1), category(2), datetime64[ns](1), float32(1), int32(1),
int8(3), uint16(1), uint8(3)
memory usage: 8.3 GB
Column 'store_nbr' has 0 null values.
Column 'item_nbr' has 0 null values.
Column 'date' has 0 null values.
Column 'unit_sales' has 207227074 null values.
Column 'onpromotion' has 226982961 null values.
Column 'holiday_local_count' has 0 null values.
Column 'holiday_national_count' has 0 null values.
Column 'holiday_regional_count' has 0 null values.
Column 'store_type' has 0 null values.
Column 'store_cluster' has 0 null values.
Column 'item_family' has 0 null values.
Column 'item_class' has 0 null values.
Column 'perishable' has 0 null values.
```

4.2: Detect negative values

- Action: Delete unit_sales if values are lower than zero –> N/A

To-do: do we want do make negative –> 0 or delete values –> Inpute later?

[28]:
```python
def negative_sales_cleaned(df):

    # Check the number of negative values before replacement
    before_replacement = (df["unit_sales"] < 0).sum()
    print(f"Number of negative values before replacement: {before_replacement}")

    # Create a boolean mask for the negative sales rows to create a 'boolean␣
↪flag-list' containing all negative rows, used to filter full df_sales df
    negative_sales_mask = df["unit_sales"] < 0

    # Use the mask to update the flagged 'unit_sales' column in the original␣
↪DataFrame
    df.loc[negative_sales_mask, "unit_sales"] = df.loc[
        negative_sales_mask, "unit_sales"
    ].where(df.loc[negative_sales_mask, "unit_sales"] >= 0, np.nan)

    # Check the number of negative values after replacement
    after_replacement = (df["unit_sales"] < 0).sum()
    print(f"Number of negative values after replacement: {after_replacement}")

    return df
```

4.3 Define new, old and closed stores

- Condition: sales for all items a given store and date are NA

- Action: Impute with 0

Label Variable for atributing numbers to store status:

- `OPEN = 0`
- `NEW = 2`
- `CLOSED = 4`
- `OLD = 6`
- `NEVER_OPENED = 8`

To-do: Write in polars??

To-do: Can the ML model run with NaN values? Or need the new / old stores also need to inputed with 0

[29]:
```python
def merge_store_status(df):

    # Label Variable for atributing numbers to store status, to save memory in␣
↪df
    OPEN = 0
    NEW = 2
    CLOSED = 4
    OLD = 6
    NEVER_OPENED = 8

    # Group by store and date, then sum sales
    df_grouped = (
        df.groupby(["store_nbr", "date"]).agg({"unit_sales": "sum"}).
↪reset_index()
    ).reset_index()

    # Sort by store and date
    df_grouped = df_grouped.sort_values(["store_nbr", "date"])

    # Create a new column for store status, label al stores as 'open' by␣
↪default and make dtype in8
    df_grouped["store_status"] = np.int8(OPEN)

    # Find the first and last day with sales for each store
    first_sale_date = (
        df_grouped[df_grouped["unit_sales"] > 0].groupby("store_nbr")["date"].
↪min()
    )
```

```
    last_sale_date = (
        df_grouped[df_grouped["unit_sales"] > 0].groupby("store_nbr")["date"].max()
    )

    # Loop trhough stores by lapeling them as 'NEW', 'CLOSED', 'OLD' or 'NEVER_OPENED' based on first sale date and last sale date
    for store in df_grouped["store_nbr"].unique():
        store_data = df_grouped[df_grouped["store_nbr"] == store]

        if store in first_sale_date.index:
            first_date = first_sale_date[store]
            last_date = last_sale_date[store]

            # Mark as 'NEW' before first sale date
            df_grouped.loc[
                (df_grouped["store_nbr"] == store) & (df_grouped["date"] < first_date),
                "store_status",
            ] = NEW
            # --> To-do: Do we call this  not opened' or a 'new store'?

            # Mark as 'closed' after first sale date if sales are 0
            df_grouped.loc[
                (df_grouped["store_nbr"] == store)
                & (df_grouped["date"] > first_date)
                & (df_grouped["unit_sales"] == 0),
                "store_status",
            ] = CLOSED

            # Mark as 'OLD' after last sale date
            df_grouped.loc[
                (df_grouped["store_nbr"] == store) & (df_grouped["date"] > last_date),
                "store_status",
            ] = OLD

        else:
            # If a store never had any sales, mark all dates as 'NEVER_OPENED' --> no records?
            df_grouped.loc[df_grouped["store_nbr"] == store, "store_status"] = (
                NEVER_OPENED
            )

    # Merging store_status on df_sales
    df = df.merge(
        df_grouped[["store_nbr", "date", "store_status"]],
```

```python
        left_on=["store_nbr", "date"],
        right_on=["store_nbr", "date"],
        how="left",
    )

    # Get list of NEW stores at 01-01-2013 and OPEN stores at 02-01-2013
    mask_new = (df["store_status"] == NEW) & (df["date"] == "2013-01-01")
    mask_open = (df["store_status"] == OPEN) & (df["date"] == "2013-01-02")

    # Get list of thores that meet both the coditions of NEW AT 01-01-2013 and␣
↪OPEN at 02-01-2013
    stores_new = set(df[mask_new]["store_nbr"].unique())
    stores_open = set(df[mask_open]["store_nbr"].unique())
    stores_status_change = stores_new.intersection(stores_open)

    # Change status of stores that are NEW on 01-01-2013 but OPEN on 02-01-2013␣
↪to CLOSED on 01-01-2013
    df.loc[
        (df["store_nbr"].isin(stores_status_change)) & (df["date"] ==␣
↪"2013-01-01"),
        ["store_status"],
    ] = [CLOSED]

    # Using a mask to flag al 'CLOSED' = 4 stores and impute 'closed' stores␣
↪with 0, not opened stores stay NA/NaN
    mask = df["store_status"] == CLOSED
    df.loc[mask, "unit_sales"] = 0

    print("-" * 72)
    print(
        f"Size of df:     {round(sys.getsizeof(df)/1024/1024/1024, 2)} GB and␣
↪end observations:       {round(df.shape[0] / 1e6, 1)} million."
    )
    print("- " * 36)
    print("df_grouped store_status value counts:")
    print(df_grouped["store_status"].value_counts())

    print("-" * 72)

    return df
```

4.4 New product –> !Polars function!

- Before the very first sale of an item, all observations are kept as NA
- After the very first sale of an item, we go to step 3:

Label Variable for atributing numbers to store status, to save memory in df - EXISTING = 1 - NEW = 3 - OLD = 7 - NEVER_SOLD = 9

TO-DO: Add polars to requirements.txt

[30]:
```
# <PATH>.\venv_case_project\Scripts\activate
# pip install polars
```

[31]:
```
import polars as pl  # later to import packages step at 0


def merge_item_status_polars(df_pandas):

    # Record the start time of the function

    start_time = time.time()

    # Label variables

    EXISTING = 1
    NEW = 3
    OLD = 7
    NEVER_SOLD = 9

    # Convert the Pandas df to Polars df
    df = pl.from_pandas(df_pandas)

    # Sort by store, item, and date
    df = df.sort(["store_nbr", "item_nbr", "date"])

    print(f"Elapsed time: {time.time() - start_time:.2f} seconds | LINE | df␣
 ↪sorted |")

    # Create a new column for item status, initialise to EXISTING
    df = df.with_columns(pl.lit(EXISTING).cast(pl.Int8).alias("item_status"))

    print(
        f"Elapsed time: {time.time() - start_time:.2f} seconds | LINE |␣
 ↪item_status added |"
    )

    # Filter for rows with unit_sales > 0 and calculate first/last sale dates
    first_sale_date = (
        df.filter(pl.col("unit_sales") > 0)
        .group_by(["store_nbr", "item_nbr"])
        .agg([pl.col("date").min().alias("first_sale_date")])
    )
```

```python
    last_sale_date = (
        df.filter(pl.col("unit_sales") > 0)
        .group_by(["store_nbr", "item_nbr"])
        .agg([pl.col("date").max().alias("last_sale_date")])
    )
    print(
        f"Elapsed time: {time.time() - start_time:.2f} seconds | LINE | first and last sale dates |"
    )

    # Join first and last sale dates to the original dataframe
    df = df.join(first_sale_date, on=["store_nbr", "item_nbr"], how="left")

    df = df.join(last_sale_date, on=["store_nbr", "item_nbr"], how="left")

    print(
        f"Elapsed time: {time.time() - start_time:.2f} seconds | LINE | joined sale dates |"
    )

    # Update the item_status column based on first and last sale dates
    df = df.with_columns(
        pl.when(pl.col("date") < pl.col("first_sale_date"))
        .then(pl.lit(NEW))
        .when(pl.col("date") > pl.col("last_sale_date"))
        .then(pl.lit(OLD))
        .otherwise(pl.col("item_status"))
        .alias("item_status")
    )

    # Handle NEVER_SOLD case where first_sale_date is null
    df = df.with_columns(
        pl.when(pl.col("first_sale_date").is_null())
        .then(pl.lit(NEVER_SOLD))
        .otherwise(pl.col("item_status"))
        .alias("item_status")
    )

    print(
        f"Elapsed time: {time.time() - start_time:.2f} seconds | LINE | updated item status |"
    )

    # Drop columns first_sale_date" and "last_sale_date" as these are not longer needed
    df = df.drop(["first_sale_date", "last_sale_date"])
```

```
    # Convert Polars df back to Pandas df
    df = df.to_pandas()

    print("-" * 72)
    print(f"Total execution time: {(time.time() - start_time) / 60:.2f}␣
↪minutes")
    print("- " * 36)
    print("df_grouped item_status value counts:")
    print(df["item_status"].value_counts())

    print("-" * 72)

    return df
```

4.8 Stockout on store level

- Perishable good: when there are missing values for two consecutive days for a given item per individual store
- Nonperishable goods: when there are missing values for 7 consecutive days for a given item and per individual store
- Action: Impute with algorithm

To-do: Add print function to keep track of type of inputations

To-do: .interpolate() –> ???

[32]:
```
import polars as pl



def impute_stockouts_polars(df_pandas, window_size=7):


    # Convert the input Pandas DataFrame to a Polars DataFrame for efficient␣
↪processing

    df = pl.from_pandas(df_pandas)


    # Sort the DataFrame by store number, item number, and date for consistent␣
↪ordering

    df = df.sort(["store_nbr", "item_nbr", "date"])

```

```
    # Nested function calc_missing_count to calculate the count of consecutive missing values in unit_sales

    def calc_missing_count(unit_sales):

        return (

            unit_sales.is_null()  # Check for null values

            .cast(pl.Int32)  # Cast to integer (1 for null, 0 for not null)

            .cum_sum()  # Cumulative sum to count sequential nulls

            .over(["store_nbr", "item_nbr"])  # Group by store_nbr and item_nbr
        )


    # Nested function to Inpute with rolling mean for missing values
    def rolling_mean_imputation(unit_sales, window_size):

        return (
            unit_sales.rolling_mean(
                window_size=window_size, min_periods=1
            )  # Impute strategy based on rolling mean
            .shift(
                1
            )  # Shift window by one day, to prevent taking the same day into account
            .over(["store_nbr", "item_nbr"])  # Group by store_nbr and item_nbr
        )


    # Apply the imputation logic based on the perishable status of the items

    df = df.with_columns(

        [

            pl.when(pl.col("perishable") == 1)  # Check if the item is perishable = 1
            .then(
                pl.when(

                    calc_missing_count(pl.col("unit_sales")) == 1

                )  # 1 missing value
```

```
                .then(0)  # --> Impute with 0
                .when(

                    calc_missing_count(pl.col("unit_sales")) > 2

                )  # More than 2 missing values

                .then(0)  # --> Impute with 0
                .when(

                    calc_missing_count(pl.col("unit_sales")) == 2

                )  # = 2 missing values
                .then(

                    rolling_mean_imputation(pl.col("unit_sales"), window_size)

                )  # --> Inpute with rolling mean for 2 missing days

                .otherwise(pl.col("unit_sales"))  # Otherwise keep original␣
↪value
            )

            .when(pl.col("perishable") == 0)  # If the item is not perishable =␣
↪0

            .then(
                pl.when(

                    calc_missing_count(pl.col("unit_sales")) > 7

                )  # More than 7 missing values

                .then(0)  # --> Impute with 0
                .when(

                    calc_missing_count(pl.col("unit_sales")) <= 7

                )  # if less 7 missing values
                .then(

                    rolling_mean_imputation(pl.col("unit_sales"), window_size)

                )  # --> Inpute with rolling mean for missing 7 or less days

                .otherwise(pl.col("unit_sales"))  # Otherwise keep original␣
↪value
            )
```

```
            .otherwise(pl.col("unit_sales"))  # For any other case not covered

            .alias("unit_sales")  # Alias the new column as 'unit_sales'

        ]
    )


    # Convert Polars df back to Pandas df

    df = df.to_pandas()

    return df
```

4.5 Promotional Data

- All missing values are interpreted a day with no promotion
- Action: Inpute onpromotion N/A with False

[33]:
```
# Fill missing N/A values in onpromotion column with False
def sales_fill_onpromotion(df):

    df["onpromotion"] = df["onpromotion"].fillna(False).astype(bool)

    return df
```

3 5 Feature construction

5.X Extracting datetime features

[34]:
```
def datetime_features(df):
    # Ensure the date column is sorted
    df = df.sort_values("date")

    # Add column with ISO year
    df["year"] = df["date"].dt.isocalendar().year.astype("int16")

    # Add column with weekday (1-7, where 1 is Monday)
    df["weekday"] = df["date"].dt.dayofweek.add(1).astype("int8")

    # Add column with ISO week number (1-53)
    df["week_nbr"] = df["date"].dt.isocalendar().week.astype("int8")

    # Calculate the date of the Monday of the first week
    first_date = df["date"].iloc[0]
    days_to_last_monday = (first_date.weekday() - 0 + 7) % 7
```

```
    monday_first_week = first_date - pd.Timedelta(days=days_to_last_monday)

    # Calculate cumulative week numbers starting from the first Monday
    df["week_number_cum"] = (
        ((df["date"] - monday_first_week).dt.days // 7) + 1
    ).astype("int16")

    return df
```

4 6. Data Manipulation and Feature construction –> Final-Function

[35]:
```
def manipulate_final_dataset(df):

    df = negative_sales_cleaned(df)

    df = merge_store_status(df)

    df = merge_item_status_polars(df)

    df = impute_stockouts_polars(df, window_size=7)

    df = sales_fill_onpromotion(df)

    df = datetime_features(df)

    return df
```

[36]:
```
df_final = manipulate_final_dataset(df_final)
```

```
Number of negative values before replacement: 7226
Number of negative values after replacement: 0
------------------------------------------------------------------------
Size of df:     8.65 GB and end observations:       320.2 million.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
df_grouped store_status value counts:
store_status
0    74347
2     4234
4      755
Name: count, dtype: int64
------------------------------------------------------------------------
Elapsed time: 26.64 seconds | LINE | df sorted |
Elapsed time: 26.86 seconds | LINE | item_status added |
Elapsed time: 47.57 seconds | LINE | first and last sale dates |
Elapsed time: 92.72 seconds | LINE | joined sale dates |
Elapsed time: 98.75 seconds | LINE | updated item status |
```

```
--------------------------------------------------------------------------------
Total execution time: 2.17 minutes
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
df_grouped item_status value counts:
item_status
1    168724590
3     82732874
9     63451920
7      5290712
Name: count, dtype: int64
--------------------------------------------------------------------------------

C:\Users\sebas\AppData\Local\Temp\ipykernel_19988\1347920879.py:4:
FutureWarning: Downcasting object dtype arrays on .fillna, .ffill, .bfill is
deprecated and will change in a future version. Call
result.infer_objects(copy=False) instead. To opt-in to the future behavior, set
`pd.set_option('future.no_silent_downcasting', True)`
  df["onpromotion"] = df["onpromotion"].fillna(False).astype(bool)
```

[37]:

```python
df_final.info()
# Count nulls per column
null_counts = df_final.isnull().sum()

# Print results
for column, count in null_counts.items():
    print(f"Column '{column}' has {count} null values.")
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 320200096 entries, 0 to 320200095
Data columns (total 19 columns):
 #   Column                  Dtype
---  ------                  -----
 0   store_nbr               uint8
 1   item_nbr                int32
 2   date                    datetime64[ns]
 3   unit_sales              float32
 4   onpromotion             bool
 5   holiday_local_count     int8
 6   holiday_national_count  int8
 7   holiday_regional_count  int8
 8   store_type              category
 9   store_cluster           uint8
 10  item_family             category
 11  item_class              uint16
 12  perishable              uint8
 13  store_status            int8
 14  item_status             int8
 15  year                    int16
 16  weekday                 int8
```

```
 17  week_nbr               int8
 18  week_number_cum        int16
dtypes: bool(1), category(2), datetime64[ns](1), float32(1), int16(2), int32(1),
int8(7), uint16(1), uint8(3)
memory usage: 12.8 GB
Column 'store_nbr' has 0 null values.
Column 'item_nbr' has 0 null values.
Column 'date' has 0 null values.
Column 'unit_sales' has 254899 null values.
Column 'onpromotion' has 0 null values.
Column 'holiday_local_count' has 0 null values.
Column 'holiday_national_count' has 0 null values.
Column 'holiday_regional_count' has 0 null values.
Column 'store_type' has 0 null values.
Column 'store_cluster' has 0 null values.
Column 'item_family' has 0 null values.
Column 'item_class' has 0 null values.
Column 'perishable' has 0 null values.
Column 'store_status' has 0 null values.
Column 'item_status' has 0 null values.
Column 'year' has 0 null values.
Column 'weekday' has 0 null values.
Column 'week_nbr' has 0 null values.
Column 'week_number_cum' has 0 null values.
```

[38]:
```python
df_final.head(10)
```

[38]:
```
           store_nbr  item_nbr       date  unit_sales  onpromotion  \
0                  1     96995 2013-01-01         NaN        False
89399856          14    421066 2013-01-01         NaN        False
279765744         48    275826 2013-01-01         NaN        False
89398168          14    420720 2013-01-01         0.0        False
89396480          14    419729 2013-01-01         0.0        False
89394792          14    418238 2013-01-01         NaN        False
279767432         48    276560 2013-01-01         NaN        False
89393104          14    418235 2013-01-01         NaN        False
89391416          14    418026 2013-01-01         0.0        False
279769120         48    278806 2013-01-01         NaN        False

           holiday_local_count  holiday_national_count  \
0                            0                       1
89399856                     0                       1
279765744                    0                       1
89398168                     0                       1
89396480                     0                       1
89394792                     0                       1
279767432                    0                       1
```

```
89393104                        0                          1
89391416                        0                          1
279769120                       0                          1

           holiday_regional_count store_type  store_cluster item_family  \
0                               0          D             13   GROCERY I
89399856                        0          C              7   GROCERY I
279765744                       0          A             14    CLEANING
89398168                        0          C              7       DAIRY
89396480                        0          C              7        DELI
89394792                        0          C              7   BEVERAGES
279767432                       0          A             14   GROCERY I
89393104                        0          C              7   BEVERAGES
89391416                        0          C              7        DELI
279769120                       0          A             14   GROCERY I

           item_class  perishable  store_status  item_status  year  weekday  \
0                1093           0             4            3  2013        2
89399856         1004           0             4            3  2013        2
279765744        3018           0             4            3  2013        2
89398168         2116           1             4            9  2013        2
89396480         2652           1             4            9  2013        2
89394792         1122           0             4            3  2013        2
279767432        1087           0             4            3  2013        2
89393104         1122           0             4            3  2013        2
89391416         2644           1             4            3  2013        2
279769120        1032           0             4            3  2013        2

           week_nbr  week_number_cum
0                 1                1
89399856          1                1
279765744         1                1
89398168          1                1
89396480          1                1
89394792          1                1
279767432         1                1
89393104          1                1
89391416          1                1
279769120         1                1
```

5 Write to Parquet fil and saves it in output_path

[39]:
```python
def save_dataframe_to_parquet(df, output_path, file_prefix="Prepped_data"):
    try:
        # Ensure the directory exists
        os.makedirs(output_path, exist_ok=True)
```

```
        # Generate today's date for the filename
        today = date.today().strftime("%Y%m%d")

        # Create the full filename with path
        filename = f"{file_prefix}_{today}.parquet"
        full_path = os.path.join(output_path, filename)

        # Save the DataFrame to a Parquet file
        df.to_parquet(full_path)

        print(f"DataFrame successfully saved to {full_path}")

        return full_path

    except Exception as e:
        print(f"Error saving DataFrame to Parquet file: {e}")

        return None
```

[40]:
```
# output_path = "C:/Users/alexander/Documents/0. Data Science and AI for␣
 ↪Experts/TEST"

output_path = "C:/Users/sebas/OneDrive/Documenten/GitHub/
 ↪Supermarketcasegroupproject/Group4B/data/interim"

saved_path = save_dataframe_to_parquet(df_final, output_path)
```

```
DataFrame successfully saved to C:/Users/sebas/OneDrive/Documenten/GitHub/Superm
arketcasegroupproject/Group4B/data/interim\Prepped_data_20240920.parquet
```

X # Function to print memory usage of DataFrames

[41]:
```
# Function to print memory usage of DataFrames
def print_memory_usage(dataframes):
    for name, df in dataframes.items():
        mem_usage = df.memory_usage(deep=True)
        total_mem = mem_usage.sum()

        print(f"DataFrame: {name}")
        print(mem_usage)
        print(f"Total Memory Usage: {total_mem} bytes\n")


# Check for DataFrames
dataframes = {
    name: obj for name, obj in globals().items() if isinstance(obj, pd.
 ↪DataFrame)
```

```
}
print_memory_usage(dataframes)
```

```
DataFrame: _
Index                       80
store_nbr                   10
item_nbr                    40
date                        80
unit_sales                  40
onpromotion                 10
holiday_local_count         10
holiday_national_count      10
holiday_regional_count      10
store_type                 472
store_cluster               10
item_family               3318
item_class                  20
perishable                  10
store_status                10
item_status                 10
year                        20
weekday                     10
week_nbr                    10
week_number_cum             20
dtype: int64
Total Memory Usage: 4200 bytes

DataFrame: df_sales
Index                128
id             501988160
store_nbr      125497040
item_nbr       501988160
unit_sales     501988160
onpromotion    250994080
day            125497040
year           125497200
month          125497324
date          1003976320
dtype: int64
Total Memory Usage: 3262923612 bytes

DataFrame: df_holidays
Index            128
date            2800
type             908
locale           650
locale_name     2476
description    12694
```

```
transferred      350
dtype: int64
Total Memory Usage: 20006 bytes

DataFrame: df_items
Index           128
item_nbr      16400
family         7408
class          8200
perishable     4100
dtype: int64
Total Memory Usage: 36236 bytes

DataFrame: df_stores
Index         128
store_nbr      54
city         2021
state        1667
type          516
cluster        54
dtype: int64
Total Memory Usage: 4440 bytes

DataFrame: df_final
Index                     2561600768
store_nbr                  320200096
item_nbr                  1280800384
date                      2561600768
unit_sales                1280800384
onpromotion                320200096
holiday_local_count        320200096
holiday_national_count     320200096
holiday_regional_count     320200096
store_type                 320200558
store_cluster              320200096
item_family                320203404
item_class                 640400192
perishable                 320200096
store_status               320200096
item_status                320200096
year                       640400192
weekday                    320200096
week_nbr                   320200096
week_number_cum            640400192
dtype: int64
Total Memory Usage: 13768607898 bytes

DataFrame: _38
```

```
Index                       80
store_nbr                   10
item_nbr                    40
date                        80
unit_sales                  40
onpromotion                 10
holiday_local_count         10
holiday_national_count      10
holiday_regional_count      10
store_type                 472
store_cluster               10
item_family               3318
item_class                  20
perishable                  10
store_status                10
item_status                 10
year                        20
weekday                     10
week_nbr                    10
week_number_cum             20
dtype: int64
Total Memory Usage: 4200 bytes
```